Parallel Network Intrusion Detection on Reconfigurable Platforms *

Chun Jason Xue[1], Zili Shao[2], MeiLin Liu[3], QingFeng Zhuge[4], and Edwin H.-M. Sha[4]

[1] City University of Hong Kong, Kowloon, Hong Kong jasonxue@cityu.edu.hk
[2] Hong Kong Polytechnic University, Hong Kong cszlshao@comp.polyu.edu.hk
[3] Wright State University, Dayton, Ohio 45435, USA meilin.liu@wright.edu
[4] University of Texas at Dallas, Richardson, Texas 75083, USA {qingfeng, edsha}@utdallas.edu

Abstract. With the wide adoption of internet into our everyday lives, internet security becomes an important issue. Intrusion detection at the network level is an effective way of stopping malicious attacks at the source and preventing viruses and worms from wide spreading. The key component in a successful network intrusion detection system is a high performance pattern matching engine that can uncover the malicious activities in real time. In this paper, we propose a highly parallel, scalable hardware based network intrusion detection system, that can handle variable pattern length efficiently and effectively. Pattern matchings are completed in O(log M) time where M is the longest pattern length. Implementation is done on a standard off-the-shelf FPGA. Comparison with the other techniques shows promising results.

1 Introduction

Network Intrusion Detection System (NIDS) performs packet inspection to identify, prevent and inhibit malicious attacks over internet. It can effectively stop viruses, worms, and spams from wide spreading. Pattern matching is the key component in the network intrusion detection systems. Using modern reconfigurable platforms, like FPGA, design and implement a parallel, high performance pattern matching engine for network intrusion detection is the goal of this paper.

Traditionally, network intrusion detection systems are implemented completely in software. Snort [20] is a well-known open source software network intrusion detection system. It matches pattern database against each packet to identify malicious target connections. With the rapid growth of pattern database, and the rapid growth of network bandwidth, software only solution can not process the internet traffic in full network link speed. A natural approach will be to move the computation intensive pattern matching to hardware. The main idea

* This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCR-0309461, USA and HK PolyU A-PH13, A-PA5X and A-PH41.

is to use specialized hardware resources along with a conventional processor. In this way, the conventional CPU can process all the general-computing tasks and the specialized co-processor can deal with string pattern matching, where parallelism, regularity of computations can be exploit by custom hardware resources.

Extensive researches exist on general pattern matching algorithms. The Boyer-Moore algorithm [6] is widely used for its efficiency in single-pattern matching problems. However, the current implementation of Boyer-Moore in Snort is not efficient in seeking multiple patterns from given payloads [3]. Aho and Corasick [1] proposed an algorithm for concurrently matching multiple strings. Their algorithm uses the structure of a finite automation that accepts all strings in the set. Two implementations of the Aho-Corasick algorithm have been done for Snort, by Mike Fisk [10] and Marc Norton [17], respectively. Fisk and Varghese [11] presented a multiple-pattern search algorithm that combines the one-pass approach of Aho-Corasick with the skipping feature of Boyer-Moore as optimized for the average by Horspool. The work by Tuck, et al. [22] takes a different approach to optimizing Aho-Corasick by instead looking at bitmap compression and path compression to reduce the amount of memory needed. All these approaches are developed mainly for software implementation. To examine packets in real time with full network link speed, a hardware solution is more favorable.

There are two main groups of hardware solutions for fast pattern matching. The first group generally applies finite state machine (FSM) to process patterns in sets. Aldwairi et al. [2] designed a memory based accelerator based on the Aho-Corasick algorithm. In their work, rules are divided into smaller sets that generate separate FSMs which can run in parallel. Hence significantly reduces the size of state tables and increases the throughput. Liu et al. [15] designed and implemented a fast string matching algorithm on network processor. Baker and Prasanna [4] proposed a pipelined, buffered implementation of the Knuth-Morris-Pratt algorithm [13] on FPGA. Li et al. [14] implemented rule clustering for fast pattern matching based on [9] with FPGA platform. Pattern size is limited by the available hardware resources. Tan and Sherwood [21] designed special purpose architecture working in conjunction with string matching algorithms optimized for the architecture. Performance improvement in this first group is generally achieved by dividing patterns into smaller sets, and deeply pipelining the pattern matching process. However, these type of approaches all have the shortcoming in scalability, when the pattern database grows exponentially, these type of approaches will suffer from extensive resources consumption and not able to maintain the same level of performance. Deep pipelining also has the side effect of increased latency, which is detrimental to some internet traffic.

The second group of hardware solutions uses hash tables as the foundation for pattern matchings. Dharmapurikar et al. [8] used bloom filters [5] to perform string matching. The strings are compressed by calculating multiple hash functions over each string. The compressed set of strings is then stored into a small memory which is queried to find out whether a given string belongs to the set. If a string is found to be a member of the bloom filter, it is declared as a possible match and a hash table or regular matching algorithm is needed to

verify the membership. Song and Lockwood [19] proposed a new version of the implementation that eliminates the need of hash table for match verification. Both of these implementation are done on FPGAs. Since patterns have different length, a bloom filter is constructed for each pattern length. This is simple but not efficient in handling variable pattern length.

This paper propose a novel hardware solution for pattern matching in NIDS. Our approach uses hash tables. However, we handle variable pattern length naturally from the beginning. Basically, we will slice each pattern into substrings of length 2^i, where $0 <= i <= k$, $k = \lfloor \log(M) \rfloor$, and M is the maximum pattern length. A hash table will be constructed for each substring length. There will be a total of k number of hash tables. Input string is processed in an iterative fashion. First, all substrings of length 2^k of the input string is matched against the hash table for substring length 2^k. Then, all substrings of length 2^{k-1} of the input string is matched against the hash table for substring length 2^{k-1}. Until all substrings of length one is matched against hash table for substring length one. A match is declared when all substrings of a pattern are matched. An extremely simple example is shown in Figure 1 to illustrate our idea.

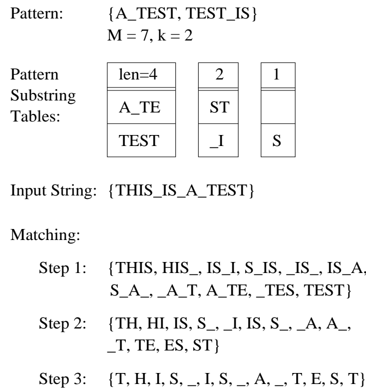


Fig. 1. An example.

In this example, there are only two patterns to be matched, "A_TEST" and "TEST_IS". The maximum pattern length $M = 7$. Hence $k = \lfloor \log(7) \rfloor = 2$. We slice patterns into substrings of length 2^2, 2^1, and 2^0. Pattern substring tables are built for each substring length. In this example, we show the exact value of the substring for illustration purpose. In reality, these tables will be hash tables for speed matching. An example input string is also shown in Figure 1. Using our approach, the matching is done in 3 steps. In step 1, all substrings of length 4 of the input string are matched against the pattern substring table with length equals 4. In step 2, all substrings of length 2 of the input string are matched against the pattern substring table with length equals 2. In step 3, all substrings of length 1 of the input string are matched against the pattern substring table

with length equals 1. A match of pattern "A_TEST" is declared when both substring "A_TE" and substring "ST" are matched during the process. The detail of the matching process and the data structures used will be presented in the rest of this paper.

In this paper, we use M to represent the maximum pattern length, example value of M could be 256 or 512. We use N to represent the number of patterns. A typical N would be 2k, which can fit the current snort rule set [20]. The main research contributions of this paper are:

– Handles variable pattern length efficiently and effectively while using hash tables.
– Finishes matching in O(log M) steps, where M is the maximum pattern length.
– Excellent scalability. Pattern matching performance is not affected by the growth of pattern database.

The remainder of this paper is organized as follows. The architecture of our technique is shown in Section 2. The concepts and data structures used in our approach are introduced in Section 3. Section 4 presents the algorithms. Section 5 presents the implementation on a reconfigurable platform. Section 6 presents the concluding remarks.

2 Architecture

The block diagram of our proposed architecture is shown in Figure 2. In this architecture, the core elements are an array of PEs (Processing Element). The number of PEs equals to the size of the input string S. A PE processes a substring of the input against all the same length substrings of the patterns. The input string is processed in rounds of different substring length. Each PE will first process all the 2^k bytes substring of the input string, then 2^{k-1}, etc.

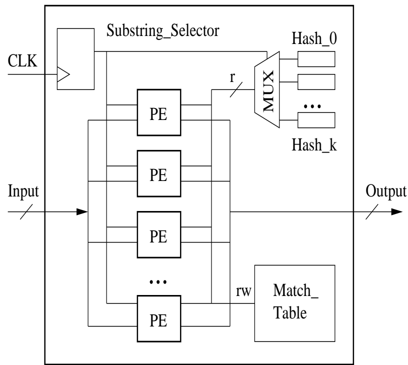


Fig. 2. Architecture Diagram.

The design diagram of a PE is shown in Figure 3. The inputs of a PE are a substring and a substring select signal that determines the length of the substring that will be worked on. First the input string will be passed to the hash function block and a hashing value will be obtained. This hash value will be used to do a hash table lookup. The result of hash table lookup will be passed to the match logic block to determine if there is a match or not. The design of each PE is kept simple. Duplicated hardware is used for the Match Logic block to increase the performance.

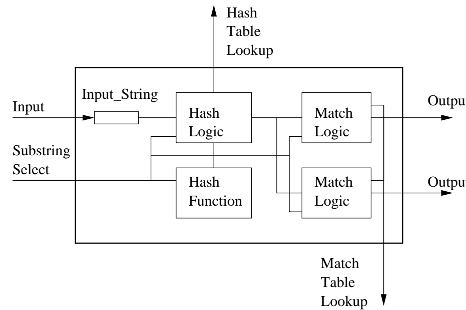


Fig. 3. A Processing Element (PE).

3 Basic Concepts and Data Structures

In this section, we introduce the basic concepts which will be used in the later sections. The data structures used in our approaches are also presented in this section.

Let us first define the problem that we are trying to solve. Assuming a packet carries a string S of length L, and we know a set of N patterns, $p[1], p[2], ..., p[N]$, the goal of Network Intrusion Detection System (NIDS) is to determine if there is any exact matching between pattern $p[i]$ and a substring of S. Let M be the maximum pattern length, and let $k = \lfloor logM \rfloor$. The main idea of our approach is to slice each pattern into substrings of length 2^i, where $0 \leq i \leq k$. Input data string S is read in as a whole and processed in rounds of different substring length. First all substrings of length 2^k are processed, then all substrings of length 2^{k-1}, etc. The whole matching are completed in k steps.

After finding a match of a substring, we will first decide if all the previous substrings in the pattern are matched, If yes, then a partial match is identified. And then, we will see if this is the last substring in the partially matched pattern. If yes, then an potential exact match is declared and a red flag will be raised by the network intrusion detection system and processed accordingly by the host system.

Three sets of data structures are used in our approach, and we will introduce them one by one. The first data structure of interest is the *Pattern_Length* table. It is an array that stores each pattern's length and indexed by the pattern ID. The binary representation of each pattern length shows what substrings that this pattern will be decomposed into. An example is shown in Figure 4. In this example, for the first pattern with pattern ID equals to 1 and length equals to 33, it will be sliced into a substring of length 32 and a substring of length 1, as depicted by its binary representation in Figure 4.

Pattern ID	Pattern Length	32	16	8	4	2	1
1	33	1	0	0	0	0	1
2	5	0	0	0	1	0	1
3	10	0	0	1	0	1	0
4	17	0	1	0	0	0	1

Fig. 4. Pattern_Length table.

The second set of data structure of interest is a set of hash tables that stores the pre-processed information for each substrings of each patterns. For pattern substrings of length 1, since there can only be 256 values, no hashing is done. Instead, a table of 256 entries is created. Each entry contains three elements, the first element is the value of this entry, the second element is the starting pattern ID, and the third element is the number of patterns that have the same value from the starting pattern ID. An example is shown in Figure 5. In this example, there are three patterns with value "a" as the last byte. Hence, in the *HASH_0* table, there is an entry with value equal to "a", starting pattern ID equal to 100, and number of consecutive patterns equal to 3.

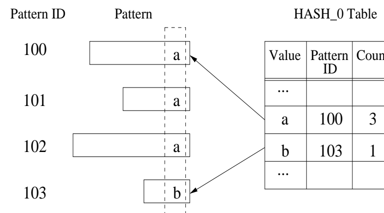
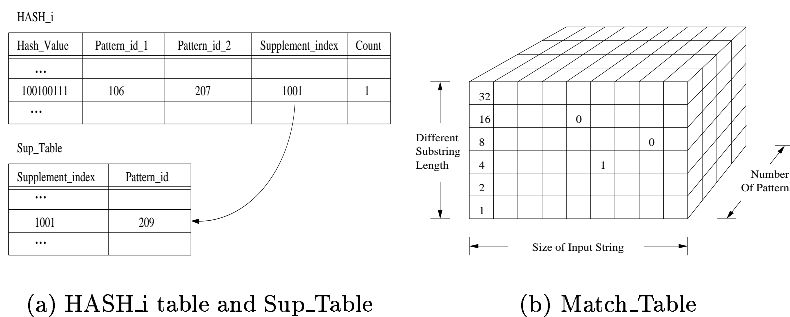


Fig. 5. HASH_0 table.

For substring length greater than 1, a hash table is constructed for each substring length. Hash table *HASH_i* correspond to substring length 2^i, where $i \neq 0$. Index of each hash table is the hashing value, and the entries in the hash tables are the pattern IDs. An example of hash table when substring length not equal to zero is shown in Figure 6(a). There are five columns in each hash table. Extra columns are used to handle hashing collisions. There are two sources of potential

hashing collisions exist in our scheme. First, different substrings could be hashed to the same hash value. Second, different patterns could have the same substring. For example, pattern “hell” and pattern “hello” have the same 4 bytes substring “hell”. To handle hashing collisions efficiently, for each hash value, we reserve two space for pattern ID in column two and column three respectively. These two pattern ID will be read in the same clock cycle and processed by hardware simultaneously. When there are more than 2 substrings are hashed to the same value, a separate table called *Sup_Table* is used to record these values. *Sup_Table* is also shown in Figure 6(a). Column four of the *HASH_i* table points to the starting Supplement_index, and column five identify the number of consecutive entries in the *Sup_Table* that have the same hash value. In the example shown in Figure 6(a), for hash value “100100111”, there are three patterns total have this hash value, pattern 106, pattern 207 and pattern 209 as recorded in *Sup_Table* in entry 1001.



(a) HASH_i table and Sup_Table

(b) Match_Table

Fig. 6. HASH Table and Match Table.

The third data structure that we use is the *Match_Table*, which is a three-dimensional bit array, with length equals to the input string length L, width equals to the number of patterns N, and the height equals to number of different substring length k. This table is used to record the substring matches found, which is in turn used for determining whole pattern match. For each substring match, a “1” will be recorded using the substring length, matched pattern id, and the position of the substring in the input string S. An example is showing in Figure 6(b). In this example, there are six different substring length, 1, 2, 4, 8, 16, and 32. Hence *Match_Table* has a height of 6.

4 Algorithms

In this section, the algorithms of our approach are presented. An example is given at the end of this section to show how the algorithms work. There are

two main algorithms in our approach. Algorithm *Init_Matching* shown in Algorithm 4.1 handles the initialization of all the necessary data structures. The second algorithm *Pattern_Matching* shown in Algorithm 4.2 processes the input strings for potential matchings.

Algorithm 4.1 Init_Matching

Require: A set of patterns p.
Ensure: Initialized data structures.

```
Sort all the odd length patterns by the value of the last byte;
for all pattern p[i] do
  Pattern_Length[i] = length(p[i]);
end for
for all pattern p[i] do
  for each substring s in p[i] do
    hashed_value = HASH(s);
    set HASH_j[hashed_value] = i;
  end for
end for
for j = 0 to 255 do
  Insert Starting Pattern ID and number of patterns into HASH_0 ;
end for
```

In algorithm *Init_Matching*, first all the odd length patterns are sorted by the value of the last byte. This is necessary for building the lookup table for substring length 1. Then for each pattern, *Pattern_Length* table is populated with the length of the pattern. Afterward, we will hash each substring of each pattern, and store the pattern ID accordingly. Based on our *HASH_i* table, there are two spaces to store pattern ID. We will first try to store the pattern ID of a particular hash value in one of these two spaces. If both of these two spaces are occupied, we will then place the pattern ID in the *Sup_Table* and update the last two columns of the *HASH_i* table accordingly. The last step of the *Init_Matching* algorithm populates the *HASH_0* table with the sorted pattern information. Updating the pattern set when we need to add or remove a pattern can be done in the similar fashion of Algorithm 4.1.

The main algorithm that processes each input string for potential matching patterns is algorithm *Pattern_Matching*. There are two functions notable used in Algorithm 4.2 , i.e., Pre_Substring(pl,i) and Post_Substring(pl,i), where pl is the pattern length and i is the current substring length. These two functions are used to determine if there is other substrings in the current pattern or not. If there are substrings before the current substring with length i in a pattern of length pl, Pre_Substring(pl,i) will return the previous substring length. Otherwise, Pre_Substring(pl,i) will return "0". Post_Substring(pl,i) will return "1" if there is any substring after the current substring with length i, and return "0" if the current substring is the last substring of the pattern. In *Pattern_Matching* algorithm, for each substring length and each substring, we will first run the

Algorithm 4.2 Pattern_Matching

```
Require: Input string S of length L.
Ensure: Yes/No. If there is a substring in the input string S that matches one pattern.
  for all substring length i do
    for all substring starting at position j of S do
      hashed_value ← HASH(substring);
      for each match in HASH_i; do
        k ← matched pattern ID;
        /* Find the pattern length for pattern k */;
        pl ← Lookup the Pattern_Length table for pattern k;
        /* Find the previous substring for pattern k */;
        pre_s ← Pre_Substring(pl,i);
        if (pre_s = 0) or ( pre_s > 0 and Match_Table[j − pre_s][pre_s][k] = 1) then
          Match_Table[j][i][k] = 1;
          if Post_Substring(pl,i) = 0 then
            Return Match_found= 1;
          end if
        end if
      end for
    end for
  end for
```

hash function to obtain a hash value. The hash value is used to lookup the corresponding hash table. If there are matches found in the hash table, for each matched pattern ID, we will examine its previous substrings and post substrings. If there is no previous substring or if there is a previous substring and it is also matched to the same pattern, we will mark "1" in the *Match_Table* for this input substring, at this substring length and this matched pattern. After we mark "1" in the Match_Table, if this substring also happens to be the last substring of the pattern, then we declare there is a potential match. Hash function is used heavily in our approach. Implementing hashing in hardware is relatively inexpensive. A class of universal hash functions called H_3 described in [18] were found to be suitable for hardware implementation. Our implementation of hash function falls into this class.

An example of the matching process is shown in Figure 7. This is a continuation of the simple example shown in the introduction section. The detail of the *Match_Table* is shown in Figure 7. In this example, there are one input string S that has 14 bytes, two patterns to be matched, and three different substring length. During the first round, where we match all substrings of length 4, there are two matches, one for "A_TE" and one for "TEST". Both matches lead to a marked "1" in the matching table. Moving on to the second round, where we match all substrings of length 2. Substring "ST" is matched, and since the previous substring of the same pattern is also marked as matched to the same pattern, "1" is marked for the "ST" substring match. Since substring "ST" has no substring after it, a match is declared. There is also a substring match of "_I"

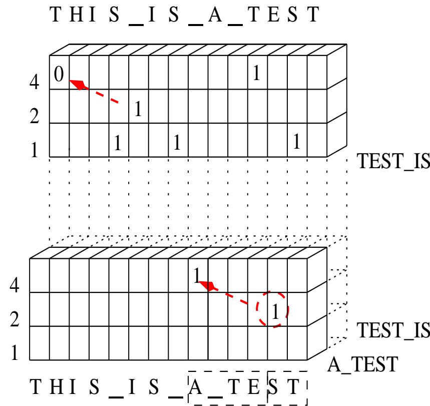


Fig. 7. Matching Example.

found, however, since the previous substring is not marked as matched, we do not mark "1" in the match table for the location where "_I" is matched.

5 Implementation

Design	Throughput	Unit Size (Logic cells)	Performance (Mb/s/logic cell)
UTD	4.7Gb/s	232	20.3
USC(no pipelining) [4]	1.8Gb/s	92	19.6
USC(pipeline)[4]	2.4Gb/s	120	20.0
Los Alamos [12]	2.2Gb/s	243	9.1
Wash U. - DFA [16]	0.952Gb/s	260	3.7
Wash U. - Bloom [8]	0.8Gb/s	0.76	1058
UCLA [7]	2.88Gb/s	160	18.0

Table 1. Comparison of throughput, unit size, and performance.

We have described our design in VHDL and targeted it to the Xilinx Virtex II architecture with -7 speed grade. We use the Xilinx ISE 7.1i and Mentor Graphic ModelSim 6.0 development tools. We have implemented a linear array of these PEs. Using a Virtex II XC2V6000, we are able to accommodate 128 PEs. This allows us to handle input string length of 128 bytes. The corresponding clock frequencies are 220 MHz. Since we need an average of six clock cycles to complete pattern matching for 128 bytes, hence our average throughput is 0.22x128/6 = 4.7 Gb/s.

Memory consideration in the implementation is very important in achieving high performance. In our design, *Match_Table* is the key in consolidating partial matches from substrings into full matches. Concurrent read/write accesses to

the *Match_Table* could be the bottleneck of our performance. In the real implementation, we actually slice the match table into thin slices. As shown in Figure 8, we can assign one slice of match table per PE. Each PE will write to its own slice of match table, and read from other slices of the match tables if needed. So the memory design requirement of the *Match_Table* becomes single write, multiple read instead of multiple write, multiple read. For our implementation on Xilinx Virtex II, we mapped each slice of match table into one 18kb Block SelectRAM. There are 3.5 Mb of total memory constituted by these 18kb Block SelectRAM on a chip [23]. We can fit all 128 slices of match tables easily. Memory implementation for the hash tables can be optimized in the same fashion. We can duplicate multiple copies of the hash tables and distribute among the PEs. Since we only need to read from the hash tables during the matching process, each copy of hash tables can be implemented using multi-port memory and shared among several PEs.

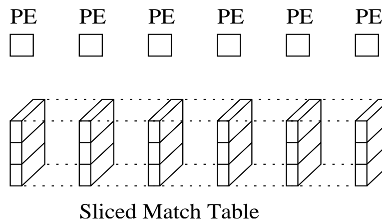


Fig. 8. Memory mapping for match_table.

Performance of our system can be further improved with the availability of more hardware resources. There are two ways that this performance gain could take place. First, we could use a larger FPGA that can accommodate 6x128 PEs. In this way, input strings can be processed in a pipelined fashion. At every clock cycle, there will be a 128 bytes string input and a 128 bytes string output. Second, multiple copies of the current design can be used in parallel to process multiple input streams at the same time. Either way, scalability can be achieved easily with the addition of new hardware resources.

The throughput, unit size, and performance of our design is compared with several other designs in Table 1. While generating high throughput, our design works relatively well in the unit size and performance. The real strength of our design comes when the number of patterns grows significantly and the speed of network increases dramatically, we do not have to make huge change in our design, only increase in hardware resources will make our design scale as needed.

6 Conclusion

In this paper, we propose a new hardware solution for NIDS. Our solution can handle variable pattern length efficiently while using the hash function approach. Pattern matching is processed in $O(\log M)$ steps, where M is the maximum pattern length. Enabling fast pattern matching is the key component in successful

network intrusion detection. As a next step, we plan to explore beyond exact pattern matching, identifying threats that are not exactly the same as the known patterns, but are variants of the known patterns.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. volume 18, pages 333–340, 1975.
2. M. Aldwairi, T. Conte, and P. Franzon. Configurable string matching hardware for speeding up intrusion detection. pages 99–107, 2005.
3. K. G. Anagnostakis, S. Antonatos, E. Markatos, and M. Polychronakis. E2xb: A domain-specific string matching algorithm for intrusion detection. 2003.
4. Z. K. Baker. Time and area efficient pattern matching on fpgas. pages 223–232, 2004.
5. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. volume 13, pages 422–426, 1970.
6. R. S. Boyer and J. S. Moore. A fast string searching algorithm. volume 20, pages 762–772, 1977.
7. Y. Cho, S. Navab, , and W. Mangione-Smith. Specialized hardware for deep network packet filtering. Sept. 2002.
8. S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. 2003.
9. C. K. et al. Automatic rule clustering for improved, signature based intrusion detection. 2002.
10. M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection. 2002.
11. M. Fisk and G. Varghese. Applying fast string matching to intrusion detection. 2004.
12. M. Gokhake, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Towards gigabit rate network intrusion detection. 2002.
13. D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in string. 1977.
14. S. Li, J. Torresen, and O. Soraasen. Exploiting reconfigurable hardware for network security. 2003.
15. R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A fast string-matching algorithm for nerwork processor-based intrusion detection system. pages 614–633, 2004.
16. J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an interenet firewall. 2003.
17. M. Norton and D. Roelker. Snort 2.0: Detection revised. 2002.
18. M. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. pages 1621–1636, 1994.
19. H. Song and J. Lockwood. Multi-pattern signature matching for hardware network intrusion detection systems. 2005.
20. Sourcefire. *Snort: The Open Source Network Intrusion Detection System*, 2003.
21. L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. pages 112–122, 2005.
22. N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. March 2004.
23. Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2004.