Configurable String Matching Hardware for Speeding up Intrusion Detection

Monther Aldwairi*, Thomas Conte, Paul Franzon

Department of Electrical and Computer Engineering, North Carolina State University,
Box 7911, Raleigh NC, 27695
{mmaldwai, conte, paulf}@ncsu.edu

***Abstract*—Signature-based Intrusion Detection Systems (IDSs) monitor network traffic for security threats by scanning packet payloads for attack signatures. IDSs have to run at wire speed and need to be configurable to protect against emerging attacks. In this paper we consider the problem of string matching which is the most computationally intensive task in IDS. A configurable string matching accelerator is developed with the focus on increasing throughput while maintaining the configurability provided by the software IDSs. Our preliminary results suggest that the hardware accelerator offers an overall system performance of up to 14Gbps.**

***Index Terms*—Intrusion detection, Snort accelerator, string matching**

I. INTRODUCTION

Traditionally networks have been protected using firewalls that provide the basic functionality of monitoring and filtering traffic. Firewall users can then write rules that specify the combinations of packet headers that are allowed through. However, not all incoming malicious traffic can be blocked and legitimate users can still abuse their rights. Intrusion Detection Systems (IDSs) go one step further by inspecting packet payload for attack signatures. Currently, most IDSs are software based running on a general purpose processor. Snort [1] is a widely used open-source IDS in which the rules refer to the header as well as to the packet payload. A sample Snort rule that detects CGI-PHF attack is shown in Fig. 1. The rule examines the protocol, source IP address, source TCP port, destination IP address and destination TCP port. The part enclosed in parenthesis is the rule options that are executed if the packet headers match. The content option indicates that the packet payload is to be matched against the string enclosed in double quotes.

Intrusion detection can be divided into two problems; packet filtering or classification based on header fields and string matching over the packet payload. The first problem was studied extensively in the literature and many algorithms were suggested [5]. A recent study [4] showed that the string matching routines in Snort account for up to 70% of the total execution time. We also have studied the snort rules and have showed that 87% of the rules contain strings to match against. Therefore, the second problem of string matching is the most computationally intensive.

```
alert tcp any any -> 10.1.1.0/24 80 (content:
                "/cgi-bin/phf")
```

Fig. 1. Sample Snort rule

The explosion of recent attacks by Code Red and MSBlast affected the productivity of computer networks all over the world. It is also becoming

*Phone 919 513 2015; fax 919 515 2285

increasingly difficult for software based IDSs running on general purpose processors to keep up with increasing network speeds (OC192 and 10Gbps at backbone networks). This has prompted the need to accelerate intrusion detection and to maintain the configurability needed to detect new attacks. Several hardware accelerators have been proposed. For example, Deterministic Finite Automata (DFA) mapped on an FPGA has been used to accelerate string matching. However, DFA based implementations achieve low throughput and are complex to build and configure. On the other hand, discrete or parallel comparators were used to achieve higher throughput at the expense of increased area and poor scalability. CAM based solutions reduce the area used by discrete comparators and achieve similar throughput. Finally, Bloom filters and hash functions were used to compress the string set, find probable matches and reduce the total number of comparisons.

This paper focuses the most computationally intensive part of the problem that is the string matching of the packet payload against hundreds of patterns at wire-speed. We suggest a memory based accelerator that is reconfigurable and have high throughput. The IDS accelerator is composed of a software based component that runs on a general purpose processor, and a standard RAM based technique for FSM implementation. The software generates an FSM from the set of strings extracted from the Snort rule database. The FSM matches multiple strings at the same time based on the Aho-Corasick string matching algorithm. This accelerator is flexible, easy to update and can achieve high throughput. The throughput increases as the on-chip RAM bandwidth increases.

The rest of the paper is organized as follows. Section II gives a background on string matching algorithms, summarizes IDS acceleration efforts and points out some of the differences between the proposed accelerator and the other architectures. Section III describes the suggested architecture for string matching. Section IV presents the simulation results, analyzes of the performance of the accelerator and compares it with previous work. Section V summarizes the contributions of this paper and discusses directions for future work.

II. RELATED WORK

Most known IDS implementations use a general purpose string matching algorithms, such as Boyer-Moore (BM) [3]. BM is the most widely used algorithm for string matching, the algorithm compares the string to the input starting from the rightmost character of the string. To reduce the large number of comparisons, two heuristics are triggered on a mismatch. The bad character heuristic shifts the search string to align the mismatching character with the rightmost position at which the mismatching character appears in the search string. If the mismatch occurs in the middle of the search string, then there is suffix that matches. The good suffix heuristic shifts the search string to the next occurrence of the suffix in the string. Fisk and Varghese suggested a set-wise Boyer-Moore-Horspoolalgorithm specifically for intrusion detection [12]. It extends BM to match multiple strings at the same time by applying the single pattern algorithm to the input for each search pattern. Obviously this algorithm does not scale well to larger string sets.

On the other hand, Aho-Corasick (AC) [2] is a multi-string matching algorithm, meaning it matches the input against multiple strings at the same time. Multi-string matching algorithms generally preprocess the set of strings, and then search all of them together over the input text. AC is more suitable for hardware implementation because it has a deterministic execution time per packet. Tuck *et al.* [13] examined the worst-case performance of string matching algorithms suitable for hardware implementation. They showed that AC has higher throughput than the other multiple string matching algorithms and is able to match strings in worst-case time linear in the size of the input. They concluded that their compressed version of AC is the best choice for hardware implementation of string matching for IDS.

We use a different method to store the AC database in an SRAM that achieves a higher throughput than Tuck's implementation while having a similar memory requirement. It works by building a tree based state machine from the set of strings to be matched as follows. Starting with a default no match state as the root node, each character to be matched adds a node to the machine. Failure links that point to the longest partial match state are added. To find

matches, the input is processed one byte at a time and the state machine is traversed until a matching state is reached. Fig. 2 shows a state machine constructed from the following strings {hers, she, the, there}. The dashed lines show the failure links, however the failure links from all states to the idle state are not shown. This gives an idea of the complexity of the FSM for a simple set of strings.

There have been several attempts to accelerate IDS recently, most of the implementations used regular expressions. Regular expressions are generated for every string in the rule set and a Nondeterministic/ Deterministic Finite Automata (N/DFA) that examines the input one byte at a time is implemented. FAs are complex, hard to implement, have to be rebuilt every time a string is added and result in designs with a modest throughput. Sidhu and Prasanna mapped an NFA into an FPGA [5]. Carver *et al.* wrote a regular expression generator in JHDL that extracts strings from the Snort database, generates regular expressions and a netlist for a Xilinx FPGA [6].

Other architectures used discrete comparators to exploit parallelism and achieve higher throughput. The disadvantage of this approach is the large area required. Cho *et al.*, for example, used four parallel comparators per string [10], and Sourdis *et al.* used pipelining as well as discrete comparators to double the throughput [7]. Several implementations [8, 11] have used CAMs and DCAMs along with comparators to reduce the area and achieve similar throughput to the discrete comparators implementations. The drawback is the high cost and the high power requirement of CAMs.

Recently, Dharmapurikar *et al.* [9] used bloom filters to perform string matching. The strings are compressed by calculating multiple hash function over each string. The compressed set of strings is stored into a small memory which is then queried to find out whether a given string belongs to the compressed set. If a string is found to be a member of a bloom filter, it is declared as a possible match and a hash table or regular matching algorithm is needed to verify the membership. Bloom filters use less memory, are easy to reprogram and achieve a higher throughput than DFA implementations.

Tuck *et al.* [13] stored the high level nodes including the pointers to the next and failure states in the RAM. Because of that a huge memory of about 53MB was needed to store the Snort rules set. They used the analogy between IP forwarding and string matching to apply bit-mapping and path compression to the AC tree, reducing its size to 2.8MB and 1.1MB, respectively. Our approach stores the state tables in the RAM and uses a minimal logic to traverse the tables and find a match. The state tables are around 3MB in size without the use of any compression techniques.

III. ACCELERATOR ARCHITECTURE

The accelerator is a part of the configurable network processor architecture shown in Fig. 3. It consists of a 2-wide multiple issue VLIW processor with hardware support for eight hyper threads. The memory system consists of multi-port RAM and a high speed DMA. A number of configurable accelerators are used to speed up specific networking tasks such as IP forwarding, quality of service and string matching for intrusion detection.

The IDS is composed of two components; a software that runs on the VLIW core and a hardware string matching accelerator. The software extracts the strings from the Snort database, creates the FSM tree and generates the state tables. The hardware is shown in Fig. 4. It implements a Mealy FSM and consists of a RAM to store the state tables, a register to hold the current state, and control logic to access the RAM and find a match.

Incoming packets need to be matched only against a subset of rules that match the packet header in the Snort database. To avoid creating one large complicated FSM for all of the strings in the database, the software performs a simple rule classification resulting in a smaller FSM or state table for every class. Rules are classified based on the header fields, mainly the protocol and port numbers, into classes such as ICMP, FTP, SMTP, Oracle, Web-CGI…etc. This makes the software faster, reduces the RAM size and exploits parallelism between packets to increase the throughput.

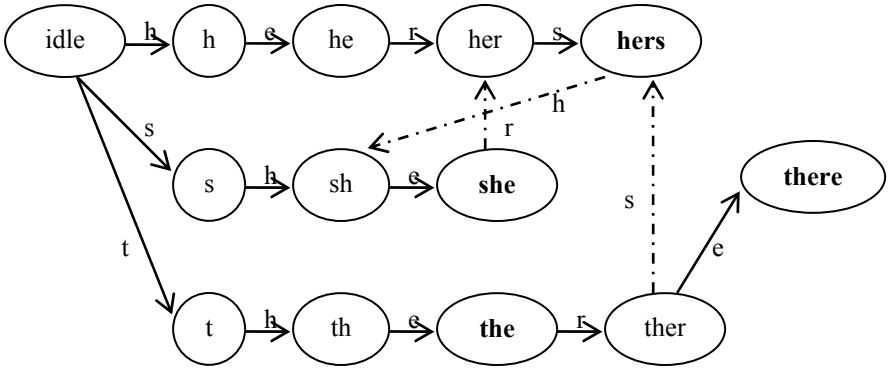

Fig. 2. Finite state machine diagram

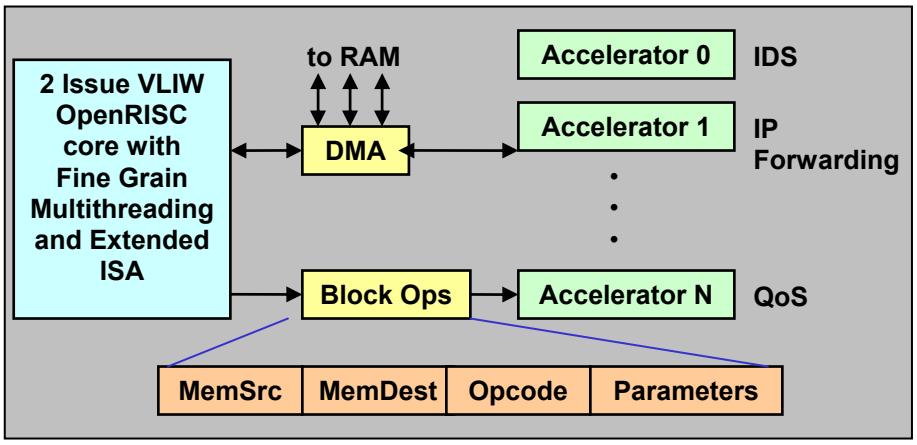

Fig. 3. Configurable network processor architecture

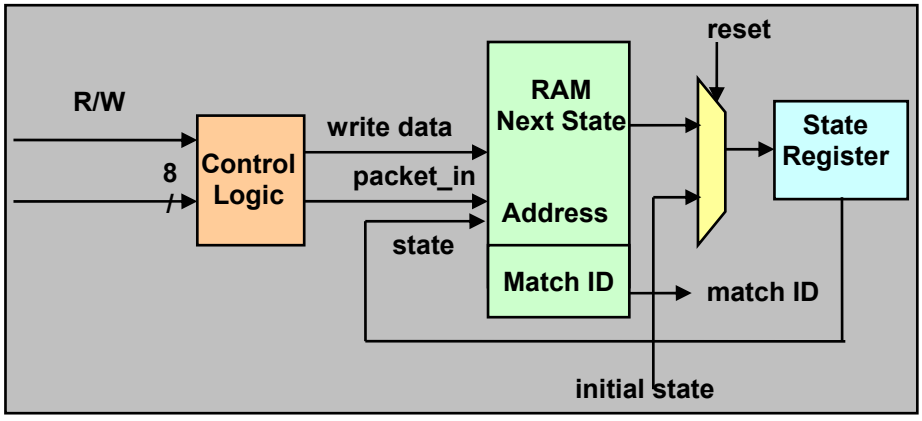

Fig. 4. String matching accelerator

Table 1. State table

Current State \ Input Character	e	h	r	s	t
- /(idle)	-,0	h,0	-,0	s,0	t,0
h	he,0	h,0	-,0	s,0	t,0
he	-,0	h,0	her,0	s,0	t,0
her	-,0	h,0	-,0	hers,1	t,0
hers	-,0	sh,0	-,0	s,0	t,0
s	-,0	sh,0	-,0	s,0	t,0
sh	she,2	h,0	-,0	s,0	t,0
she	-,0	h,0	her,0	s,0	t,0
t	-,0	th,0	-,0	s,0	t,0
th	the,3	h,0	-,0	s,0	t,0
the	-,0	h,0	ther,0	s,0	t,0
ther	there,4	h,0	-,0	hers,1	t,0
there	-,0	h,0	-,0	s,0	t,0

The state table for the string set and state diagram discussed in the previous section is shown in Table 1. The rows are indexed by the current state and the columns by the input character. Every element contains a pair of values; the next state and a matching string ID. The table is mapped onto a RAM, where the address of the RAM is the concatenation of the current state and the input character. The content of the RAM is the concatenation of the next state and the Match ID. Although the address and content of the RAM are assumed to be 32 in the simulations, they could be any value depending on the number of states. The right most 24 bits represent the state and the left most 8 bits represent the input character or the Match ID.

The control logic is responsible for building and updating the string tables as well as matching the strings. A simple FSM matches the packet content against the string set a byte at a time. The FSM generates the RAM address, reads the next state and Match ID, and exits if Match ID doesn't equal to zero (i.e. a match is found). If Match ID is zero the FSM processes the next input byte until the end of packet content or a match is reached. To add a new string, the state table has to be rebuilt and written to the RAM. Because the rules are classified into several classes, only one state table needs to be rebuilt.

IV. RESULTS

In this section we present the results of our Snort rules study and the simulations of the accelerator. The first subsection presents statistics about snort rule set. In Subsection B we derive an equation for memory size and present the experimental memory requirements. In Subsection C we examine the performance of our accelerator. Finally, in Subsection D we compare our design to the other hardware accelerators.

A. Snort Rules Analysis

Our study of the Oct. 2003 Snort rules set showed that 1542 of the total 1777 rules studied (about 87%) contained strings to match against the packet payload. This demonstrates the strong need for hardware acceleration of the string matching aspect of the IDS problem. Fig. 5 shows the distribution of the string lengths in bytes. We can see that the average string length is 14 bytes and the majority of the strings are shorter than 26 bytes. It is also clear that there is a non-negligible number of strings longer the 40 bytes. Our simulator took into consideration all ASCII characters including the non-printable characters and parsed the hexadecimal strings included in most Snort rules. To make sure that the strings accurately represent the rules they were extracted from. Multiple strings in the same rule within a distance of zero were combined into one string.

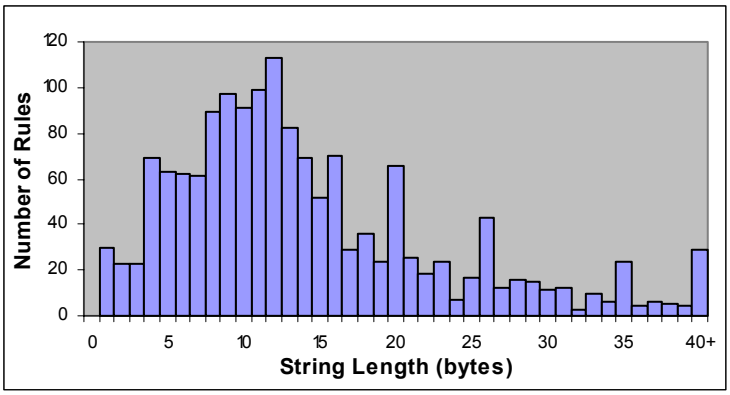

Fig. 5. Distribution of the string lengths in the Snort database

B. Memory Size

The RAM is the major component in the accelerator; it dictates the size and limits the throughput. As shown in Fig. 6 the disadvantage is that the memory requirement increases linearly with the number of states which in turn depends on the number of characters in the string set. The memory requirement in bits was derived in terms of the number of states, number of strings and the number of different characters in the string set. Equation (1) shows the memory requirement, where s is the number of states, n is the number of strings and c is the number of characters per set.

$$RAM = \left(\lceil \log_2 s \rceil + \lceil \log_2 n \rceil\right) * s * c \qquad (1)$$

The number of states depends on the number of strings and the number of characters per string. As we mentioned earlier the rules were classified by headers to reduce the sizes of the FSMs and state tables. Table 2 shows the RAM requirement only for the largest rule classes. The RAM requirement for the largest rules classes (web-cgi, web-misc) is around 750KB. The size of the state tables for all of Snort 2003 rule set is around 3MB which can be fitted on-chip. By applying state minimization and compression techniques we expect to shrink the state tables' sizes even more.

Table 2. RAM size in bytes for different rule classes

Rule class	Rules	Strings	States	RAM (bytes)
FTP	50	49	268	43,997
SMTP	18	24	362	56,840
ICMP	22	11	138	17,501
RPC	124	58	720	132,623
Oracle	25	25	265	40,366
Web-CGI	311	311	3133	747,939
Web-Misc	275	275	3242	768,975
Web-IIS	108	108	1514	314,652
Web-PHP	58	58	914	172,132
Web-Coldfusion	35	35	572	98,081
Web-Frontpage	34	34	367	59,926
Other classes	717	554	-	709,963
Total	1777	1542	-	3,118,996

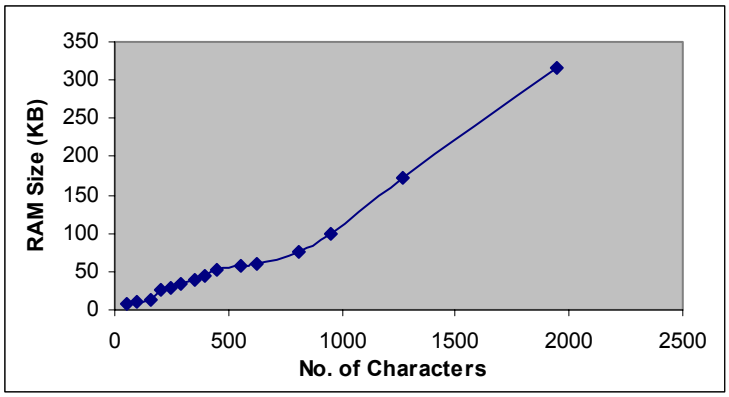

Fig. 6. RAM size in bytes for different character counts

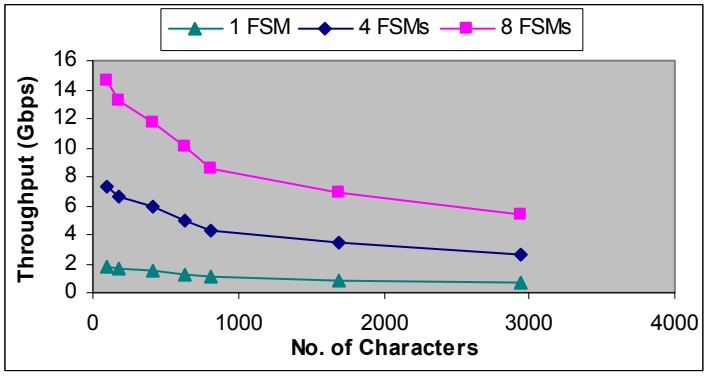

Fig. 7. Throughput for different character counts

Table 3. Comparison of string matching implementations

Description	Input Bits	Device	Through-put(Gbps)	Logic Cells/Char
Aldwairi *et al.* State tables/RAM	64	Altera EP20k400E	10.1	15
	32	Altera EP20k400E	5.0	
Sourdis *et al.* [8] Pre-decoded CAMs	32	Virtex2 6000	9.7	3.56
Gokhale *et al.*[11] CAMs/Comparators	32	VirtexE-1000	2.2	15.2
Cho *et al.*[10] Discrete Comparators	32	Altera EP20K	2.9	10.6
Sidhu *et al.*[5] NFAs/Regular Expression	8	Virtex 100	0.75	~31

C. Performance

The AC string matching algorithm has a deterministic worst-case lookup time. Once the state tables are generated and stored in the RAM, the packet is processed one byte at a time, and every byte requires one access to the RAM. The processing time mainly depends on the length of the packet and the RAM access time. The RAM access time depends on the RAM size which is proportional to the number of strings and the number of characters per string. As discussed in Section III, using a simple classification technique to divide the rules into smaller rule sets generates separate FSMs that can run in parallel. This

not only significantly reduces the size of the state tables but also increases the throughput by exploiting parallelism between the packets and different rule classes.

Fig. 7 shows the throughput of the accelerator where CACTI version 3.2 [14] was used to model the on-chip RAM. The figure plots the performance in terms of processing throughput (Gbps) for different FSM counts (1, 4, 8) and for rule sets with sizes up to 3,000 characters. We see that higher throughput is achieved by using more FSMs in parallel. By using 8 FSMs a throughput of around 14Gbps is achieved as opposed to 7 and 2Gbps for 4 and 1 FSM(s), respectively. We also notice that the throughput degrades as the number of characters grows. The throughput for 8 parallel FSMs decreases to about 5Gbps for 3000 characters. The performance degradation is due to the fact that as the number of characters increases, the number of states increases and the state table size increases as well.

D. Comparison with Previous Work

Table 3 compares the performance of our design with regular expressions/FAs, discrete comparators and CAMs based designs. The performance numbers for Bloom filters implementation were not available. The data for the other designs was obtained from Sourdis *et al.* [8]. It is clear that our design out performed Sidhu's NFAs, Cho's discrete comparators, and Gokhale's CAMs in terms of throughput. On the other hand, Sourdis's pre-decoded CAMS used extensive fine grain pipelining to increase the throughput to 9.7 Gbps. By using 8 FSMs (i.e. 64 bit input) our accelerator achieves a throughput of about 10 Gbps and exceeds pre-decoded CAMs speed. Another advantage over Sourdis's design is the low cost and power requirements of RAMs compared to CAMs.

V. CONCLUSIONS AND FUTURE WORK

We have studied Snort rules set and have shown that 87% of the rules have content. This further emphasizes the need for hardware acceleration for content matching. We have also presented a configurable string matching accelerator based on a memory implementation of the AC FSM where the state tables are directly stored in the RAM rather than the high level tree data structure. This results in a small memory requirement that is likely to fit in on-chip SRAM. We have shown that the accelerator can achieve up to 14Gbps throughput with a simple classification algorithm which highlights the importance of classification algorithms. We also showed that our design out performed the previous work published in this area.

Despite the work done in this area there is still room for improvement, our immediate goals are to apply FSM minimization and compression techniques to further reduce the size of the RAM. Furthermore, it would be interesting to use a better classification algorithm to exploit parallelism and achieve higher throughput.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their useful comments on this paper. This work was supported by DARPA under contract F33615-03-C-410.

REFERENCES

[1] M. Roesch. Snort - lightweight intrusion detection for networks, in *Proceedings of LISA99, the 13th Systems Administration Conference*. 1999.

[2] A. Aho and M. Corasick. Efficient string match-ing: An aid to bibliographic search, In Communications of the ACM, vol. 18, no. 6, pp.333-343, June 1975.

[3] R. Boyer and J. Moore. A fast string searching algorithm. In Communications of the ACM, vol.20, no 10, pp762–772, October 1977.

[4] S. Antonatos, K. Anagnostakis, and E. Markatos. Generating realistic workloads for network intrusion detection systems. In ACM Workshop on Software and Performance, 2004.

[5] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In IEEE Symposium on Field Programmable Custom Computing Machines (FCCM01), April 2001.

[6] D. Carver, R. Franklin, and B. Hutchings. Assisting network intrusion detection with reconfigurable

hardware. In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM02), April 2002.

[7] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a network intrusion detection system. In Proceedings of 13th International Conference on Field Programmable Logic and Applications, 2003.

[8] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In Proceedings of 12th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM04), April 2004

[9] S. Dharmapurikar, M. Attig and J. Lockwood. Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters. In the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04), April 2004

[10] Y. Cho, S. Navab and W. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In Proceedings12th International Conference on Field-Programmable Logic and Applications, Sept. 2002

[11] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, Sept. 2002

[12] M. Fisk and G. Varghese. Applying Fast String Matching to Intrusion Detection, SEP 2002

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

[14] http://research.compaq.com/wrl/people/jouppi/CACTI.html