A Virus Scanning Engine Using a Parallel Finite-Input Memory Machine and MPUs

Hiroki Nakahara†, Tsutomu Sasao†, Munehiro Matsuura†, and Yoshifumi Kawamura††
† Kyushu Institute of Technology, Japan †† Renesas Technology Corp., Japan

Abstract

This paper presents a virus scanning engine. After showing the difference between ClamAV (an anti-virus software) and SNORT (an intrusion detection software), we show a new architecture for the virus scanning engine, which is different from that of the intrusion detection engine. The new architecture consists of a parallel finite-input memory machine (PFIMM) and general purpose MPUs. It uses two-stage matching. That is, in the first stage, the parallel hardware filter quickly scans the text to find partial matches, and in the second stage, the MPU scan the text to find the total match. To reduce the memory size, compressed match vectors are used. The system is implemented on the Stratix III FPGA, where 65,536 ClamAV virus patterns are stored. As for the area-performance ratio, our system is 1.2-26.3 times more efficient than existing ones.

1 Introduction

A **malware** (a composite word from <u>mal</u>icious soft<u>ware</u>) intends to damage computer systems. With the wide use of the Internet, users can easily access and download dangerous data. So, the risk of infection by the malware is increasing. Malware secretly installs a bot virus, a back door, or a keylogger. As a result, the exploitation of the password, the stealing of the information, and illegal remote operation can do damage to computer users. Although a software-based virus scanning system can clean and isolate the malware, throughput for software-based scanning is at most tens of mega bits per second (Mbps) [12]. Thus, the software-based approach cannot keep up with the modern Internet throughput which is more than one giga bits per seconds (Gbps). Malware is becoming more prevalent and more complex, and so virus scanning on computer systems will be a bottleneck in the future. Recently, hardware based virus scanning systems are attached to the gateway between the Internet and the Intranet [16]. The most important part of the virus scanning system is the virus scanning engine. Other part can be realized by the conventional technique. So, in this paper, we show a virus scanning engine that uses the parallel finite-input memory machine and MPUs. Some virus scanning software, e.g., Kaspersky [8], updates the virus data every an hour. Although, the random logic implementation of the virus scanning circuit on the FPGA [14] is fast and compact, the time for the place-and-route is longer than the period for the virus pattern update. This implies that such system must be suspended during the update. To reduce the memory size, we introduce the finite-input memory machine (FIMM) that quickly scans the text to find partial matches. Then, we partition the FIMM to further reduce memory size. Since the FIMM only detects partial matches, an MPU is used to find the total match. This is called the two-stage matching [5, 18]. The proposed engine is memory-based, so the power consumption is lower than the TCAM-based ones [3, 18].

The rest of the paper is organized as follows: Chapter 2 introduces the virus scanning; Chapter 3 describes the two-stage matching using FIMM; Chapter 4 shows the compression methods of the FIMM; Chapter 5 shows the implementation results on Altera FPGA; and Chapter 6 concludes the paper.

2 Virus Scanning

2.1 Virus Scanning Problem

A **virus scanner** detects the malware on executable code or data. **Text** denotes a string of characters in which there is a possible virus. The malware is specified by a **pattern**[1] represented by a **restricted regular expression**. Virus scanning corresponds to the pattern matching that detects variable length patterns in the text.

2.2 Restricted Regular Expression for Virus Scanning in ClamAV

A pattern consists of **sub-patterns**, and each sub-pattern is represented by a regular expression consisting of **characters** and **meta characters**. A character is represented by 8 bits, or a pair of hexadecimal numbers. The **length** of a pattern is the number of characters in the pattern. In this paper, k denotes the number of patterns, and c denotes the length of a pattern. Table 1 shows meta characters used in the ClamAV [6], and Table 2 shows examples of virus patterns in ClamAV. Table 3 compares the current version of ClamAV (v.0.94.2) with that of SNORT (v.2.8.3.2) [13]. It shows that regular expressions for ClamAV are simpler

[1]Pattern is also called a **signature**

than that for SNORT. However, the number of patterns in ClamAV is much larger than that in SNORT. Thus, in the virus scanning system, a memory efficient architecture is required.

Table 1. Meta Characters Used in ClamAV.

	Meaning
??	An arbitrary character
*	Repetition of more than zero (Kleene closure)
()	Specify the priority of the operation
|	Logical OR
$\{n, m\}$	Repetition (more than n and less than m)

Table 2. Virus Patterns in ClamAV.

Virus Name	Pattern
Trojan.Bat.DelY-3	64656c74726565{-1}2f(59|79)20633a5c2a2e2a
Trojan.Bat.DelY	44454c54524545202f(59|79)20633a5c2a2e2a
Trojan.Bat.MkDir.B	406d64202572616e646f6d25**????**676f74 6f20486f6f
W32.Gop	736d74702e796561682e6e65*2d20474554 204f494351
Worm.Bagle-67	6840484048688d5b0090eb01ebeb0a5ba9ed46

Table 3. Comparison ClamAV with SNORT

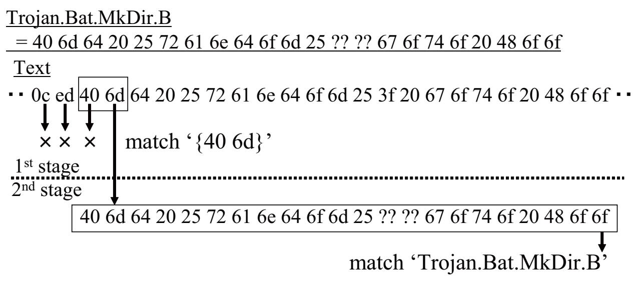
	ClamAV	Snort
# of patterns	514,287	3,533
average pattern length	32.9	193.7
average # of meta-characters	0.081	46.7

2.3 Aho-Corasick Method

Here, we briefly introduce the **Aho-Corasick method (AC method)** [1]. The AC method performs the string matching using an **AC automaton**. To obtain the AC automaton, first, the given patterns are represented by a text tree (Trie). Next, the failure paths that indicate the transitions for the mismatches are attached to the text tree. Since the AC automaton stores failure paths, no backtracking is required. By scanning the text only once, the AC automaton can detect all the patterns. The AC automaton can be realized by a state machine consisting of a the memory and a register. The memory stores the state transition functions and the output functions, while the register stores the state variables. The number of bits for the register is much smaller than that for the memory, so we ignore the bits for the register. In this paper, *the amount of memory* denotes the memory bits that stores the state transition functions and the output functions.

2.4 Two-stage Matching

An AC automaton accepting patterns of length c. Table 3 shows that, for ClamAV, the average length of pattern is $\bar{c} = 32.9$. Hence, the AC automaton is quite large. ClamAV uses the **two-stage matching** to achieve a high-speed and memory-efficient system. The first stage scans the text to find a partial match consisting of three characters using a compact AC automaton [5]. The second stage exactly scans the text to find the total match using hash tables when the first stage detects the partial match. To perform the second stage, additional off-chip memory and the MPU are used.



Trojan.Bat.MkDir.B
= 40 6d 64 20 25 72 61 6e 64 6f 6d 25 ?? ?? 67 6f 74 6f 20 48 6f 6f
Text
・・ 0c ed 40 6d 64 20 25 72 61 6e 64 6f 6d 25 3f 20 67 6f 74 6f 20 48 6f 6f ・・
× × ×
match '{40 6d}'
1st stage
2nd stage
40 6d 64 20 25 72 61 6e 64 6f 6d 25 ?? ?? 67 6f 74 6f 20 48 6f 6f
match 'Trojan.Bat.MkDir.B'

Figure 1. An Example of the Two-stage Matching.

Although the two-stage matching is slower than the original AC method, it drastically reduces the memory size for the AC automaton.

Example 2.1 *Fig. 1 illustrates the two-stage matching that detects the pattern of Trojan.Bat.MkDir.B in Table 2. First, it detects* $406d$*, then it detects the whole pattern of Trojan.Bat.MkDir.B.* *(End of Example)*

2.5 Profile Analysis for Virus Scanning

To analyze the profile of the two-stage matching on a PC, we selected 512 patterns from ClamAV, and performed the two-stage matching for 10 executable codes. The profile analysis shows that the first stage spends 83% of the CPU time, while the second stage spends 17% of the CPU time. Thus, to improve the throughput, we considered the following:

High-speed AC automaton to improve the first stage. We use hardware instead of software. For the virus scanning, since the packet can be inspected independently, a parallel processing using hardware is applied.

Reduction of the partial matches in the first stage to reduce the load of the second stage. Increasing the length c reduces the partial matches, and reduces the work for the MPU. However, this increases the amount of memory in the first stage.

3 Two-stage Matching Using FIMM

3.1 Finite-Input Memory Machine

Since the state transitions for the AC automaton is complex, the size of memory tends to be large. For the intrusion detecting system, bit partitioning is used to reduce the size of the circuit [15]. However, for virus scanning system, the size of the circuit would be too large even if the bit partitioning method is used[2]. By restricting transitions, we have a **finite-input memory machine (FIMM)** [9] shown in Fig. 2. In Fig. 2, *Reg* denotes an 8-bit parallel-in parallel-out register. The FIMM stores the past c inputs to the shift register. The memory produces the match number. In the FIMM, it contains many equivalent states. Thus, the number of states of the FIMM is much larger than that of AC

[2]It will be shown in Table 4

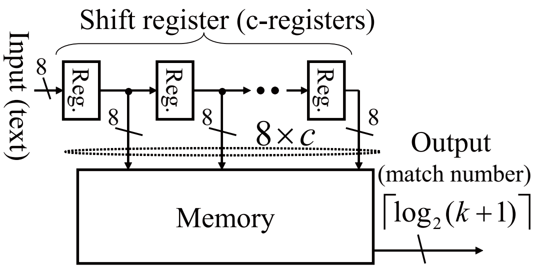


Figure 2. Finite-Input Memory Machine (FIMM).

automaton. However, this eliminates the circuits for transition functions. Furthermore, we can reduce the size of the memory drastically by bit partitioning to be explained in the next part.

3.2 Bit Partitioned FIMM

Let M_{FIMM} be the size of memory to realize the output function of the FIMM, k be the number of patterns, and c be the length of the patterns. Then, we have

$$M_{FIMM} = 2^{8c}\lceil log_2(k+1)\rceil. \quad (1)$$

As started before, the amount of memory for the state variables realized by the shift register is much smaller than that for the output functions, so it is neglected. The number of inputs for the memory is $8c$, so the direct implementation of the FIMM by a single memory and a register is expensive. To further reduce the memory size, we partition the FIMM into r units. In other words, we construct r independent sequential machines. Fig. 3 shows the **bit-partitioned FIMM** (r FIMM). Since each memory only scans $\frac{8}{r}c$ bits out of $8c$ bits, non-stored patterns may be matched. To solve this problem, the output for each FIMM is encoded to 1-hot code to form a **MV (Match Vector)**. Then, the match number is detected by the bitwise-AND operation of all the MVs. Let k be the number of patterns, then the MV consists of k bits. Let M_{rFIMM} be the total amount of memory for r FIMM. Then, we have

$$M_{rFIMM} = 2^{\frac{8}{r}c}k \times r, \quad (2)$$

where $r = 1, 2, 4$, and 8. When $r = 8$, Expr. (2) takes its minimum. Thus, we partition the FIMM into eight small FIMMs. The resulting circuit (**8_FIMM**) consists of eight memories, eight shift registers, and a bitwise-AND. From Expr. (2), the total amount of memory for the 8_FIMM is

$$M_{8FIMM} = 2^{c}k \times 8 = 2^{c+3}k. \quad (3)$$

This is much smaller than the size of memory obtained by [15].

3.3 Virus Scanning Engine

Fig. 4 shows the virus scanning engine consisting of 128 units of 8_FIMMs. It is called **PFIMM(Parallel FIMM)1024**, since it uses 1024 memories. The FIFO stores the pattern numbers of partial matches and the positions for the detected sub-patterns. When the sub-pattern is detected, the FIFO sends an interrupt signal (IRQ) to the MPU. When the MPU accepts an IRQ, it scans the full text to check if it is a total match or not.

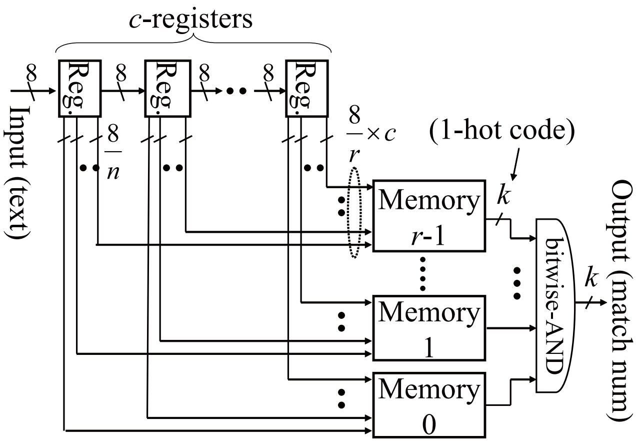


Figure 3. Bit-Partitioned FIMM (r_FIMM).

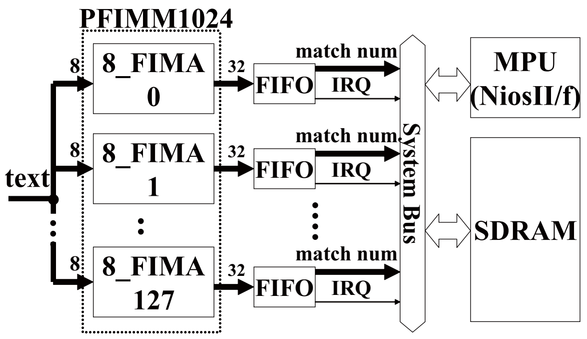


Figure 4. Virus Scanning Engine.

4 Compression of the Match Vectors

4.1 CMV (Compressed Match Vector)

For the 8_FIMM storing k patterns, an MV consists of k bits. When k is large, e.g., $k = 512$, the memory is too large to implement, and the bitwise-AND circuit is too large. This decreases the throughput. In this paper, we compress the

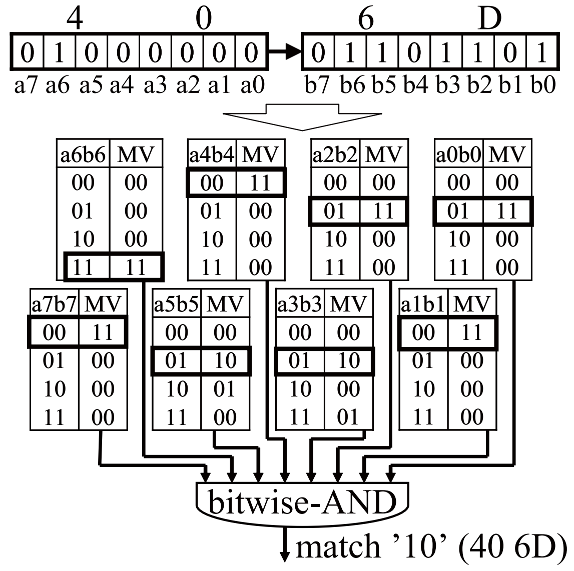


Figure 5. 8_FIMM Matches {40 6D}.

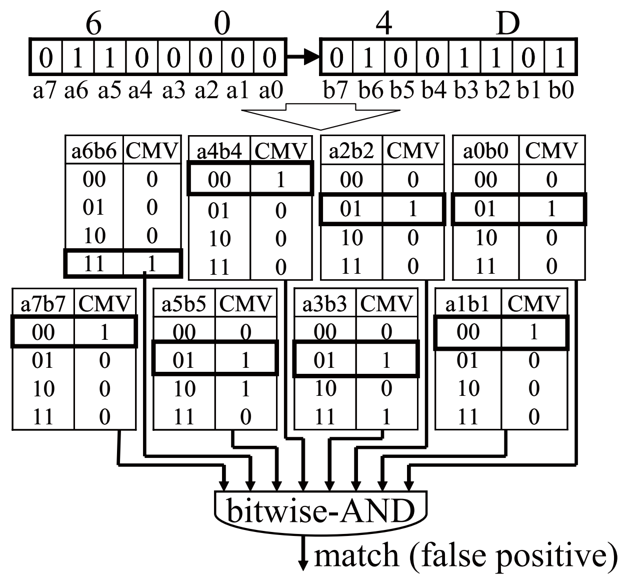


Figure 6. An Example of the False Positive.

MV to the **CMV (Compressed Match Vector)**. To obtain the CMV, the MV is partition into groups of m bits, and the OR operation is applied to the m bits in each group. In other word, if a group contains '1' anywhere, we compress the group to '1', otherwise to '0'. m is called the **compression ratio**. Let M_{CMV} be the total amount of memory for an 8_FIMM producing the CMV. From Expr.(3), we have

$$M_{CMV} \quad = \quad 2^c \frac{k}{m} \times 8. \qquad (4)$$

Since the CMV is a lossy compression, non-stored pattern can be matched (**false positive**). However, this compression never miss the total matches. Next example illustrates the false positive.

Example 4.2 *Fig. 6 compress the MV in Fig. 5 to the CMV, where the compression ratio is $m = 2$. In Fig. 6, it shows matched patterns are {40 6D, 68 40} and {60 4D}, but the last one is not a total match.* *(End of Example)*

5 Experimental Results

5.1 Determination of Parameters from Simulation

Increasing the compression ratio m increases the false positive, so partial matches in the 8_FIMM also increases. As a result, the number of interrupts for the MPU increases. When the interrupt occurs during the matching operation in the MPU, the system suspends the 8_FIMM. We obtained the maximum ratio m that does not suspend 8_FIMM, experimentally. Let $D(m)$ (characters) be the average interval of matches in the 8_FIMM, T_{8FIMM} be the matching time for one character on the 8_FIMM, $\bar{T}_{MPU}$ be the average matching time on the MPU, and n_{8FIMM} be the number of 8_FIMMs. To avoid the suspension in 8_FIMMs, the following relation must be true:

$$\frac{1}{n_{8FIMM}} T_{8FIMM} D(m) \quad \gg \quad \bar{T}_{MPU}. \qquad (5)$$

To obtain the maximum m that does not violate the above relation, we implemented a cycle-accurate virus scanning engine in C-language. The assumption for the target device is Altera FPGA StratixIII EP3SL340H1152C3NE5 (containing 1,040 M9ks) at 200 MHz. We obtained $\bar{T}_{MPU} = 1,384$ nsec by performing the total match on the NiosII/f (embedded MPU) at 100 MHz[3]. When 8_FIMM operates at 200 MHz, we have $T_{8FIMM} = 5$ nsec/character. Therefore, when $n_{8FIMM} = 128$, from Expr.(5), we have $D(m) \gg 17,612.8$ characters. Let the number of patterns k be 512, the pattern length c be eight[4], and the number of 8_FIMMs be 128. Thus, 128 8_FIMM can store 65,536 ClamAV virus sub-patterns. To estimate $D(m)$, our virus scanning engine scanned selected 100 executable codes in Cygwin distribution [7] for our simulation. From the simulation results, for m =8, 16, 32, 64, and 128, the maximum ratio m that satisfies the condition $D(m) \gg 17,612.8$ is 16. Thus, the number of bits for the CMV is $512/16 = 32$. Note that, to fit the embedded memory of the Altera FPGA, m is selected to a power of 2. From Expr.(4), when $c = 8$, $k = 512$, and $m = 16$, the total amount of memory for the 8_FIMM is $2^8 \times 32 \times 8 = 8,192 \times 8$ bits. The 8_FIMM of this size efficiently fits the embedded memory of the Altera FPGA (9 Kbits). Thus, in our implementation, we can store 512 patterns in the 8_FIMM. For 128 8_FIMMs, the necessary number of embedded memories is 1,024 that can be implemented to our target device. Also, we implement a single MPU with the off-chip memory.

5.2 Implementation of the Virus Scanning Engine

We implemented the virus scanning engine using the PFIMM1024 on the Altera FPGA StratixIII EP3SL340H1152C3NE5 (containing 270,400 ALUTs, and 1,040 M9ks). To perform the total match, we used the embedded processor NiosII/f. We also stored the executable code into the 1 giga bytes DDR2-SDRAM. For the synthesis tool, we used QuartusII (v.8.0). The PFIMM1024 operates at 199.40 MHz, and consumes 75,826 ALUTs that is 28% of the total ALUTs. Also, the NiosII/f operates at 100.00 MHz, and consumes 1,478 ALUTs. Our virus scanning engine scans one character in every clock. Thus the throughput Th is $0.1994 \times 8 = 1.595$ Gbps. The 8_FIMM can store up to 512 patterns with at most 8 characters. Since the PFIMM1024 consists of 128 units of 8_FIMMs, it can store up to 65,536 patterns. The amount of memory for

[3]The 8_FIMM produces a unique index for each sub-pattern, while the bloom filter only shows the match or mismatch. Thus, the work for total match in our method is simpler and easier.

[4]When the pattern length is less than eight, the pattern is expanded into a set of 8-character patterns in the 8_FIMM. Fortunately, the number of these short patterns is less than 100. So, even if we can expanded them, the increased hardware is small.

Table 4. Comparison with Existing Methods.

Method	Th Gbps	# of Patterns	MUC Bytes/char	Th/MUC	Implementation
Aho-Corasick (AC Method)[17]	6.0	1,533	2896.2	0.0020	ASIC
Aldwari et al.[2]	14.0	1,542	126.0	0.1111	FPGA+SRAM
Bitmap compressed Aho-Corasick[17]	8.0	1,533	154.0	0.0519	ASIC
Path compressed Aho-Corasick[17]	8.0	1,533	60.0	0.1333	ASIC
Alicherry et al.[3]	20.0	100	48.0	0.4166	FPGA+TCAM
Yu et al.[18]	2.0	1,768	3.0	0.6666	FPGA+TCAM+MPU
USC RegExpController[5]	1.4	1,316	46.0	0.0304	FPGA+MPU
Hardware Bloom Filter[4]	0.5	35,475	1.5	0.3333	FPGA+SDRAM
Proposed method	1.6	65,536	2.0	0.8000	FPGA+MPU+SDRAM

the 8_FIMM is 8 KBytes(1024 Bytes$\times$8). Let the **memory utilization coefficient (MUC)** be the necessary amount of memory per a character. Then, MUC for the PFIMM1024 is $\frac{8{,}192\times128}{65{,}536\times8} = 2.000$ Bytes/Char.

5.3 Comparison with Existing Methods

Table 4 compares existing methods for the regular expression matching. They use different methods on technologies: FPGAs and ASICs. To make a fair comparison, we use the **normalized throughput**[5]Th/MUC, where $Th/MUC = \frac{Th}{MUC}$. In Table 4, Th denotes the throughput for a pattern matching engine (Gbps); *# of patterns* denotes the number of patterns; MUC denotes the memory utilization coefficient (Bytes/Char); and Th/MUC denotes the normalized throughput.

Table 4 shows that only our method can store 65,536 patterns in a single FPGA. Also, as for Th/MUC, our method is about 400 times better than AC method [17], and is 1.2-26.3 times better than other methods. The reason for our efficiency is that MUC for our method is quite small, since we partition the FIMM into 8_FIMM and compress the MV to the CMV. As for Th/MUC, Yu et al.[18] is the second to our method. However, they use the TCAM which is quite expensive and dissipates much power. If we consider the cost of TCAM, our method is much more efficient than Yu et al.[18].

6 Conclusion and Comments

This paper showed a virus scanning engine using the PFIMM1024 and general purpose MPUs. To perform efficient prefiltering, we used the parallel finite-input memory machine (PFIMM). To further reduce the memory size, we used CMV. We successfully stored 65,536 ClamAV virus patterns on the PFIMM1024. For the area-performance ratio, our system is 1.2-26.3 times more efficient than existing ones.

Our virus scanning engine has a vulnerability. When the attacker sends a sequence of sub-patterns stored in our engine (performance attack), it generates an IRQs for every clock and overflows the MPU. Kumar et al. [10] has proposed a method to protect against performance attack. It attaches a flow counter to the FIFO in Fig. 4. When the value of the counter exceeds some threshold, the circuit detects the performance attack. Our virus scanning engine can adopt Kumer's method.

[5]A similar measure is used in [14].

7 Acknowledgments

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, and the grant of Innovative Cluster Project of MEXT (the second stage). Discussions with Prof. Jon T. Butler and Mr. Hisashi Kajiwara were quite useful.

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, 18(6):333-340, 1975.

[2] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *SIGRACH. Compt. Archit. News*, vol. 33, no. 1, pp.99-107, 2005.

[3] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," *IEEE Int. Conf. on Network Protocols (ICNP'06)*, pp.187-196, 2006.

[4] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of bloom filters for string matching," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pp.322-323, 2004.

[5] Z. K. Baker, H. Jung, and V. K. Prasanna, "Regular expression software deceleration for intrusion detection systems," *16-th Int. Conf. on Field Programmable Logic and Applications (FPL'06)*, pp. 28-30, 2006.

[6] Clam AntiVirus, http://www.clamav.net/

[7] Cygwin distribution, http://cygwin.com/

[8] Kaspersky, http://www.kaspersky.com/

[9] Z. Kohavi, *Switching and Finite Automata Theory, McGraw-Hill Inc.*, 1979.

[10] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," *3rd ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS'07)*, pp. 155-164, 2007.

[11] PCRE: Perl Compatible Regular Expressions, http://www.pcre.org/

[12] H. C. Roan, W. J. Hawang, and C. T. Dan Lo., "Shift-or circuit for efficient network intrusion detection pattern matching," *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL'06)*, pp.785-790, 2006.

[13] SNORT, http://www.snort.org/.

[14] I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis, "Scalable Multigigabit Pattern Matching for Packet Inspection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Vol. 16, Issue 2, pp.156-166, Feb. 2008.

[15] L. Tan, and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," *Proceedings of the 32nd Int. Symp. on Computer Architecture (ISCA'05)*, pp.112-122, 2005.

[16] TrendMicro, http://us.trendmicro.com/.

[17] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *23-th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, pp.333-340, 2004.

[18] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern matching using TCAM," *IEEE Int. Conf. on Network Protocols (ICNP'04)*, pp.174-183, 2004.