PAPER Special Section on Reconfigurable Systems

A Virus Scanning Engine Using an MPU and an IGU Based on Row-Shift Decomposition

Hiroki NAKAHARA[†a)], Tsutomu SASAO[††b)], *and* Munehiro MATSUURA[††c)], *Members*

SUMMARY This paper shows a virus scanning engine using two-stage matching. In the first stage, a binary CAM emulator quickly detects a part of the virus pattern, while in the second stage, the MPU detects the full length of the virus pattern. The binary CAM emulator is realized by an index generation unit (IGU) based on row-shift decomposition. The proposed system uses two off-chip SRAMs and a small FPGA. Thus, the cost and the power consumption are lower than the TCAM-based system. The system loaded 1,290,617 ClamAV virus patterns. As for the area and throughput, this system outperforms existing two-stage matching systems using FPGAs.

key words: *pattern matching, virus scanning, index generation function, CAM*

1. Introduction

1.1 Virus Scanning System

A **computer virus*** intends to damage computer systems. The growth of the Internet requires a high-speed virus scanning on an e-mail and a file servers. The throughput of the software-based virus scanning is at most tens of mega bits per second (Mbps) [1], which is too low. Thus, a hardware-based virus scanning is necessary. We consider a low-cost and high-performance virus scanning system shown in Fig. 1 for low-end users such as SOHO (small office and home office) and enterprise with the following features:

High throughput: The throughput is higher than one Gbps and is higher than servers (hundreds Mbps).

Low power and low cost: It uses a low-end (*i.e.*, a small) FPGA and SRAMs instead of a high-end FPGA and a ternary content addressable memory (TCAM). Table 1 shows that the TCAM dissipates much higher power than the SRAM. Although we can implement the CAM function on the FPGA [2], [3], for the virus scanning, it requires excessive amount of resources of the FPGA.

Reconfigurable: It uses a memory-based realization rather than the random logic realization. Although the random logic realization on the FPGA is fast and compact, the required time for place-and-route is longer than the periods for the virus pattern update. Some virus scanning software, *e.g.*, Kaspersky [4], updates the virus data every hour.

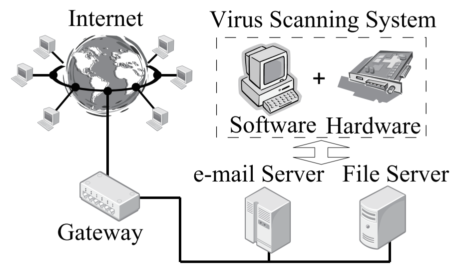

Fig. 1 Virus scanning system for an e-mail server and a file server.

Table 1 Comparison of TCAM with SRAM (18 Mbit chip) [5].

	TCAM	SRAM
Max. Freq. [MHz]	266	400
Power Dissipation [W]	12-15	≈ 0.1
# of transistors per a bit	16	6

1.2 Related Works

Our virus scanning engine uses two-stage matching [6]. The first stage scans sub pattern by the hardware, while the second stage exactly scans full length pattern by the software. Various two-stage matching implementations have been reported: A TCAM with a general purpose processor (MPU) [7]; a bit-partitioned Aho-Corasic DFA [8] with a special purpose MPU [9]; a method using cuckoo hashing [10]; bit-partitioned finite-input memory machines (FIMMs) with an MPU [11]; a method using index generation units (IGUs) of different sizes and an MPU [12]; Bloom filter (PERG-Rx) [13]; and a method using four IGUs of the same sizes and an MPU [14]. Many methods [8]–[12], [14] use memory-based approach rather than power-hungry TCAM-based ones. Since, previous methods [8]–[12] used many more on-chip memories, it was a bottleneck for the virus scanning engine. This paper proposes the two-stage matching engine that uses the smallest on-chip memory.

1.3 Contributions of the Paper

Implementation of more than one million ClamAV virus patterns: ClamAV [15] is an open source (GPL) antivirus

Manuscript received November 9, 2012.
Manuscript revised March 7, 2013.

†The author is with the Faculty of Engineering, Kagoshima University, Kagoshima-shi, 899–0065 Japan.
††The authors are with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.
a) E-mail: nakahara@eee.kagoshima-u.ac.jp
b) E-mail: sasao@ieee.org
c) E-mail: matsuura@cse.kyutech.ac.jp

DOI: 10.1587/transinf.E96.D.1667

*It is also called **a malware** (a composite word from malicious software). In this paper, a virus means a computer virus.

engine, which scans a mail server (Postfix) using pattern matching. This paper presents an index generation unit (IGU) based on row-shift decomposition [16], [17] to realize a scanning engine for the ClamAV virus pattern. Previous implementation [14] used four off-chip SRAMs by a linear transformation, while the method in this paper uses only two SRAMs by a row-shift decomposition. Thus, the cost is lower than the previous one.

High-level optimization of the system throughput by the hardware and the software: We implement two-stage matching by the hardware and the software. We maximized the system throughput by finding the optimal size of the hardware experimentally.

Comparison of various two-stage matching methods: We compare our method with various two-stage matching implementations with respect to throughput and area efficiency.

The rest of the paper is organized as follows: Sect. 2 introduces the virus scanning based on two-stage matching; Sect. 3 describes the binary CAM emulator using the IGU; Sect. 4 shows the design method for the IGU based on row-shift decomposition; Sect. 5 shows the implementation results of the virus scanning engine; and Sect. 6 concludes the paper.

This paper is based on previous publications [11], [12], [14].

2. A Virus Scanning Based on Two-Stage Matching

2.1 Definitions

A **virus scanning** detects the virus on a **text** (executable codes or e-mails). A **pattern** is represented by a **regular expression** consisting of **characters** and **meta-characters**. A **pattern matching** is to detect (variable-length) patterns in the text. Table 2 shows the meta-characters used in ClamAV. Note that, ClamAV represents a character by two hexadecimal characters. For example, "AB" denotes "11001101" in binary. A **length** is the number of characters. A **subpattern** is a part of the pattern consisting characters only†. In this paper, k denotes the number of patterns in the pattern set, r denotes the length of a pattern, and m ($m \leq r$) denotes the length of a subpattern. Note that, r and m vary by patterns.

2.2 ClamAV Virus Pattern

As of December 1st, 2010, ClamAV (version 0.96.5) contains 1,290,808 patterns [15]. Table 3 shows the pattern types, the number of patterns, and their detection methods. An **MD5 checksum pattern** is a hash value (128 bits) of the virus, which is detected by the hardware. A **basic pattern** is a **regular expression** representing a part of the virus, which is detected by the hardware. A **Google safe browsing database pattern** is the MD5 hash value of the abnormal address obtained from the Google safe browsing API [18], which is detected by the hardware. A **combination pattern** is a combination of basic patterns. It is represented by the logical operations such as "AND", "OR", and "NOT" of basic patterns, and detected by an MPU. A **compressed file analysis pattern** includes a file size, a file name, or header characteristics. Since the ClamAV committee announces that this pattern will be not supported, we do not detect this.

Figure 2 shows the virus scanning system. Since the detection time for the Google safe browsing API and the basic pattern combination are significantly short, they are realized by software. The MD5 checksum generator is implemented by the commercial IP core [19]. Therefore, in this paper, $k = 1,290,617$ *patterns including the MD5 checksum pattern, the basic pattern, and the Google safe browsing database pattern* are detected by a **virus scanning engine** on the hardware (a small FPGA and SRAMs).

Example 2.1: Table 4 shows examples of ClamAV patterns. For "W32.Gop", "736D74702E79656168" and "2D20474554204F49" are subpatterns. ■

Table 2 Meta-characters used in ClamAV.

<table>
<tr><th>Meta-Char</th><th>Meaning</th><th>Example</th></tr>
<tr><td>??</td><td>An arbitrary character</td><td></td></tr>
<tr><td>*</td><td>Repetition of more than zero "??"</td><td>AA*BB={AABB,AA??BB,
AA????BB,AA??????BB,···}</td></tr>
<tr><td>(AA|BB)</td><td>Alternation of "AA" and "BB"</td><td>(AA|BB)={AA,BB}</td></tr>
<tr><td>{n-m}</td><td>Repetition of n or more than n "??" and m or less than m "??"</td><td>AA{1-2}BB={AA??BB,
AA????BB}</td></tr>
</table>

Table 3 Virus patterns in ClamAV (version 0.96.5, December, 1st, 2010) and our implementation.

Pattern type	#Patterns	Implementation	Realized
MD5 checksum	761,527	Hardware	Yes
Basic pattern	94,227	Hardware	Yes
Google safe browsing database	434,863	Hardware	Yes
Combination pattern	85	Software	No
Compression file analysis	106	Software	No
Total	1,290,808		

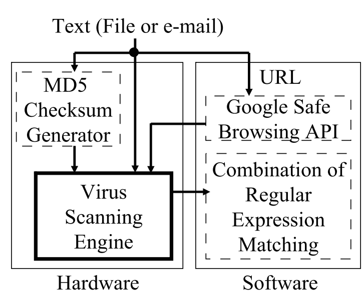

Fig. 2 Virus scanning system.

Table 4 Examples of ClamAV patterns.

<table>
<tr><th>Virus Name</th><th>Pattern</th></tr>
<tr><td>Trojan.DelY-3</td><td>64656C74726565{-1}2F(59|79)20633A5C2A2E2A</td></tr>
<tr><td>Trojan.MkDir.B</td><td>406D64202572616E646F6D25????676F746F2048</td></tr>
<tr><td>W32.Gop</td><td>736D74702E79656168*2D20474554204F49</td></tr>
<tr><td>Worm.Bagle-67</td><td>6840484048688D5B0090EB01EbEB0A5BA9ED46</td></tr>
</table>

†However, a meta-character "??" is permitted.

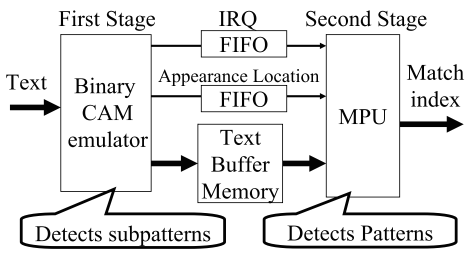

Fig. 3 Virus scanning engine using two-stage matching.

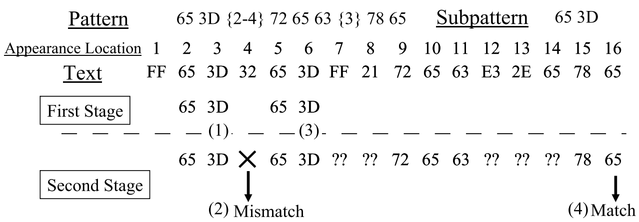

Fig. 4 Example of two-stage matching.

2.3 Virus Scanning Engine Using Two-Stage Matching

A regular expression for a ClamAV pattern consists of subpatterns and meta-characters representing the distance. To detect patterns, we use **two-stage matching** shown in Fig. 3. Since no subpattern contains meta-characters, in the first stage, we use **a binary CAM emulator** to detect subpatterns. When a subpattern is detected, **the IRQ (interrupt request) signal** and **the appearance location** are sent to the MPU. For the pattern that contains meta-characters, in the second stage, the embedded MPU performs PCRE (Perl compatible regular expression) [20] matching for the full length of the pattern. To detect other subpatterns during the MPU operation, FIFOs are attached between the first stage and the second stage to store IRQ signals and appearance locations. Also, **a text buffer memory** is attached to store inputs.

Example 2.2: Fig. 4 shows an example of two-stage matching. First, at the appearance location "3", the first stage finds the subpattern "653D" (Fig. 4 (1)). After this, the second stage finds mismatch (Fig. 4 (2)). Next, at the appearance location "6", the first stage finds the subpattern "653D" (Fig. 4 (3)). Finally, the second stage detects the pattern (Fig. 4 (4)). ■

2.4 Subpattern Length m

For ClamAV, since most patterns are MD5 checksums or MD5 hash values consisting 16 characters (128 bits)[†], we assume that $m \leq 16$. The binary CAM emulator stores k subpatterns with length m from k patterns. In virus patterns, since a character consists of eight bits, the total number of bits to represent a pattern with length m is $2^8 m = 256m$. Then, the **subpattern detection probability** $P(m)$ is $\frac{k}{m2^8}$[††].

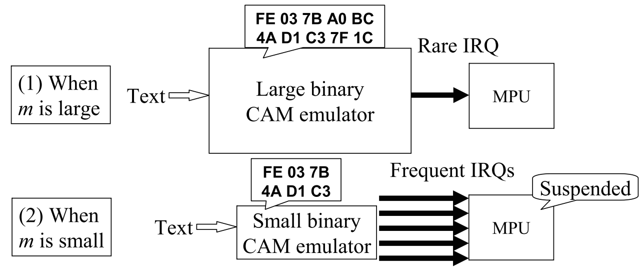

Fig. 5 Relation between the subpattern length m and the number of IRQs.

When m is large, since $P(m)$ is small, the IRQ signal rarely occurs[†††]. However, in this case, the size of the binary CAM emulator becomes large (Fig. 5 (1)). On the other hand, when m is small, the binary CAM emulator becomes small. Since $P(m)$ is large, the IRQ signal frequently occurs (Fig. 5 (2)). In this case, the binary CAM emulator is suspended until the MPU finishes the operation, thus the system throughput decreases. Thus, to minimize the size of the binary CAM emulator without sacrificing the performance, we find the minimum m that does not suspend the MPU.

Problem 2.1: Let k be the number of subpatterns, m be the length of the subpatterns, T_{MPU} be the processing time for the regular expression matching by the MPU, $P(m) = \frac{k}{m2^8}$ be the subpattern detection probability, and T_{bCAMe} be the operation time of the binary CAM emulator to shift a character. Obtain the minimum m that satisfies the condition:

$$\frac{T_{bCAMe}}{P(m)} \gg T_{MPU}. \tag{1}$$

$\frac{1}{P(m)}$ denotes the average distance of appearance locations of virus, and $\frac{T_{bCAMe}}{P(m)}$ denotes the **average IRQ period**. Here, we assume that subpatterns are uniformly distributed. The optimum value of m is obtained experimentally in Sect. 5.1.

3. Binary CAM Emulator Using an Index Generation Unit

3.1 Index Generation Function [21]

Definition 3.1: A mapping $F(\vec{X}) : B^n \to \{0, 1, \ldots, k\}$, is an **index generation function**, where $F(\vec{a}_i) = i$ $(i = 1, 2, \ldots, k)$ for k different **registered vectors**[††††], and $F = 0$ for other $(2^n - k)$ non-registered vectors, and $\vec{a}_i$ $(i = 1, 2, \ldots, k)$ are different vectors in B^n. In other words, an index generation

[†] For the basic patterns consisting of more than 16 characters, only the first 16 characters are checked in the first stage.

[††] When the distribution of the characters in the subpatterns is uniform.

[†††] For a subpattern shared by multiple patterns, the second stage using the PCRE library detects the multiple patterns.

[††††] In this paper, k also denotes the number of patterns in ClamAV.

Table 5 Example of an index generation function.

x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

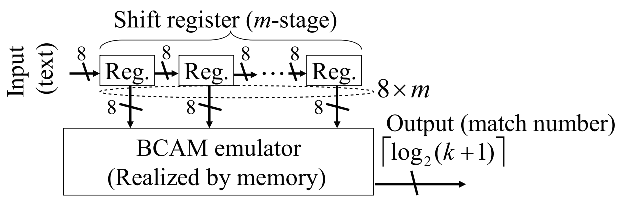

Fig. 6 Finite Input Memory Machine (FIMM).

function produces unique indices ranging from 1 to k for k different registered vectors, and produces 0 for other vectors.

Example 3.3: Table 5 shows an example of an index generation function, where $n = 6$ and $k = 7$. ■

In a virus scanning, a registered vector corresponds to a subpattern of a virus pattern, while an index corresponds to the unique number for each subpattern.

3.2 Finite Input Memory Machine to Detect Subpatterns

Figure 6 shows **a finite input memory machine (FIMM)** [22] that accepts k subpatterns with length m. In Fig. 6, *Reg* denotes an 8-bit parallel-in parallel-out shift register. The m-stage shift register stores the past m inputs, and the memory produces the match number. Let M_{FIMM} be the size of the memory† of the FIMM, then, we have $M_{FIMM} = 2^{8m}\lceil log_2(k+1)\rceil$. Thus, a straightforward implementation of the memory is impractical for a large m.

3.3 Index Generation Unit (IGU) [17], [23]

In this paper, to realize the FIMM compactly, we use **an index generation unit (IGU)** [23].

Table 6 is a **decomposition chart** for the index generation function f shown in Example 3.3. The columns labeled by $X_1 = (x_2, x_3, x_4, x_5)$ denotes the **bound variables**, and rows labeled by $X_2 = (x_1, x_6)$ denotes the **free variables**. The corresponding chart entry denotes the function value. We can represent the non-zero elements of f by the **main memory** $\hat{f}$ whose input is X_1. Table 7 shows the function $\hat{f}(X_1)$ of the main memory. The main memory realizes a mapping from a set of 2^p elements to a set of $k + 1$ elements, where $p = |X_1|$. The output for the main memory does not always represent f, since $\hat{f}$ ignores X_2. Thus, we must check whether $\hat{f}$ is equal to f or not by using an **auxiliary (AUX) memory**. To do this, we compare the input X_2 with the output for the AUX memory by a **comparator**.

Table 6 Decomposition Chart for $f(X_1, X_2)$, where $X_1 = (x_2, x_3, x_4, x_5)$ and $X_2 = (x_1, x_6)$.

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_5
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_4 X_1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_2
00	0	0	0	0	0	0	0	0	1	2	3	0	0	0	4	0	
01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
10	5	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0	
11	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0	0	
x_6, x_1 X_2																	

Table 7 Decomposition Chart for $\hat{f}(X_1)$, where $X_1 = (x_2, x_3, x_4, x_5)$.

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_5
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_4
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_2
5	0	0	0	0	0	0	0	1	2	3	6	0	7	4	0	

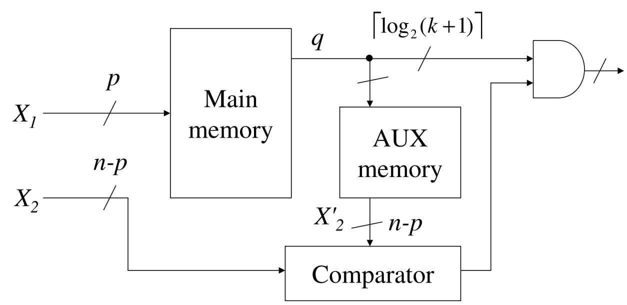

Fig. 7 Index Generation Unit (IGU).

The AUX memory stores the values of X_2 when the output of $\hat{f}(X_1, X_2)$ is non-zero. Figure 7 shows the index generation unit (IGU). First, the main memory finds the possible index corresponding to X_1. Second, the AUX memory produces the corresponding inputs X_2' ($n - p$ bits). Third, the comparator checks whether X_2' is equal to X_2 or not. Finally, the AND gates produce the correct value f. We implement the main memory and the AUX memory by a single memory device with $|X_1|$ ($= p$ *bits*) inputs and $q + |X_2'|$ ($= q + n - p$) outputs.

Example 3.4: Figure 8 shows an example of the IGU realizing the index generation function shown in Table 6. When the input vector is $X(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 1, 1, 0, 1, 1)$, the corresponding index is "6". First, the main memory produces the index. Second, the AUX memory produces the corresponding X_2'. Third, the comparator checks X_2 and X_2'. Since the corresponding input X_2 is correct, the AND gates produces the index. In this case, $n = 6$, $p = 4$, and $q = 3$. ■

3.4 Row-Shift Decomposition [16]

The decomposition chart shown in Table 6 is an ideal case,

†Since the amount of memory of the state variables for the shift register is much smaller than that for the output functions, when we calculate the memory size, we neglect it.

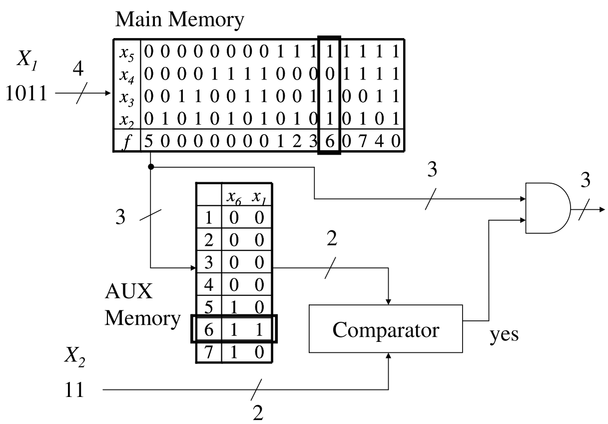

Fig. 8 IGU for Table 6.

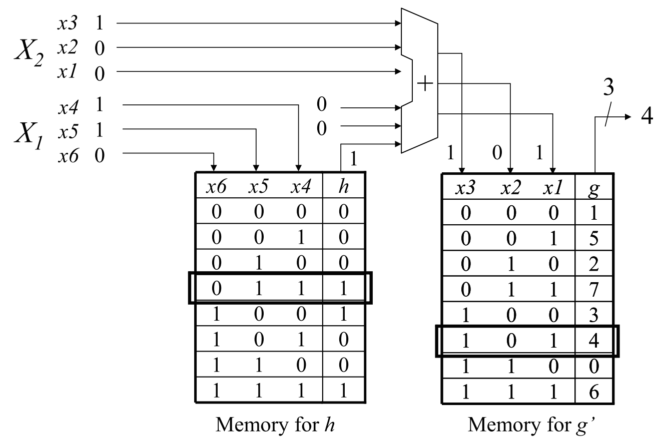

Fig. 9 An example of row-shift decomposition.

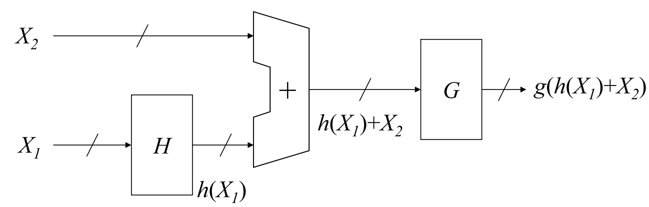

Fig. 10 Row-shift decomposition.

Table 8 Decomposition chart for g.

	0	0	0	0	1	1	1	1	x_3	
	0	0	1	1	0	0	1	1	x_2	X_2
	0	1	0	1	0	1	0	1	x_1	
000	0	0	0	0	0	0	0	0		
001	0	0	0	0	0	0	0	0		
010	1	0	2	0	3	0	0	0		
011	0	0	0	0	4	0	0	0		
100	5	0	0	0	0	0	0	0		
101	0	0	0	0	0	0	0	0		
110	0	0	0	0	0	0	0	6		
111	0	0	7	0	0	0	0	0		
x_6, x_5, x_4 X_1										

Table 9 Decomposition chart for g'.

	0	0	0	0	1	1	1	1	x_3	
	0	0	1	1	0	0	1	1	x_2	X_2
	0	1	0	1	0	1	0	1	x_1	
000	0	0	0	0	0	0	0	0		
001	0	0	0	0	0	0	0	0		
010	1	0	2	0	3	0	0	0		
011	0	0	0	0	→	4	0	0		
100	→	5	0	0	0	0	0	0		
101	0	0	0	0	0	0	0	0		
110	0	0	0	0	0	0	0	6		
111	0	0	→	7	0	0	0	0		
x_6, x_5, x_4 X_1										

since each column has at most one non-zero element. When a column of a decomposition chart has two or more non-zero elements, it has a **collision**. Table 8 shows a decomposition chart for the index generation function g, where $X_2 = (x_3, x_2, x_1)$ and $X_1 = (x_6, x_5, x_4)$. The number of collisions is three in Table 8. Consider the decomposition chart for g' shown in Table 9 that is obtained from Table 8 by shifting one bit to the right in the rows for $X_1 = (x_6, x_5, x_4) = (0, 1, 1)$,$(1, 0, 0)$, and $(1, 1, 1)$. Table 9 has at most one non-zero element in each column. Thus, the modified function g' can be realized by the main memory with inputs X_1. Let X_1 be the row variables, and X_2 be the column variables. In Fig. 10, assume that the memory for H stores the number of bits to shift ($h(X_1)$**: shift value**) for each row specified by X_1, while the memory for G stores the non-zero element of the column after the shift operation: $h(X_1) + X_2$, where "+" denotes an unsigned integer addition. We call this **row-shift decomposition**. This method requires fewer memories than previous methods [11], [12], [14].

Example 3.5: Figure 9 shows an example of row-shift decomposition realizing non-zero elements for an index generation function shown in Table 5. Note that, $X_1 = (x_6, x_5, x_4)$ and $X_1 = (x_3, x_2, x_1)$. When the input vector $X(x_1, x_2, x_3, x_4, x_5, x_6)$ is $(0, 0, 1, 1, 1, 0)$, the corresponding index is "4". First, the memory for h produces the shift value "1" as $h(X_1)$. Then, the adder produces $(1, 0, 1)$ as $h(X_1) + X_2$. Finally, the memory for g' produces the index "4". ■

In Example 3.5, the row-shift decomposition represents g', while the target function is g. To realize g using g', we use an AUX memory, a comparator, and an AND gates.

Example 3.6: Figure 11 shows the IGU based on row-shift decomposition realizing the index generation function shown in Table 5. To reduce the number of memories, we realize both the memory for g' and the AUX memory by a single memory. From Example 9, when the input vector is $X = (0, 0, 1, 1, 1, 0)$, the memory for g' produces the index and the corresponding X_1 simultaneously. Next, the comparator checks X_1. Finally, the AND gates produces the correcting index. ■

Example 3.7: To realize the index generation function g shown in Table 8, a single-memory realization requires

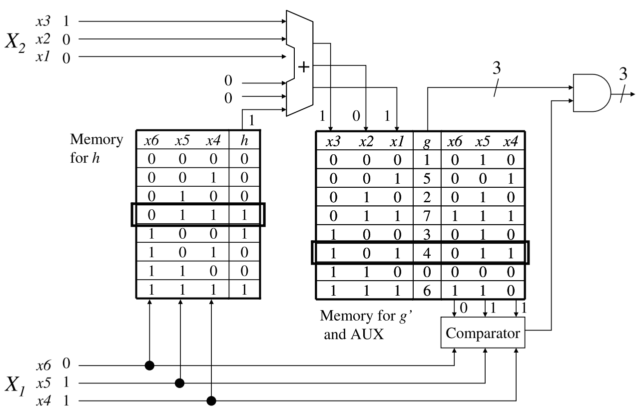

Fig. 11 IGU based on row-shift decomposition.

$2^6 \times 3 = 192$ bits. On the other hand, in the IGU based on row-shift decomposition shown in Fig. 11, since **the maximum value of the shift** is one, the first memory requires only $2^3 \times 1 = 8$ bits. And, the second memory requires $2^3 \times 6 = 48$ bits. Thus, the IGU requires 56 bits in total. In this way, we can reduce the total amount of memory by using the row-shift decomposition. ■

Compared with the single-memory realization, the row-shift decomposition requires an adder, the comparator, and the AND gates in addition to the memory. However, these are negligible for the modern FPGA. Thus, the IGU based on row-shift decomposition is suitable for the low-cost implementation. In the experimental results, we will demonstrate this.

3.5 Capability of the IGU based on row-shift decomposition

Example 3.6 shows that the row-shift decomposition reduces the amount of memory. However, in an extreme case, the row-shift decomposition cannot represent a function with n inputs for 2^n different elements. We refer following Conjecture for a limitation of the IGU based on row-shift decomposition.

Conjecture 3.1: [23] When the number of the input variables is sufficiently large, more than 95% of incompletely specified index generation functions with weight k ($k \leq 7$), can be represented with $n = 2\lceil log_2(k+1)\rceil - 3$ variables.

When $k << 2^n$, the IGU can represent most functions. Fortunately, in the virus scanning problem, k is about 2^{20}, and $n = 40$.

4. Design of IGU Using Row-Shift Decomposition [16]

In Example 3.6, we could represent the function without increasing the columns. However, in general, we must increase the columns to represent the function. Since each column has at most one non-zero element after the row-shift operations, at least k columns are necessary to represent a function with weight k.

We assume that the virus scanning system updates its virus pattern every one hours. In this case, it is impractical to find an optimal solution by spending much computation time. We use the **first-fit method** [24], which is simple and efficient.

Algorithm 4.1: (Find row-shifts)

1. Sort the rows in decreasing order by the number of non-zero elements.
2. Compute the row-shift value for each row at a time, where $h(X_1)$ for row X_1 denotes the smallest value such that no non-zero element in row X_1 is in the same position as any non-zero element in the previous rows.
3. Terminate.

When the distribution of non-zero elements among the rows is uniform, Algorithm 4.1 reduces the memory size effectively. To reduce the total amount of memories, we use the following:

Algorithm 4.2: (Row-shift decomposition [16])

1. Reduce the number of variables by the method [23], which eliminates the redundant variables. If necessary, use a linear transformation [17], which reduces variables by applying EXOR operations to X_1 and X_2, where $X = (X_1, X_2)$ denotes the partition of the inputs. Let n be the number of variables after reduction.
2. From $t = -2$ to $t = 2$, perform Steps 2.1 through 2.4.
   - 2.1. $p \leftarrow \lceil \frac{n}{2} \rceil + t$.
   - 2.2. Partition the inputs X into $(X_1, X_2)^{\dagger}$, where $X_1 = (x_p, x_{p-1}, \ldots, x_1)$ denotes the rows, and $X_2 = (x_n, x_{n-1}, \ldots, x_{p+1})$ denotes the columns.
   - 2.3. Obtain the row-shift value by Algorithm 4.1.
   - 2.4. Obtain the maximum of the shift value, and compute the total amount of memories.
3. Find t that minimizes the total amount of memories.
4. Terminate.

5. Experimental Results

5.1 Optimum Subpattern Length m

We obtained the minimum m that satisfies the relation (1). To do this, first, we implemented a cycle-accurate simulator for the IGU based on row-shift decomposition in C-language. Then, we scanned 2,963 cygwin executable codes. We assume that the IGU reads the data from the SRAM running at 400 MHz. From this, we have $T_{bCAMe} = \frac{1}{400} \times 10^6 \mu$ sec. We obtained the average operation time of the MPU (T_{MPU}) and the maximum T_{MPU} by matching

†In the row-shift decomposition, we assume that non-zero elements are uniformly distributed in the decomposition chart. In the virus pattern, distribution of non-zero elements is uniform. Thus, unlike ordinary functional decompositions, the influence of the partition (X_1, X_2) is relatively small.

Table 10 Comparison with other methods.

	FPGA (Op.Freq.)	Synthesis Tool (version)	#Pattern (#Char)	#LC	On-chip Mem [Bytes]	Th [Gbps]	#LC/ #Char	On-chip Mem/ #Char	Off-chip Memories	Off-chip Mem/ #Char
USC RegExp Controller (2006) [9]	XC4VFX100 (412 MHz)	ISE (7.1.2)	1,316 (16,715)	41,787	768,819.2	1.40	2.4999	45.9957	SDRAM	
Cuckoo Hashing (2007) [10]	XC4VLX25 (285 MHz)	ISE (8.1i)	4,748 (68,266)	2,982	142,848.0	2.20	0.0436	2.0925	SRAM	
Parallel FIMMs (2009) [11]	EP3SL340H (199 MHz)	Quartus II (8.0)	65,536 (524,288)	77,304	1,048,576.0	1.59	19.3150	2.0000	None	
Standard Parallel Sieve Method (2009) [12]	EP3SL340H (200 MHz)	Quartus II (8.0)	497,172 (3,977,376)	5,268	3,500,880.0	1.60	0.0013	0.8801	Three SRAMs (8 MB×3)	50.6179
PERG-Rx (2009) [13]	XC2VP100 (180 MHz)	ISE (9.1)	85,625 (8,645,488)	42,809	387,072.0	1.30	0.0049	0.0447	SRAM (4 MB)	3.8811
4IGU method (2012) [14]	XC5VLX50T (400 MHz)	ISE (11.1)	1,290,617 (42,461,299)	13,857	39,116.8	3.20	0.0003	0.0009	Four SRAMs (16 MB×4)	12.6437
IGU based on row-shift decomposition (proposed)	XC5VLX50T (400 MHz)	ISE (14.2)	1,290,617 (42,461,299)	9,147	39,116.8	3.20	0.0002	0.0009	Two SRAMs (20 MB + 256 KB)	3.9906

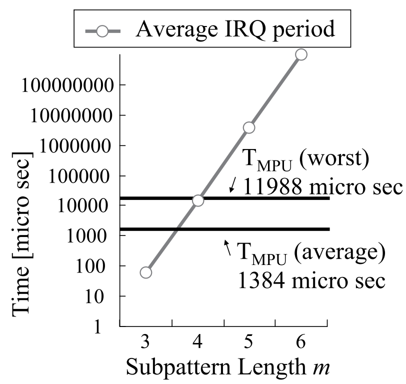

Fig. 12 Average and maximum operation times of MPU T_{MPU} and average IRQ period for different values of *m*.

2,963 cygwin executable codes on the MicroBlaze [25] running at 100 MHz using the Perl Compatible Regular Expression library (PCRE) [20]. We used the hardware IRQ handler and the software context switch in the MicroBlaze. Figure 12 shows the average T_{MPU}, the maximum T_{MPU}, and the average IRQ period $\frac{T_{bCAMe}}{P(m)}$ for different *m*. From this, we chose $m = 5$ (40 bits) for implementation to satisfy the condition (1) of Problem 1.

5.2 Realization of ClamAV Virus Subpatterns

We implemented Algorithm 4.2, and applied to ClamAV virus subpatterns. The number of subpatterns is 1,290,617. For the design, we used a PC with Intel's Core 2 Duo CPU running at 2.53 GHz and 4.0 GB RAM, on Windows XP Professional Operation System. Algorithm 4.2 produced the circuit having the architecture shown in Fig. 11. As for the memory *H*, the number of input bits was 18, and the number of output bits was five. As for the memory *G*, the number of input bits was 22, and the number of output bits was 39†. Thus, the memory for *H* can be implemented by a 256 KB SRAM (18 inputs and eight outputs), and that for *G* can be implemented by a 20 MB SRAM (22 inputs and 40 outputs). In this design, the linear decomposition [17] was not used.

5.3 Implementation Results

We implemented a proposed virus scanning engine shown in Fig. 3 consisting of the IGU and the MicroBlaze (MPU) on the Inrevium Corp. PCI Express Evaluation Board (FPGA: Xilinx Inc., Virtex5 VLX50T-GB-R). We used two SRAMs running at 400 MHz for the IGU, and used one 512 MBytes SO-DIMM module running at 266 MHz for the MicroBlaze. The synthesis tool is the Xilinx ISE Design Suite ver. 14.2. In the implementation, the IGU based on row-shift decomposition used 1,560 logic cells (LCs); the MicroBlaze used 1,263 LCs; the DDR2-SDRAM controller used 6,324 LCs and 10 BRAMs; and the text buffer memory used 10 BRAMs. In total, the virus scanning engine used 9,147 LCs and 20 BRAMs. The IGU operated at 508.2 MHz, while the MicroBlaze operated at 100 MHz. Since the clock frequency is set to 400 MHz and the IGU shifts 8 bits per one clock, the system throughput is $0.4 \times 8 = 3.2$ Gbps.

Table 10 compares various FPGA realizations. As for the throughput (*Th*), our system is 1.45-2.46 times higher than the previous ones except for [14]. As for the LC requirement per a character (*#LC/#Char*), our system is 4.3 times lower than that for the standard parallel sieve method [12]; as for the on-chip memory requirement per a character (*On-chip Mem/#Char*), our system is 49.6 times lower than that for the PERG-Rx. As for the off-chip SRAM requirement per a character (*Off-chip Mem/#Char*), our system is 3.1 times lower than that for the 4IGU method. As for the number of off-chip SRAMs, our system requires a half of the 4IGU method [14].

Compared with the 4IGU method, *#LC/#Char* for the proposed one is 1.5 times smaller than the 4IGU method. The reason is that the proposed one uses only a single IGU unit, while the 4IGU method uses four IGU units.

The limitation on the number of pins and the board layout often prevent us attaching many off-chip SRAMs to modern FPGA. Also, many off-chip SRAMs increase the

†The number of output bits for the main memory was 21, and that for the AUX memory was 18.

cost, decrease the reliability, and lose of the connection. Thus, the cost for our system is lower than that for the 4IGU method.

6. Conclusion and Comments

This paper showed a virus scanning engine using two-stage matching. In the first stage, the IGU detects the subpatterns, while in the second stage, the MicroBlaze MPU detects the full length of patterns using PCRE library. Our system using Xilinx FPGA and two SRAMs stored 1,290,617 ClamAV virus patterns, and has the throughput of 3.2 Gbps. Compared with previous systems, our virus scanning engine has lower cost and higher performance.

Our virus scanning engine has a vulnerability for the performance attack. When the attacker sends a sequence of stored subpatterns, the first stage generates an IRQ for every clock, and overflows the second stage. Kumar et al. [6] have proposed a method to protect against the performance attack. It attaches a flow counter to the FIFO in Fig. 3. When the value of the counter exceeds the threshold, the circuit detects the performance attack. Our virus scanning engine can incorporate the Kumar's method.

In our experiment, to find the optimum subpattern length *m*, we scanned cygwin executable codes. However, it is also possible to use other binary codes. One candidate is Windows executable codes, since many commercial virus scanners scan them. Also, we implemented the interface with the hardware IRQ and the software context switch. Since the hardware context switch can switch the context quickly, it may increase system throughput, however, this also increases the amount of hardware. Considering practical simulation setup is the one of the future works.

Acknowledgements

This research is supported in part by Strategic Information and Communications R&D Promotion Program (SCOPE), and the Grants in Aid for Scientistic Research of JSPS. Discussion with Prof. J.T. Butler was quite useful. Reviewer's comments were quite useful to improve the presentation.

References

[1] H.C. Roan, W.J. Hawang, and C.T. Dan Lo, "Shift-or circuit for efficient network intrusion detection pattern matching," FPL2006, pp.785–790, 2006.
[2] J. Ditmar, K. Torkelsson, and A. Jantsch, "A dynamically reconfigurable FPGA-based content addressable memory for internet protocol," FPL2000, pp.19–28, 2000.
[3] P.B. James-Roxby and D.J. Downs, "An efficient content-addressable memory implementation using dynamic routing," FCCM2001, pp.81–90, 2001.
[4] Kaspersky, http://www.kaspersky.com/
[5] W. Jiang, Q. Wang, and V.K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," INFOCOM2008, pp.1786–1794, 2008.
[6] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," ANCS2007, pp.155–164, 2007.
[7] F. Yu, R.H. Katz, and T.V. Lakshman, "Gigabit rate packet pattern matching using TCAM," ICNP2004, pp.174–183, 2004.
[8] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," ISCA2005, pp.112–122, 2005.
[9] Z.K. Baker, H. Jung, and V.K. Prasanna, "Regular expression software deceleration for intrusion detection systems," FPL2006, pp.28–30, 2006.
[10] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," ICFPT2007, pp.121–128, 2007.
[11] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "A virus scanning engine using a parallel finite-input memory machine and MPUs," FPL2009, pp.635–639, 2009.
[12] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "The parallel sieve method for a virus scanning engine," DSD2009, pp.809–816, 2009.
[13] J.T.L. Ho and G.G.F. Lemieux, "PERG-Rx: A hardware pattern-matching engine supporting limited regular expressions," FPGA2009, pp.257–260, 2009.
[14] H. Nakahara, T. Sasao, and M. Matsuura, "A low-cost and high-performance virus scanning engine using a binary CAM emulator and an MPU," LNCS 7199, pp.202–214, 2012.
[15] ClamAV, http://www.clamav.net/
[16] T. Sasao, "Row-shift decompositions for index generation functions," DATE2012, pp.1585–1590, 2012.
[17] T. Sasao, "Linear decomposition of index generation functions," ASPDAC2012, pp.781–788, 2012.
[18] Google, "Google Safe Browsing API," http://code.google.com/intl/ja/apis/safebrowsing/
[19] CAST inc., "MD5 IP Core," http://www.cast-inc.com/ip-cores/encryption/md5/
[20] PCRE: Perl compatible regular expressions, http://www.pcre.org/
[21] T. Sasao, M. Matsuura, and H. Nakahara, "A realization of index generation functions using modules of uniform sizes," IWLS'10, pp.201–208, June 2010.
[22] Z. Kohavi, Switching and Finite Automata Theory, McGraw-Hill, 1979.
[23] T. Sasao, Memory-Based Logic Synthesis, Springer, 2011.
[24] R.E. Tarjan and A.C-C. Yao, "Storing a sparse table," Commun. ACM, vol.22, no.11, pp.606–611, 1979.
[25] Xilinx inc, "MicroBlaze," http://www.xilinx.com/

Hiroki Nakahara received the B.E., M.E., and Ph.D. degrees in computer science from Kyushu Institute of Technology, Fukuoka, Japan, in 2003, 2005, and 2007, respectively. He has held a research position at Kyushu Institute of Technology, Iizuka, Japan. Now, he is an assistant professor at Kagoshima University, Japan. He received the 8th IEEE/ACM MEMOCODE Design Contest 1st Place Award in 2010, the SASIMI Outstanding Paper Award in 2010, IPSJ Yamashita SIG Research Award in 2011, and the 11th FIT Funai Best Paper Award in 2012, respectively. His research interests include logic synthesis, reconfigurable architecture, digital signal processing and embedded systems. He is a member of the IEEE and the ACM.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in Electronics Engineering from Osaka University, Osaka Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan; IBM T. J. Watson Research Center, Yorktown Height, NY; the Naval Postgraduate School, Monterey, CA; and Kyushu Institute of Technology, Iizuka, Japan. Now, he is a Professor of Department of Computer Science, Meiji University, Kawasaki, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design including, *Logic Synthesis and Optimization*, *Representation of Discrete Functions*, *Switching Theory for Logic Synthesis*, *Logic Synthesis and Verification*, and *Memory-Based Logic Synthesis*, in 1993, 1996, 1999, 2001, and 2011, respectively. He has served Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan in 1998. He received the NIWA Memorial Award in 1979, Takeda Techno-Entrepreneurship Award in 2001, and Distinctive Contribution Awards from IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003 and 2004. He has served an associate editor of the *IEEE Transactions on Computers*. He is a Fellow of the IEEE.

Munehiro Matsuura studied at the Kyushu Institute of Technology from 1983 to 1989. He received the B.E. degree in Natural Sciences from the University of the Air, in Japan, 2003. He has been working as a Technical Assistant at the Kyushu Institute of Technology since 1991. He has implemented several logic design algorithms under the direction of Professor Tsutomu Sasao. His interests include decision diagrams and exclusive-OR based circuit design.