The Parallel Sieve Method for a Virus Scanning Engine

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

Abstract

This paper shows a new architecture for a virus scanning system, which is different from that of an intrusion detection system. The proposed method uses two-stage matching: In the first stage, a hardware filter quickly scans the text to find partial matches, and in the second stage, the MPU scans the text to find a total match in the ClamAV 514,287 virus pattern set. To make the hardware filter simple, we use a finite-input memory machine (FIMM). To reduce the memory size of the FIMM, we introduce the parallel sieve method. The proposed method is memory-based, so it is quickly reconfigurable and dissipates lower power than a TCAM-based method. The system is implemented on the Stratix III FPGA with three off-chip SRAMs and an SDRAM, where all ClamAV 514,287 virus patterns are stored. Compared with existing methods, our method achieves 1.41-31.36 times more efficient area-throughput ratio.

1 Introduction

A **malware** (a composite word from malicious software) 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, the throughput for software-based scanning is at most tens of mega bits per second (Mbps) [16]. 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 [22]. Fig. 1 shows an example of a virus scanning system. To detect the malware, first the packet receiver assembles the data from the incoming packets. Also, for compressed data, the packet receiver decompresses it. Then, the virus scanning engine inspects the data to see if it contains the malware. Finally, the packet sender assembles the data to packets, and sends them to the Intranet. The most important part of the virus scanning system is the virus scanning engine. Other parts can be realized by the conventional technique. So, in this paper, we focus on the virus scanning engine.

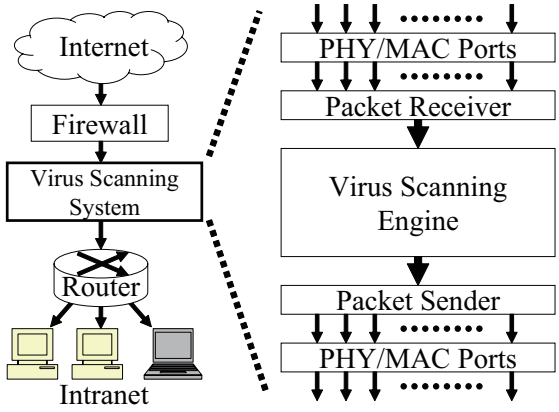

Figure 1. Virus Scanning System.

In a typical commercial virus scanning system[22], the throughput is about 1.2 Gbps, the power dissipation is 450 W, and the price is around $10,000. Here, we consider a virus scanning engine with the following features:

High throughput It has a throughput with more than one Gbps.

Low power It is SRAM-based rather than TCAM-based [3, 24]. The power dissipation for the TCAM is higher than that for the SRAM. Also, the number of transistors per bit for the TCAM is larger than that for the SRAM [14]. Table 1 [9] compares the SRAM with the TCAM.

High-speed reconfigurable It uses a memory-based realization rather than the hard-wired realization. Some virus scanning software, e.g., Kaspersky [10], updates the virus data every hour. Although, the random logic implementation of the virus scanning circuit on the FPGA [6] is fast and compact, the time for place-and-route is longer than the period for the virus pattern update. Thus, the hard-wired system is unsuitable for quick update.

Memory efficient Various memory-based methods have been proposed. For example, in [5], the patterns are embedded into the memory of a sequential circuit for an Aho-Corasick automaton. It requires 46 bytes per character. For the current version (v.0.94.2) of ClamAV [7] (the most popular open source anti-virus software), the number of patterns is 514,287. To store all the patterns, tens of high-end

FPGAs are required. Thus, the method [5] is too expensive for virus scanning.

Table 1. Comparison of TCAM with SRAM (18Mbit chip) [9]

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

Since the number of virus patterns is large, conventional methods cannot be used. In this paper, we show a new architecture for the virus scanning system, which is different from that of the intrusion detection system. The proposed method uses two-stage matching [5, 24], which is area-throughput efficient. 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 scans the text to find the total match. To make the hardware filter simple, we use a finite-input memory machine (FIMM). To reduce the memory size of the FIMM, we introduce the parallel sieve method. The proposed method is memory-based, so the power consumption is lower than the TCAM-based method.

The rest of the paper is organized as follows: Chapter 2 introduces the virus scanning system; Chapter 3 describes the finite-input memory machine (FIMM); Chapter 4 shows the parallel sieve method that reduces the memory for the FIMM; Chapter 5 shows implementation results on an Altera FPGA; and Chapter 6 concludes the paper.

2 Virus Scanning System

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] that consists of the **sub-pattern**. Virus scanning corresponds to the pattern matching that detects variable length patterns in the text.

2.2 Virus Pattern in ClamAV

The current version of ClamAV (v.0.94.2) [7] contains three types patterns: the MD5 checksum pattern, the basic pattern, and the restricted regular expression pattern. The MD5 checksum pattern and the basic pattern is represented by **characters**, while the restricted regular expression pattern is represented by **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 2 shows meta characters used in the ClamAV, and Table 3 shows examples of virus patterns in ClamAV.

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

Table 2. 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 3. 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

2.3 Virus Patterns in ClamAV

Table 4 compares the current version of ClamAV (v.0.94.2) with that of SNORT (v.2.8.3.2) [20]. It shows that regular expressions for ClamAV are simpler than that for SNORT. However, the number of patterns in ClamAV is much larger than that in SNORT. This implies that, in the virus scanning system, a memory efficient architecture is required.

Table 4. 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

2.4 Two-stage Matching for the Virus Scanning

ClamAV uses **two-stage matching** to achieve a high-speed and memory-efficient system. The first stage searches sub-patterns using an Aho-Corasick (AC) automaton [1] to find **partial matches** [5]. 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 fixed-length patterns. The AC automaton can be realized by a state machine consisting of a memory and a register. The memory stores the state transition functions and the output functions, while the register stores the state variables[2]. The state transitions for the AC automaton is complex, the size of memory tends to be large. To realize the compact automaton, ClamAV uses the AC automaton with the length $c = 3$. The second stage scans patterns us-

[2]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 that stores the state transition functions and the output functions (in bits).

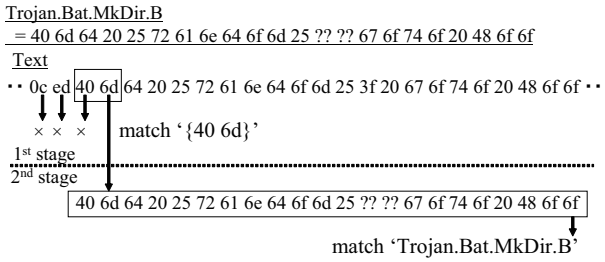

Figure 2. An Example of the Two-stage Matching.

ing PCRE (Perl Compatible Regular Expression library for C-language) [15] to find **total matches**, while the first stage detects the partial match. To perform the second stage, additional off chip memory and an MPU are used.

Example 2.1 *Fig. 2 illustrates two-stage matching that detects Trojan.Bat.MkDir.B in Table 3. First, it detects* 406d*, then it detects the whole pattern of Trojan.Bat.MkDir.B.*

2.5 Profile Analysis for the Virus Scanning

To analyze the profile of two-stage matching on a PC, we selected 512 patterns from ClamAV, and performed two-stage matching for 10 executable codes. A 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 throughput, we should consider the following:

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

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

From an experiment on a PC, the interval for the partial match occurrence (the number of characters) is about 100, when $c = 3$. For our virus engine, in the second stage, an embedded MPU on an FPGA is used. However, the embedded MPU is slower than a PC. When an interrupt happens during the matching operation on the embedded MPU, the system must suspend the hardware filter in the first stage. To avoid suspension, we increase the length of patterns to $c = 4$ to reduce the number of partial matches[3].

3 Virus Scanning Engine Using a Finite-Input Memory Machine and an MPU

In two-stage matching for virus scanning, it is necessary to reduce the hardware filter. To make the hardware filter compact, first we introduce the finite-input memory machine (FIMM). Also, in this section, we describe the two-stage matching using an FIMM and an MPU.

[3]We can increase the length more than $c = 4$. However, it increases the amount of memory. Detail is shown in Section 4.4.

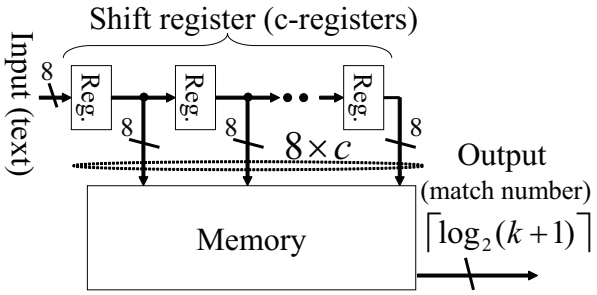

Figure 3. Finite-Input Memory Machine (FIMM).

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 [21]. However, for the virus scanning system, the size of the circuit would be too large even if bit partitioning method is used. By restricting the transitions, we have a **finite-input memory machine (FIMM)** [11]. Fig. 3 shows the architecture of an FIMM that accepts k patterns of length at most c. In Fig. 3, *Reg* denotes an 8 bit parallel-in parallel-out register. The FIMM stores the past c inputs in a shift register. The memory produces the match number. The FIMM is smaller and faster than the state machine for the AC-automaton, since no circuit for the state transition functions is required. The FIMM has 2^{8c} states. 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

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

Since the amount of memory for the state variables realized by the shift register is much smaller than that for the output functions, it is ignored.

3.2 Virus Scanning Engine

Fig. 4 shows the virus scanning engine consisting of an FIMM and an MPU. The FIFO stores indices for partial matches and the positions for the detected sub-patterns. When a partial match 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.

4 Realization of the FIMM using the Parallel Sieve Method

For the virus scanning engine using two-stage matching, in the first stage, the FIMM scans the text of length c to find partial matches, while in the second stage, the MPU scans

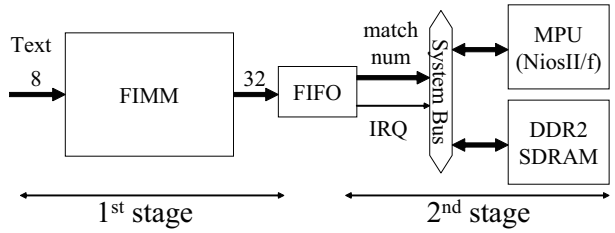

Figure 4. Virus Scanning Engine.

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

full length text to find the total match. When the output function of the FIMM is implemented by a single memory, the necessary memory size M_{FIMM} is

$$M_{FIMM} = 2^{8\times c}\lceil log_2(k+1)\rceil \simeq 2^{51}[bits], \quad (1)$$

when $k = 514,287$ and $c = 4$. So, a single-memory implementation is impractical. To reduce the memory size, we use the parallel sieve method which uses multiple index generation units (IGUs).

4.1 Index Generation Function

Definition 4.1 *A mapping $F(\vec{X}) : B^n \rightarrow \{0, 1, \ldots, k\}$, is an* **index generation function with weight *k***. *if $F(\vec{a}_i) = i$ $(i = 1, 2, \ldots, k)$ for k different* **registered vectors**, *and $F = 0$ for other $(2^n - k)$ input vectors, where $\vec{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$. In other words, an index generation function produces indices ranging from 1 to k for k different registered vectors, and produces 0 for other vectors.*

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

Example 4.2 *Table 5 shows an example of an index generation function f, where $n = 6$ and $k = 7$.*

An FIMM produces an index generation function as shown in Fig. 3. A content addressable memory (CAM) implements the index generation function directly. However, an FPGA realization of a CAM requires a large number of elements [8, 13]. Also, the CAM dissipates more power than the SRAM as shown in Table. 1. Thus, SRAMs are used to implement the index generation function.

4.2 Index Generation Unit

Let $f(X_1, X_2)$ be an index generation function of n variables with weight k. A single-memory requires $2^n\lceil \log_2(k+1)\rceil$ bits.

Table 6. Decomposition Chart for $f(X_1, X_2)$.

	0 0 0 0 1 1 1 1	x_3
	0 0 1 1 0 0 1 1	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	
$x_6x_5x_4$		

Table 7. Decomposition Chart for $\hat{f}(Y_1, X_2)$.

	0 0 0 0 1 1 1 1	y_3
	0 0 1 1 0 0 1 1	y_2
	0 1 0 1 0 1 0 1	y_1
000	0 0 0 0 0 0 0 0	
001	0 0 0 0 0 0 0 0	
010	2 0 1 0 0 0 3 0	
011	0 0 4 0 0 0 0 0	
100	0 5 0 0 0 0 0 0	
101	0 0 0 0 0 0 0 0	
110	0 0 0 0 6 0 0 0	
111	0 0 0 0 0 7 0 0	
$x_6x_5x_4$		

Table 8. Decomposition Chart for $\hat{f}_1(Y_1, X_2)$.

	0 0 0 0 1 1 1 1	y_3
	0 0 1 1 0 0 1 1	y_2
	0 1 0 1 0 1 0 1	y_1
000	0 0 0 0 0 0 0 0	
001	0 0 0 0 0 0 0 0	
010	2 0 1 0 0 0 3 0	
011	0 0 0 0 0 0 0 0	
100	0 5 0 0 0 0 0 0	
101	0 0 0 0 0 0 0 0	
110	0 0 0 0 6 0 0 0	
111	0 0 0 0 0 7 0 0	
$x_6x_5x_4$		

Table 9. Main Memory $\hat{h}(Y_1)$ for $\hat{f}_1(Y_1)$.

y_3	0	0	0	0	1	1	1	1
y_2	0	0	1	1	0	0	1	1
y_1	0	1	0	1	0	1	0	1
$\hat{h}$	2	5	1	0	6	7	3	0

Let $\hat{f}(Y_1, X_2)$ be the function whose variables $X_1 = (x_1, x_2, \ldots, x_p)$ are replaced by $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$, $x_j \in \{X_2\}$, and $p \geq \lceil \log_2(k+1)\rceil$.

Example 4.3 *Table 6 shows a* **decomposition chart** *for the index generation function shown in Example 4.2. The column labeled $X_1 = (x_1, x_2, x_3)$ denotes* **bound variables**, *and row labeled $X_2 = (x_4, x_5, x_6)$ denotes* **free variables**. *The corresponding chart entry denotes the function value. Table 7 shows the decomposition chart for $\hat{f}(Y_1, X_2)$, where $Y_1 = (x_1 \oplus x_6, x_2 \oplus x_5, x_3 \oplus x_4)$. In Table 7, column labels denote $Y_1 = (y_1, y_2, y_3)$, and row labels denote $X_2 = (x_4, x_5, x_6)$. If a column of a decomposition chart has two or more non-zero elements, then the column has a* **collision**. *The number of collisions is three in Table 6, while the number of collisions is only one in Table 7.*

In Table 7, assume that the element '4' in the column (0,1,0) is realized by an other circuit. By removing '4' from $\hat{f}$, we have $\hat{f}_1$ whose decomposition chart is shown in Table 8, where no collision occurs. Note that, we can represent the non-zero elements of $\hat{f}_1$ by a **main memory** $\hat{h}$ whose input is Y_1. Table 9 shows the function $\hat{f}_1(Y_1)$ of the main memory. The main memory realizes a mapping from a set of 2^p elements to a set of at most 2^p elements. The output for the main memory does not always represent f, since $\hat{f}_1$ ignores X_2. Thus, we must check whether $\hat{f}_1$ is equal to f or not by using an **auxiliary memory**. To do this, we compare the input X_2 with the output for the auxiliary memory

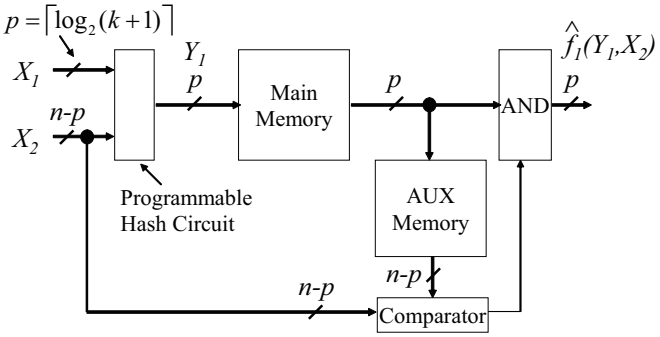

Figure 5. Index Generation Unit (IGU).

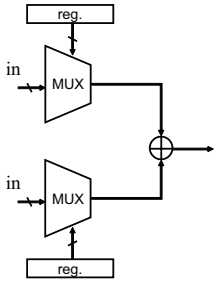

Figure 6. Programmable Hash Circuit.

by a **comparator**. The auxiliary memory stores the values of X_2 when the output of $\hat{f}_1(Y_1, X_2)$ is non-zero. Fig. 5 shows the **index generation unit (IGU)** [18]. First, the **programmable hash circuit** shown in Fig. 6 generates the hashed inputs Y_1 from the primary inputs (X_1, X_2), where $|X_1| = |Y_1|$. Second, the main memory finds the possible index corresponding to Y_1. Third, the auxiliary memory produces the corresponding inputs X_2'. Fourth, the comparator checks whether X_2' is equal to X_2 or not. And, finally, the **AND gates** produce the correct value $\hat{f}(Y_1, X_2)$.

Example 4.4 *Fig 7 shows the circuit of 6-variable function in Table 6. The element '4' is realized by the AND gate, while the other elements are realized by the IGU. The binary representation of '4' is (1,0,0). The output of the function is realized by ORing the most significant bit of the output of IGU and the output of the AND gate.*

4.3 Capability of the Index Generation Unit

Theorem 4.1 *[17] Let f be an n-variable index generation function with weight k. Let the non-zero elements of f be uniformly distributed in the decomposition chart of f. Then, the fraction of registered vectors realized by the index generation unit (IGU) in Fig. 5 is given by*

$$\delta \simeq 1 - \frac{1}{2}(\frac{k}{2^p}) + \frac{1}{6}(\frac{k}{2^p})^2,$$

where $p = |Y_1|$ denotes the number of bound variables in the decomposition chart for $f(Y_1, X_2)$, and $k \leq 2^p$.

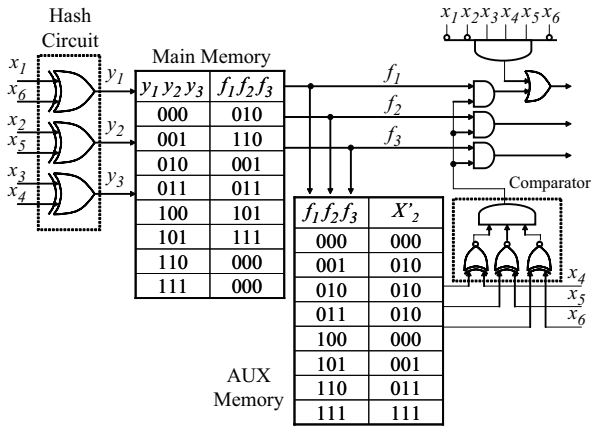

Figure 7. A realization of 6-variable function shown in Table 6.

Example 4.5 *When $\frac{k}{2^p} = \frac{1}{1}$, we have $\delta = \frac{2}{3} \simeq 0.666$. In this case, the main memory realizes 66.6% of the registered vectors. Note that the programmable hash circuit is used to make f uniformly distributed.*

Experimental results show that, by increasing the number of inputs for the main memory, we can store virtually all vectors.

Conjecture 4.1 *[19] Consider a set of uniformly distributed index generation functions with weight k. In most cases, an index generation function can be represented by an IGU with the main memory having at most*

$$p = 2\lceil log_2(k+1) \rceil - 1$$

inputs.

4.4 The Parallel Sieve Method

From Theorem 4.1, given p and k, we can estimate the number of vectors realized by the IGU. Consider virus scanning where the total number of registered vectors k is 514,287, and the length of the patterns c is 4. As shown in (1), the single-memory realization requires $2^{4\times 8} \times 514,287 \simeq 2^{51}$ [bits], which is impractical. Suppose that all the registered vectors are stored in a single IGU. From Conjecture 4.1, the number of inputs for the main memory is $p = 2\lceil log_2(514,287+1) \rceil - 1 = 37$, which is also too large, since the memory size is 2^{37} [bits]. Thus, we implement the function by applying Theorem 4.1 many times. Consider the case, where $\frac{514,287}{2^p} \simeq \frac{1}{1}$. The number of inputs p is 19, which is practical for modern SRAMs. In this case, we have, $\delta = \frac{2}{3} \simeq 0.666$. By storing registered vectors separately in the main memories of IGUs shown in Fig. 8, we can implement most of the vectors in a series of main memories of IGUs. A small number of remaining vectors can be implemented by an additional IGU by using Conjecture 4.1.

Example 4.6 *When the number of remaining vectors is 200, by Conjecture 4.1, an IGU having a main memory with 15 inputs can implement all the remaining vectors.*

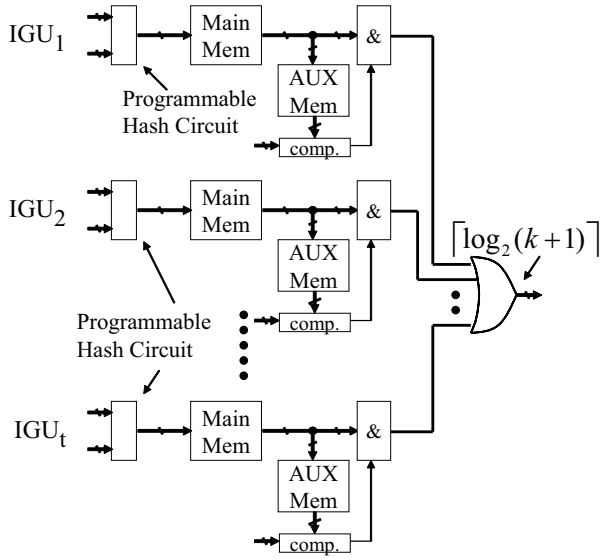

Figure 8. Parallel Sieve Method.

Definition 4.2 *The* **parallel sieve method** *is an implementation of an index generation function using the circuit consisting of multiple IGUs as shown in Fig. 8. IGU_{i+1} is used to realize a part of the registered vectors not realized by IGU_1, IGU_2, ..., or IGU_i. The OR gate in the output combines the indices to form a single output.*

4.5 Number of IGUs in the Parallel Sieve Method

Let k be the number of registered vectors, and p be the number of inputs for the first main memory. From Theorem 4.1, when $\frac{k}{2^p} = 1$, the main memory stores $\delta = \frac{2}{3}$ of the registered vectors. In this case, the fraction for the remaining vectors is $\gamma = 1 - \delta = \frac{1}{3}$. Next, choose p' so that the second main memory stores $\delta = \frac{2}{3}$ of the remaining vectors. Then, the fraction of vectors realized by the second main memory is $\gamma\delta = \frac{1}{3} \times \frac{2}{3} = \frac{2}{9}$. Therefore, the fraction for the vectors realized by two main memories is $\frac{2}{3} + \frac{2}{9} = \frac{6+2}{9} = \frac{8}{9}$.

For each step, we choose the smallest integer p_j such that 2^{p_j} is greater than the number of remaining registered vectors. By generalizing this, we have the following:

Theorem 4.2 *Let k be the total number of registered vectors, t be the number of index generation units (IGUs), and r be the number of vectors not realized by these t IGUs. Then,*

$$t = \lceil log_\gamma(\frac{r}{k})\rceil, \quad (2)$$

where δ is given by Theorem 4.1, $\gamma+\delta = 1$, and the number of inputs for each main memory p_j satisfies the relation: (the number of remaining registered vectors) $\leq 2^{p_j}$.

(Proof) Let δ be the fraction of the vectors realized by the i-th IGU, and let γ be the fraction for the remaining vectors. Then, the fraction of vectors realized by t IGUs is

$$\delta + \gamma\delta + \gamma^2\delta + \cdots + \gamma^{t-1}\delta = \delta\frac{1-\gamma^t}{1-\gamma} = 1 - \gamma^t.$$

Since we try to realize k vectors, and r is the number of vectors not realized by t IGUs, we have

$$r = k - k(1-\gamma^t) = k\gamma^t.$$

By solving the above expression for t, we have

$$t = log_\gamma(\frac{r}{k}).$$

Since t is an integer, we have (2). (Q.E.D.)

Given the number of remaining vectors r, Theorem 4.2 shows the necessary number of IGUs. To store most vectors in IGUs, the number of IGUs should be large. This makes the circuit complicated. In this paper, we store vectors in the IGUs using off-chip SRAMs until the remaining vector can be stored in the embedded RAMs of the FPGA. Let the number of remaining registered vectors be k_{t+1}. By Conjecture 4.1, the number of inputs p_{t+1} for the $(t+1)$-th main memory must satisfy the relation $p_{t+1} \leq 2\lceil log_2(k_{t+1}+1)\rceil - 1$. Note that, the size of the embedded memory for the Altera FPGA is 9 kbit. When $k_{t+1} \leq 255$, we have $p_{t+1} \leq 15$, which is a practical value for the embedded memory. In this way, we can store all the remaining vectors in an embedded memory of an FPGA.

Example 4.7 *Let k_j be the number of registered vectors to be implemented by $IGU_j, IGU_{j+1}, \ldots$, and IGU_{t+1}, p_j be the number of inputs for the j-th main memory, t be the number of index generation units (IGUs), and r be the number of vectors not realized by these t IGUs. Consider virus scanning with $k = 514,287$. When $2^{p_j} = k_j$, we have $\gamma = \frac{1}{3}$. From Theorem 4.2, when $r = 255$, the number of necessary IGUs is*

$$t = \lceil \log_{\frac{1}{3}} \frac{255}{514,287}\rceil = 7.$$

Finally, we use an additional IGU to store the remaining 255 vectors. Thus, eight IGUs are used to store all the patterns.

Example 4.8 *Table 10 compares the estimated values of stored vectors with that for the experimental values. In Table 10, IGU_j denotes the j-th index generation unit; $\hat{k}_j$ denotes the number of vectors implemented by IGU_j consists of four characters rather than the original registered vector k, where $\hat{k} < k$; and r_j denotes the number of remaining vectors. We can see that the necessary number of IGUs is eight which is derived in Example 4.7.*

In the parallel sieve method, we consider the patterns consisting of four characters. Thus, the number of unique patterns is $\hat{k} = 497,172$, rather than $k = 514,287$.

5 Experimental Results

5.1 Implementation of a Virus Scanning Engine

We implemented a virus scanning engine using the parallel sieve method on an Altera FPGA. In our implementation, we used StratixIII EP3SL340H1152C3NE5 (containing 270,400 ALUTs, and 1,040 M9ks). To perform the total

Table 10. Comparison of the Estimated Value with the Experimental Value.

		Estimated Value			Experimental Value		
	j	p_j	$\hat{k}_j$	r_j	p_j	$\hat{k}_j$	r_j
IGU_1	1	19	331,414	165,757	19	321,659	175,513
IGU_2	2	18	110,493	55,263	18	128,398	47,115
IGU_3	3	16	36,838	18,424	16	33,791	13,324
IGU_4	4	15	12,281	6,142	14	9,267	4,057
IGU_5	5	13	4,094	2,047	12	2,641	1,416
IGU_6	6	11	1,364	682	11	1,055	361
IGU_7	7	10	454	227	9	277	84
IGU_8	8	15	227	0	9	84	0

match, we used the embedded processor NiosII/f. We stored the executable code and the full length of $k = 514,287$ patterns into the 1 Giga bytes DDR2-SDRAM. Note that, we also stored $\sum_{j=1}^{8} \hat{k}_j = 497,172$ patterns consisting of four characters into SRAMs for the parallel sieve method. For the FPGA synthesis tool, we used QuartusII (v.8.0). Table 11 shows the memories used in the parallel sieve method. In Table 11, *IGU_j* denotes the j-th index generation unit; $\hat{k}_j$ denotes the number of stored vectors in IGU_j; *INOUT* denotes the number of inputs (i) and outputs (o); *Memory* denotes the type of the memory, where *SRAM* denotes the 8MB off-chip SRAM, *M144k* and *M9k* denote the embedded memory on the Altera FPGA, respectively. In our implementation, the parallel sieve method uses three off-chip SRAMs, 3,790 ALUTs, 31 M9ks, and 13 M144ks[4]. Note that these values do not include the resource for the NiosII/f. The FPGA operates at 370.1 MHz. However due to a limitation on the clock frequency by the off-chip SRAM, we set the system clock to 200 MHz. Our virus engine scans one character in every clock. Thus, the throughput is $0.200 \times 8 = 1.600$ Gbps. The system stores 514,287 patterns, while the parallel sieve method stores unique 497,172 patterns consisting of four characters. From Table 11, the total amount of memory is 3,500,880 Bytes. Let the **memory utilization coefficient (MUC)** be the necessary amount of memory per a character. Then, MUC for the parallel sieve method is $\frac{3,500,880}{497,172 \times 4} = 1.7$ Bytes/Char.

5.2 Comparison with Existing Methods

Table 12 compares existing regular expression matching methods. They use various methods in different technologies: FPGAs and ASICs. To make a fair comparison, we use the **normalized throughput** Th/MUC. In Table 12, Th denotes the throughput for a pattern matching engine [Gbps]; *# of patterns* denotes the number of stored patterns; MUC denotes the memory utilization coefficient [Bytes/Char]; and Th/MUC denotes the normalized throughput.

Table 12 shows that only our method can store 497,172 virus patterns on a single FPGA and off-chip SRAMs. Also, as for Th/MUC, our method is about 470.5 times better than the AC method [23], and is 1.4-31.3 times better than other methods. The reason is that the MUC for our method is quite small, since the parallel sieve method stores patterns in compact memories. As for Th/MUC, Yu et al.'s method [24] is second to our method. However, they use a TCAM which is quite expensive and dissipates much power. If we consider the cost of TCAM in Table 1, our method is much better than Yu et al.[24].

[4]Excludes the MD5 checksum hardware.

[5]Since the off-chip SRAM has 72 outputs, three main memories are stored in a single SRAM.

6 Conclusion and Comments

In this paper, we showed a new architecture for a virus scanning system, which is different from that of the intrusion detection system. The proposed method uses two-stage matching: In the first stage, the hardware filter quickly scans the text to find partial matches, and in the second stage, the MPU scans the full text to find the total match. To make the hardware filter simple, we used a finite-input memory machine (FIMM). To reduce the memory size for the FIMM, we introduced the parallel sieve method. The proposed method is memory-based, so it is quickly reconfigurable and dissipates lower power than a TCAM-based method. The system for 514,287 virus patterns was implemented on the Stratix III FPGA, three off-chip SRAMs and an SDRAM. Compared with existing methods, our method achieved 1.41-31.36 times more efficient area-throughput ratio.

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. [12] 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 the threshold, the circuit detects the performance attack. Kumer's method can be applied to our virus scanning engine.

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

Table 11. Memories used in the Sieve Method.

			Main Memory		AUX Memory	
	j	$\hat{k}_j$	INOUT	Memory	INOUT	Memory
IGU_1	1	321,659	i=19,o=19	SRAM[5]	i=19,o=13	SRAM
IGU_2	2	128,398	i=18,o=18	SRAM[5]	i=18,o=14	SRAM
IGU_3	3	33,791	i=16,o=16	SRAM[5]	i=16,o=16	8 M144k
IGU_4	4	9,267	i=14,o=14	2 M144k	i=14,o=18	3 M144k
IGU_5	5	2,641	i=12,o=12	6 M9k	i=12,o=20	10 M9k
IGU_6	6	1,055	i=11,o=11	3 M9k	i=11,o=21	6 M9k
IGU_7	7	277	i=9,o=9	1 M9k	i=9,o=23	2 M9k
IGU_8	8	84	i=9,o=7	1 M9k	i=9,o=23	2 M9k

Table 12. Comparison with Existing Methods.

Method	Th [Gbps]	# of Patterns	MUC [Bytes/char]	Th/MUC	Comment
Aho-Corasick Method (AC method)[23]	6.0	1,533	2,896.2	0.0020	ASIC implementation
Aldwari et al.[2]	14.0	1,542	126.0	0.1111	FPGA+SRAM
Bitmap compressed Aho-Corasick[23]	8.0	1,533	154.0	0.0519	ASIC implementation
Path compressed Aho-Corasick[23]	8.0	1,533	60.0	0.1333	ASIC implementation
Alicherry et al.[3]	20.0	100	48.0	0.4166	with TCAM
Yu et al.[24]	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
The parallel sieve Method	1.6	497,172	1.7	0.9417	FPGA+MPU+SRAM+SDRAM

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] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *16-th Int. Conf. on Field Programmable Logic and Applications (FPL'06)*, pp.119-126, 2006.

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

[8] J. Ditmar, K. Torkelsson, and A. Jantsch, "A dynamically reconfigurable FPGA-based content addressable memory for internet protocol," *International Conference on Field Programmable Logic and Applications 2000*, (FPL2000), pp.19-28.

[9] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," *27-th IEEE Int. Conf. on Computer Communications (INFOCOM2008)*, pp.1786-1794, 2008.

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

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

[12] 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.

[13] K. McLaughlin, N. O'Connor, and S. Sezer, "Exploring CAM design for network processing using FPGA technology," *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services* (AICT/ICIW 2006), p.84.

[14] K. Pagiamtzis and A. Sheikholeslami, "A Low-power content-addressable memory (CAM) using pipelined hierarchical search scheme, " *IEEE Journal of Solid-State Circuits*, Vol. 39. No. 9, Sept. 2004, pp.1512-1519.

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

[16] 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.

[17] T. Sasao, "A Design method of address generators using hash memories," *IWLS-2006*, Vail, Colorado, U.S.A, June 7-9, 2006, pp.102-109.

[18] T. Sasao and M. Matsuura, "An implementation of an address generator using hash memories," *DSD 2007, 10th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Aug. 27 - 31, 2007, Lubeck, Germany, pp.69-76.

[19] T. Sasao, "On the number of variables to represent sparse logic functions," *ICCAD-2008*, San Jose, California, USA, Nov.10-13, 2008, pp. 45-51.

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

[21] 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.

[22] TrendMicro, *Network Virus Wall Enforcer*, http://us.trendmicro.com/.

[23] 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.

[24] 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.