# A Low-Cost and High-Performance Virus Scanning Engine Using a Binary CAM Emulator and an MPU

Hiroki Nakahara[1], Tsutomu Sasao[2], and Munehiro Matsuura[2]

[1] Kagoshima University, Japan
[2] Kyushu Institute of Technology, Japan

**Abstract.** 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 four index generation units (IGUs). The proposed system uses four 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 FPGA-based implementations.

## 1 Introduction

### 1.1 Virus Scanning System

A **computer virus**[3] 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) [16], 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 that of servers (hundreds Mbps). It has a throughput with higher than one Gbps.

*Low power and low cost:* It uses a low-end (i.e., a small) FPGA and SRAMs[4], instead of a high-end FPGA and a TCAM. Table 1 shows that the TCAM dissipates much higher power than the SRAM. Although we can implement the CAM function on the FPGA [5, 8], for the virus scanning, it requires excessive amount of resources for 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 time for place-and-route is longer than the periods for the virus pattern update. Some virus scanning software, e.g., Kaspersky [10], updates the virus data every hour.

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

[4] As of Nov. 2011, the retail prices for semiconductor devices are as follows: a TCAM is hundreds USD (U.S. dollar); a high-end FPGA is more than ten thousand USD; a low-end FPGA is several USD; and an SRAM is tens USD [4].

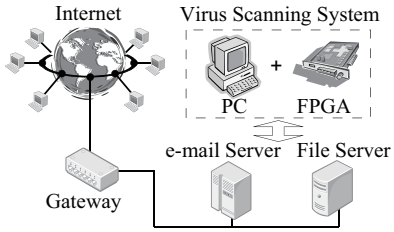


**Fig. 1.** Virus scanning system for an e-mail server and a file server.

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

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

### 1.2 Related Works

Our virus scanning engine uses two-stage matching to make the circuit compact [11]. Various two-stage matching implementations have been reported: A TCAM with a general purpose processor (MPU) [23]; a bit-partitioned Aho-Corasic DFA [20] with a special purpose MPU [1]; hash methods using cuckoo hashing [21]; parallel FIMMs with an MPU [13]; the parallel sieve method with an MPU [14]; and Bloom filter (PERG-Rx) [7].

### 1.3 Contributions of The Paper

*Implementation of more than one million ClamAV virus patterns:* We used a parallel sieve method.

*High-level characterization of the bandwidth by the hardware and the software:* We implement two-stage matching by the hardware and the software. We maximized the bandwidth 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: Chapter 2 introduces the virus scanning based on two-stage matching; Chapter 3 describes the binary CAM emulator for the FIMM; Chapter 4 shows the implementation results of the virus scanning engine; and Chapter 5 concludes the paper.

**Table 2.** Meta characters used in ClamAV

| Meta Char | Meaning | Example |
|---|---|---|
| ?? | An arbitrary character | |
| * | Repetition of more than zero "??" | AA*BB={AABB,AA??BB, AA????BB,AA??????BB,$\cdots$} |
| (AA\|BB) | Set of characters | (AA\|BB)={AA,BB} |
| {$n$-$m$} | Repetition of $n$ or more than $n$ "??" and $m$ or less than $m$ "??" | AA{1-2}BB={AA??BB, AA????BB} |

## 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 written by a **regular expression** consisting of **characters** and **meta characters**. A

**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 | | |

**pattern matching** is to detect variable-length patterns in the text. A character is represented by a pair of hexadecimal numbers. Table 2 shows the meta characters used in ClamAV. A **length** is the number of characters. A **subpattern** is a part of the pattern consisting characters only[5]. In this paper, $k$ denotes the number of patterns, $r$ denotes the length of a pattern, and $m$ $(m \leq r)$ denotes the length of a subpattern.

### 2.2 ClamAV Virus Pattern

As of December 1st, 2010, ClamAV (version 0.96.5) contains 1,290,808 patterns [3]. Table 3 shows the pattern types, the number of patterns, and their detection methods. An **MD5 checksum pattern** is the MD5 hash value (128 bits) of the virus. It is detected by the hardware. A **basic pattern** is a **regular expression** of a part of the virus. It 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 [6]. It is detected by the hardware. A **combination pattern** is a combination of basic patterns. It is detected by the logical operations of software such as "AND", "OR", and "NOT" of the basic patterns. 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 implement this.

Fig. 2 shows the virus scanning system. Since the computing 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 [2]. 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 realized by a **virus scanning engine** on the hardware (a small FPGA and SRAMs).

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

### 2.3 Virus Scanning Engine Using Two-Stage Matching

A ClamAV pattern consists of subpatterns and meta characters representing the distance. To detect patterns, we use **two-stage matching**. Fig. 3 shows the virus scanning engine using two-stage matching. Since no subpattern contains meta characters, in the

[5] However, a meta character "??" is permitted.

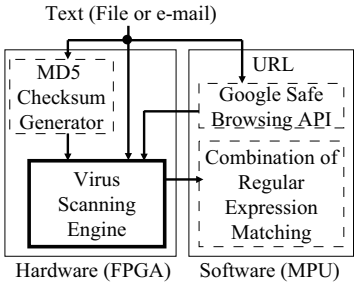


**Fig. 2.** Virus scanning system.

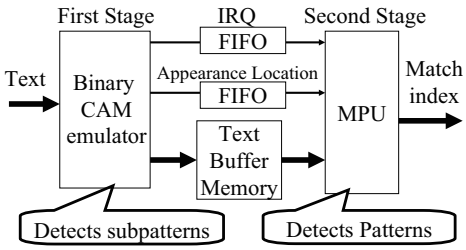


**Fig. 3.** Virus scanning engine using two-stage matching.

**Table 4.** Examples of ClamAV patterns

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

first stage, we use **a binary CAM emulator** to detect the subpattern. When a subpattern is detected, **the IRQ (interrupt request) signal** and **the appearance location** are sent to the MPU. Since the pattern contains meta characters, in the second stage, the embedded MPU performs PCRE (Perl compatible regular expression) [15] matching for the full length of the pattern. Since other subpatterns may be detected 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.

**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)). At this point, 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)[6], $m$ is at most 16. Let $k$ be the number of subpatterns with

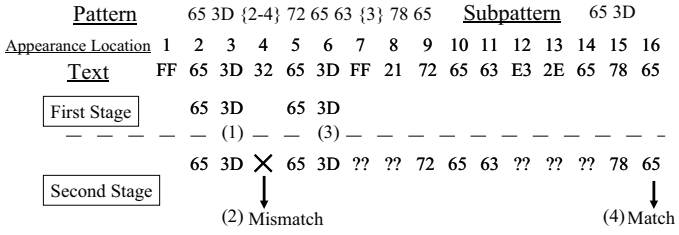


**Fig. 4.** Example of two-stage matching.

[6] For the basic patterns consisting of more than 16 characters, we extract the first 16 characters.

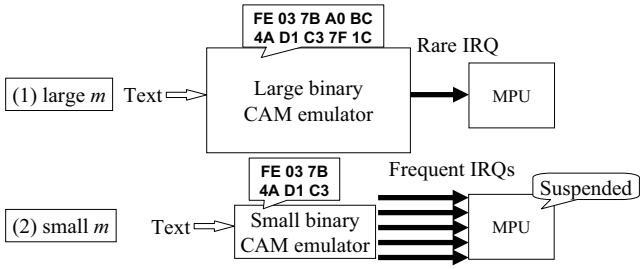


**Fig. 5.** Relation between the subpattern length $m$ and the number of IRQs.

length $m$ to be stored to the binary CAM emulator, then the **subpattern detection probability** $P(m)$ is $\frac{k}{m2^8}$ [7]. When $m$ is large, since $P(m)$ is small, the IRQ signal rarely occurs[8]. However, 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, so that the system throughput decreases. Thus, to minimize the size of the binary CAM emulator, we must find the minimum $m$ that does not suspend the MPU.

**Problem 2.1** *Let $k$ be the number of subpatterns with length $m$ to be stored to the binary CAM emulator, $T_{MPU}$ 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 characters. 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, and $\frac{T_{FIMM}}{P(m)}$ denotes the **average IRQ period**. Here, we assume that subpatterns are uniformly distributed. The actual value of $m$ is obtained experimentally in Section 4.1.

# 3 Binary CAM Emulator Using Four Index Generation Units

## 3.1 Index Generation Function

**Definition 3.1** *[18] A mapping $F(\boldsymbol{X}) : B^n \to \{0, 1, \ldots, k\}$, is an* **index generation function with weight $k$**, *where $F(\boldsymbol{a}_i) = i$ $(i = 1, 2, \ldots, k)$ for $k$ different* **registered vectors**, *and $F = 0$ for other $(2^n - k)$ non-registered vectors, and $\boldsymbol{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$. In other words, an index generation 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.

[7] when the distribution of the characters in the subpatterns is uniform.

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

**Table 5.** An 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 |

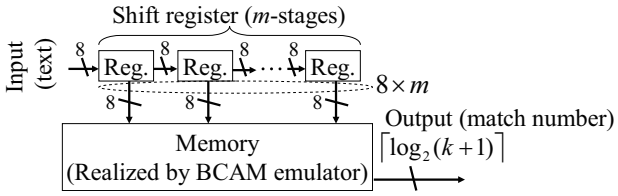


**Fig. 6.** Finite Input Memory Machine (FIMM).

### 3.2 Finite Input Memory Machine to Detect a Subpattern

Fig. 6 shows **a finite input memory machine (FIMM)** [12] 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 [9] of the FIMM, then, we have $M_{FIMM} = 2^{8m}\lceil log_2(k+1)\rceil$. Thus, a single-memory implementation is impractical for a large $m$.

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

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $x_3$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $x_2$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $x_1$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 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 |
| $x_6, x_5, x_4$ | | | | | | | | |

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

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $y_3$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $y_2$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $y_1$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 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 |
| $x_6, x_5, x_4$ | | | | | | | | |

### 3.3 Index Generation Unit For FIMM

In this paper, to realize the FIMM compactly, we use multiple index generation units (IGUs) [19].

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 $y_i = x_i \oplus x_j$, $x_i \in \{X_1\}$, $x_j \in \{X_2\}$, and $p \geq \lceil \log_2(k+1) \rceil$. Table 6 shows a **decomposition chart** for the index generation function shown in Example 3.3. The columns labeled by $X_1 = (x_1, x_2, x_3)$ denotes **bound variables**, and rows labeled by $X_2 = (x_4, x_5, x_6)$ 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 $Y_1 = (x_1 \oplus x_6, x_2 \oplus x_5, x_3 \oplus x_4)$, and the column labels denote $Y_1 = (y_1, y_2, y_3)$, and the row labels denote $X_2 = (x_4, x_5, x_6)$. When a column of a decomposition chart has two or more non-zero elements, it has a **collision**. The number of collisions is three in Table 6, while the number of collisions is only one in Table 7.

[9] 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.

**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 | |
| $x_6, x_5, x_4$ | | |

**Table 9.** Main memory for $\hat{h}(Y_1)$.

| $y_3$ | 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 |
| $\hat{f}_1$ | 2 | 5 | 1 | 0 | 6 | 7 | 3 | 0 |

In Table 7, assume that the element '4' in the column (0,1,0) is realized by other IGU. 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 the **main memory** $\hat{h}$ whose input is $Y_1$. Table 9 shows the function $\hat{h}(Y_1)$ of the main memory. The main memory realizes a mapping from a set of $2^p$ elements to a set of $k+1$ elements. 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 (AUX) memory. To do this, we compare the input $X_2$ with the output for the AUX memory by a **comparator**. The AUX memory stores the values of $X_2$ when the output of $\hat{f}_1(Y_1, X_2)$ is non-zero. Fig. 7 shows the **index generation unit (IGU)**. First, the **hash circuit** generates the transformed inputs $Y_1$ from the primary inputs $(X_1, X_2)$, where $|X_1| = |Y_1|$. The detailed design method for the hash circuit is described in [18]. Second, the main memory finds the possible index corresponding to $Y_1$. Third, the AUX memory produces the corresponding inputs $X_2'$ ($n-q$ bits, where $q = \lceil log_2(k+1) \rceil$). Fourth, the comparator checks whether $X_2'$ is equal to $X_2$ or not. Finally, the AND gates produce the correct value $\hat{f}(Y_1, X_2)$. We implement the main memory and the AUX memory (gray part in Fig. 7) by a single memory device with $|Y_1|$ $(= p\ bits)$ inputs and $q + |X_2'|$ $(= n\ bits)$ outputs.

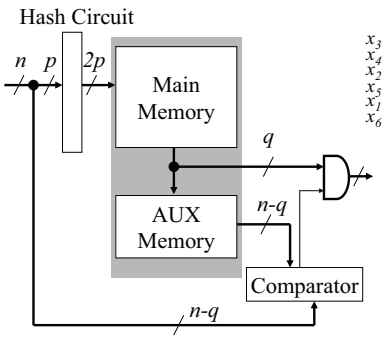


**Fig. 7.** Index generation unit (IGU) realizing the memory of the FIMM.

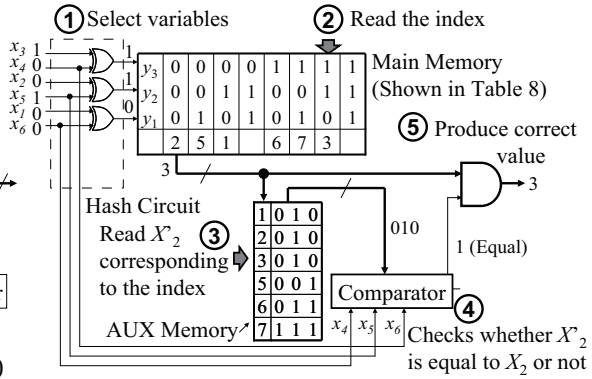


**Fig. 8.** Operation of IGU.

**Example 3.4** *Fig. 8 shows an example of operation of an IGU realizing* $\hat{f}_1(Y_1, X_1)$ *shown in Table 8.* ■

**Example 3.5** *When the decomposition chart shown in Table 8 is realized by a single memory, the memory size is* $2^6 \times \lceil log_2(7+1) \rceil = 192$ *(bits). On the other hand, the*

*memory size for the IGU shown in Fig. 8 is* $2^3 \times (3+3) = 48$ *(bits). Hence, the IGU can reduce the memory size.* ■

**Table 10.** Estimated value vs. experimental value for ClamAV 1,290,617 subpatterns stored in the standard parallel sieve method.

| | Estimated | | Experimental | |
|---|---|---|---|---|
| | $p$ | Stored | $p$ | Stored |
| $IGU_1$ | 21 | 963,815 (74.678%) | 21 | 934,999 |
| $IGU_2$ | 19 | 243,186 (18.842%) | 19 | 303,310 |
| $IGU_3$ | 17 | 61,816 (4.789%) | 16 | 30,967 |
| $IGU_4$ | 15 | 15,921 (1.233%) | 15 | 16,056 |
| $IGU_5$ | 13 | 4,195 (0.325%) | 13 | 4,255 |
| $IGU_6$ | 11 | 1,148 (0.089%) | 11 | 638 |
| $IGU_7$ | 10 | 416 (0.032%) | 9 | 329 |
| $IGU_8$ | 11 | 78 (remain) | 6 | 63 |

**Table 11.** Estimated value vs. experimental value for ClamAV 1,290,617 subpatterns stored in the 4IGU.

| | Estimated | | Experimental | |
|---|---|---|---|---|
| | $p$ | Stored | $p$ | Stored |
| $IGU_1$ | 21 | 963,815 (74.678%) | 21 | 953,221 |
| $IGU_2$ | 21 | 280,778 (21.755%) | 21 | 311,943 |
| $IGU_3$ | 21 | 45,028 (3.489%) | 21 | 25,276 |
| $IGU_4$ | 17 | 996 (remain) | 16 | 177 |

### 3.4 Capability of the Index Generation Unit

The fraction of registered vectors realized by the IGU has been analyzed [17].

**Theorem 3.1** *[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 $\delta$ of registered vectors realized by the index generation unit (IGU) is given by $\delta \simeq \frac{1-e^{-\xi}}{\xi}$, where $p$ denotes the number of inputs to the main memory, $k \leq 2^p$, and $\xi = \frac{k}{2^p}$.*

**Example 3.6** *When $\frac{k}{2^p} = 1$, we have $\delta = 1 - e^{-1} \simeq 0.632$. In this case, the main memory realizes 63.2% of the registered vectors. Note that the hash circuit is used to make the function uniformly distributed.* ■

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

**Conjecture 3.1** *[17] Consider a set of uniformly distributed index generation functions with weight $k$ ($\geq 7$). If $p \geq \lceil log_2(k+1) \rceil - 3$, then, more than 95% of the functions can be represented by an IGU with the main memory having $p$ inputs.*

### 3.5 Realization of the Index Generation Function Using Parallel Sieve Method [14]

From Theorem 3.1 and Conjecture 3.1, we can estimate the number of registered vectors $k$ stored in the main memory with $p$ inputs. The parallel sieve method stores all the subpatterns in multiple IGUs.

**Definition 3.2 The parallel sieve method** *is an implementation of an index generation function using multiple IGUs. $IGU_{i+1}$ is used to realize a part of the registered vectors not stored by $IGU_1, IGU_2, \ldots, IGU_i$. The OR gate in the output combines the indices to form a single output. In* **the standard parallel sieve method***, the number of inputs to the main memory is chosen as $p_i = \lceil log_2(k_i+1) \rceil$, where $k_i$ denotes the number of registered vectors to be realized by $IGU_j$, $(j \geq i)$.*

**Example 3.7** *Table 10 compares the numbers of estimated stored vectors with that for the experimental ones for ClamAV $k = 1,290,617$ subpatterns with length* [10] *$n = 40$.*

[10] $n = 40$ is obtained experimentally in our implementation described in Section 4.1.

*We can see that, the necessary number of IGUs is obtained from the given number of vectors $k$ by using Theorem 3.1 and Conjecture 3.1. In this case, the total amount of memory is $\sum_{i=1}^{8} 2^{p_i} n = 13.33$ MBytes.* ■

### 3.6 Realization of the Index Generation Function Using Four IGUs [18]

In this section, we show that most index generation functions can be realize with only four IGUs with the uniformed size. We call this **four IGUs method (4IGU)**.

**Example 3.8** *Table 11 compares the estimated stored vectors with that for the experimental ones for ClamAV $k = 1,290,617$ subpatterns with length $n = 40$. By using Theorem 3.1 and Conjecture 3.1, we can show that, the 4IGU can store all given vectors. In this case, the total amount of memory is $2^p n \times 4 = 40$ MBytes.* ■

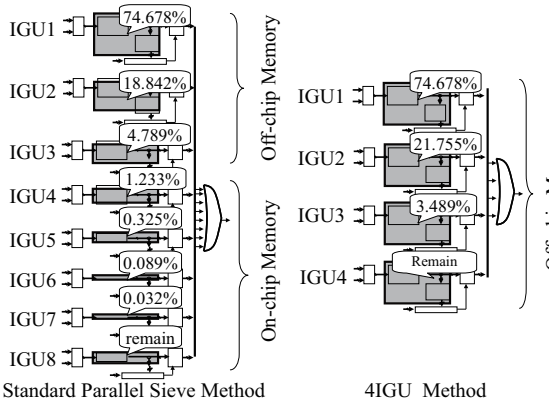


**Fig. 9.** Comparison the standard parallel sieve method with the 4IGU method.

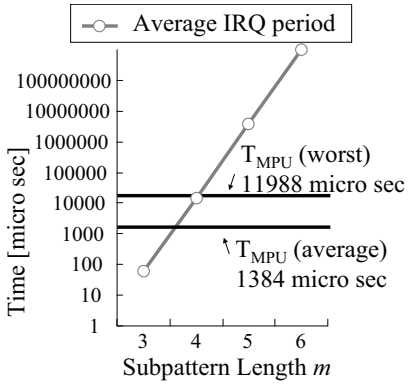


**Fig. 10.** Average and maximum operation times of MPU $T_{MPU}$ and average IRQ period for different values of $m$.

### 3.7 Discussion

Fig. 9 compares the standard parallel sieve method with the 4IGU method, when the number of registered vectors is 1,290,617 and $n = 40$. From theoretical analysis, as for the total amount of memory, the standard parallel sieve method (13.33 MBytes) requires less memory than the 4IGU method (40.00 MBytes). However, the 4IGU method is more suitable than the standard parallel sieve method for a small FPGA implementation;

1. The 4IGU is easy to update the registered vectors: The standard parallel sieve method requires many IGUs. This is inconvenient for the update of registered vectors.
2. The 4IGU uses off-chip memory only: The standard parallel sieve method requires many memories with different sizes. When we use the off-chip memory only, we have a problem since the FPGA has a limited number of pins. Also, if we use the on-chip memory only, the FPGA has a limited amount of on-chip memory. Although the standard parallel sieve method uses both on-chip and off-chip memories, it requires much on-chip resource on the FPGA.

So, the standard parallel sieve method is unsuitable for a small FPGA implementation. In Section 4.2, as for FPGA resources, we will show that the 4IGU method uses less resource than the standard parallel sieve method experimentally.

## 4 Experimental Results

### 4.1 Minimum Subpattern Length $m$

We obtained the minimum $m$ that satisfies the relation (1). To obtain the subpattern detection probability $P(m)$, first, we implemented a cycle-accurate simulator for the 4IGU in C-language. Then, we scanned 2,963 cygwin executable codes. We assume that the 4IGU reads the data from the SRAM running at 400 MHz. Thus, 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 2,963 cygwin executable codes on the MicroBlaze [22] running at 100 MHz using the Perl Compatible Regular Expression library (PCRE) [15]. We used the hardware IRQ handler and the software context switch in the MicroBlaze. Fig. 10 shows the average $T_{MPU}$, the maximum $T_{MPU}$, and the average IRQ period $\frac{T_{bCAMe}}{P(m)}$ for different $m$. Since both the average $T_{MPU}$ and the maximum $T_{MPU}$ are smaller than $\frac{T_{bCAMe}}{P(m)}$, we chose $m = 5$ (40 bits) for implementation.

**Table 12.** Comparison with Other Methods.

| | #Pattern (#Char) | #LC | On-chip Mem [Bytes] | Th [Gbps] | #LC/ #Char | On-chip Mem/ #Char | Off-chip Memories |
|---|---|---|---|---|---|---|---|
| USC RegExp Controller(2006) [1] | 1,316 (16,715) | 41,787 | 768,819.2 | 1.40 | 2.4999 | 45.9957 | SDRAM |
| Cuckoo Hashing (2007) [21] | 4,748 (68,266) | 2,982 | 142,848.0 | 2.20 | 0.0436 | 2.0925 | SRAM |
| Parallel FIMMs (2009) [13] | 65,536 (524,288) | 77,304 | 1,048,576.0 | 1.59 | 19.3150 | 2.0000 | None |
| Standard Parallel Sieve Method (2009) [14] | 497,172 (3,977,376) | 5,268 | 3,500,880.0 | 1.60 | 0.0013 | 0.8801 | Three SRAMs |
| PERG-Rx (2009) [7] | 85,625 (8,645,488) | 42,809 | 387,072.0 | 1.30 | 0.0049 | 0.0447 | SRAM |
| 4IGU method (Proposed Method) | 1,290,617 (42,461,299) | 13,857 | 39,116.8 | 3.20 | 0.0003 | 0.0009 | Four SRAMs |

### 4.2 Implementation Results

We implemented a proposed virus scanning engine shown in Fig. 3 consisting of the 4IGU and the MicroBlaze (MPU) on the Inrevium Corp. PCI Express Evaluation Board (FPGA: Xilinx Inc., Virtex5 VLX50T-GB-R). We used four 16MBytes SRAMs running at 400 MHz for the 4IGU, 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. 11.1. In the implementation, the 4IGU used 6,279 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 13,857 LCs and 20 BRAMs. The 4IGU operated at 508.2 MHz, while the MicroBlaze operated at 100 MHz. Since we used four SRAMs running at 400 MHz, the 4IGU shifts 8 bits per one clock. Thus, the system throughput is $0.4 \times 8 = 3.2$ Gbps.

Table 12 compares various FPGA realizations. As for the throughput (*Th*), our system is 1.45-2.46 times higher. 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; and as for the on-chip memory requirement per a character (*Mem/#Char*), our system is 49.6 times lower than that for the PERG-Rx. This shows that our virus scanning engine is suitable for a small FPGA implementation. Although it requires four SRAMs, the cost for off-chip SRAMs is much lower than that for the high-end FPGA. Table 12 shows that our virus scanning engine is low-cost and high-performance.

## 5 Conclusion and Comments

This paper showed the virus scanning engine using two-stage matching. In the first stage, the 4IGU 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 four SRAMs stored 1,290,617 ClamAV virus patterns, and has the throughput of 3.2 Gbps. Experimental results showed that our virus scanning engine is suitable for a low-cost and high-performance system.

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. [11] 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 possible to use other binary codes. One candidate is Windows executable codes, since many commercial virus scanner scans 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 reduce system throughput, however, this also increases the amount of hardware. Considering practical simulation setup is the one of future work.

## 6 Acknowledgments

This research is supported in part by the grant of Regional Innovation Cluster Program (Global Type, 2nd Stage). Reviewer's comments were useful to improve the paper.

## References

1. Z. K. Baker, H. Jung, and V. K. Prasanna, "Regular expression software deceleration for intrusion detection systems," *FPL2006*, 2006, pp. 28-30.
2. CAST inc., "MD5 IP Core," http://www.cast-inc.com/ip-cores/encryption/md5/
3. ClamAV, http://www.clamav.net/
4. Digi-key Corp., http://www.digikey.com/

5. J. Ditmar, K. Torkelsson, and A. Jantsch, “A dynamically reconfigurable FPGA-based content addressable memory for internet protocol,” *FPL2000*, pp.19-28.
6. Google, “Google Safe Browsing API,” http://code.google.com/intl/ja/apis/safebrowsing/
7. J. T. L. Ho and G. G. F. Lemieux, “PERG-Rx: A hardware pattern-matching engine supporting limited regular expressions,” *FPGA2009*, pp.257-260.
8. P. B. James-Roxby and D.J. Downs, “An efficient content-addressable memory implementation using dynamic routing,” *FCCM2001*, 2001, pp.81-90.
9. W. Jiang, Q. Wang, and V. K. Prasanna, “Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup,” *INFOCOM2008*, 2008, pp.1786-1794.
10. Kaspersky, http://www.kaspersky.com/
11. S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” *ANCS2007*, 2007, pp. 155-164.
12. Z. Kohavi, *Switching and Finite Automata Theory, McGraw-Hill Inc.*, 1979.
13. H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, “A virus scanning engine using a parallel finite-input memory machine and MPUs,” *FPL2009*, 2009, pp.635-639.
14. H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, “The parallel sieve method for a virus scanning engine,” *DSD2009*, 2009, pp.809-816.
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,” *FPL2006*, 2006, pp.785-790.
17. T. Sasao, *Memory-Based Logic Synthesis, Springer.*, 2011.
18. T. Sasao, M. Matsuura and H. Nakahara, “A realization of index generation functions using modules of uniform sizes,” *IWLS’10*, June 18-20, 2010, pp.201-208.
19. T. Sasao and M. Matsuura, “An implementation of an address generator using hash memories,” *DSD2007*, Aug. 27 - 31, 2007, pp.69-76.
20. L. Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” *ISCA2005*, 2005, pp.112-122.
21. T. N. Thinh, S. Kittitornkun, and S. Tomiyama, “Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS,” *ICFPT2007*, 2007, pp.121-128.
22. Xilinx inc, “MicroBlaze”, http://www.xilinx.com/
23. F. Yu, R. H. Katz, and T. V. Lakshman, “Gigabit rate packet pattern matching using TCAM,” *ICNP2004*, 2004, pp.174-183.