社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

並列ふるい法とMPUを用いたウイルス検出エンジンについて

中原　啓貴† 　笹尾　勤† 　松浦　宗寛† 　川村　嘉郁††

† 九州工業大学 情報工学部 〒820–8502 福岡県飯塚市大字川津 680–4
†† ルネサステクノロジ 〒100–0004 東京都千代田区大手町 2-6-2

あらまし　本論文では並列ふるい法とMPUを用いたウイルス検出エンジンについて述べる. アンチ・ウイルスソフトClamAVと侵入検知ソフトSNORTのパターンの違いを述べ, 侵入検知システムとは異なるウイルス検出エンジンの構成について述べる. ウイルス検出エンジンはMPUとFIMMで構成し, 二段階マッチングを行ってウイルスを検出する. 第一段階では, 並列ハードウェアフィルタを用いて高速に部分マッチングを行い, 第二段階では, MPUを用いてウイルスパターンを厳密にマッチングする. 大量のウイルスパターンを効率よく格納するため, 並列ふるい法を提案する. 外付けSRAMとSDRAMとFPGA1個で514287個のClamAVのウイルスパターンを全て格納した. 単位面積で正規化したスループットにおいて, 提案手法は従来手法よりも1.41倍-31.36倍優れている.

A Virus Scanning Engine Using a Parallel Sieve Method and the MPU

Hiroki NAKAHARA†, Tsutomu SASAO†, Munehiro MATSUURA†, and Yoshifumi KAWAMURA††

† Department of Computer Science and Electronics, Kyushu Institute of Technology
680–4, Kawazu, Iizuka, Fukuoka, 820–8502 Japan
†† Renesas Technology Corp., Tokyo, 100-0004, Japan

Abstract　In this paper, we show a new architecture for the virus scanning machine, which is different from that of the intrusion detection machine. The proposed method uses the two-stage matching, which is area-throughput efficient. That is, in the first stage, the hardware filter quickly scans to find possible matches, and in the second stage, the MPU scans the real match by a brute-force method. To make the hardware filter simply, we will introduce finite-input memory machine (FIMM). To reduce the memory size in the FIMM, we will introduce the parallel sieve method. The proposed method uses memory, so the power consumption is lower than the TCAM-based method. The system is implemented on the Stratix III FPGA and three off chip SRAMs, where all ClamAV virus patterns (514287) are stored. Comparison with existing methods, as for the area-throughput ratio, our method is 1.41-31.36 times more efficient.

1. はじめに

コンピュータウイルス, ワーム, スパイウェア, スパムメール等悪意を持ったソフトウェア (Malicious software) などをマルウェア (Malware) という. 近年のインターネットの普及に伴い, 利用者はネットワークを経由してプログラムを入手することが可能なため, コンピュータがマルウェアに侵される危険性が増している. マルウェアの感染により, ボットウイルス, バックドア, キーロガーが仕掛けられ, IDやパスワードの搾取, 情報の盗難, 不正な遠隔操作が行われており社会問題となっている. マルウェアを駆除, 隔離する簡単な方法は個々のコンピュータにウイルス検出ソフトを導入することである. しかしながら, ウイルス検出をソフトウェアで実現した場合, その性能は高々数10 Mbps [16] であり数 Gbps に達しつつある今日のネットワークには対応できない. マルウェアは日々複雑化, 多様化しており, 今後コンピュータのパフォーマンス低下や転送速度のボトルネックとなるのは明白である. 近年, ゲートウェイやホストセンタにネットワーク型ウイルス検出装置を設置し, マルウェアの検出・駆除が行われている [23]. 図 1 にネットワーク型ウイルス検出装置を示す. ウイルス検出装置は, まず, 転送されたパケットを PHY/MAC ポートで受信し, PacketReciver で元のデータに復元する. その際, 圧縮されたデータは展開される. そして, ウイルス検出エンジンでウイルス (マルウェア) 検出を行う. PacketSender は検出済みのパケットを PHY/MAC ポートを経由してイントラネットに転送する. ウイルス検出装置において, ウイルス検出エンジン以外は既存技術を流用できるので, 本論文ではウイルス検出エンジンのみ述べる. ゲートウェイ設置型のウイルス検出装置 [23] はスループットが高々1.2 Gbps であり, 消費電力も 450 W と大きく価格も$10000 と高価である.

ウイルス検出エンジンは以下の項目が求められる.

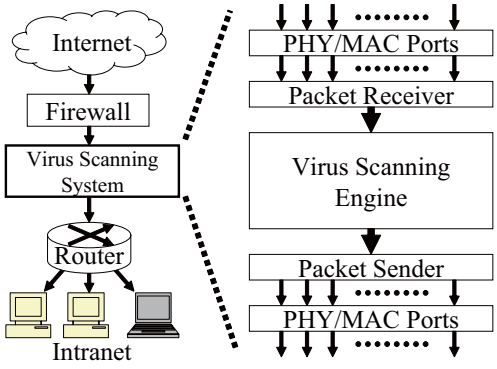

図 1　ウイルス検出システム.

a）高スループット
少なくとも転送速度が数 G bps 以上の性能が必要.

b）低消費電力
ウイルス検出機器に TCAM を用いた手法 [3], [25] が提案されているが TCAM は表 1 に示すように, 消費電力が高く, ビット単位の面積が大きい. SRAM 等の低消費電力メモリを用いるのが望ましい.

c）容易に更新可能
現在, 最も更新頻度が早いウイルス検出ソフトでは, ウイルスパターンを 1 時間に 1 度更新している [11]. よって, ウイルス検出機器を停止させずに, 高速に更新できる手法が求められる. FPGA に直接検出回路を実装する手法 [6] もあるが, FPGA の配置配線には数時間〜数日かかり, ウイルスパターンの更新間隔に間に合わない. よって, メモリのみを書換える手法を用いる.

d）コストパフォーマンス
ウイルス検出機器を実装する手法は多く提案されている. しかし, 既存の手法はウイルスのパターン当りの必要メモリ量が多く, 現時点のウイルスパターン (約 50 万個) を全て実装すると, ハイエンド FPGA を数 10 個必要とするため, 非常に高価となる.

表 1　TCAM と SRAM の比較 (18Mbit チップ) [10]

	TCAM	SRAM
最大動作周波数 [MHz]	266	400
消費電力 [W]	12-15	≈ 0.1
ビット当りのトランジスタ数	16	6

本論文では二段階マッチングを行い, ウイルスを検出する. 第一段階では有限入力メモリ機械を用いてウイルスである可能性のあるパターンを高速に検出する. 第二段階では MPU を用いてウイルスパターンを厳密に検出する. 有限入力メモリ機械は状態遷移が制限されたオートマトンを実現する. 通常のオートマトンと比較して, 受理できるパターンが制約されるが, 単純な回路で実現できる. しかし, 有限入力メモリ機械をメモリで直接実現するとメモリ量が大きくなり実用的でない. よって, 本論文ではインデックス生成回路を用いて有限入力メモリ機械を実現する. 更に, メモリ量を削減するためインデックス生成回路を複数用いた並列ふるい法を用いる.

第 2 章ではウイルス検出の説明を行い, 第 3 章ではインデックス生成関数回路を用いた有限入力メモリ機械の実現法について述べ, 第 4 章では並列ふるい法について述べ, 第 5 章ではウイルス検出エンジンを実装した結果を述べ, 第 6 章で本論文のまとめを行う.

本論文は, [14] を日本語に翻訳したものである.

2. ウイルス検出

2.1 ウイルス検出問題

実行プログラムやデータ内に埋込まれた不正な動作を引き起こすコードを検出することを**ウイルス検出**という. 検出対象の実行プログラムやデータを**テキスト**という. 不正な動作を引き起こすコードはテキスト内に埋込まれることが多い. これらは特定のバイトコードで記述されており, **シグネチャ(パターン)** と呼ぶ. ウイルス検出問題は, 可変長のテキストの中から特定のパターンを探し出す**文字列照合** (パターンマッチング) 問題に帰着できる.

2.2 シグネチャで用いられている制限された正規表現

シグネチャは**文字**と特殊な文字である**メタ文字**から成る**制限された正規表現**で記述される. 1 文字は 2 桁の 16 進数で表現される. 以降, シグネチャのことを**パターン**と呼ぶ. 本論文ではパターン数を k, パターン長を c で表す. ソースコードが公開されている ClamAV [7] で用いられているメタ文字を表 2 に示す. 表 3 にシグネチャの例を示す.

表 2　ClamAV のシグネチャで用いられる正規表現のメタ文字

表記	意味	例
??	任意の 1 文字	
A*	0 文字以上の繰返し (連接閉包)	AA*={A,AA,AAA,···}
()	連結の変更	
A|B	論理和	A|B={A,B}
$\{n,m\}$	n 文字以上 m 文字以下	A{2,3}={AA,AAA}

表 3　シグネチャの例

ウイルス名	パターン
Trojan.Bat.DelY-3	64656c74726565**{**-1**}**2f**(**59|79**)**20633a5c2a2e2a
Trojan.Bat.DelY	44454c54524545202f**(**59|79**)**20633a5c2a2e2a
Trojan.Bat.MkDir.B	406d64202572616e646f6d25**????**676f746f20486f6f
W32.Gop	736d74702e796561682e6e65*****2d20474554204f494351
Worm.Bagle-67	6840484048688d5b0090eb01ebeb0a5ba9ed46

表 4　ClamAV と SNORT のパターンの比較.

	ClamAV	Snort
パターン数	514287	3533
平均パターン長	32.9	193.7
平均メタ文字数	0.081	46.7

2.3 2 段階マッチングを用いたウイルス検出

表 4 に 2009 年 2 月の時点における ClamAV (v.0.94.2) と SNORT (v.2.8.3.2) [21] のパターンを比較した結果を示す. 表 4 に示すように, ClamAV のパターンは SNORT のパターンよりもメタ文字数が少ないため, 正規表現としては簡単である. しかし, ClamAV のパターン数は SNORT のパターン数よりも遥かに多い. よって, ウイルス検出ではパターンを効率よくメモリに格納する手法が求められる.

Aho-Corasick 法 (AC 法) はパターン検索法の代表的な手法である [1]. AC 法はパターンを AC オートマトンで表現し, これを用いて文字列検索を行う. c バイトのシグネチャを AC オートマトンで実現する場合, 最悪 $O(256^c)$ のメモリが必要であるため, ウイルスのシグネチャの平均長 $\bar{c} = 32.9$ のオートマトンを直接実現することは現実的でない. よって, ClamAV では高速かつ省メモリを達成するため, パターンマッチングを 2 段階に分けて行う. 第一段階では, シグネチャの先頭 3 文字に関してオートマトンを用いてウイルスの可能性のあるパターンを検出する. 第二段階では, AC オートマトンでマッチした部分のみ, ハッシュ法を用いてシグネチャと一致するか厳密に調べる. この方法では, メモリに全パターンを格納する場合と比較して, AC オートマトンのメモリ量を削減できる. 2 段階マッチングの例を以下に示す.

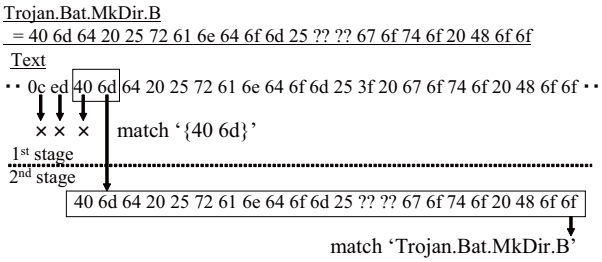

図 2　2 段階マッチングの例.

［例 2.1］ 図 2 は表 3 に示したパターン Trojan.Bat.MkDir.B を 2 段階マッチングで検出する例を示している. パターンの一部 $\{406D\}$ が検出されるので, パターンと完全に一致するかハッシュ法を用いて正規表現マッチを行い Trojan ウイルスを検出する. (例終)

2.4　ウイルス検出処理のプロファイル

ClamAV で行われている 2 段階マッチングのプロファイル解析を行った. ClamAV と同じく, シグネチャの先頭 3 文字を AC 法でマッチングを行い, AC 法でマッチした部分をフリーの正規表現ライブラリ PCRE [15] を用いて正規表現マッチングを行った. シグネチャ数は 512 とし, 10 個の実行コードに対してマッチング時間のプロファイル解析を行った. その結果, AC 法の処理が 83%であり, 正規表現マッチの処理が 17%であった. よって性能向上を達成するには以下の 2 点を考慮すべきであることがわかる.

1.　(AC オートマトンの処理速度, 第一段階の向上) AC オートマトンをハードウェア化する. ウイルス検出では複数のテキストを個別にマッチングできるので並列処理可能である.

2.　(AC オートマトンのマッチ確率, 第二段階の向上) AC オートマトンに格納するパターン長 c を増やせばマッチ確率は低下する. 一方, パターン長 c に対して AC 法では $O(256^c)$ のメモリを必要とする.

予備実験から, 第一段階で 3 文字のパターンに対してマッチが発生するテキストの間隔 (文字数) は約 100 文字であった. 第二段階では FPGA 上の組込みプロセッサを用いるが, PC の MPU と比較すると性能が劣るため, 第一段階でのパターン長 c を 4 文字とし, マッチが発生するテキストの間隔を伸ばし, FPGA 上の組込みプロセッサでも処理できるようにする. ウイルスパターン数 k は現時点で 51 万個以上あるから, 第一段階をコンパクトなメモリで実現する手法が求められる.

3.　インデックス生成回路を用いた有限入力メモリ機械の実現

二段階マッチングを用いたウイルス検出問題では, 第一段階でのハードウェアのメモリ量を削減する手法が要求される. 提案手法は, 第一段階を状態遷移を制限したオートマトンで実現し, 第二段階を FPGA 上の組込みプロセッサで実現する. まず, 状態遷移を制限したオートマトンである有限入力メモリ機械 (FIMM) について述べ, 次に, FIMM と MPU を併用したウイルス検出機器について述べる.

3.1　有限入力メモリ機械 (FIMM)

AC オートマトンは状態遷移が複雑であり, 回路も複雑になる. AC オートマトンのビット分割法 [22] が提案されているが, パターン数 k が多いウイルス検出では回路が複雑になってしまう. そこで, 状態遷移が比較的簡単な**有限入力メモリ機械** (FIMM: Finite-Input Memory Machine) を考える [13]. 長さ c のパターンを k 個格納する FIMM を実現する回路を図 3 に示す. 図 3 に示すように, この回路では常に入力がシフトされ

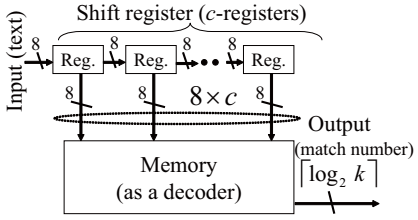

図 3　FIMM を模擬する回路.

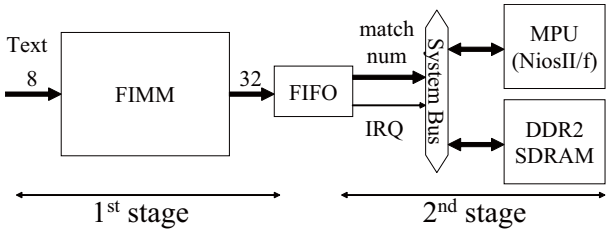

図 4　ウイルス検出エンジン.

フィードバック入力はないため, シフトレジスタは過去の c 個の入力のみ記憶している. また, シフトレジスタ実現のため, 入次数と出次数はそれぞれ 2^8 個に制限されている. メモリは各状態に対する出力関数を実現する. FIMM は回路構造に制限があるため, 連接閉包'$*$' などを受理できない. FIMM は表現能力を制限したオートマトンなので, 簡単な回路で実現できる. FIMM の出力関数を実現するために必要なメモリ量(注1)は

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

である.

3.2　MPU と併用したウイルス検出機器

図 4 にウイルス検出エンジンを示す. FIMM を用いてテキスト中から c 文字で構成されたパターンの一部を検出する. 検出されたパターンのインデックスは FIFO に格納する. FIFO はパターンのインデックスと割込み信号 (IRQ) を MPU に送る. MPU はウイルスパターンか否かを厳密に判別する.

4.　並列ふるい法を用いた FIMM の実現

二段階マッチングを用いたウイルス検出エンジンでは, 第一段階のマッチングは FIMM で行なう. FIMM の出力関数を直接メモリで実現した場合, 表 4 より, $k = 514287$, $c = 4$ であるから, 式 (1) より必要メモリ量は

$$M_{FIMM} = 2^{8\times 4}\lceil log_2(k+1)\rceil \simeq 2^{37} \tag{2}$$

ビットとなり, 現実的でない. 本論文では, FIMM のメモリ量を削減する並列ふるい法を提案する. まず, FIMM の出力関数の数学的モデルであるインデックス生成関数について述べる. 次に, インデックス生成関数を小さなメモリで実現するインデックス生成回路について述べる. 最後に, 複数のインデックス生成回路を用いてメモリ量を更に削減する並列ふるい法について述べる.

4.1　インデックス生成関数

［定義 4.1］ k 個の異なる登録ベクトルに対して 1 から k までの固有のインデックスを対応させた表を, **インデックス表** [20] という.

ウイルス検出問題では, 登録ベクトルはウイルスパターンに対応し, インデックスは各ウイルスパターンに割当てた固有の

(注1)：状態遷移を記憶するシフトレジスタのビット数は出力関数を記憶するメモリのビット数よりも遥かに小さいので無視できる. よって本論文では**メモリ量**とは FIMM の出力関数を記憶するメモリのビット数を表すものとする.

表 5　インデックス生成関数の例.

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	0	1	1	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7
otherwise						0

表 6　$f(X_1, X_2)$ の分解表.

	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 0 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 4 0 0 6	
111	0 0 7 0 0 0 0 0	
x_6, x_5, x_4		

表 7　$\hat{f}(Y_1, X_2)$ の分解表.

	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 4 0 6 0 0 0	
111	0 0 0 0 0 7 0 0	
x_6, x_5, x_4		

番号に対応する. ただし, 第一段階ではウイルスパターンの最初の c 文字のみを検査しているので厳密にはこの定義は成立しないこともある.

図 3 に示した FIMM の出力関数はインデックス表で表せる. インデックス表は Content Addressable Memory(CAM) [12] で直接実現できるが, FPGA 上に CAM 機能を実現するには大量の論理素子が必要であり [8], [9], 消費電力も大きい. よって, 本論文ではメモリを用いてインデックス生成関数を実現する.

[定義 4.2]　$B = \{0, 1\}$ とする. 関数 $f(\vec{X}) : B^n \rightarrow \{0, 1, \ldots, k\}$ において k 個の異なる登録ベクトル $\vec{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$ に対して, $f(\vec{a}_i) = i$ $(i = 1, 2, \ldots, k)$ が成立し, それ以外の $(2^n - k)$ 個の入力ベクトルに対しては, $f = 0$ が成立するとき, $f(\vec{X})$ を**重み k のインデックス生成関数**という. インデックス生成関数は, k 個の異なる 2 値ベクトルに対して, 1 から k までのアドレス (インデックス) を生成する.

[例 4.2]　表 5 に $n = 6, k = 7$ のインデックス生成関数 f の例を示す. (例終)

4.2　インデックス生成回路

$f(X_1, X_2)$ を重み k のインデックス生成関数とする. n 入力インデックス生成関数を直接メモリで実現する場合, メモリ量は $2^n \lceil \log_2(k+1) \rceil$ となり, n の値が 32 程度であるウイルス検出では実用的ではない.

$X_1 = (x_1, x_2, ..., x_p)$ を $Y_1 = (y_1, y_2, ..., y_p)$ に置き換えた関数を $\hat{f}(Y_1, X_2)$ とする. ただし, $y_i = x_i \oplus x_j$, $x_j \in \{X_2\}$, $p \geq \lceil \log_2(k+1) \rceil$ である.

[例 4.3]　例 4.2 に示したインデックス生成関数の**分解表**を表 6 に示す. 列ラベルは $X_1 = (x_1, x_2, x_3)$ を表し, 行ラベルは $X_2 = (x_4, x_5, x_6)$ を表す. 表の値は関数値を表す. $Y_1 = (x_1 \oplus x_6, x_2 \oplus x_5, x_3 \oplus x_4)$ と変数変換を行った場合の $\hat{f}(Y_1, X_2)$ の分解表を表 7 に示す. 列ラベルは Y_1 を示し, 行ラベルは X_2 を示す. f の分解表では非零要素を 2 つ以上持つ列が 3 つであるのに対し, $\hat{f}$ では非零要素を 2 つ以上持つ列が 1 つに減少している. (例終)

表 8　$\hat{f}_1(Y_1, X_2)$ の分解表.

	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		

表 9　$\hat{f}_1(Y_1)$ の主メモリ.

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

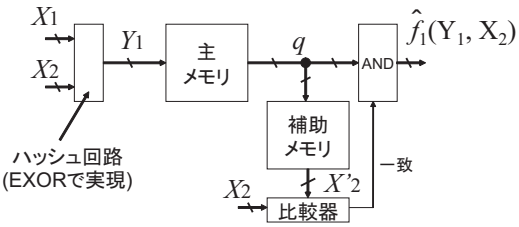

図 5　インデックス生成回路 (IGU).

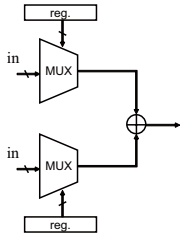

図 6　プログラマブル・ハッシュ回路.

表 7 において, 列 010 の要素 4 を別の回路で実現すれば, この要素は $\hat{f}$ から削除できる. $\hat{f}$ から要素 4 を削除した関数を $\hat{f}_1$ とする. 表 8 に $\hat{f}_1$ の分解表を示す. $\hat{f}_1$ の各列には非零要素が高々1 つしか存在しない. よって $\hat{f}_1$ は Y_1 のみを入力とした**主メモリ**で実現できる. 表 9 に $\hat{f}_1$ の主メモリを示す. 主メモリは 2^p の集合を 2^p の集合へ写す写像を表現できる. 主メモリは $\hat{f}_1$ の出力値を与えるが, この値は必ずしも f の値と等しいとは限らない. 主メモリの出力が非零の場合でも, X_2 の値を調べなければ, $\hat{f}_1$ の値が正しい値か否かわからない. $\hat{f}_1$ の値が零の場合は, f の値も零である. そこで**補助メモリ**を付加し, 補助メモリに主メモリに登録したベクトルに対応する X_2 を登録する. そして入力 X_2 と比較を行い, 主メモリの値の正誤判定を比較器で行う. 図 5 に**インデックス生成回路 (IGU: Index Generation Unit)** を示す. 図 6 に示す**プログラマブル・ハッシュ回路**を用いて入力 (X_1, X_2) からハッシュ入力 Y_1 を生成する. ハッシュ関数の生成法は文献 [18], [19] で述べられている. ここで, $|X_1| = |Y_1|$ である. Y_1 を用いて主メモリを参照し出力 q を得る. q を用いて補助メモリを参照し出力 X_2' を得る. X_2' と入力 X_2 を比較し, 一致すれば q を出力する. 不一致の場合は, 0 ベクトルを出力する.

4.3　インデックス生成回路で実現可能な登録ベクトルの割合

主メモリの入力数 (アドレス空間) に対する登録ベクトルの格納率が知られている.

[定理 4.1]　[19] 重み k のインデックス生成関数において, 主メモリで実現される登録ベクトルの割合は

$$\delta \simeq 1 - \frac{1}{2}(\frac{k}{2^p}) + \frac{1}{6}(\frac{k}{2^p})^2 \qquad (3)$$

である. ただし, 主メモリの入力数を p とすると $k \leqq 2^p$ であり, インデックス生成関数の分解表において, 非零要素は一様に分布しているものとする.

[例 4.4]　$\frac{k}{2^p} = \frac{1}{1}$ とすると $\delta = \frac{2}{3} \simeq 0.666$ となり, 主メモリには登録ベクトルを約 66.6%格納できる. ただし, 登録ベクトルを一様に分布させるためにプログラマブル・ハッシュ回路が必要になる. (例終)

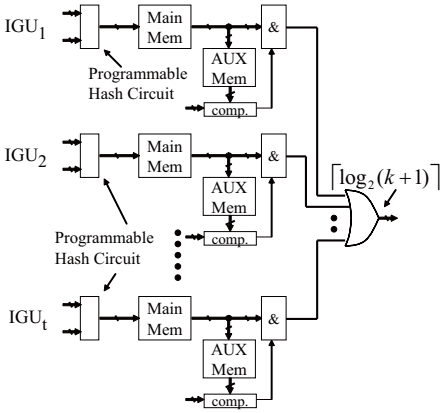

図 7　並列ふるい法.

主メモリの入力数を増やしてアドレス空間を十分に広くした場合, 登録ベクトルをほぼ全て格納可能であることが経験的に知られている.

[推論 4.1]　[17] 重み k のインデックス生成関数を実現するために必要な主メモリの入力数 p は高々

$$p = 2\lceil log_2(k+1) \rceil - 1 \qquad (4)$$

である. ただし, インデックス生成関数の分解表において, 非零要素は一様に分布しているものとする.

4.4　並列ふるい法

定理 4.1 と推論 4.1 を用いると, 入力数 p の主メモリに格納可能な登録ベクトル数 k を推定可能である. 登録ベクトルを 4 文字づつメモリに直接格納する場合, 式 (2) に示すように, 大量のメモリが必要である. 全てのベクトルを図 5 に示すインデックス生成回路 (IGU) に格納する場合, 推論 4.1 より, 主メモリの入力数は $p = 2\lceil log_2(514287+1) \rceil - 1 = 37$ であり, 必要なメモリ量は 2^{37} ビットとなる. 従って, この手法も実用的ではない. 定理 4.1 より, $\frac{514287}{2^p} \simeq \frac{1}{1}$ の場合, $p = 19$ であり, $\delta \simeq 0.66$ となる. つまり, 入力数 19 の主メモリには 51 万個の登録ベクトルのうちの約 66%を格納できる. 入力数が 18〜22 程度の SRAM は使用可能である. 従って, 図 7 に示すように, 登録ベクトルの一部を入力数が大きい主メモリに格納し, 残りのベクトルを別の主メモリに格納することを繰り返せば, 登録ベクトルをほぼ全て複数の主メモリに格納できる. 最後に残った登録ベクトルが僅かであれば, 主メモリに全て格納したとしても, 主メモリ入力数はそれほど増加しない.

[例 4.5]　残りのベクトル数を 200 個とする. このとき, 推論 4.1 より高々15 入力の主メモリを用意すれば全てのベクトルを格納できる.　(例終)

[定義 4.3]　図 7 に示すように, 複数の IGU を用いて登録ベクトルを分散格納し, 全ての登録ベクトルを実現する手法を**並列ふるい法**と呼ぶ. IGU_{i+1} は IGU_1, IGU_2, ..., IGU_i に格納しない登録ベクトルを実現する. 各 IGU の出力を OR ゲートで接続し, インデックス生成関数の出力を決定する.

4.5　並列ふるい法における主メモリの入力数

IGU_1 の主メモリの入力数を p_1 とし, 登録ベクトル数を k_1 とし, $p_1 = \lceil log_2(k_1+1) \rceil$ とする. IGU_1 の主メモリに格納できるベクトルの割合 δ_1 は定理 4.1 より計算できる. また, IGU_1 の主メモリのメモリ量は $2^{p_1}\lceil log_2(p_1+1) \rceil$ ビットである. この時, 格納できない登録ベクトルの割合は $\gamma_1 = 1 - \delta_1$ である. 残りの $k_1\gamma_1$ 個のベクトルのうち, δ_2 を格納する p_2 入力の IGU_2 の主メモリには $k_1\gamma_1\delta_2$ 個のベクトルを格納できる. 従って, p_1 入力と p_2 入力である主メモリを 2 個用いることで $k_1 \times \delta_1 + k_1 \times \gamma_1\delta_2$ 個の登録ベクトルを格納できる.

ALTERA 社の FPGA の組込みメモリの大きさは 9 キロビットなので, 推論 4.1 より, $k_{t+1} = 127$ 以下になるまでは, 登録ベクトルを t 個の IGU に格納し, 残りのベクトルを FPGA の組込みメモリで実現した IGU_{t+1} に格納する.

ClamAV のウイルスパターンに対して, 定理 4.1 を $r_t \leqq 127$ になるまで t 回繰返し適用し, $t+1$ 番目の IGU_{t+1} には推論 4.1 を適用し, 登録されるベクトル数と IGU の台数の推定値を求めた. また, ClamAV のウイルスパターンを実際に格納して得られた実験値も求めた. 表 10 に推定値と実験値を示す. 表 10 において, p_j は IGU_j の主メモリの入力数を表し, $\hat{k}_j$ は IGU_j に格納した登録ベクトル数を表し, r_j は残りの登録ベクトル数を表す. 並列ふるい法は 4 文字のパターンを格納するので, ユニークなベクトル数 $\hat{k}$ は 497172 個である. 表 10 より, 8 台の IGU で登録ベクトルを全て格納できていることが確認できる. また, 実験結果と推定結果では IGU の台数が一致している.

表 10　登録ベクトルの推定値と実験値.

		推定値			実験値		
	j	p_j	$\hat{k}_j$	r_j	p_j	$\hat{k}_j$	r_j
IGU_1	1	19	335955	161217	19	321659	175513
IGU_2	2	18	121805	39412	18	128398	47115
IGU_3	3	16	29936	9476	16	33791	13324
IGU_4	4	14	7263	2213	14	9267	4057
IGU_5	5	12	1721	492	12	2641	1416
IGU_6	6	9	330	162	11	1055	361
IGU_7	7	8	120	42	9	277	84
IGU_8	8	11	42	0	9	84	0

表 11　インデックス生成回路に用いたメモリ.

			Main Memory		AUX Memory	
	j	$\hat{k}_j$	INOUT	Memory	INOUT	Memory
IGU_1	1	321659	i=19,o=19	SRAM(注2)	i=19,o=13	SRAM
IGU_2	2	128398	i=18,o=18	SRAM(注2)	i=18,o=14	SRAM
IGU_3	3	33791	i=16,o=16	SRAM(注2)	i=16,o=16	8 M144k
IGU_4	4	9267	i=14,o=14	2 M144k	i=14,o=18	3 M144k
IGU_5	5	2641	i=12,o=12	6 M9k	i=12,o=20	10 M9k
IGU_6	6	1055	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

5.　実 装 結 果

5.1　提案ウイルス検出エンジンを実装した結果

表 10 で得られた値を元に図 4 に示したウイルス検出エンジンを Altera 社の FPGA に実装した. 実装に用いた評価ボードは Terasic 社の DE3 ボードであり, 使用 FPGA は EP3SL340H1152C3NE (ALUT: 270400 個, M9k: 1040 個, M144k: 48 個) である. また, 主メモリを実現するため外付け SRAM ボードを 3 枚使用した. 外付け SRAM は入力 21 ビット, 出力 72 ビット (パリティ8 ビット) である. 表 11 にインデックス生成回路に用いたメモリを示す. 表 11 において, IGU_j は j 番目の IGU を表し, $\hat{k}_j$ は IGU_j 格納した登録ベクトル数を表し, *INOUT* は各メモリの入力数 (i) と出力数 (o) を表し, *Memory* は使用したメモリの種類を表す. Quartus II (v.8.0) による実装では, ALUT は 3790 個, M9k は 31 個, M144k は 13 個であった. また組込みプロセッサは NiosII/f を用い, 必要な ALUT は 1312 個であった. ClamAV の全パターンを格納するために DDR2-SODIMM を取り付けた. 並列ふるい法では 4 文字のユニークなパターンを 497172 個格納している.

(注2)：複数の主メモリをハッシュ入力を共通にして, 72 出力の外付けの SRAM に纏めている. ただし, 各主メモリのハッシュ入力を個別に指定する場合と比較して格納できる登録ベクトル数は若干少ない.

表 12　他の手法との比較.

Method	Th [Gbps]	# of Patterns	MUC [Bytes/char]	Th/MUC	Comment
AC 法 [24]	6.0	1533	2896.2	0.0020	ASIC 実装
Aldwari et.al [2]	14.0	1542	126.0	0.1111	FPGA+SRAM
Bitmap compressed Aho-Corasick [24]	8.0	1533	154.0	0.0519	ASIC 実装
Path compressed Aho-Corasick [24]	8.0	1533	60.0	0.1333	ASIC 実装
Alicherry et.al [3]	20.0	100	48.0	0.4166	FPGA+TCAM
Yu et.al [25]	2.0	1768	3.0	0.6666	FPGA+TCAM+MPU
USC RegExpController [5]	1.4	1316	46.0	0.0304	FPGA+MPU
Hardware Bloom Filter [4]	0.5	35475	1.5	0.3333	FPGA+SDRAM
提案手法	1.6	497172	1.7	0.9417	FPGA+MPU+SRAM

FPGA 内のインデックス生成回路の動作周波数は 371.0[MHz] であったが, 外付けの SRAM ボードの動作周波数の上限により 200[MHz] で動作させた. 1 文字 (8 ビット) を 1 クロックで評価できるので, スループットは $Th = 0.200 \times 8 = 1.6$ [Gbps] である. 1 文字当りの使用メモリ量を**メモリ利用係数 (MUC: Memory Utilization Coefficient)** とする. 設計したウイルス検出エンジンの使用メモリ量は表 10 より, 3500880 バイトであり, 4 文字の登録ベクトルを 497172 個格納するので, $MUC = \frac{3500880}{497172 \times 4} = 1.7$ [Bytes/Char] である.

5.2　他の手法との比較

提案手法とメモリを用いた正規表現マッチング手法を比較した結果を表 12 に示す. 引用した結果は実装デバイスによってパフォーマンスや使用メモリ量が異なる. よって, パターンマッチング回路 1 台のスループットと格納したパターンのサイズを考慮したコストパフォーマンスで比較した. 表 12 において, パターンマッチング回路 1 台当りのスループットを Th[Gbps], パターンを格納するのに必要なメモリ量をメモリ利用率とし, MUC[Bytes/Char] で表す. パターンマッチング回路 1 台当りのコストパフォーマンスをメモリ利用率で正規化したスループットとし, Th_{MUC} とする. 従って, $Th_{MUC} = \frac{Th}{MUC}$ である.

表 12 より, 提案手法は 497172 個のウイルスパターンを全て実装できた. また, AC 法と比較して Th_{MUC} が 470.5 倍優れており, 他の手法と比較して, 1.41-31.36 倍優れている. 提案手法が Th_{MUC} で優れている理由は MUC が特に小さいからである. すなわち, インデックス生成回路がウイルスパターンを効率よく格納できたからである. Th_{MUC} の値が本手法に次いで優れているのは Yu [25] の手法である. しかし, Yu の手法は TCAM を使っており, Th_{MUC} は表 1 に示した消費電力とビット単位のトランジスタ数を考慮していない. これらを考慮すれば, 提案手法は消費電力・価格でさらに優れている.

6. ま と め

本論文では MPU と FIMM を用いたウイルス検出エンジンについて述べた. 提案エンジンは二段階マッチングを行ってウイルスを検出する. 第一段階では, 並列ハードウェアフィルタを用いてウイルスの可能性のあるパターンを高速に検出し, 第二段階では, MPU を用いてウイルスパターンを厳密にマッチする. 少量のメモリを用いて FIMM を実現するため, インデックス生成回路を複数用いた並列ふるい法を提案した. 外付け SRAM3 個と SDRAM1 個と FPGA1 個で ClamAV のウイルスパターン 514287 個を全て格納した. 単位面積で正規化したスループットにおいて, 提案手法は従来手法よりも 1.41 倍-31.36 倍優れている.

7. 謝　　辞

本研究は, 一部, 日本学術振興会・科学研究費補助金, および, 文部科学省・知的クラスター創成事業 (第二期) の補助金による. 日立情報制御ソリューションズ梶原久志氏には有益な助言を頂いた.

文　　献

[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] P. B. James-Roxby and D.J. Downs, "An efficient content-addressable memory implementation using dynamic routing," *FCCM'01 2001*, pp.81- 90, 2001.
[10] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based paralel multi-pipeline architecture for terabit IP lookup," *27-th IEEE Int. Conf. on Computer Communications (INFOCOM2008)*, pp.1786-1794, 2008.
[11] Kaspersky, http://www.kaspersky.com/
[12] T. Kohonen, *Content-Addressable Memories*, Springer Series in Information Sciences, Vol. 1, Springer Berlin Heidelberg 1987.
[13] R. McNaughton and S. Papert. "Counter-FreeAutomata," *MIT Press*, 1971.
[14] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "The Parallel sieve method for a virus scanning engine," *12th EUROMICRO Conference on Digital System Design (DSD2009)*, pp. 809-816, Patras, Greece, Aug. 27-29, 2009.
[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, "On the number of variables to represent sparse logic functions," *ICCAD-2008*, San Jose, California, USA, Nov.10-13, 2008, pp. 45-51.
[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, "A Design method of address generators using hash memories," *IWLS-2006*, Vail, Colorado, U.S.A, June 7-9, 2006, pp.102-109.
[20] T. Sasao, "Design methods for multiple-valued input address generators," *ISMVL-2006*(invited paper), Singapore, May 17-20, 2006.
[21] SNORT, http://www.snort.org/
[22] 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.
[23] TrendMicro, *Network Virus Wall Enforcer*, http://us.trendmicro.com/.
[24] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrustion detection," *23-th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, pp.333-340, 2004.
[25] 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.