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・ウイルスソフトウェアである ClamAV と侵入検知ソフトウェアである SNORT のパターンの違いを述べ, 侵入検知システム用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 ボードと FPGA1 個で ClamAV のウイルスパターン 514287 個全て格納した．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 Seive 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

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．マルウェアに感染することにより, 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] [18] 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, ネットワークの入口 (ゲートウェイ) やホストセンタにウイルス検出専用の装置を設置し, 個々の PC にデータを転送する前にマルウェアを検出・駆除するサービスや専用のハードウェアが注目されている [24]．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インターネットとイントラネット間に設置され, 転送されてくるデータからマルウェアの検出を行う．まず, 転送されたパケットを PHY/MAC ポートで受信し, PacketReciver で組立てて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 は検出済みのデータを 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のルータ等に用いられている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性能向上についてのみ述べる．ゲートウェイ設置型のウイルス検出装置 [24] 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．

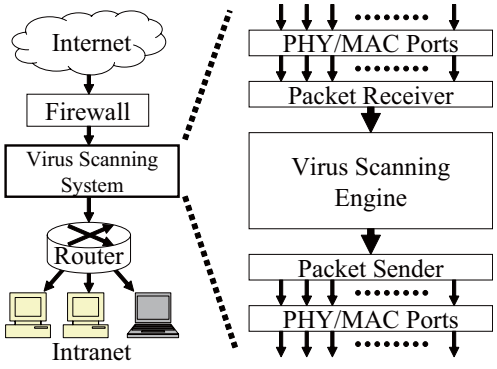

図 1　ウイルス検出システム.

a）高スループット (High throughput)

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 (Low power)

ウイルス検出機器に TCAM を用いた手法 [3], [26] 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 (Easy and quick update)

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 [12]. 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.

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 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}

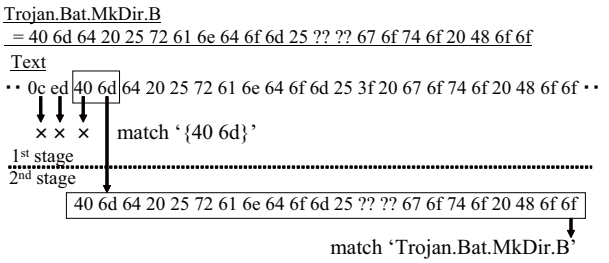

図 2　2 段階マッチングの例.

［例 2.1］ 表 3 にシグネチャの例を示す. (例終)

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) のパターンを比較する. 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(ver0.94.2) 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. 第一段階では, 3 文字しかチェックしないので, AC オートマトンのマッチ確率は上昇する. しかし, 読戻しを必要とするハッシュ法を用いたマッチングだけを用いた場合と比較して, 高速にマッチングできる. 2 段階マッチングの例を次に示す.

［例 2.2］ 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)

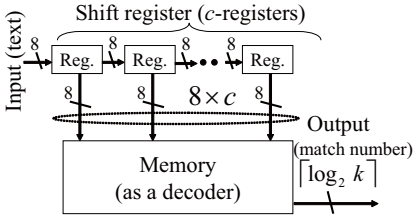

図 3　FIMM を模擬する回路.

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 [17] 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.　インデックス生成関数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. 本章では, オートマトンの部分を少ないメモリで実現する手法について述べる. 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, 第一段階の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, 第二段階である 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. 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 [15]. 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を実現する. 表現能力を制限することで, 回路を単純にでき, 必要メモリ量を削減できる. 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 $M_{FIMM} = 2^{8\times4}\lceil log_2(k+1)\rceil \simeq 2^{51}$ ビットとなり, 現実的でない. 本章では, 並列ふるい法を用いて FIMM をコンパクトに実現する手法について述べる. 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効率よく小さなメモリに実現す

(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.

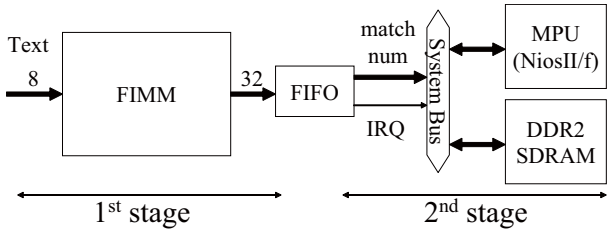

図 4　ウイルス検出エンジン.

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	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	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									

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ウイルスパターンを効率よくメモリに格納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** [22] 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) [13] 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, 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 $f(\vec{X}) : B^n \to \{0, 1, \ldots, k\}$ 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 4.3］　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　ハッシュ法を用いたインデックス生成関数の実現

$f(X_1, X_2)$ 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, メモリ量が $2^n \lceil \log_2(k+1) \rceil$ 必要であり, 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.

［例 4.4］　表 4 に示すように, ウイルス検出問題では $n = 8c = 32, c = 4$, $k = 514287$ である.

$X_1 = (x_1, x_2, ..., x_p)$ 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 4.5］　例 4.3 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 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

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	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									

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

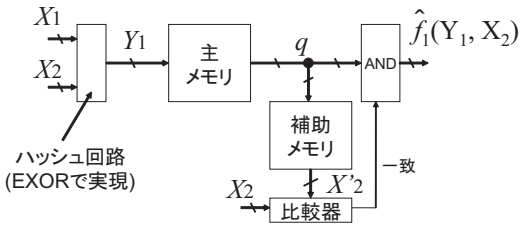

図 5　インデックス生成回路 (ハッシュ法).

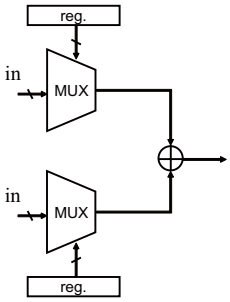

図 6　プログラマブル・ハッシュ回路.

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)** 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 [20], [21] 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］　[21] 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

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

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 4.6］　$\frac{k}{2^p} = \frac{1}{1}$ とすると $\delta = \frac{2}{3} \simeq 0.666$ である. 例えば, 登録ベクトル数とほぼ等しいアドレス空間 $(k \simeq 2^p)$ を持つ主メモリには約 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 4.7］　図 7 と図 8 に主メモリで実現できる登録ベクトルの割合を示す. 図 7 と図 8 において, 縦軸は主メモリで実現できる登録ベクトルの割合を表し, 横軸は登録ベクトル数を表す. 図 7 はランダムな登録ベクトルを格納した場合を表し, 図 8 は英単語帳, すなわち偏りのある登録ベクトルを格納した場合を表す. ハッシュ関数 1 は 1 変数を図 6 に示すプログラマブル・ハッシュ回路で 1 変数を選択した場合である. すなわち, 任意の変数を選択した場合である. ハッシュ関数 2 は図 6 に示すプログラマブル・ハッシュ回路を用いた場合である. 偏りがあるデータに関してはプログラマブル・ハッシュ回路を用いると, 分解表において非零要素を

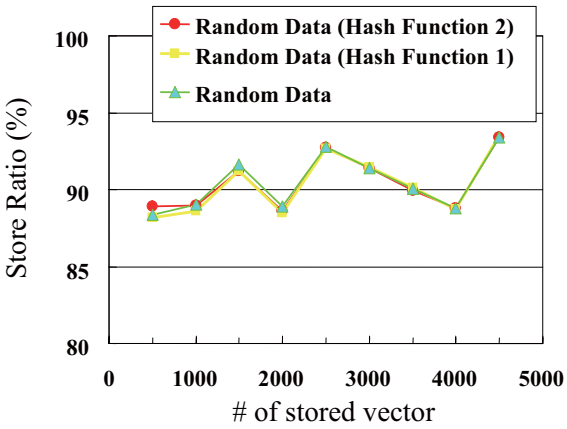

図 7　主メモリで実現できる割合 (ランダムデータ).

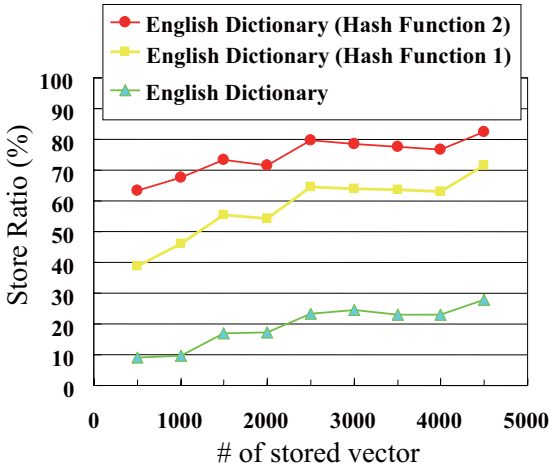

図 8　主メモリで実現できる割合 (ブロック英単語帳).

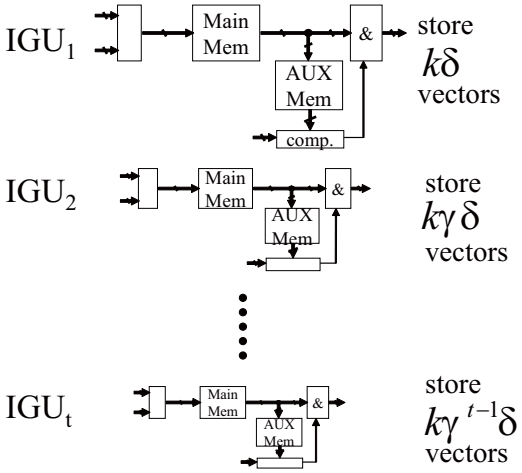

図 9　並列ふるい法.

一様に分散させることができるため, 主メモリに格納できる割合が増加する. しかしながら, ランダムなデータに対してはプログラマブル・ハッシュ回路を用いても主メモリに格納できる割合は増加しない. (例終)

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]　[19] 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

$$p = 2\lceil log_2(k+1) \rceil - 1 \tag{3}$$

である. ただし, 主メモリの入力数を p とするとき, $k \leqq 2^p$ 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.

4.4　並列ふるい法

定理 4.1 や推論 4.1 より, 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. 実現すべきウイルス検出回路では登録ベクトル数は $k = 514287$ であるから, 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 1 より必要なメモリ量は $2^{4\times 8} \times 514287 \simeq 2^{51}$ ビットとなり, 現在の技術では実現できない. 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 9 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 4.8]　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.

[定義 4.3]　図 9 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(注2).

4.5 並列ふるい法における主メモリの入力数

登録ベクトル数を k とし, 主メモリの入力数を p とする. 定理 4.1 より, $\frac{k}{2^p}=1$ のとき, 第一番目の主メモリに格納できるベクトルの割合は $\delta=0.666$ である. また, 主メモリのメモリ量は $2^p\lceil log_2(p+1)\rceil$ 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 $\gamma=1-\delta=0.334$ である. 残りの $k\gamma$ 個のベクトルのうち, $\delta=0.666$ を格納する p' 入力の第二番目の主メモリには $k\gamma\delta=0.222k$ 個のベクトルを格納できる. 従って, p 入力と p' 入力である主メモリを 2 個用いることで $k\times0.666+k\times0.222=0.888k$ 個の登録ベクトルを格納できる.

よって, 本論文では $k<2^p$ を満たす最小の p を第一番目の主メモリの入力数とする.

[定理 4.2] t 個の IGU を有し k 個の登録ベクトルを格納する並列ふるい回路 (図 9) において, 残りのベクトルの個数を r とする. 関係

$$t=\lceil log_{\frac{1}{\gamma}}(\frac{k}{r})\rceil \tag{4}$$

が成立する. ただし, $\gamma+\delta=1.00$ であり, δ は定理 4.1 で得られる値である. また, 各主メモリの入力 p は $k<2^p$ を満たすものとする.

証明 i 番目の IGU の主メモリに格納できる登録ベクトルの割合を δ とし, 残りのベクトルの割合を γ とする. t 個のインデックス生成回路を用いて格納可能な登録ベクトルの割合は

$$\begin{aligned}&\delta+\gamma\delta+\gamma^2\delta+\cdots+\gamma^{t-1}\delta \\ &=\delta\frac{1-\gamma^t}{1-\gamma}=1-\gamma^t\end{aligned} \tag{5}$$

となる. よって, t 個の IGU を用いて k 個の登録ベクトルを格納した場合, 残りのベクトルの個数を r とすると,

$$\begin{aligned}r&=k-k(1-\gamma)^t \\ &=k\gamma^t\end{aligned} \tag{6}$$

となる. これを t について解くと

$$t=log_{\frac{1}{\gamma}}(\frac{k}{r}) \tag{7}$$

を得る. t は整数であるから, 式 (4) を得る. (証明終)

定理 4.2 より, 残りの登録ベクトル数が与えられたとき, 並列ふるい回路において, 必要な IGU の台数 t が得られる. ほぼ全てのベクトルを格納する場合 ($\frac{r}{k}$ を零に近づける場合), t が大きくなり, 多数のインデックス生成回路が必要となる. 従って, それらの周辺回路が複雑になる. よって, 本論文では, FPGA の組込みメモリに登録ベクトルを全て格納できるまで繰り返しインデックス生成回路を適用する. 補題 4.1 より, k 個の登録ベクトルをほぼ格納するのに必要な主メモリの入力数 p は高々$2\lceil log_2(k+1)\rceil-1$ であり ALTERA 社の FPGA の組込みメモリの大きさは 9 キロビットである. 従って, $r=255$ 以下になるまでは, 登録ベクトルを外付け SRAM で実現したインデックス生成回路に格納し, $r=255$ 以下のベクトルを FPGA の組込みメモリ等で実現した 1 個のインデックス生成回路に格納する.

[例 4.9] 実現すべきウイルス検出回路のベクトル数を $k=514287$ とする. 定理 4.2 より, $\gamma=\frac{1}{3}$ ($\frac{k}{2^p}=\frac{1}{1}$) となるように主メモリの入力数 p を設定する場合, 必要なインデックス生成回路の台数は

$$t=\lceil\log_3\frac{514287}{255}\rceil=7 \tag{8}$$

となる. 残りの 255 個以下の登録ベクトルは FPGA 内の 1 つのインデックス生成回路で実現するため, 全ての登録ベクトルを 8 台のインデックス生成回路で実現できる. (例終)

[例 4.10] 表 10 に登録ベクトルの推定値と実験値を示す. 表 10 において, p_j は j 番目の IGU の主メモリの入力数を表し, 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. 表 10 より, 例 4.9 で求めた 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. (例終)

実験値では $\frac{k}{2^p}$ の値は $\frac{1}{1}$ よりも大きくなった. 定理 4.1 より, $\frac{k}{2^p}$ が大きくなると主メモリに多くの登録ベクトルを格納できる. よって, 残りのベクトルは推定値よりも少なくなり, 残りのベクトルを全て格納するために必要なインデックス生成回路の主メモリも小さくなった.

(注2):並列ふるいとは複数のインデックス生成回路によって登録ベクトルを徐々に削減する (ふるいにかける) ことによる.

表 10　登録ベクトルの推定値と実験値.

	推定値			実験値		
	p_j	k_j	r_j	p_j	k_j	r_j
IGU_1	19	331414	165757	19	321659	1775513
IGU_2	18	110493	55263	18	128398	47115
IGU_3	16	36838	18424	16	33791	13324
IGU_4	15	12281	6142	14	9267	4057
IGU_5	13	4094	2047	12	2641	1416
IGU_6	11	1364	682	11	1055	361
IGU_7	10	454	227	9	277	84
IGU_8	15	227	0	9	84	0

表 11　インデックス生成回路に用いたメモリ.

		主メモリ		補助メモリ	
	k_j	入出力数	使用メモリ	入出力数	使用メモリ
IGU_1	331414	i=19,o=19	Off chip SRAM	i=19,o=13	Off chip SRAM
IGU_2	110493	i=18,o=18	Off chip SRAM	i=18,o=14	Off chip SRAM
IGU_3	36838	i=16,o=16	Off chip SRAM	i=16,o=16	8 M144k
IGU_4	12281	i=14,o=14	2 M144k	i=14,o=18	3 M144k
IGU_5	4094	i=12,o=12	6 M9k	i=12,o=20	10 M9k
IGU_6	1364	i=11,o=11	3 M9k	i=11,o=21	6 M9k
IGU_7	454	i=9,o=9	1 M9k	i=9,o=23	2 M9k
IGU_8	227	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 Off チップ SRAM ボードを 3 枚使用した. Off チップ 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(注3). 表 11 において, k は格納した登録ベクトル数を表す. 実装結果より, 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. また ClamAV のパターンを格納するために DDR2-SODIMM を取り付けた. 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, 設計したウイルス検出エンジンは 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

$$Th = 0.200 \times 8 = 1.6[Gbps] \tag{9}$$

である. また, 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 514287 個格納するので,

$$MUC = \frac{3500880}{514287 \times 4} = 1.7\ [Bytes/Char]. \tag{10}$$

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} は

$$Th_{MUC} = \frac{Th}{MUC} \tag{11}$$

である.

(注3)：複数の主メモリを外付けの SRAM で実現する場合, ハッシュ入力を共通にして, 1 つの 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 [25]	6.0	1533	2896.2	0.0020	ASIC 実装
Aldwari et.al [2]	14.0	1542	126.0	0.1111	with SRAM
Bitmap compressed Aho-Corasick [25]	8.0	1533	154.0	0.0519	ASIC 実装
Path compressed Aho-Corasick [25]	8.0	1533	60.0	0.1333	ASIC 実装
Alicherry et.al [3]	20.0	100	48.0	0.4166	with TCAM
Yu et.al [26]	2.0	1768	3.0	0.6666	with TCAM+MPU
USC RegExpController [5]	1.4	1316	46.0	0.0304	AC 法 + MPU
Hardware Bloom Filter [4]	0.5	35475	1.5	0.3333	with SDRAM
提案手法	1.6	514287	1.7	0.9417	MPU+SRAM

表 12 より, 提案手法だけが 514287 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といえる. Yu [26] の手法と比較すると, Th_{MUC} が 1.41 倍と最も差がないが, 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 SRAM ボードと FPGA1 個で ClamAV のウイルスパターン 514287 個全て格納した. 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] H-J. Jung, Z. K. Baker, and V. K. Prasanna, “Performance of FPGA implementation of bit-split architecture for intrusion detection systems,” *Proceedings of the Reconfigurable Architectures Wrokshop at IPDPS (RAW'06)*, 2006.
[12] Kaspersky, http://www.kaspersky.com/
[13] T. Kohonen, *Content-Addressable Memories*, Springer Series in Information Sciences, Vol. 1, Springer Berlin Heidelberg 1987.
[14] 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.
[15] R. McNaughton and S. Papert. “Counter-FreeAutomata,” *MIT Press*, 1971.
[16] 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.
[17] PCRE: Perl Compatible Regular Expressions, http://www.pcre.org/
[18] 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.

[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] 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.

[21] 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.

[22] T. Sasao, "Design methods for multiple-valued input address generators," *ISMVL-2006*(invited paper), Singapore, May 17-20, 2006.

[23] 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.

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

[25] 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.

[26] 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.