情報処理学会論文誌 Vol.56 No.3 892–902 (Mar. 2015)

文書型マルウェアに対するエントロピーとエミュレーションを用いたシェルコード特定方法

岩本 一樹[1,2,a)] 和﨑 克己[2,b)]

受付日 2014年6月29日, 採録日 2014年12月3日

概要：アプリケーションの脆弱性を攻撃する文書型マルウェアを動的に解析するためには，該当する脆弱性を持つアプリケーションを準備する必要がある．しかし脆弱性の種類を特定することは困難な場合があり，またアプリケーションが入手できない可能性もある．一方，脆弱性を攻撃した後に動作する不正なプログラム（シェルコード）は脆弱性やアプリケーションに関係なく独立して動作することが多い．そこで本研究では脆弱性の種類を特定することなく，またアプリケーションがなくても文書型マルウェアの動的解析が行えるようにするために，文書型マルウェアに含まれるシェルコードを特定して実行する方法を提案する．提案手法では文書ファイルのエントロピーから算出したシェルコードの候補の優先順位に基づいて，文書ファイル内のバイト列をエミュレータで実行し，シェルコードの特徴を観測することで特定を行う．我々が作成したシステムに 88 種類の文書型マルウェアを投入した結果，74 種類でシェルコードを特定できた．また 74 種類のシェルコードを動的解析したところ，51 種類でマルウェアとしての動作を確認できた．

キーワード：マルウェア，シェルコード，エントロピー，動的解析，脆弱性

The Method for Shellcode Extraction from Malicious Document File Using Entropy and Emulation

Kazuki Iwamoto[1,2,a)] Katsumi Wasaki[2,b)]

Received: June 29, 2014, Accepted: December 3, 2014

Abstract: The following document is an analysis of malicious documents which exploit vulnerability in applications dynamically, the application must have appropriate vulnerability. Therefore, we have to analyze the document statically to identify the type of vulnerability. Moreover it is difficult to identify unknown vulnerability, and the application may not be available even if we could identify the type of vulnerability. However malicious code which is executed after exploiting does not have relation with vulnerability in many cases. In this paper, we propose a method to extract and execute shellcode for analyzing malicious documents without identification of vulnerability and application. Our system extracts shellcode by executing byte sequence to observe the features in document file in order of priority decided on the basis of entropy. When 88 malware samples were analyzed by our system, it extracted shellcode from 74 samples. And 51 of extracted shellcodes behaved as malicious software in dynamic analysis.

Keywords: malware, shellcode, entropy, dynamic analysis, vulnerability

1. はじめに

標的となる組織からの情報窃取を目的とした標的型攻撃の導入として，標的型メールにより不正なプログラムが送り込まれることが報告されている [1], [2]．標的型メールとは攻撃対象を特定の者に狙い定めて送られるもので，このメールには不正なプログラムが組み込まれた文書ファイル

1 株式会社セキュアブレイン先端技術研究所
Advanced Research Laboratory, SecureBrain Corporation, Chiyoda, Tokyo 102–0083, Japan
2 信州大学大学院総合工学系研究科
Interdisciplinary Graduate School of Science and Technology, Shinshu University, Matsumoto, Nagano 390–8621, Japan
a) kazuki_iwamoto@securebrain.co.jp
b) wasaki@cs.shinshu-u.ac.jp

が添付されている場合がある．標的となった者はこの文書ファイルが攻撃目的で作成されたことを知らずに開封し，利用しているコンピュータで不正なプログラムを実行させてしまう．標的型攻撃対策の一環としては，このような文書ファイルを動的解析することで不正なプログラムの挙動を分析する．

しかし，従来の動的解析ではオペレーティング・システム（OS）やアプリケーションといった実行環境に依存することが多く，脆弱性がある環境を再現できずに動的解析が行えない場合もある．一方で脆弱性を攻撃した後に動作する文書ファイルに埋め込まれた不正なプログラム（シェルコード）は汎用性があり，特定の OS やアプリケーションで実行環境を構成する必要がないことが多い．そこで本システムは，環境に依存する文書型マルウェアを解析するために，まず文書ファイル内のシェルコードの位置を特定することを目的とする．また，特定したシェルコードを直接実行することで動的解析を実施するために実行可能ファイルを出力する．

システムを作成する前にシェルコードの位置が特定できている検体を用いて事前調査を行う．事前調査では文書ファイルで除外すべき領域，シェルコードの候補の優先順位を決めるアルゴリズムやエントロピーの算出方法を決定する．また本システムはシェルコードの候補となるバイト列をエミュレータで実行してシェルコードの特徴を観測する．特徴が観測できるまでに必要なエミュレータが実行する命令の数（ステップ数）も事前調査で決定する．

本論文は以下で構成される．まず，2 章では対象となる環境やファイル形式，シェルコードの特徴，エントロピーの定義などを明らかにすることでシステムの設計方針を示す．3 章では検体セットを用いて最も適したシステムのアルゴリズムやパラメータを決定する．4 章では実際に作成したシステムに検体を投入し，パフォーマンスの評価を行う．5 章では 3 章の事前調査や 4 章の実験結果に対して考察する．6 章では関連研究と本システムの比較を行い，最後に 7 章で今後の課題をまとめる．

2. 対象とする環境と提案手法

本システムはシェルコードの候補となるファイル内部のバイト列をエミュレータで実行し，シェルコードの特徴が観測できたときには，そのバイト列をシェルコードと見なす．事前にシェルコードの候補の絞り込みを行い，またシェルコードの可能性が高いファイル内部のバイト列から順にエミュレーションを行うことで，効率良くシェルコードを特定する．

システムの全体の流れを**図 1** に示す．

2.1 対象とする環境

本システムでは下記のファイル形式の 32 ビット Windows の文書型マルウェアを対象とする．

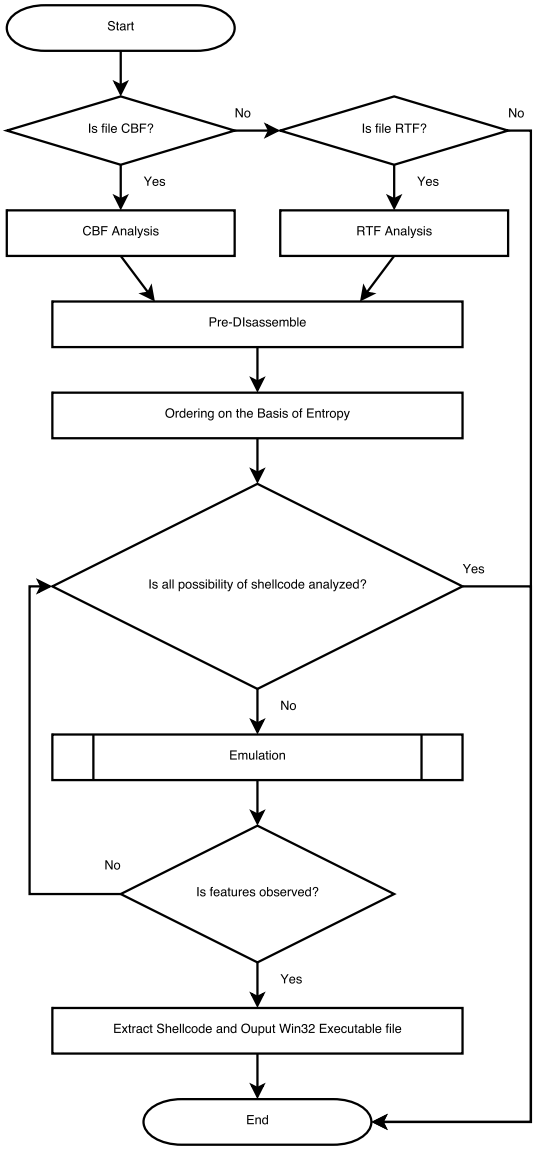

図 1 フローチャート
Fig. 1 Flowchart.

- Microsoft Office Word（doc）
- Microsoft Office Excel（xls）
- Microsoft Office PowerPoint（ppt）
- Rich Text Format（rtf）

本システムでは実際に文書型マルウェアを開くアプリケーションを必要としない．そのため下記のようなアプリケーションに依存するマルウェアには対応しない．

Return Oriented Programming（ROP）[3]
スタックに格納された戻り値に基づいてアプリケーションまたは OS などのコードが実行される ROP では，文書ファイル内部にシェルコードが含まれていない場合があり，本システムでは対応しない．シェルコードがある場合でも，特定のアドレスにシェルコードが読み込まれることが前提なっている場合には本システムでは対応できない．

文書ファイル内部にシェルコードがないマルウェア
脆弱性が攻撃されることで，文書ファイル以外のファイルのコードが実行される場合には，文書ファイル内部に存在しないので本システムでは対応しない．たとえばCVE-2011-1980 [4] では文書ファイルと同じフォルダにあるDynamic Link Library（DLL）が実行される．

環境に強く依存する文書型マルウェア
シェルコードに汎用性がなく，メモリの確保やシェルコードのアドレスなど特定の条件が前提なっている場合には本システムでは対応しない．

2.2 候補の絞り込みと優先順位

本システムは次の方法でシェルコードの候補を絞り込み，優先順位を決定する．CFB解析およびRTF解析による絞り込みはファイルを対象とする本システムであるから可能であり，ネットワークのパケットからシェルコードを抽出する研究 [5], [6], [7] には見られない．ファイルの構造を解析する研究 [8], [9] も提案されているが，それらはシェルコードの抽出を目的とはしていないので本システムとは異なる．

2.2.1 CFB 解析

2.1 節のファイル形式はRTFを除いてCompound File Binary（CFB）形式 [10], [11] である．CFB形式のファイルは**図 2** のようにファイルシステムを模した構造になっており，Header，DiFAT，FAT，Mini FAT，Directory，Stream，Mini Stream，Freeの各要素に分類できる．Headerはファイルの先頭の情報領域である．DiFATとFAT，Mini FATはファイルシステムのFAT，Directoryはディレクトリエントリ，StreamとMini Streamはファイルに対応する．Freeは未使用の領域である．本システムではCFBを解析してファイル内をこれらの領域に分け，その中の特定の領域をシェルコードの候補とする．

なお上記のCFBの仕様上存在する要素以外にも，実際にはFATから参照されない不正な領域（Illegal）やファイルの末尾に付加されたデータ（Extra）が存在する．それらの仕様外の領域も本システムでは上記の要素と同様に扱う．

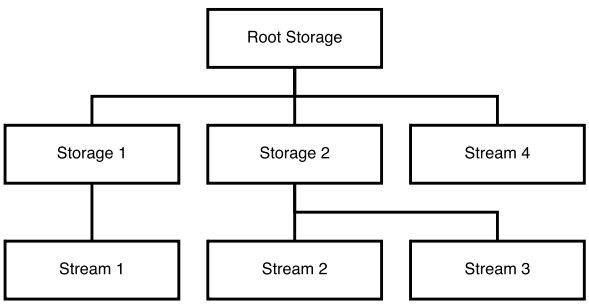

図 2 CFB 形式の構造
Fig. 2 Compound File Binary hierarchy.

2.2.2 RTF 解析

RTFはテキスト形式であるが，内部にはテキストにエンコードされたバイナリデータ（文字列を含む）がある [12]．シェルコードはバイナリデータの中にあることは明らかなので，RTFの場合にはこのバイナリデータを対象とする．また少なくとも2.3節であげた特徴をコード内に含む必要があるので，小さなバイナリデータの中にシェルコードを収めることはできない．本システムでは2.3節の特徴を持つ最小のコードは128バイト程度であると見積もり，128バイト未満のデータは対象としない．

2.2.3 事前逆アセンブル

本システムではエミュレーションを行う前に，対象となるバイト列を逆アセンブルする．逆アセンブルが正しく行えないときにはエミュレーションを行わずに，そのバイト列にはシェルコードが存在しないと見なす．この処理はエミュレータの起動に時間がかかるため，エミュレータを起動する回数を減らすために行う．本システムでは逆アセンブルの可否を絞り込みのみに利用しており，逆アセンブル結果を用いる他のシェルコードを抽出する方法 [5], [6], [7] とは異なる．

2.2.4 エントロピーによる優先順位

仮にファイルの先頭から順番にシェルコード判定を行った場合，明らかにシェルコードではない箇所もエミュレーションされることになり効率が悪い．そこで他のシェルコードを抽出する方法 [5], [6], [7] とは異なり，エントロピーを用いてシェルコードの候補の順番を決定する．

バイト列 $(a_1, a_2, \cdots, a_{n-1}, a_n)$ のエントロピーは

$$H(X) = \sum_{i=0}^{255} -P_i \log_2 P_i \tag{1}$$

で求めることができる．P_i は i の確率（バイト列の中の i の数をバイト列のサイズで割った値）であり，

$$P_i = \frac{\sum_{j=1}^{n} \begin{cases} 1(a_j = i) \\ 0(a_j \neq i) \end{cases}}{n} \tag{2}$$

で求めることができる．$H(X)$ の範囲は $0 \leq H(X) \leq 8$ である．なお，$P_i = 0$ のときには $-P_i \log_2 P_i = 0$ とする．

シェルコードは実行可能で意味のあるプログラムであるので，ファイルの中ではシェルコードの部分はエントロピー（乱雑さ）が高くなる．一方，シェルコード以外の部分は文書ファイルのデータになるため，エントロピーはシェルコードよりも低くなる．またファイル内のデータ領域の隙間に相当する部分は，同じ値が連続することになり，エントロピーは非常に小さい．

たとえば**表 1** ではファイルのアドレスの5E00hからシェルコードが開始される．シェルコードの手前は00hが連続するのに対して，シェルコード部分は意味のある実行可能

表 1 シェルコード付近のバイナリイメージ
Table 1 Binary image around shellcode entry point.

5DD0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5DE0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5DF0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5E00	60 B9 A4 05 00 00 EB 0D 5E 56 46 8B FE AC 34 FC
5E10	AA 49 75 F9 C3 90 E8 ED FF FF FF 61 15 C1 FE FC
5E20	FC AA CF 3C 98 77 BC CC 77 BC F0 77 8C E0 51 77

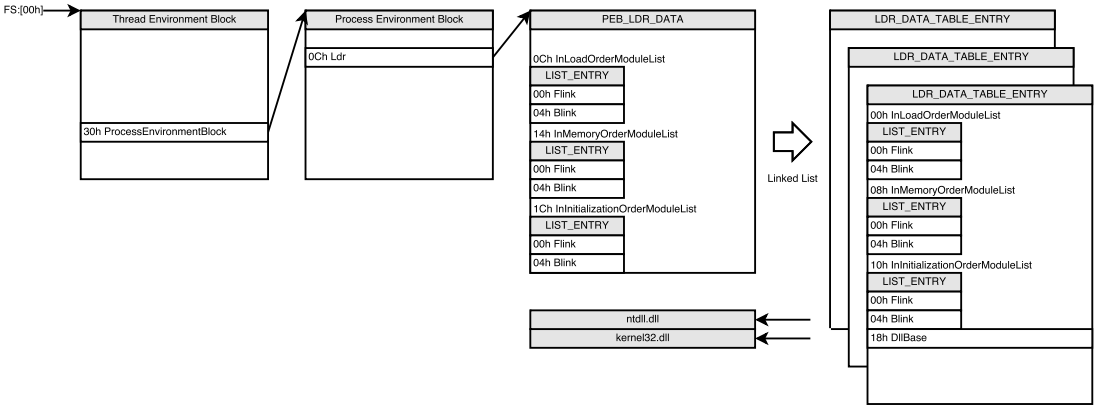

図 3 DLL のアドレスを取得するための構造体
Fig. 3 Structures to get DLL base address.

なコードなので値の分布に大きな偏りはない．

本システムは入力された文書ファイルを一定の範囲で区切り，それぞれの範囲のエントロピーを求め，「エントロピーが高いバイト列」または「エントロピーの差が大きいバイト列」を列挙する．本システムは列挙したシェルコードの存在する可能性が高いバイト列から順にエミュレーションを行う．

我々がツール類に頼らずに文書型マルウェアからシェルコードを特定するときには，表 1 のような文書ファイル中のシェルコードが存在しそうな場所を，バイナリエディタで探索して逆アセンブルを行っていた．本システムはこの作業をエントロピーを利用した定量的な評価により自動化する．

2.3 シェルコード判定

本システムはシェルコードが存在する可能性が高いファイル内部のバイト列を 32 ビット Windows で動作するコードと見なしてエミュレータで実行する．仮にそのバイト列からシェルコードが開始するならば，継続的にエミュレータで実行が可能であり，エミュレータの実行中にシェルコードに特徴的な動作が観測できる．エミュレーションが継続ができなくなったとき，または一定のステップ数の実行後に特徴的な動作が観測できないときには，本システムはエミュレーションを打ち切り，次のシェルコードの候補に対して同様の処理を行う．すべてのシェルコードの候補で，特徴的な動作が観測できないときにはシェルコードが存在しないと判断する．

本システムでは下記の動作をシェルコードの特徴とする．

(1) 自身が書き換えたメモリを実行する．
(2) FS レジスタ経由で Process Environment Block（PEB）へのアクセスが発生する．
(3) Application Programming Interface（API）が呼び出される．

シェルコードの本体が暗号化されている場合には，シェルコードの最初に実行されるコードが暗号化された本体のコードを復号した後で，本体のコードが実行される．そのため (1) の動作が発生する．ただし本体が暗号化されていないときには (1) の動作は起こらず，(2) の動作が発生する．

32 ビット Windows では CPU のレジスタの 1 つである FS レジスタに実行中のスレッドに関する情報を格納する Thread Environment Block（TEB）のアドレスが設定されている．**図 3** のように TEB から PEB，PEB_LDR_DATA へと構造体内のポインタをたどると LDR_MODULE への連結リストがあり，LDR_MODULE から DLL のアドレスを取得できる．シェルコードは図 3 の構造体から DLL のアドレスを取得することで必要とされる API のアドレスを取得するので (2) の動作をシェルコードの特徴とする．

シェルコードは API のアドレスが取得できた後には，その API を呼び出す．ゆえに (3) の動作をシェルコードの特徴とする．

本システムは (1) の後に (2)，または (2) の後に (3) が観測できたときには，そのバイト列からシェルコードが開始されると見なす．

2.4 実行可能ファイル

シェルコードを見つけたときには，本システムはシェルコードを実行するための 32 ビット Windows 実行可能ファイルを出力する．この実行可能ファイルには文書ファイルとファイル名，シェルコードのファイル内のアドレスが格納されている．シェルコードはアプリケーションが開いているファイルのハンドルを列挙することで，シェルコード自身のファイルのハンドルの取得を試みることが多い．そのため本システムが出力する実行可能ファイルは実際のアプリケーションの状態を再現するために，ファイルが実行されると，テンポラリフォルダに文書ファイルを作成して開いた後にシェルコードをメモリに配置して実行する．また実行可能ファイルは GetCommandLine と GetModuleFileName をフックして実行可能ファイルの名称を標準で文書ファイルを開くアプリケーション（たとえば「WINWORD.EXE」など）に偽装する．

動的解析のために実行可能ファイルを出力するという方法はほかにもすでに存在する [7], [13]．API をフックしてアプリケーションを偽装する方法もシェルコードを解析するツール [14] としてすでに存在する．

3. 事前調査

シェルコードを特定するにあたり，最適なアルゴリズムやパラメータを決定するために検体セットを準備して下記の事前調査を行った．

3.1 検体セット

我々は本システムの事前調査と実験のために，入手したマルウェアの検体の中から 2.1 節の条件に合致すると推測できる検体をランダムに選び，静的解析を行いシェルコードの存在が確認できた検体から検体セットを作成した．したがって検体セットの検体はシェルコードのアドレスが確認できている．CFB 形式の文書型マルウェアではランダムに選んだ 125 の検体の中から 73 が条件に合致し，RTF の文書型マルウェアではランダムに選んだ 25 の検体の中から 15 が条件に合致した．検体セットのファイル形式別の脆弱性の内訳は**表 2** である．脆弱性は表 2 のとおり 15 種類あり，3 つのファイルは脆弱性の詳細などが不明である．脆弱性が異なるかシェルコードのファイル内のアドレスが異なるユニークな検体は 42 種類ある．

CFB 形式のファイルを一括してランダムで選んだため表 2 では各ファイル形式の比率に偏りがある．検体の入手の段階ではファイル形式の比率が均等になることは意図しておらず，表 2 の比率が入手できた検体のファイル形式のおおよその比率であり，我々が対象としている攻撃のファイル形式の比率を反映していると思われる．表 2 の脆弱性にも偏りがあるが，同様に攻撃で使われる脆弱性の比率を反映していると思われる．

表 2 ファイルの種類と脆弱性
Table 2 File type and vulnerability.

Vulnerability	doc	xls	ppt	rtf	Total
CVE-2006-2389	4				4
CVE-2006-2492	13				13
CVE-2006-6456	2				2
CVE-2007-0671			1		1
CVE-2008-2244	5				5
CVE-2008-4841	1				1
CVE-2009-0556			1		1
CVE-2009-0563	1				1
CVE-2009-3129		24		5	29
CVE-2010-0822		2			2
CVE-2010-1901				1	1
CVE-2010-3333				6	6
CVE-2011-1269	1		2		3
CVE-2012-0158	13			2	15
CVE-2014-1761				1	1
UNKNOWN	3				3
Total	43	26	4	15	88

表 3 観測された特徴
Table 3 Observed feature.

Feature	Number
(1) Self-modifying, (2) PEB access, (3) API call	55
(1) Self-modifying, (2) PEB access	2
(2) PEB access, (3) API call	17
None	14

表 4 最大ステップ数
Table 4 Maximum step.

Feature	Step
Start to (1) Self-modifying	35,847
Start to (2) PEB access or (1) Self-modifying to (2) PEB access	857
(2) PEB access to (3) API call	2,772,706

3.2 シェルコードが存在する CFB の要素

検体セットのシェルコードが CFB 形式のどの領域に存在するか調べたところ，すべて Stream 領域に存在した．なお，本システムでは Mini Stream と Stream は区別していない．

3.3 ステップ数測定

エミュレータで実行する際に 2.3 節のシェルコードの特徴を観測するために必要なステップ数を決定する．そのため解析済みの検体のシェルコードの先頭からエミュレータで実行し，2.3 節の特徴が観測できるまでのステップ数を測定したところ**表 3** と**表 4** の結果になった．

3.4 エントロピー算出対象のバイト数とアルゴリズム

エントロピーを求める場合，ファイルの中のどの程度の

表 5 エミュレーション回数の比率の平均
Table 5 Average of ratio of emulation trials.

Size	Entropy Order	Delta Order
128	0.561	0.33
192	0.581	0.317
256	0.578	0.288
384	0.554	0.268
512	0.593	0.27
1,024	0.715	0.305
1,536	0.817	0.403
2,048	0.882	0.55

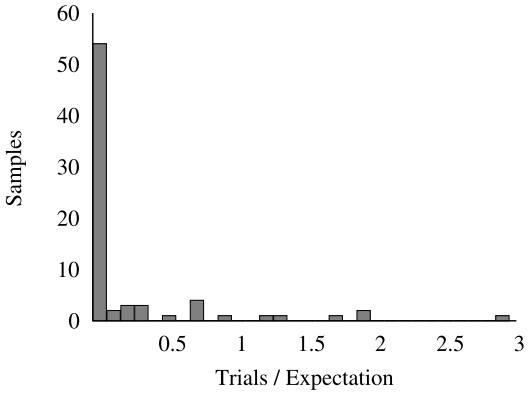

図 4 エントロピーが高い順の試行回数の比率
Fig. 4 Ratio of number of trials (Entropy order).

長さのバイト列からエントロピーを算出するのかが問題となる．そこで最も適切なバイト数，式 (2) の n を求めるために 128 バイトから 2,048 バイトで検体セットのエントロピーを算出した．エントロピーの差は求めるバイト列の前のバイト列との差とする．実装上はファイルの範囲を超えてしまう場合にはファイルの先頭または末尾からエントロピーを求めることとする．また，すべてのバイト列のエントロピーを求めると時間がかかりすぎるので，16 バイトごとにエントロピーを算出した．

表 5 では，検体セットの検体に対して「エントロピーが高い順」と「エントロピーの差が大きい順」にシェルコードを探索した場合に，エミュレーションの試行回数と期待値（ランダムにバイト列を選びシェルコードの特定を試みた場合の試行回数）の比率の平均値をエントロピー算出対象のバイト数ごとにまとめた．

エントロピー算出対象のバイト数が 384 バイトで「エントロピーの差が大きい順」のときに最も試行回数が少なかった．このときシェルコードを探索した場合の試行回数と期待値の比率の分布を**図 4** に示す．比率が小さいほど効率が良く，比率が 1 よりも小さければランダムに選ぶよりも効率が良いといえる．

また参考のため，ファイルの先頭から順番にシェルコードを探索した場合の試行回数と期待値の比率の分布を**図 5** に示す．図 5 には示していないが，比率が 3 を超える検体は 2 つあった．

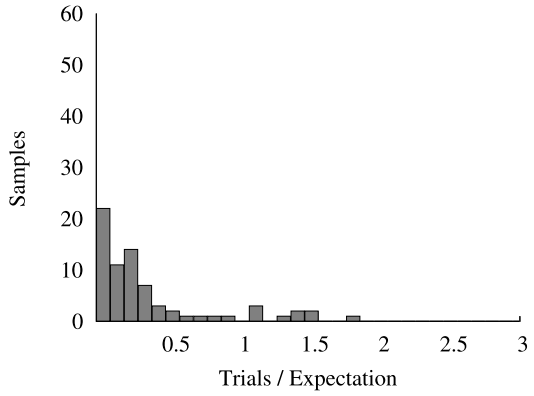

図 5 ファイルの先頭から順の試行回数の比率
Fig. 5 Ratio of number of trials (Address order).

4. 実験

3.2 節の調査結果より，本システムは Stream 領域だけを探索する．3.3 節の調査結果より，本システムではエミュレータによる実行で書き換えたメモリの実行が観測されたときには 16,384 ステップ延長し，PEB へのアクセスがあったときには最大 4,194,304 ステップ先まで実行して API 呼び出しを観測する．また 3.4 節の調査結果より，本システムではエントロピーを求める範囲のバイト数は 384 バイトで「エントロピーの差が大きい順」にシェルコードを探索する方法を実装する．

4.1 CFB・RTF 解析と逆アセンブルによる絞り込み

検体セットの検体に対して CFB 解析を行ったところ Stream 領域はファイル全体の 30.83%，RTF 解析ではファイルのバイトサイズに対してバイナリデータは 11.95%であった．またすべてのファイルに対して逆アセンブルできたのは 95.92%であった．

4.2 False Positive 検査

3 章の検体セットとは別に，正常なファイルを 125 種類（doc: 50，xls: 25，ppt: 25，rtf: 25）準備し，本システムでシェルコードを探索したところ，すべてのファイルでシェルコードを見つけることはなかった．正常なファイルは 2014 年 5 月頃にインターネット上のサイトからダウンロードした．すべてのファイルは Virus Total[*1]でマルウェアとして検出されていない．CFB 形式のファイルでは，ファイルのサイズは 18,432 バイトから 11,030,528 バイト，平均のサイズは 1,030,640 バイトである．RTF のファイルでは，ファイルのサイズは 217 バイトから 12,173,558 バイト，平均のサイズは 1,078,672 バイトである．CFB 形式で 5 MB を超えるファイルは 8 つ，RTF では 2 つある．

[*1] http://www.virustotal.com/

表 6 実験環境
Table 6 Environment.

	Shellcode Extraction	Dynamic Analysis	
		Host	Guest
CPU	Pentium M 1.20 GHz	Core i7 3.40 GHz	
		(4 cores)	(1 core)
Memory	1 GB	24 GB	512 MB
OS	Ubuntu 10.04 LTS	Windows 7	Windows XP SP3

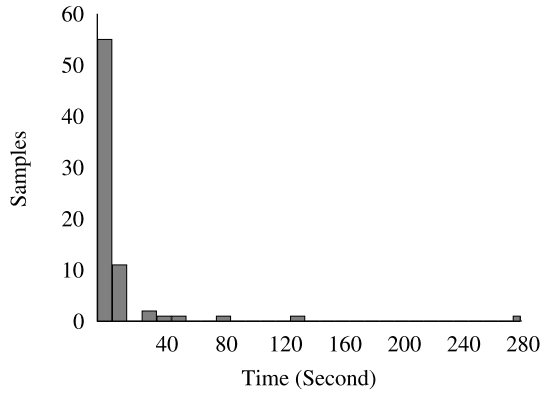

図 6 シェルコード抽出時間
Fig. 6 Time of shellcode extraction.

4.3 シェルコード特定

表 6 の実験環境で本システムで検体セットの 88 の検体に対してシェルコードの抽出を試みたところ，74 の検体でシェルコードを特定することができた．

全体では約 9,468 秒，シェルコードが特定できた検体だけでは約 5,389 秒かかった．しかし約 5,389 秒のうち 1 つの検体で約 4,399 秒かかっていた．エミュレータを 1 回実行するのにかかった時間は平均で約 1.638 ミリ秒であった．約 4,399 秒かかった検体を除くシェルコードが特定できたときにかかった時間の分布を**図 6** に示す．

なお，ファイルの先頭から順番にシェルコードを探したときには，シェルコードを特定できた 74 検体の中からランダムに選んだ 57 検体で約 6,804 秒かかった．エントロピーによってシェルコードの候補を決定する方法と性能を比較できれば十分なので 5,389 秒を超えたところで実験を打ち切った．シェルコードが特定できた検体 74 検体で約 2 時間以上，全体では約 3 時間以上はかかると推計できる．

4.4 動的解析の結果

本システムでシェルコードを特定した 74 の検体で出力された 32 ビット Windows 実行可能ファイルを仮想環境で実行したところ，結果は**表 7** になった．シェルコードがファイルを書き出して実行を試みた検体が 50 あり，1 検体は壊れたファイルが作成された．残りの 49 検体では 32 ビット Windows 実行可能ファイルが作成されて実行された．この 49 検体のハッシュ値を Virus Total で調べたところ，27 検体は Virus Total に登録されており，26 検体はマルウェアとして検出されていた．ネットワークに接続を試みた検体が 1 つあったが，サーバからの応答が得られずに通信は失敗した．一方，マルウェアとしての動作の前に無効な命令の実行や無効なメモリのアクセスなどが発生して継続して実行できなくなった検体が合計で 10 検体あった．またループから抜け出せなくなっている検体もあった．

表 7 出力されたファイルの実行結果
Table 7 Result of execution.

Success	Drop	Executable	Malware	26
			Benign	1
			Unregistered	22
		Broken		1
	Communication			1
Failure	Memory			6
	Instruction			3
	Unknown			1
Infinity Loop				13

5. 考察

一般的にシェルコードは汎用性があり，シェルコードが存在するメモリのアドレスやシェルコードが開始する時点でのレジスタなどが特定の値である必要はない．しかし汎用性がなく攻撃に使われる脆弱性が固定され，特定のメモリのアドレスやレジスタの値が必要なシェルコードもあったため，シェルコードの特定ができない検体があった．

5.1 エミュレーションの絞り込み

4.1 節の絞り込みにより，全体の約 30%が対象となった．しかし逆アセンブルによる絞り込みの効果は小さかった．エミュレータの初期化コードを改良するか，あるいはコンパイラや実行環境が異なるならば，逆アセンブルは不要になる可能性がある．

5.2 エミュレーションの試行順番

本システムではエントロピーを用いて候補となるバイト列の優先順位を決定した．図 4 より多くの検体では期待値に対して十分に少ないエミュレーションの試行回数でシェルコードを特定できることが分かった．しかし 74 検体中 6 検体は期待値を超える回数のエミュレーションが必要な（ランダムに選ぶよりも効率が悪い）検体であった．

図 5 から，シェルコードの開始位置はファイルの先頭付近に偏っており，ランダムに候補を選ぶよりは効率が良いことが分かる．しかし図 4 と比べると図 5 は効率が良いとはいえない．4.3 節の実験では，57 検体で約 6,804 秒かかっており実際の実行結果でもエントロピーを用いて候補を求めるより効率が悪かった．ランダムに候補を選ぶ場合の実験は行っていないが，図 4 および図 5 から，ランダム

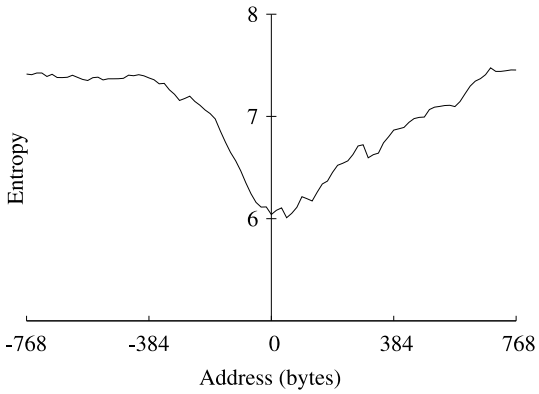

図 7 シェルコード付近のエントロピー
Fig. 7 Entropy near shellcode.

に候補を選ぶ方法が良いとは推測できない．

4.3 節の実験では，シェルコードの特定に 4,399 秒かかった検体があった．この検体のファイルサイズは 397,824 バイトであり，その中で Stream 領域でありかつ逆アセンブルできエミュレーションの対象となったのは 338,212 バイトであった．本システムはシェルコードを特定するまでに 314,549 回のエミュレーションを試みていた．これは対象の約 93%をエミュレーションしており，明らかにランダムに選ぶよりも効率が悪い．

検体のシェルコードのエントリポイント付近における，エントロピー値の変化を**図 7** に示す．図 7 の横軸はシェルコードのアドレスからの差であり，0 はシェルコードのアドレスになる．縦軸はエントロピーを示す．この検体のシェルコードの開始位置のエントロピーが約 6.041 であるのに対して，その前の文書ファイルのデータのエントロピーの方が高く約 7.377 であった．そのためエントロピーの差が負になったことでシェルコードの開始位置の優先順位が下がり効率が悪くなった．

今回の実験では，エントロピーの差が極端に低くなる検体はシェルコードを特定できた 74 検体の中で 6 検体であった．仮にシェルコードの前のデータのエントロピーを高くすることが可能ならば，本システムによるシェルコードの特定にかかる時間を増大させることができる．シェルコードの前のデータのエントロピーを高くすることができるかは，文書ファイルの仕様や攻撃する脆弱性に依存する．またエントロピーに差が生じないような文書ファイル形式があるならば，本システムの手法を用いることはできない．

5.3 動的解析の結果

シェルコード特定のためのエミュレータによる実行は最初の API 呼び出しで打ち切られるが，本システムが出力した実行可能ファイルではその後のコードも実行される．そのため表 7 の Failure に示すように，シェルコードが特定できても実際には実行できない検体もあった．シェルコードがロードされるメモリのアドレスやアプリケーションが確保しているメモリなどの再現や偽装が不十分であったことが原因であると考えられる．

一方，ファイルの作成と実行や通信の発生というようなシェルコードにおけるマルウェアとしての動作を確認できた検体もあった．たとえ作成されたファイルが実行できなかったりマルウェアとしての動作が観測できなくても，あるいは通信に失敗しても，表 7 の Success の結果は脆弱性があるアプリケーションでシェルコードが動作したときと同じ動作であると思われる．対象となる脆弱性があるアプリケーションを準備せずにマルウェアを動的解析するという本システムの目的は達成しているので，これらの検体については成功したと見なした．

5.4 破損した検体

表 7 ではシェルコードが想定するファイルのサイズよりも実際のファイルサイズが小さいために無限ループに陥った検体があった．これらの検体は仮に脆弱性があるアプリケーションでシェルコードが動作しても，同様に無限ループに陥ってしまいマルウェアとしては成立しないと考えられる．

検体セットを作る前に，無限ループに陥るコードまで静的解析を行っていれば，これらを検体セットに含めないこともできた．しかしこれは本システムを構築しシェルコードを実行できるようになったから判明したことであり，本システムが有用であるともいえる．

6. 関連研究

6.1 ネットワーク通信解析

Polychronakis らはネットワークのパケットからシェルコードを検出する方法 [5] を提案している．文献 [5] ではシェルコードが自身のアドレスを取得するためのコードに注目してシェルコードの候補を決定している．また本システムと同様にコードをエミュレータで実行し，自身が書き換えたメモリを実行するという特徴にも注目している．しかし PEB の取得や API 呼び出しといった他のシェルコードの特徴には対応していないので，自己書き換えがない場合には検出できない．

藤井らは自己書き換えがない場合にも対応する方法 [6] を提案している．また神保らは藤井らの提案する方法を用いてシェルコードを検知し，実行可能ファイルを作成して解析を行っている [7]．

本システムは文献 [5], [6], [7] を文書型マルウェアに応用したものであるといえる．

6.2 文書ファイルの構造解析

Li らは統計的に分析することで Microsoft Word の文書ファイルがマルウェアであるか判定する方法 [8] を提案している．文献 [8] ではエントロピーについて言及されてい

るが，この研究はシェルコードを特定するものではない．また文献 [8] では，実際に文書ファイルを開くアプリケーションの実装を多数準備して動的解析を行っており，アプリケーションを必要としない我々のシステムとは異なる．

大坪らは文書ファイルの構造を調べることで文書型マルウェアを検知する方法 [9] を提案している．文献 [8] とは異なり，文書ファイルを開くアプリケーションを必要とせず，シェルコードの有無は問わず静的な解析のみで検知する．しかし文献 [9] の目的は検知であり，本システムのシェルコードの特定とは異なる．

6.3 シェルコード解析

Cova らは悪意のある JavaScript を検出する方法 [15] を提案しており，この中でシェルコードの抽出も行っている．JavaScript が生成するコードを抽出するので，バイナリデータの中から実行可能なコードを探すことでシェルコードを特定する本システムとは異なる．また Fratantonio らは文献 [15] で抽出したシェルコードを動的に解析するためのツール [14] を提案している．本システムはシェルコードの特定を行い 32 ビット Windows 実行可能ファイルを出力することで，他の動的解析環境を利用することを前提としている．そのため，文献 [14] のように API のリストを出力するような機能はなく，文献 [14] であげられているような動的解析を行ううえでの問題点は，本システムでは他の動的解析環境が解決することを期待している．しかし本システムが出力する 32 ビット Windows 実行可能ファイルは，文献 [14] と同様に GetCommandLine と GetModuleFileName をフックして実行可能ファイルの名称を標準で文書ファイルを開くアプリケーションに偽装している．本システムは文献 [16], [17], [18] のような動的解析環境を想定しているが，本システムが特定したシェルコードを文献 [14] のツールで動的解析することができる可能性はある．

6.4 シェルコード特定

Boldewin は文書型マルウェアを解析する OfficeMalScanner [13] を提案している．OfficeMalScanner ではシェルコードに特徴的なコードである GetEIP などのパターンを探すことでシェルコードを特定しているので，文献 [13] で想定されていない同等の動作をするコードが使われているときにはシェルコードを特定できない．パターンを探す方法ではパターンに柔軟性を持たせたならば，検知できる可能性は高くなるが誤認の可能性も生じる．さらにシェルコードが暗号化されているときには PEB へのアクセスや API 呼び出しなど，GetEIP 以外のパターンを見つけることはできない．本システムでは，コードをエミュレータで実行しているので，コードのパターンには依存しない．また暗号化されたシェルコードもエミュレータによる実行で復号できる．ゆえに本システムは文献 [13] で特定できないシェルコードも特定できる可能性がある．しかしエミュレータで実行しているため，その実効速度は文献 [13] よりも遅くなる．

OfficeMalScanner に含まれる MalHost-Setup は，本システムの 32 ビット Windows 実行可能ファイルを出力する機能と同様の機能を提供する．本システムが特定したシェルコードのアドレス情報を用いて MalHost-Setup で 32 ビット Windows 実行可能ファイルを出力することは可能であり，逆に OfficeMalScanner が特定したシェルコードのアドレス情報を用いて本システムで 32 ビット Windows 実行可能ファイルを出力することも可能である．ただし文献 [13] では，2.4 節であげられているような実際に文書ファイルを開くアプリケーションの状態を再現する方法については言及されていない．

6.5 文書型マルウェア内部の実行可能ファイル抽出

Boldewin が提案する OfficeMalScanner [13] には文書ファイルに埋め込まれた実行可能ファイルを探す機能がある．三村らは文書ファイルに埋め込まれた実行可能ファイルを抽出する方法 [19] を提案している．これらの方法ではシェルコードで用いられているエンコード方式を試すことで実行可能ファイルを特定する．本システムと同様に脆弱性を持つアプリケーションを準備する必要はなく，2.1 節で示した本システムが対応できないマルウェアにも対応できる．しかし提案する方法が想定していないエンコード方式が使われている場合には実行可能ファイルを特定できない．また，これらの方法ではシェルコードそのものの解析を行うことはできないので，ファイルをダウンロードする場合はその URL および作成されるファイルのパスや名前などを知ることができない．

7. 今後の課題

本システムにより，文書型マルウェアが対象とする脆弱性を持つアプリケーションを準備することなくシェルコードを特定することができた．本システムは文書型マルウェアを解析するのに有効・有用であると考えられる．

本システムは文書型マルウェアの中でも 2.1 節で示すファイル形式だけが対象であり，またその中でシェルコードが存在してアプリケーションなしでも実行が可能であることを前提条件としている．本システムが対象とする文書型マルウェアが，文書型マルウェア全体のどの程度の割合になるのかは正確には分からないが，本システムが限定的であることは間違いない．ROP をどのように自動解析するか，あるいはアプリケーションに依存する文書型マルウェアをどのように扱うかは課題の 1 つである．

一方，対応するファイル形式を増やすことで対象となる文書型マルウェアを増やすことも可能である．CFB 形式は

他のアプリケーションでも使われているので，対応することは難しくないかもしれない．また Microsoft Office 2007 以降の Office Open XML 形式にも応用できる可能性がある．その他，圧縮またはエンコードされていても，元のバイナリイメージが取り出せるならば，本システムの方法は有効であると考えられる．

シェルコードを特定するためのエミュレーションの段階で特徴が観測できない検体もあったので，エミュレータの精度を高める必要がある．

少数ではあるがランダムに選ぶよりも効率が悪い検体があった．エントロピーからシェルコードの候補を求める方法では，図 4 より多くは 10%程度以下のエミュレーションでシェルコードを見つけることができる．ゆえに 10%程度を探索してもシェルコードを見つけることができないときには，ファイルの先頭から順番にシェルコードを探すなど，異なる方法に切り替えることも検討できるかもしれない．

本システムが出力した実行可能ファイルの中にはマルウェアとしての動作を確認できない検体もあった．動作を確認できなかった原因を調査し，本システムが作るシェルコードの実行環境を実際のアプリケーションの環境に近づける改良も必要である．

参考文献

[1] 独立行政法人情報処理推進機構：標的型攻撃メールの傾向と事例分析〈2013 年〉，独立行政法人情報処理推進機構（オンライン），入手先 ⟨http://www.ipa.go.jp/files/000036584.pdf⟩（参照 2014-12-10）．

[2] Mandiant: Mandiant APT1: Exposing One of China's Cyber Espionage Units, Mandiant (online), available from ⟨http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf⟩ (accessed 2014-12-10).

[3] Prandini, M. and Ramilli, M.: Return-Oriented Programming, *Security Privacy*, Vol.10, No.6, pp.84–87, IEEE (online), DOI: 10.1109/MSP.2012.152 (2012).

[4] Microsoft: Vulnerabilities in Microsoft Office Could Allow Remote Code Execution (2587634), Microsoft (online), available from ⟨https://technet.microsoft.com/library/security/ms11-073⟩ (accessed 2014-12-10).

[5] Polychronakis, M., Anagnostakis, K. and Markatos, E.: Network-level polymorphic shellcode detection using emulation, *Journal in Computer Virology*, Vol.2, No.4, pp.257–274 (online), DOI: 10.1007/s11416-006-0031-z (2007).

[6] 藤井孝好，吉岡克成，四方順司，松本　勉：エミュレーションに基づくシェルコード検知手法の改善，マルウェア対策研究人材育成ワークショップ 2010 (2010).

[7] 神保千晶，吉岡克成，四方順司，松本　勉，衛藤将史，井上大介，中尾康二：CPU エミュレータと Dynamic Binary Instrumentation の併用によるシェルコード動的分析手法の提案，電子情報通信学会技術研究報告 ICSS，情報通信システムセキュリティ，Vol.110, No.266, pp.59–64（オンライン），入手先 ⟨http://ci.nii.ac.jp/naid/110008152390/⟩ (2010).

[8] Li, W.J., Stolfo, S., Stavrou, A., Androulaki, E. and Keromytis, A.: A Study of Malcode-Bearing Documents, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Hämmerli, B.M. and Sommer, R. (Eds.), Lecture Notes in Computer Science, Vol.4579, pp.231–250, Springer Berlin Heidelberg (online), DOI: 10.1007/978-3-540-73614-1_14 (2007).

[9] 大坪雄平，三村　守，田中英彦：ファイル構造検査による悪性 MS 文書ファイルの検知，情報処理学会論文誌，Vol.55, No.5, pp.1530–1540 (2014).

[10] Rentz, D.: The Microsoft Compound Document File Format, OpenOffice.org (online), available from ⟨http://www.openoffice.org/sc/compdocfileformat.pdf⟩ (accessed 2014-12-10).

[11] Microsoft: Compound File Binary File Format, Microsoft (online), available from ⟨http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMS-CFB%5D.pdf⟩ (accessed 2014-12-10).

[12] Microsoft: Word 2007: Rich Text Format (RTF) Specification, version 1.9.1, Microsoft (online), available from ⟨http://www.microsoft.com/en-us/download/details.aspx?id=10725⟩ (accessed 2014-12-10).

[13] Boldewin, F.: Analyzing MSOffice malware with OfficeMalScanner, www.reconstructer.org (online), available from ⟨http://www.reconstructer.org/papers/Analyzing%20MSOffice%20malware%20with%20OfficeMalScanner.zip⟩ (accessed 2014-12-10).

[14] Fratantonio, Y., Kruegel, C. and Vigna, G.: Shellzer: A tool for the dynamic analysis of malicious shellcode, *Proc. Symposium on Recent Advances in Intrusion Detection* (*RAID*), S. Francisco, CA (2011).

[15] Cova, M., Kruegel, C. and Vigna, G.: Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code, *Proc. World Wide Web Conference* (*WWW*), Raleigh, NC (2010).

[16] Inoue, D., Yoshioka, K., Eto, M., Hoshizawa, Y. and Nakao, K.: Automated Malware Analysis System and Its Sandbox for Revealing Malware's Internal and External Activities, *IEICE Trans. Information and Systems*, Vol.92, No.5, pp.945–954 (online), DOI: 10.1587/transinf.E92.D.945 (2009).

[17] Inoue, D., Yoshioka, K., Eto, M., Hoshizawa, Y. and Nakao, K.: Malware Behavior Analysis in Isolated Miniature Network for Revealing Malware's Network Activity, *IEEE International Conference on Communications, ICC '08*, pp.1715–1721 (online), DOI: 10.1109/ICC.2008.330 (2008).

[18] Yoshioka, K., Inoue, D., Eto, M., Hoshizawa, Y., Nogawa, H. and Nakao, K.: Malware Sandbox Analysis for Secure Observation of Vulnerability Exploitation, *IEICE Trans. Information and Systems*, Vol.92, No.5, pp.955–966 (online), DOI: 10.1587/transinf.E92.D.955 (2009).

[19] 三村　守，田中英彦：Handy Scissors：悪性文書ファイルに埋め込まれた実行ファイルの自動抽出ツール，情報処理学会論文誌，Vol.54, No.3, pp.1211–1219 (2013).

岩本 一樹 （学生会員）

1998 年東京電機大学理工学部情報科学科卒業，2008 年信州大学大学院工学系研究科修士課程情報工学専攻修了，2010 年同大学院総合工学系研究科システム開発工学専攻博士課程入学．1998 年日本コンピュータセキュリティリサーチ株式会社入社，2012 年独立行政法人情報処理推進機構非常勤研究員．2013 年株式会社セキュアブレイン先端技術研究所入社，マルウェア解析と研究開発に従事．1999 年 Anti Virus Asia Researchers（AVAR）会員，2011 年，2012 年 AVAR 理事．電子情報通信学会会員．

和﨑 克己 （正会員）

1991 年信州大学工学部情報工学科卒業，1993 年同大学大学院工学系研究科博士前期課程情報工学専攻修了，1994 年同博士後期課程システム開発工学専攻退学，同年長野工業高等専門学校電子制御工学科助手，1998 年信州大学工学部情報工学科助手，2001 年同大学工学部助教授．現在，信州大学工学部教授．博士（工学）．2003 年，2005 年カナダ・アルバータ州立大学計算科学科客員研究員．2008 年独立行政法人新エネルギー・産業技術総合開発機構（NEDO）技術評価委員．並列分散システムのモデル化と解析，非同期システムの数学モデルと形式検証，モデル検査系向けハードウェアコンパイラに関する研究に従事．IEEE，電子情報通信学会，電気学会，教育システム情報学会各会員．