Vol.2014-CSEC-65 No.13
Vol.2014-IOT-25 No.13
2014/5/22

# 電子文書型マルウェアから シェルコードを抽出する方法の提案

岩本 一樹[1,2]　神薗 雅紀[3,1]　津田 侑[3]　遠峰 隆史[3]　井上 大介[3]　中尾 康二[3]

**概要**：アプリケーションの脆弱性を攻撃する電子文書型マルウェアを動的に解析するためには，該当する脆弱性をも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無くても電子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，電子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．

# Proposal for Shellcode Extraction from Malicious Document File

Kazuki Iwamoto[1,2]　Masaki Kamizono[3,1]　Yu Tsuda[3]　Takashi Tomine[3]　Daisuke Inoue[3]
Koji Nakao[3]

***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.

## 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]．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が添付されている場合が多い．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）は実行環境に依存しないことが多い．そこで本稿は，環境に依存するマルウェアを解析することを目的に，まず文章ファイル内のシェルコードの位置を特定するを目的とする．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ことができるよう実行可能ファイ

[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, Chiyodaku, Tokyo, 102-0083
[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
[3] 独立行政法人 情報通信研究機構
National Institute of Information and Communications Technology, Koganei, Tokyo, 184-8795

ⓒ 2014 Information Processing Society of Japan

ルを出力し，実行できることを検証する．

## 2. システム設計方針

本システムはシェルコードの候補となるファイルオフセット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，そのファイルオフセットをシェルコードとみなす．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オフセットから順にエミュレーションを行うことで，効率よくシェルコードを特定する．

### 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 の電子文書型マルウェアを対象とする．

- Microsoft Office Word（doc 形式）
- Microsoft Office Excel（xls 形式）
- Microsoft Office PowerPoint（ppt 形式）

本システムでは実際に電子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できない．

- ROP（Return Oriented Programming）
- 環境に強く依存する電子文書型マルウェア
- 文書ファイル内部にシェルコードがない電子文書型マルウェア

### 2.2 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オフセットにあるデータを IA-32 のコードとみなしてエミュレータで実行する．仮にそのファイルオフセットがシェルコードの開始位置ならば，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 レジスタ経由でメモリアクセスが発生する
(3) 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．

シェルコードは必要とされる API のアドレスを，FS レジスタを利用して Process Environment Block にアクセスしてシェルコード自身で取得するので，(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，そのファイルオフセットからシェルコードが開始されるとみなす．

### 2.3 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．

#### 2.3.1 CBF 解析

2.1 節のファイル形式はすべて CBF（Compound Binary File）形式 [2] である．CBF 形式のファイルは Header，DiFAT，FAT，Mini FAT，Directory，Stream，Mini Stream，Free の各要素に分類できる．本システムでは CBF 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ファイルオフセットの候補とする．

なお上記の CBF 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．

#### 2.3.2 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，対象となるファイルオフセット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行わずに，そのファイルオフセットにはシェルコードが存在しないとみなす．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．

#### 2.3.3 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．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．

例えば**表 1** ではファイルオフセットの 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

ⓒ 2014 Information Processing Society of Japan

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 Pooint

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5DD | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 5DE | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 5DF | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 5E0 | 60 | B9 | A4 | 05 | 00 | 00 | EB | 0D | 5E | 56 | 46 | 8B | FE | AC | 34 | FC |
| 5E1 | AA | 49 | 75 | F9 | C3 | 90 | E8 | ED | FF | FF | FF | 61 | 15 | C1 | FE | FC |
| 5E2 | FC | AA | CF | 3C | 98 | 77 | BC | CC | 77 | BC | F0 | 77 | 8C | E0 | 51 | 77 |

表 2　ファイルの種類
**Table 2**　File Type

| File Type | Number |
|---|---|
| doc | 43 |
| ppt | 4 |
| xls | 26 |

表 3　脆弱性
**Table 3**　Vulnerability

| Name | Number |
|---|---|
| CVE-2006-2389 | 4 |
| CVE-2006-2492 | 13 |
| CVE-2006-6456 | 2 |
| CVE-2007-0671 | 1 |
| CVE-2008-2244 | 5 |
| CVE-2008-4841 | 1 |
| CVE-2009-0556 | 1 |
| CVE-2009-0563 | 1 |
| CVE-2009-3129 | 24 |
| CVE-2010-0822 | 2 |
| CVE-2011-1269 | 3 |
| CVE-2012-0158 | 13 |
| UNKNOWN | 3 |

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ファイルオフセット」または「エントロピーの差が大きいファイルオフセット」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ファイルオフセットから順にエミュレーションを行う．

## 3. 事前調査

シェルコードのファイルオフセット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 2.1 節の条件に合致する**表 2** のハッシュ値が異なる 73 の電子文書型マルウェアを準備した．これらの検体は既にシェルコードのファイルオフセットは特定できている．これらの検体が対象とする脆弱性は**表 3** のとおり 12 種類あり，3 つファイルは脆弱性の詳細等が不明である．脆弱性が異なるかシェルコードのファイルオフセットが異なるユニークな検体は 37 種類ある．

表 4　観測された特徴
**Table 4**　Extracted Feature

| Feature | Number |
|---|---|
| (1)Self-modifying, (2)PEB access, (3)API call | 45 |
| (1)Self-modifying, (2)PEB access | 2 |
| (2)PEB access, (3)API call | 13 |
| None | 13 |

表 5　最大ステップ数
**Table 5**　Maximum Step

| Feature | Step |
|---|---|
| Start to (1)Self-modifying | 35,847 |
| Start to (2)PEB access<br>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 CBF の要素

検体セットのシェルコードのファイルオフセットが CBF 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

2.2 節のシェルコードの特徴を観測するためには，エミュレータで実行する十分なステップ数を測定する必要がある．そのため解析済みの検体のシェルコードのファイルオフセットからエミュレータで実行し，2.2 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 4** と**表 5** の結果になった．特徴が観測できなかった検体が 14 あるが，それらで脆弱性が異なるかシェルコードのファイルオフセットが異なるユニークな検体は 6 種類であった．

### 3.4 エントロピー算出対象のバイト数

エントロピーを求める場合，ファイルの中のどの程度の長さのバイト列からファイルオフセットのエントロピーを算出するのが問題となる．そこで最も適切なバイト数を求めるために 128 バイトから 2,048 バイトで検体セットのファイルオフ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ファイルオフセットの前の範囲との差とする．実装上はファイルの範囲を越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，すべてのファイルオフセットのエントロピーを求めると時間がかかりすぎるので，16 バイト毎にエントロピーを算出した．

**表 6** 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，最も少ない回数でシェルコードを見つけることができた件数をエントロピー算出対象のバイト数ごとにまとめた．**表 7** では，検体セットの検体に対して

ⓒ 2014 Information Processing Society of Japan

表 6　最小エミュレーション回数の件数
**Table 6**　Minimum Number of Emulation Trials

| Size | Entropy Order | Delta Order | Total |
|---|---|---|---|
| 128 | 10 | 19 | 29 |
| 192 | 6 | 6 | 12 |
| 256 | 5 | 3 | 8 |
| 384 | 6 | 2 | 8 |
| 512 | 18 | 17 | 35 |
| 1,024 | 13 | 22 | 35 |
| 1,536 | 11 | 1 | 12 |
| 2,048 | 4 | 3 | 7 |

表 7　エミュレーション回数の比率の平均
**Table 7**　Avarage of Ratio of Emulation Trials

| Size | Entropy Order | Delta Order |
|---|---|---|
| 128 | 0.604 | 0.107 |
| 192 | 0.209 | 0.050 |
| 256 | 0.127 | 0.028 |
| 384 | 0.089 | 0.020 |
| 512 | 0.074 | 0.018 |
| 1,024 | 0.063 | 0.016 |
| 1,536 | 0.064 | 0.024 |
| 2,048 | 0.065 | 0.039 |

「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ランダムにファイルオフセットを選びシェルコードのファイルオフセット特定を試みた場合の試行回数の期待値の比率をエントロピー算出対象のバイト数ごとにまとめた．

### 3.5　アルゴリズムの比較

エントロピー算出対象のバイト数が 512 バイトと 1,024 バイトのときに最も試行回数が少なかった．そこで 512 バイトのときに，「エントロピーが高い順」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 1** に示す．比率が小さいほど効率が良いく，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言える．同様に 1,024 バイトのときに，「エントロピーの差が大きい順」の試行回数と期待値の比率の分布を**図 2** に示す．

シェルコードを探索するとき「エントロピーの差が大きい順」の方が試行回数が少なかったのは 37 件，「エントロピーが高い順」の方が試行回数が少なかったのは 32 件，同一であったのは 4 件であった．また図 2 より，エントロピーの差をとる方法ではランダムに選ぶよりも効率が悪くなる件数は少なかった．

## 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では書き換えたメモリの実行が観測されたときにはエミュレーションを 16,384 ステップ延長し，Process Environment Block 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 節および 3.5 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 1,024 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最も効率が良い方法を実装する．

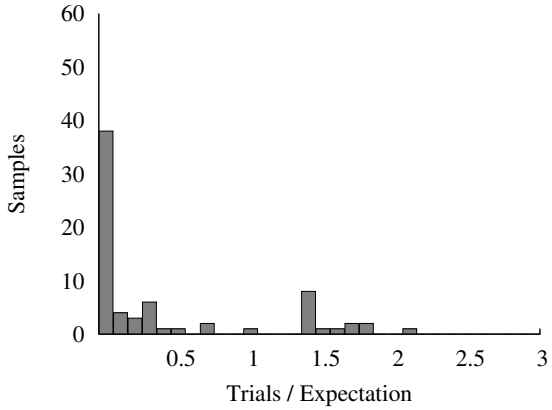


図 1　エントロピーが高い順の試行回数の比率
**Fig. 1**　Ratio of Number of Trials (Entropy Order)

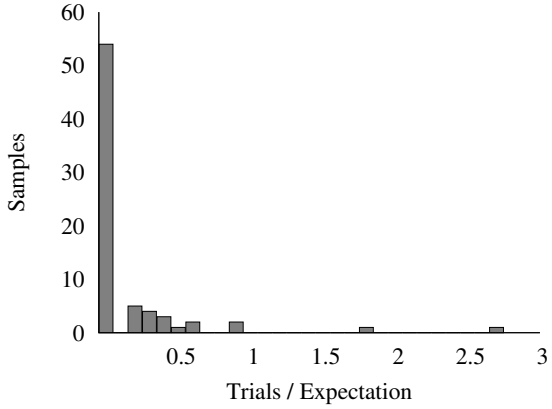


図 2　エントロピーの差が大きい順の試行回数の比率
**Fig. 2**　Ratio of Number of Trials (Delta Order)

### 4.1　CBF 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 CBF 解析を行ったところ，Stream 領域はファイル全体の 30.83%，Stream 領域で逆アセンブルできたのは 96.82%であった．ファイル全体に対して 29.85%がエミュレーションを試行する対象となった．

### 4.2　False Positive 検査

3 節の検体セットとは別に，正常なファイルを 100 種類（doc:50，ppt:25，xls: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．

ⓒ 2014 Information Processing Society of Japan

### 4.3 シェルコード抽出

本システムで検体セットの検体に対してシェルコードの抽出を試みたところ，61 の検体でシェルコードのファイルオフセットを特定することができた．

## 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や実行できない検体もあった．

シェルコード特定のためのエミュレータでの実行は最初の API 呼び出しで打ちきられるが，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．そのためシェルコードのファイルオフセットが特定できても，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等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．

### 5.1 実行速度

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，あるいはコンパイル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シェルコードのファイルオフセットの優先順位を決定した．図 2 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のファイルオフセットを特定できることがわかった．しかし少数ではあるが期待値を越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）検体もあった．

### 5.3 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オフセットが格納されている．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ファイルオフセットから実行する．また実行可能ファイルは GetCommandLine，GetModuleFileName をフックして自身の名称を文書ファイルを開くアプリケーションに偽装する．

表 8　出力されたファイルの実行結果
**Table 8**　Result of Execution

| | | |
|---|---|---|
| Success | Drop | 20 |
| | Communication | 1 |
| Failture | Memory | 22 |
| | Instruction | 2 |
| | Unknown | 3 |
| Infinity Loop | | 13 |

#### 5.3.1 出力されたファイルの実行結果

本システムでシェルコードのファイルオフセットを特定した 61 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 8** 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場合，またはネットワークに接続を試みたときにはマルウェアとして動作したとみなした．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ときには失敗したとみなした．またループから抜け出せなくなっている検体もあった．

### 5.4 検体セットの問題

本システムを構築するために 3.1 節に示す 73 の検体を準備したが，その不備が本システムを構築することで見つかった．

#### 5.4.1 解析者の間違い

解析者が特定したファイルオフセットからシェルコードをエミュレータで実行した結果が表 4 であり，合計で 60 の検体でシェルコードの特徴が観測された．一方，4.3 節の実際に本システムを実行した結果では，61 の検体でシェルコードのファイルオフセットを特定している．1 検体多く本システムがシェルコードを特定している理由は，解析者によるシェルコードの特定が間違っていたためである．該当検体では，3.3 節で誤ったファイルオフセットからシェルコードのエミュレーションを開始したため特徴を観測できなかった．

今回は解析者の間違いを本システムで発見することができた．

#### 5.4.2 破損した検体

図 8 ではシェルコードが想定するファイルのサイズと実際のファイルサイズが一致しないために無限ループに陥っ

ⓒ 2014 Information Processing Society of Japan

た検体があった．ファイルサイズがシェルコードの想定よりも短いことがファイルサイズが一致しない原因であった．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言える．

## 6. 関連研究

鵜飼らは解析ツールの 1 つとしてシェルコード展開ツール [3] を作成している．本システムによるシェルコード特定と同じ方針ではあるが，このツールではファイルの構造に関係なく先頭からシェルコードを探していることと，シェルコードのエミュレーションを行っていないところが異なる．

藤井らはシェルコード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方法を提案 [4] 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 [5]．これらは通信データのシェルコードに対する提案であるが，本システムはこれを電子文書型マルウェアに応用していると言える．

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 [6] を提案している．本システムと同様に，脆弱性をもつアプリケーションを準備する必要はない．この方法では，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．しかし提案された方法では．実行可能ファイルを内包していない電子文書型マルウェアは対象外であり，またシェルコードそのものの解析を行うことはできない．

## 7. 今後の課題

本システムにより，電子文書型マルウェアが対象とする脆弱性をも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．本システムが電子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．

本システムは電子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電子文書型マルウェアが，電子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わからないが，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電子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電子文書型マルウェアを増やすことも可能である．CBF 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 RTF や 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．エミュレータの精度を高める必要がある．

本システムが出力した実行可能ファイルの中には実行できない検体が多数あった．実行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．

また，本システムによる電子文書型マルウェアの動的解析の結果を従来の動的解析によるマルウェア対策に連携させて，今後のマルウェア対策に生かしていきたいと考えている．

### 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 2014-3-25）．

[2] 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 2014-3-25).

[3] 鵜飼裕司，小林偉昭，　中野学：脆弱性を利用した標的型攻撃のための解析ツール，マルウェア対策研究人材育成ワークショップ 2008 (2008).

[4] 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).

[5] 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).

[6] 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).

ⓒ 2014 Information Processing Society of Japan