博士論文

ファイルフォーマットからの逸脱に着目した悪性文書ファイル検知方式の研究

Yuhei OTSUBO

大坪　雄平

情報セキュリティ大学院大学

情報セキュリティ研究科

情報セキュリティ専攻

2016年9月

目　　次

頁

第 1 章

序　論

はじめに，研究の背景を述べ，目的と範囲を朗らかにした後，方針及び論文の構成について述べる．

1.1　標的型メール攻撃の概要

近年では，特定の組織や個人を狙って情報摂取等を行う標的型攻撃の脅威が顕在化している．2015 年には，多くの組織で標的型メールを受信するだけでなく，マルウェア感染による情報漏えいにまで至っていることが発覚し，大きな社会問題となった [1]．

標的型メール攻撃の流れを，図 1.1 に示す．標的型メール攻撃では，受信者が不審に思わないような件名および本文の電子メールにマルウェアが添付されることが多い．受信者が不審に思わずに添付ファイルを開封すると，端末がマルウェアに感染する．この場合のマルウェアは，RAT（Remote Access Trojan または Remote Access Tool）と呼ばれる遠隔操作を目的としたマルウェアであることが多い．端末が，攻撃者により遠隔操作され，端末・システム内に保存された情報が外部に送信される．

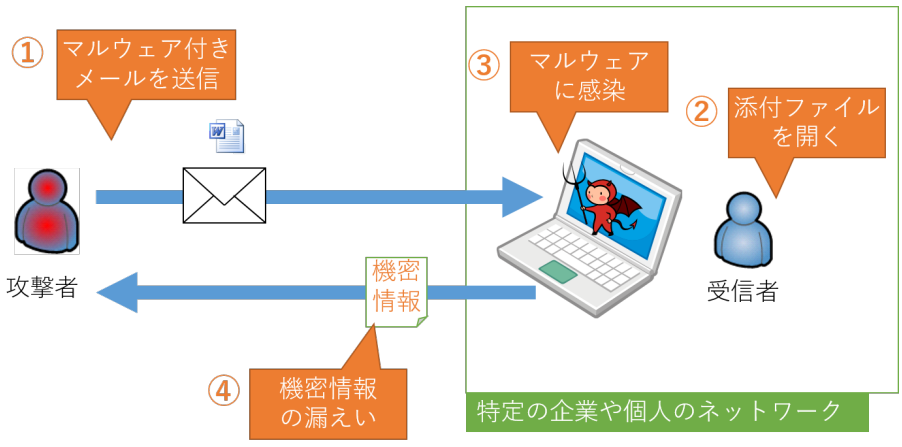

図 1.1　標的型メール攻撃の流れ

1.2　標的型メール攻撃の気づきにくさ

標的型メール攻撃では，受信者が不審に思わないような件名および本文の電子メールが送信されることが多い．さらに，図 1.2 に示すように，実際に業務でやり取りしているメールの情報が，何らかの理由により攻撃者に漏えいし，標的型メールに悪用されることがある．図の例では，実際のメールから 10 時間後には，本文等を盗用した標的型メールが送信されている．このようなメールを送信されると，メールの本文，件名，添付ファイル名等から受信者が標的型メールに気づくことは非常に困難になる．

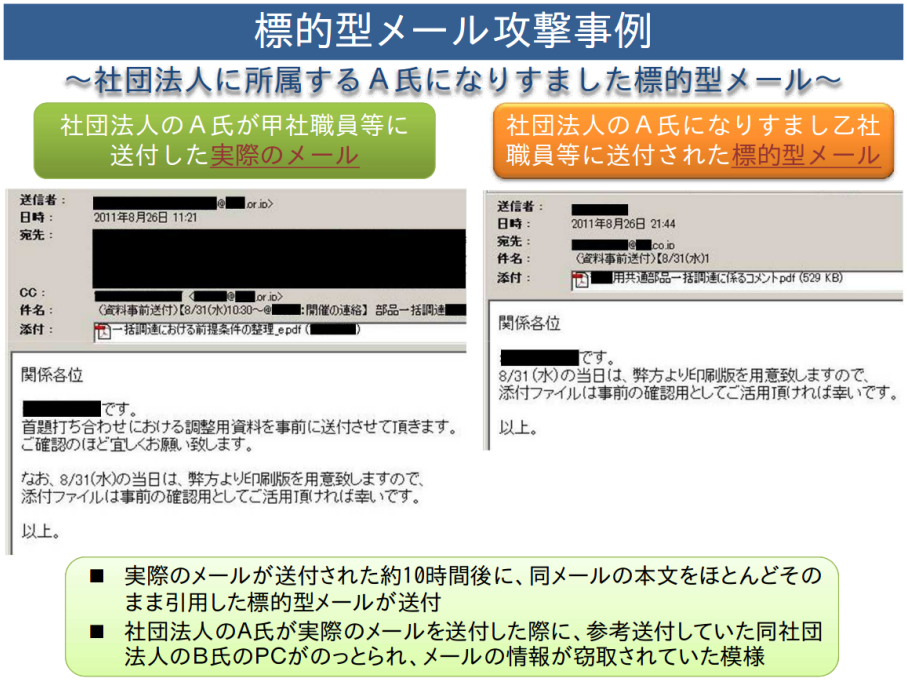

図 1.2　標的型メールの例．警察庁発表資料 [2] より引用

標的型メールに添付されるマルウェアのほとんどは新種のマルウェアであり，メールを受信した時点では，パターンマッチング方式のウイルス対策ソフトでマルウェアを検知することは困難である．2014 年の標的型メール攻撃における添付ファイルの傾向を見ると，国内では実行ファイルが約 7 割で，残りの約 3 割が文書ファイルとなっている [3]．一方，海外も含めると，約 6 割が文書ファイルとなっている [4]．添付されたファイルが実行ファイルであった場合，拡張子を見れば受信者が不審に感じる可能性もある．しかしながら，添付ファイルが文書ファイルであった場合は，ウイルス対策ソフトでも検知できない以上，受信者が添付ファイルを開くことなくマルウェアに気づくのは困難な状況となっている．

1.3　研究の対象と研究の目的

本研究は，標的型攻撃における被害を抑止し，情報システムのセキュリティを向上させることを目的とする．本研究では，標的型攻撃における初期段階のメールを用いた攻撃に着目する．標的型メールに添付されるマルウェアが悪性文書ファイルの場合，受信者側で拡張子などの特徴から不審な添付ファイルと気づくことが困難な状況であることから，悪性文書ファイルを研究対象とする．そのような状況でも悪性文書ファイルを高確率で検知できる手法を提案する．

1.4　研究対象とする悪性文書

本研究の対象とする文書ファイルの分類を図 1.3 に示す．文書ファイルは，悪性（malicious）なものと無害（benign）なものに分類される．本論文の対象とする悪性文書ファイルは，ソフトウェアの脆弱性（プログラムの不具合や設計上のミスが原因となって発生した情報セキュリティ上の欠陥 [5]）を攻撃する exploit が含まれている文書ファイルを悪性文書ファイルとする．したがって，本文中のハイパーリンクのクリックや，埋め込みオブジェクトのダブルクリックなどの閲覧者による明示的な動作が必要なものや，動作にマクロを利用したものは，ここでは対象としない．

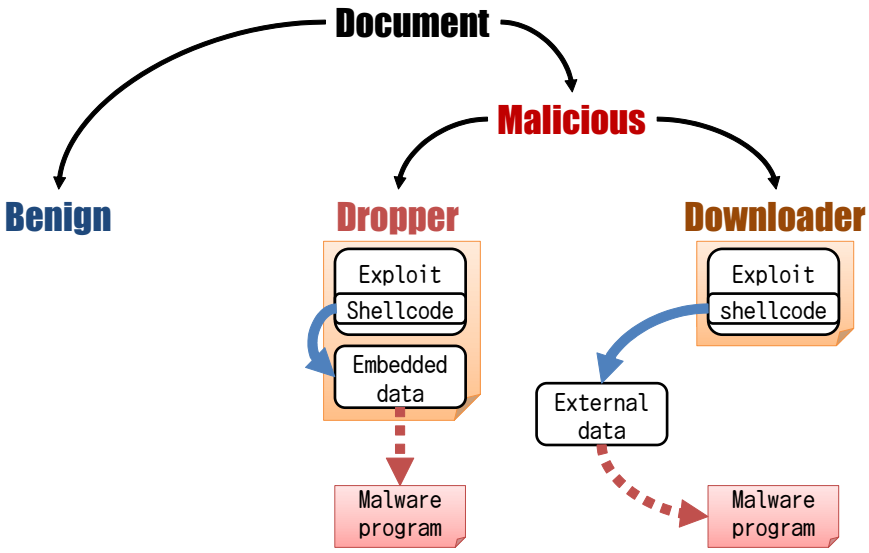

図 1.3　悪性文書ファイルの分類

さらに，悪性文書ファイルを，dropper と downloader に分類する．

1.4.1 dropper

dropper は，感染活動に必要な実行ファイルなどのデータを自身のファイルから取り出すものである．dropper の多くは，自身のファイルから実行ファイル（exe や dll）を端末上に取り出して実行するものであるが，本論文で言う dropper には，実行ファイルを作成せずに，自身のファイルから攻撃コードをメモリ上にだけ展開して実行するものも含まれる．

典型的な dropper 型の悪性文書ファイルの構造及び動作を図 1.4 に示す．悪性文書ファイルを開くと，閲覧ソフトの脆弱性を攻撃する exploit と呼ばれる部分が動作し，shellcode（脆弱性を利用してコンピュータに実行させる任意のコードの総称）が実行される．shellcode は文書ファイルに埋め込まれた実行ファイル（exe や dll）やダミー表示用の文書ファイルを取り出し，実行ファイルを実行したりダミー表示用の文書ファイルを表示したりする．これによって悪性文書ファイルを閲覧した端末はマルウェアに感染する．

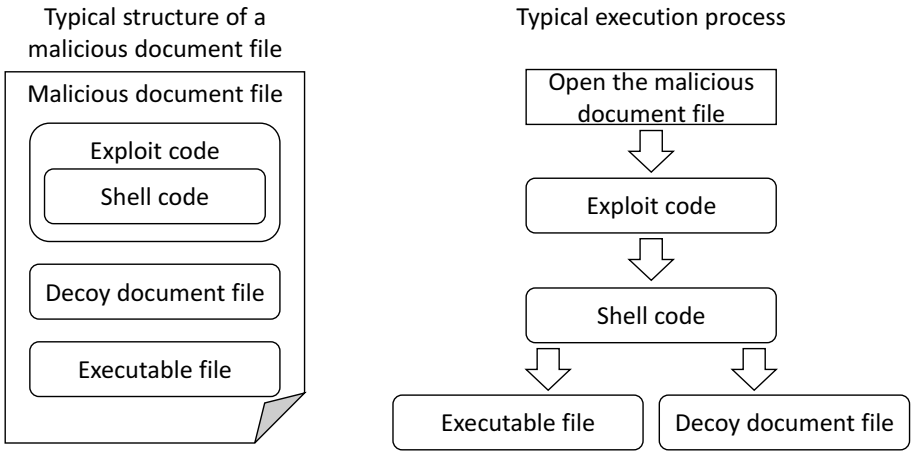

図 1.4　悪性文書ファイルの構造及び動作

1.4.2 downloader

downloader は，必要なデータを外部のサーバから取り出すものであり，それ以外については dropper と同じである．

1.5 論文の構成

本論文では，まず第 2 章において，悪性文書ファイルの検知に関連する先行研究について整理する．第 3 章において，悪性文書ファイルの傾向について調査を行い，本研究の目的である標的型攻撃対策に資する研究対象の絞り込みを行う．第 4 章では，標的型メール攻撃に使用される悪性文書ファイルの特徴を整理し，

ファイルフォーマットからの逸脱に着目した検知方式を提案する．第5章では，3種類のファイルフォーマットごとに，仕様の概要を整理し，悪性文書ファイルの仕様からの逸脱とその検知方法について明らかにする．第6章では，第5章で明らかになった8種類の仕様からの逸脱構造を検知するために開発した “o-checker” というプログラムの概要および使用方法について述べる．第7章では，4種類のデータセットを使用し，o-checker の検知率等の実験を行う．第8章では，実験結果を踏まえ，本研究の提案の効果，限界等について考察し，第9章でまとめる．

第2章

関連研究

悪性文書フィルの検知を目的とした先行研究は，動的解析を用いるものと静的解析を用いるものの大きく2つに分類することができる．動的解析を行うためには，解析対象のマルウェアが動作する環境を準備する必要がある．解析対象が文書ファイルの場合，マルウェアの本体である実行ファイルが動作する環境を準備することに加え，悪性文書ファイルが利用する脆弱性が動作する環境を準備する必要がある．OSや閲覧ソフトのバージョン，適用しているパッチの種類など組み合わせは膨大であり，全てに対応させる場合には膨大な環境を準備する必要がある．

一方，静的解析は，exploitを含む不正なコードを動作させずに解析する．静的解析の場合，解析環境がマルウェアの動作する環境に依存しないという特徴がある．本論文では，静的解析に着目し，その新たな検知方式を提案する．静的解析によって悪性文書ファイルを検知する既存の手法を以下に示す．

2.1 パターンマッチングを用いる検知手法

パターンマッチングを用いる検知手法は，検査対象のプログラムとパターンファイル（マルウェアの特徴を記述したデータベース）をマッチングすることでマルウェアを検知する手法である．既知のマルウェアに対し高い検知率を示し，誤検知も少ない．しかしながら，2015年では，1日あたり平均30万個以上新たなマルウェアが作成されている[6]．攻撃者は，マルウェア作成ツールを使用し，ウイルス対策ソフトで検知できないマルウェアを簡単に作ることができることから，攻撃の度に新たなマルウェアを作成していると思われる．このような状況で，パターンマッチングを用いる検知手法では，未知のマルウェアを用いた攻撃に先行して対応することは困難である．

2.2 exploitに着目した検知手法

RTF（Rich Text Format：rtf拡張子）[7] ファイルおよびCFB（Compound File Binary：doc，xls，ppt，jtd拡張子など）[8] ファイル専用の解析ツールであるOfficeMalScanner[9](OMS)は，RTFファイルおよびCFBファイルから不正なコードによく利用されるコードを検索したり，文書ファイルに埋め込まれた実行ファイルやダミー表示用の文書ファイルのヘッダに使われる文字列を検索するこ

とにより悪性文書ファイルを検知することができる．

文献 [10] では，PDF（Portable Document Format：pdf 拡張子）[11] ファイルの中に含まれる JavaScript を対象に機械学習を適用することで，不審な PDF ファイルを高速に検知する手法が提案されている．

文献 [12] では，PDF ファイルの中に含まれる不正な JavaScript の分類および検知に利用可能な特徴点を抽象構文解析木を用いて抽出する手法が提案されている．この手法は HTML ファイルに埋め込まれた JavaScript を対象としているが，exploit に着目しており，悪性 PDF ファイルの exploit に利用された JavaScript の分類および検知にも適用可能であると考えられる．

一方，文書編集ソフトの脆弱性は様々なものがあり [13–17]，毎年のように新たな脆弱性が発見されている．このため，exploit に利用されるものは JavaScript を始め Flash，フォント，画像等があり，それぞれに対応した検知手法を検討する必要がある．さらに，exploit 部分はパターンマッチングによる検知から回避するためエンコードされていることが多い．したがって，そのデコード手法についても検討する必要がある．

2.3 shellcode に着目した検知手法

OMS は，既に述べたとおり，RTF ファイルおよび CFB ファイルから不正なコードによく利用されるコードを検索することができる．

文献 [18],[19] では，バイナリデータの統計分析に加え，動的解析を組合せて文書ファイルに隠された shellcode 等の不正なコードを検出する手法が提案されている．

しかしながら，前述したとおり，exploit 部分はエンコードされていることが多く，expoit が内包する shellcode もエンコードされていることが多い．したがって，デコード手法についても検討する必要がある．

文献 [20] は，PDF ファイルを構文解析して JavaScript を抜き出し，得られた JavaScript を動的解析することによって shellcode を検知する MDScan というツールが提案されている．この手法は，exploit に利用されている JavaScript がエンコードされていても静的解析と動的解析を組み合わせることで検知をすることができる．しかしながら，この手法は，exploit に JavaScript を用いているものを対象としており，悪性 PDF ファイルが exploit に利用するものは JavaScript 以外にもたとえば Flash，フォントや画像等があり，それぞれに対応した検知手法を検討する必要がある．

2.4 実行ファイルに着目した検知手法

OMS は，既に述べたとおり，RTF ファイルおよび CFB ファイルに埋め込まれた実行ファイルのヘッダに使われる文字列を検索することができる．

文献 [21] では，バイナリデータの値を統計的に分析をすることによって，文書ファイルに隠された実行ファイル等の不正なコードを検出する手法が提案されている．

文献 [22] では，様々な形式の悪性文書ファイルに埋め込まれた実行ファイルを自動的に抽出する Handy Scissors（HS）というツールが提案されている．この手法では，実行ファイルを埋め込む際に使用される様々なエンコード方式を自動的に解読し，実行ファイルに頻出する文字列を検索することで実行ファイルを抽出することができる．

OMS は，単純な 1 byte 鍵の XOR 方式に，HS は複数のエンコード方式に対応している．しかしながら，新たなエンコード方式が現れるたびに，その解読方法を検討しなければならないという課題がある [23].

2.5 文書ファイルの構造に着目した検知手法

これまでに述べたエンコードによる検知回避の課題を解決する検知手法として，文書ファイルの構造に着目した手法がある．exploit，shellcode，実行ファイルなどの攻撃者によってエンコードされる部分以外を対象に，文書ファイルの構造を解析することで，エンコードを用いる検知回避に対応することができる．

文献 [24] では，PDF を開いた際に JavaScript などの命令を実行させるフィルタや外部のアプリケーションを起動するアクションなど，潜在的に危険なアクションを伴うフィルタを対象に機械学習を適用することで，不審な PDF ファイルを高速に検知する手法が提案されている．

文献 [25] は PDF のタグの数や位置，オブジェクトの大きさなど，202 種類の特徴を対象に機械学習を適用することで，不審な PDF ファイルを高確率で検知する手法が提案されている．

文献 [26] では，PDF のドキュメント階層構造を対象に機械学習を適用することで，不審な PDF ファイルを高速に検知する手法が提案されている．

これらは，機械学習等の学習サンプルの必要な方式である．これらの方式は，検知の基準を学習サンプルをもとに自動的に設定するため，悪性文書ファイルの傾向が将来変わったとしても，新たな学習サンプルを準備すれば対応でき得るという柔軟性がある．しかしながら，学習するためのサンプルを集める必要があるだけでなく，その検知精度は学習のサンプルに依存する．標的型攻撃に使用された文書ファイルには，ダミー文書内に対象組織の機微な情報が含まれていたり，埋め込まれた実行ファイルの中に組織内のシステムの設定情報が含まれていたりすることがあり，この点が情報共有の障壁となっている．したがって，機械学習等を標的型攻撃対策に適応させるには，学習用サンプルをいかに集めるかという課題がある．

文献 [27] では，CFB ファイルを対象に，文書ファイルの構造を検査すること

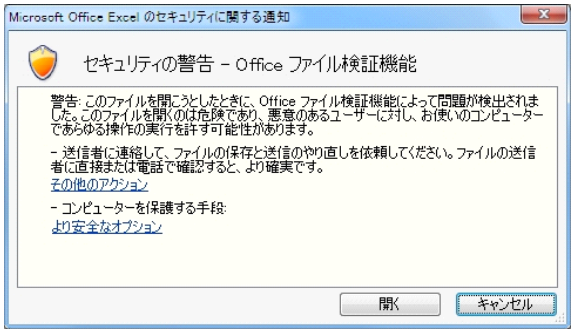

図 2.1　Office ファイル検証機能

により，文書ファイルに埋め込まれた，表示内容と関係のないデータを解析するツールが提案されている．このツールは CFB ファイル内でデータが秘匿される可能性がある 4 種類の場所を表示する．表示内容と関係ないデータがある文書ファイルは，不審である可能性が考えられる．しかしながら，通常の文書ファイルにも表示内容と関係のないデータが存在する．例えば，CFB のファイル構造は，ある固定のサイズ単位で分割して管理されており，論理的に割り当てる領域と実際に使用する領域に差分がある場合，未使用の領域が発生する．これは表示内容と関係のないデータである．このことから，仮にこのツールを悪性文書ファイル検知に活用した場合，悪性文書ファイルだけでなく通常の文書ファイルも検知してしまうという課題がある．

Microsoft 社が Microsoft Office 2010 から導入した Office ファイル検証機能 [28] は，Office ファイル（xls，doc，ppt，pub 拡張子）を開こうとした場合に，正常なファイル構造の Office ファイルか否かを検証し，正常なファイル構造でない場合に閲覧するか否か確認するポップアップウインドウが表示される機能である（図 2.1）．この Office ファイル検証機能がどのような仕組みでファイル構造を検証しているかは不明であるため，本論文では検知率を調査することで本論文による提案との差異を間接的に評価する．

第3章

悪性文書ファイルに対する予備調査

本章では，標的型攻撃に使用されたものを中心に悪性文書ファイルの傾向について調査を行う．さらに，標的型メール攻撃による感染を抑止するという本研究の目的を効率的に達成するため，調査結果を元に研究対象とする悪性文書ファイルの絞り込みを行う．

3.1 調査対象のデータセット

3.1.1 tar(09–12)

このデータセットは，2009年1月から2012年12月までの間に日本の複数の組織に送付された標的型メールから採取した悪性文書ファイルである．特定の脆弱性の種類，検知名，RATの種類等について，同一のものが多数含まれるといった検体の偏りが生じるのを防ぐため，検体の採取期間に標的型攻撃に用いられたメールとして提供を受けたもの全てから添付ファイルを取り出し，拡張子が文書ファイルのものを機械的に選定した．ただし，拡張子と実際の中身が一致していないものがあり，それらについては実際の中身に基づいて分類した．その上で同一ハッシュ値を持つものは取り除いた．

このデータセットの概要を表3.1に示す．表中の括弧内の数値はdownloaderの内数を示している．

表 3.1 Summary of the tar(09–12).

Type	Ext.	Num.	Avg. Size (KB)
RTF	rtf	99 (1)	266.5
CFB	doc	36 (0)	252.2
	xls	48 (0)	180.4
	jtd/jtdc	17 (0)	268.5
PDF	pdf	170 (7)	351.2
Total	Num.	370 (8)	291.8

3.1.2 mal(con)

このデータセットは，マルウェアダンプサイト contagio で調査・研究用に公開された悪性文書ファイルである [29]．このデータセットの収集方法の詳細は不明であるものの，そのほとんどは標的型攻撃によらないものである．

このデータセットの概要を表 3.2 に示す．表中の括弧内の数値は downloader の内数を示している．

表 3.2 Summary of the mal(con).

Type	Ext.	Num.	Avg. Size (KB)
PDF	pdf	11,101 (10,964)	26.5

3.2 データセットとしての tar(09–12) の有効性

2012 年の tar(09–12) のうち dropper が使用する脆弱性を表 3.3 に示す．なお，1 つの検体で複数の脆弱性を使用しているものがあるため，表の「Rate」欄の合計は 100 %にならない．

表 3.3　2012 年の tar(09–12) のうち dropper が利用する脆弱性

Type	Ext.	Vulnerability	Num.	Rate
RTF	rtf	MS10-087[30]	31 / 70	44.3 %
		MS12-027[31]	39 / 70	55.7 %
CFB	doc	MS12-027[31]	6 / 23	26.1 %
		APSB11-07[32]	1 / 23	4.3 %
		APSB12-03[33]	1 / 23	4.3 %
		APSB12-18[34]	10 / 23	43.5 %
		APSB12-22[35]	5 / 23	21.7 %
CFB	xls	MS09-067[36]	8 / 18	44.4 %
		MS11-021[37]	2 / 18	11.1 %
		MS12-027[31]	6 / 18	33.3 %
		APSB11-07[32]	1 / 18	5.6 %
		APSB12-03[33]	1 / 18	5.6 %
PDF	pdf	APSB09-04[38]	2 / 15	13.3 %
		APSB10-02[39]	1 / 15	6.7 %
		APSB10-07[40]	1 / 15	6.7 %
		APSB10-21[41]	8 / 15	53.3 %
		APSB11-08[42]	10 / 15	66.7 %
		APSB11-30[43]	1 / 15	6.7 %

これらの脆弱性は 2012 年に発生した悪性文書ファイルを用いた標的型攻撃で悪用された脆弱性として報告されたもの [44],[45] すべてを包含している．このことから，tar(09–12) を調査することにより，2009 年から 2012 年の間にに日本国内で発生した標的型メール攻撃の傾向を分析できることが期待できる．

3.3 悪性文書ファイルの形式

tar(09–12)，mal(con) および 2014 年の標的型攻撃に用いられた悪性文書ファイルの傾向 [3],[4] を見ると，RTF（Rich Text Format：rtf 拡張子）[7]，CFB（Compound File Binary：doc，xls，ppt，jtd 拡張子など）[8] および PDF（Portable Document Format：pdf 拡張子）[11] の 3 種類の形式が，悪性文書ファイルのほとんどを占めていた．

近年，利用が広がっている文書ファイルの形式として OOXML（Office Open XML；docx，xlsx，pptx 拡張子など）[46] がある．現時点では，マクロを用いたバラマキ型の攻撃は確認されているものの，本論文の対象である脆弱性を攻撃する悪性文書ファイルの例が少ない．このため，OOXML 形式の悪性文書ファイルを対象とした検知手法を提案したとしても，その有効性について議論を行うことが難しい．

したがって，本論文の研究対象は，RTF，CFB および PDF の 3 種類の形式とする．OOXML 形式の悪性文書ファイルについては，今後 OOXML 形式の悪性文書ファイルを用いた標的型攻撃が増加した場合に，その対応を検討することとする．

3.4 拡張子偽装と閲覧ソフトの動作

Windows の場合，拡張子はファイルの種類を判別する際に使用され，ファイルを開く際に拡張子に対応したアプリケーションが動作することになっている．したがって，Windows の場合は，拡張子と実際のファイル形式（ファイルに記録されたデータ）が一致していることが期待されている．しかしながら，拡張子は容易に変更することが可能であり，拡張子と実際のファイル形式が一致していない場合のアプリケーションの動作については閲覧ソフト開発ベンダー側に委ねられている．

多くのベンダーは閲覧ソフトの利便性を向上させるため，拡張子と実際のファイル形式にズレがあった場合でも，ヘッダ等の情報からファイル形式を推測し，対応しているファイル形式であれば，問題なく動作するように閲覧ソフトを設計する．攻撃者はこの仕組みを悪用した攻撃を行っている．

tar(09–12) を調べたところ，370 個の検体のうち 94 個の検体でファイルのヘッダから判断した実際のファイル形式と拡張子が一致しないものがあった．それらはすべて，実際のファイル形式は RTF であるにもかかわらず，拡張子を “doc” に偽装していた．Microsoft Word を標準の設定でインストールしている場合，拡張子が “rtf” であっても “doc” であっても，Word が起動する．したがって，Word の脆弱性を突く悪性 RTF ファイルの動作には，拡張子の偽装は必ずしも必要ではない．それにも関わらず，攻撃者が拡張子を偽装するのは，OOXML が Word の標準形式に採用されるまで “doc” 拡張子が Word の標準形式であり，“rtf” 拡

張子よりも一般に馴染みのある “doc” 拡張子を使用することで，攻撃に気づかせにくくする意図があると推測される．

表 3.4 は，拡張子と実際のファイル形式の組み合わせで Microsoft Word 2016 がどのように動作するか示している．表の「○」は，何の支障もなく閲覧可能であることを示している．拡張子が，“rtf” および “doc” の場合，Word 2016 は実際のファイル形式に沿った動作を行う．拡張子と実際のファイル形式にズレがあった場合でも，何の警告も出ないため，利用者が拡張子と実際のファイル形式とのズレを意識することはない．一方，拡張子が “docx” であった場合，拡張子と実際のファイル形式が一致していない限り，Word 2016 は動作を停止する（図 3.1 および図 3.2）．

表 3.4　拡張子偽装に対する Word 2016 の動作.

	Filename Extension		
Real File Type	docx	doc	rtf
OOXML (docx)	○	○	○
CFB (doc)	図 3.1	○	○
RTF (rtf)	図 3.2	○	○

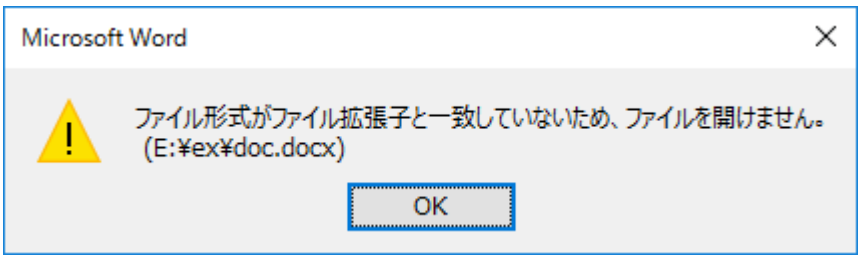

図 3.1　docx 拡張子の CFB ファイルを開いた際の Word 2016 の動作

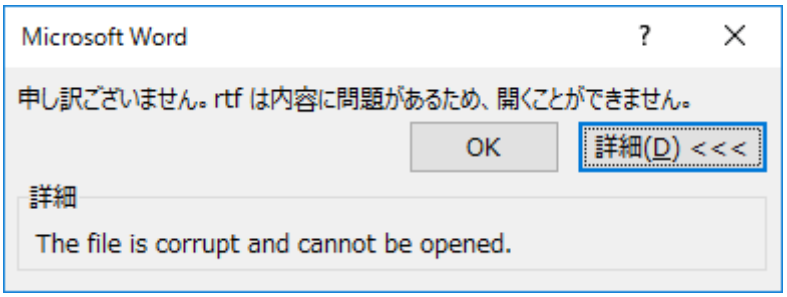

図 3.2　docx 拡張子の RTF ファイルを開いた際の Word 2016 の動作

Word 2016 の動作を見ると，“docx” 拡張子の場合，拡張子と実際のファイル形式が一致しているか厳密にチェックしているにも関わらず，“rtf” 拡張子および “doc” 拡張子の場合はこのズレを許容していることがわかる．拡張子偽装を使った攻撃を受けるリスクと Word の過去のバージョンから挙動を変更した場合のリスクを比較した結果，Microsoft 社が互換性を優先したため，このような実装になっているものと推測される．

このように，Word のような広く一般に普及したソフトウェアの場合，セキュリティを向上させる技術的な解決策があったとしても，互換性が優先されることがある．

3.5　攻撃種別に着目した悪性文書ファイルの特徴

tar(09–12) を見ると，downloader は 370 検体中 8 体と全体のわずか 2.2 %であった．一方，mal(con) を見ると，downloader は 11,101 検体中 10,964 体と全体の 98.8 %を占めていた．このことから，悪性文書ファイルの傾向として，少なくとも 2013 年までは，標的型攻撃に使用されるもののほとんどは dropper であり，標的型攻撃以外に使用されるもののほとんどは dropper であったことがわかる．

3.5.1　悪性文書ファイルの分類と特徴

悪性文書ファイルは，ダミー表示をするか否か，表示をする場合は，ダミー表示用の文書ファイルを自身から取り出すか否かの 3 種類に分類できる．したがって，マルウェア本体の格納場所の 2 種類と組み合わせると，悪性文書ファイルは 6 種類に分類できる．各悪性文書ファイルについて，攻撃者の意図どおりに動作するかという観点で分析した結果を表 3.5 に示す．表の各欄の意味について以下に示す．

表 3.5　悪性文書ファイルの分類と特徴

		Online		Offline		Shell	
Exe	Dec.	Inf.	Dec.	Inf.	Dec.	code	Target
	None	OK	–	NG	–	Small	Wide
Down	Down	OK	Slow	NG	NG	Small	Wide
	Drop	OK	Fast	NG	Fast	Big	Wide
	None	OK	–	OK	–	Small	Narrow
Drop	Down	OK	Slow	OK	NG	Big	Narrow
	Drop	OK	Fast	OK	Fast	Small	Narrow

(1)　Exe

マルウェア本体を外部のサーバからダウンロードするもの（downloader）を “Down”，自身から取り出すもの（dropper）を “Drop” とした．

(2)　Decoy（Dec.）

ダミー表示をしないものを “None”，ダミー表示をするもののうちダミー表示用の文書ファイルを外部のサーバからダウンロードするものを “Down”，自身から取り出すものを “Drop” とした．

(3)　Online / Offline

悪性文書ファイルの侵入経路として，Web，メール，USB 等様々なものが想定され，悪性文書ファイルを閲覧する際の端末の環境は，攻撃手法によっては，攻撃者が制御することが難しい場合がある．そこで，端末の環境として，端末が外部のネットワークに繋がっているか否かで，“Online” と “Offline” に分け，それぞれの状況で，攻撃者の意図どおりに悪性文書ファイルが動作することが可能か否かを分析した．

- Infection（Inf.）
  マルウェア本体が攻撃者の意図どおり実行されるものを “OK”，実行されないものを “NG” とした．例えば，downloader の場合，Online であれば，外部のサーバからマルウェア本体がダウンロード可能であることから，マルウェア本体が攻撃者の意図どおり実行される可能性が高いことから “OK” とした．一方，Offline であれば，外部のサーバからマルウェア本体をダウンロードできず，マルウェア本体の実行が不可能であることから “NG” とした．

- Decoy（Dec.）
  悪性文書ファイルの中からダミー表示用の文書ファイルを取り出す場合，既にメモリ上に展開されたデータをファイルに保存したり，端末上に保存された悪性文書ファイルから必要な部分を切り出すだけである．そのため，悪性文書ファイルを開いてからダミー表示用の文書ファイルが表示されるまでの時間は，通常の文書ファイルを表示する場合の時間と大差がない．
  一方，外部のサーバからダミー表示用の文書ファイルをダウンロードする場合，表示にかかる時間は，攻撃対象のネットワーク環境に依存する．さらに，悪性文書ファイルを開く環境がオンラインである保証もないため，ダミー表示用の文書ファイルのダウンロードに失敗する可能性もある．
  ダミー表示する機能がないものを “–”，ダミー表示に失敗してしまうものを “NG”，ダミー表示に成功するもののうち悪性文書ファイルを開いてからダミー表示されるまでの時間で “Fast” と “Slow” に分類した．

(4) Shellcode

Shellcode は，使用する脆弱性によりサイズを制限されることが多い．言い換えると，shellcode のサイズが小さければ，攻撃者が攻撃に使用することができる脆弱性の種類が増えることになる．そのため，shellcode には，コードサイズを小さくするための様々な工夫が施されている．ダミー表示用の文書ファイルを作成する手法とマルウェア本体の実行ファイルを作成する手法に着目し，両手法を同一のものにすることで，共通で使えるコードを増やし，shellcode のサイズを小さくすることができるものを “Small” とした．また，ダミー表示をしないものも “Small” にした．それ以外を “Big” とした．

(5) Target

攻撃対象の範囲．downloader は，感染端末の環境，攻撃目的，時期等に応じて柔軟に，ダウンロードするマルウェア本体を変更することができることから “Wide” に分類した．dropper は，埋め込んだマルウェア本体を後から変更することができず，一度想定した感染端末の環境，攻撃目的を変更することが困難であることから，“Narrow” とした．

3.5.2 悪性文書ファイルの評価

本章では，攻撃者にとってのメリットという観点で表 3.5 を攻撃種別ごとに再評価を行うことにより，悪性文書ファイルのうち，標的型メール攻撃に使用されるもののほとんどが dropper であり，それ以外の攻撃に使用されるもののほとんどが downloader である理由について考察を行う．

(1) 標的型メール攻撃

標的型メール攻撃は，特定の個人や組織を対象にした攻撃であり，攻撃対象が興味を引くようにメールの件名や本文を調整することにより，メールに添付され

たマルウェアを受信者に開封させる攻撃である．マルウェアの多くは，端末の遠隔操作を目的としたRATと呼ばれるものである．攻撃者は，RATを使用することで，攻撃対象の保有する情報を窃取したりする．

表3.5を基準に，標的型メール攻撃に使用される悪性文書ファイルについて，攻撃の成功率を高めるものを“+1”，攻撃に気づかれる確率を上げるものを“-1”，どちらでもないものを“0”として評価を行った．評価の結果を表3.6に示す．

表 3.6　標的型メール攻撃で使用される悪性文書ファイルの評価

		Online		Offline		Shell	
Exe	Dec.	Inf.	Dec.	Inf.	Dec.	code	Target
	None	+1	-1	0	-1	+1	0
Down	Down	+1	-1	0	-1	+1	0
	Drop	+1	0	0	0	0	0
	None	+1	-1	+1	-1	+1	0
Drop	Down	+1	-1	+1	-1	0	0
	Drop	+1	0	+1	0	+1	0

各項目ごとの評価基準について以下に示す．

- Infection（Inf.）
  マルウェア本体の実行に成功（OK）のものを“+1”とした．また，マルウェア本体の実行はダミー表示のバックグラウンドで処理されている．マルウェア本体の実行という処理は被害者が意識をしているものではないことから，その処理に失敗したとしても，被害者が攻撃に気づくことには繋がらない．したがって，マルウェア本体の事項に失敗（NG）のものは“0”とした．

- Decoy（Dec.）
  この攻撃の初期段階であるマルウェアへの感染には，添付ファイルを開くという受信者の操作が必須である．受信者は，メールに添付された悪性文書ファイルがメールの本文や件名に関連した内容が表示されることを期待している．悪性文書ファイルを開いた際に，何も表示しなかったり，閲覧ソフトの動作が停止したりする動作は，受信者の期待を裏切る動作である．そのような動作があった場合，受信者がメールを不審に感じ，攻撃に気付く可能性が高くなる．マルウェアへの感染という攻撃者の初期の目的を達

成できたとしても，受信者が短い時間で感染に気付いてしまうため，情報窃取という攻撃者の最終目的が達成できなくなってしまう．そのため，標的型メール攻撃に使用される悪性文書ファイルを開くと，ダミー表示用の文書ファイルが表示されるものを “+1”，表示されないものを “-1” とした．

- Shellcode
  Shellcode を小さくすることで，攻撃することのできる脆弱性の種類が増えたり，shellcode 内に実装できる検知回避技術の種類が増えることから，シェルコードのサイズが小さくなるものを “+1” と評価した．

- Target
  攻撃対象を広く取れることは，一般的に考えるとメリットである．しかしながら，標的型攻撃は，攻撃対象を絞っているため，攻撃の成功率に対し，この項目が大きく貢献することはないと考えられる．したがって，すべて “0” とした．

(2) ドライブバイダウンロード（DBD）攻撃

DBD 攻撃とは，主に Web ブラウザを介して，閲覧者に気づかれずにマルウェアに感染させる攻撃である．マルウェア感染までの流れを以下に示す．攻撃者は，Web サイトを改ざんする．Web サイトは様々なコンテンツの組み合わせで構成されている．ある 1 つの url のページを指定すると，Web ブラウザはそのページに関連する様々な関連するコンテンツをダウンロードする．攻撃者は，Web サイトを改ざんする際に，この関連するコンテンツとして悪性文書ファイルを追加する．閲覧者が改ざんされた Web サイトを表示しようとすると，Web ブラウザは，正規のコンテンツの表示と並行して，バックグラウンドで悪性文書ファイルをダウンロードして表示をしようとする．これにより，改ざんされた Web サイトを表示した端末はマルウェアに感染する．

表 3.5 を基準に，DBD 攻撃に使用される悪性文書ファイルについて，攻撃の成功率を高めるものを “+1”，攻撃に気づかれる確率を上げるものを “-1”，どちらでもないものを “0” として評価を行った．DBD 攻撃には攻撃対象を限定しないバラマキ型の攻撃と，攻撃対象を限定した水飲み場型の攻撃の 2 種類がある．水飲み場型の事例がバラマキ型と比較して少ないため，ここではバラマキ型を想定して評価を行う．評価の結果を表 3.7 に示す．

表 3.7　DBD 攻撃で使用される悪性文書ファイルの評価

		Online		Offline		Shell	
Exe	Dec.	Inf.	Dec.	Inf.	Dec.	code	Target
	None	+1	0	–	–	+1	+1
Down	Down	+1	-1	–	–	+1	+1
	Drop	+1	-1	–	–	0	+1
	None	+1	0	–	–	+1	0
Drop	Down	+1	-1	–	–	0	0
	Drop	+1	-1	–	–	+1	0

各項目ごとの評価基準について以下に示す.

- Offline
  DBD 攻撃は，攻撃の起点が Web サイトの閲覧であり，被害者の端末がネットワークに繋がっていることが前提となっている．したがって，被害者の端末がネットワークに繋がっていない場合については評価を行わない.

- Infection（Inf.）
  マルウェア本体の実行に成功（OK）のものを “+1” とした．また，マルウェアの実行はバックグラウンドで処理されており，マルウェアの実行に失敗したとしても，被害者が攻撃に気づくことには繋がらないことから “0” とした.

- Decoy（Dec.）
  DBD 攻撃の初期段階である，マルウェアへの感染の起因となった被害者の行動は，改ざんされた Web サイトの表示である．被害者は正規の Web サイトのコンテンツを期待して改ざんされた Web サイトを表示する．Web ブラウザは，改ざんされた Web サイトに関連するコンテンツとして悪性文書ファイルを表示する．しかしながら，この悪性文書ファイルの表示に関する処理は，閲覧者の行動と直接結びついたものではない．閲覧者は，この悪性文書ファイルの存在自体意識していない．したがって，標的型メール攻撃の場合と異なり，ダミー表示は必要ない．逆に，ダミー表示をすると，閲覧者が文書ファイルを開いていることを意識するため，閲覧者が攻撃に気づく可能性が高くなってしまう．そのため，ダミー表示用の文書ファイルが表示されるものを “-1”，表示されないものを “+1” とした.

- Shellcode
  Shellcode を小さくすることで，攻撃することのできる脆弱性の種類が増えたり，shellcode 内に実装できる検知回避技術の種類が増えることから，シェルコードのサイズが小さくなるものを “+1” と評価した．

- Target
  DBD 攻撃の多くは攻撃対象を限定しない，いわゆるバラマキ型の攻撃である．したがって，攻撃対象を広く取れることは，感染端末の増加につながり，攻撃の成功率を高めるものである．したがって，攻撃対象を広げるもの（Wide）を “+1” とし，それ以外を “0” とした．

3.5.3 各攻撃に最適な悪性文書ファイル

1 回の攻撃で被害者が攻撃に気づく確率を k とすると，攻撃に気づかれるまでに，攻撃者が攻撃できる回数の期待値 n は以下の数式で示せる．

$$n = \frac{1}{k} - 1 \tag{3.1}$$

また，1 回の攻撃での攻撃の成功確率を s とすると，攻撃に気づかれるまでに，マルウェア感染に成功する回数の期待値 N は以下の数式で示せる．

$$N = s \times n = s \times (\frac{1}{k} - 1) \tag{3.2}$$

攻撃に気づかれる確率（n) とマルウェア感染の成功確率（s）から，マルウェア感染に成功する回数の期待値（N）を求めた結果を表 3.8 に示す．

表 3.8 マルウェア感染の成功回数の期待値

		s				
k	n	1 %	25 %	50 %	75 %	99 %
1 %	99	0.99	24.75	49.50	74.25	98.01
25 %	3	0.03	0.75	1.50	2.25	2.97
50 %	1	0.01	0.25	0.50	0.75	0.99
75 %	0.3	0.00	0.08	0.17	0.25	0.33
99 %	0.01	0.00	0.00	0.01	0.01	0.01

マルウェアの感染という攻撃の初期段階の目的を達成するためには，マルウェア感染に成功する回数は 1 回で十分である．したがって，攻撃対象が限定されて

いる場合，マルウェア感染に成功する確率を上げるよりも，いかに攻撃に気づかれないかが重要になってくる．一方，攻撃対象を特に限定しない場合，いかに攻撃範囲を広げるかが重要になってくる．

この結果を踏まえ，各攻撃ごとに最適な悪性文書ファイルについて考察を行う．

(1)　標的型メール攻撃

標的型メール攻撃の場合，攻撃対象は特定の組織または個人である．攻撃対象が限定されているため，攻撃に気づかれる確率を下げることが重要になる．したがって，表 3.6 で “-1” と評価されている悪性文書ファイルは攻撃に適していない．その上で，よりマルウェア感染の確率が高くなるものを選択した場合，マルウェア本体およびダミー表示用の文書ファイルを悪性文書ファイルの中から取り出すタイプの悪性文書ファイルになるものと推測される．

(2)　DBD 攻撃

DBD 攻撃の場合，その多くが攻撃対象を限定しない，いわゆるバラマキ型攻撃である．この場合，以下に攻撃範囲を広げるかが重要になってくるため，表 3.7 の “Target” で “+1” と評価されていることが重要である．また，ダミー表示をするコストをかけた上で得られる効果は，攻撃に気づかれやすくなるというデメリットであるため，ダミー表示をすることは通常考えられない．したがって，DBD 攻撃で使用される悪性文書ファイルは，ダミー表示を行わない downloader になるものと推測される．

3.5.4　研究の効率化

調査の結果，攻撃種別ごとに悪性文書ファイルの特徴に顕著な差があることが明らかになった．この差が生じる理由を推測するため，攻撃の成功率を高めるという攻撃者の視点にたった考察を行った．各攻撃には攻撃者の目的があり，その目的を達成するために最適な悪性文書ファイルの種類を攻撃者が選択した結果，悪性文書ファイルの特徴に偏りが出ていることが推測される．

本研究では，標的型攻撃における被害を抑止することを目的に研究を行う．研究を効率的に行うため，この章では，dropper に焦点をあてて調査を行う．

3.6　実行ファイルに着目した検知方式

3.6.1　埋め込まれた RAT のエンコード方式

tar(09–12) の dropper に埋め込まれた実行ファイルは，そのまま文書ファイルに埋め込まれる場合もあるが，多くの場合に何らかの方式でエンコードされて文書ファイルに埋め込まれていた．tar(09–12) から，主にバイナリエディタを用いた静的解析により，実行ファイルを展開・実行するコードを抽出して逆アセンブラで解析した．その結果，判明したエンコード方式に用いられていた主な演算を表 3.9 に示す．エンコード方式は，主としてこれらの基本的な演算を組み合わせて構成されており，換字による方式，転置による方式およびファイル形式固有の

方式に分類することができる．換字による方式は，実行ファイルをエンコードする場合に必ず使用される基本的なエンコード方式である．転置による方式およびファイル形式固有の方式は，換字による方式に加えて使用される付加的なエンコード方式であり，複数の方式を組み合わせている場合もある．

表 3.9　Main instructions to encode executable files.

演算	説明
XOR	排他的論理和
ADD	算術加算
SUB	算術減算
ROL	左論理ローテート
ROR	右論理ローテート

HEX

```
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
```

ASCII

```
MZ..............
........@.......
................
................
........!..L.!Th
is program canno
t be run in DOS
mode....$.......
```

図 3.3　Example of an executable file.

(1)　換字による方式

換字による方式は，何らかの方式で作成した鍵との排他的論理和（XOR）を計算することで，実行ファイルを別の文字列に変換する方式である．換字による方式では，元の実行ファイルのオフセット（先頭から所定の位置までのバイト数）はエンコード後にも変化しない．

XOR

XOR は，実行ファイルと任意の 1 byte の鍵を XOR 演算するエンコード方式であり，最も基本的かつ頻繁に用いられている．図 3.4 は XOR でエンコードされた実行ファイルの例を示している．通常の XOR による方式では，実行ファイ

ルと鍵の XOR 演算を実施すると，実行ファイルの NULL の部分は鍵と同一の値となる．そのため，実行ファイルの NULL の部分は鍵と同一の値となる．そのため，実行ファイルの NULL の部分に着目することで，容易にエンコードに用いる鍵を特定することが可能である．

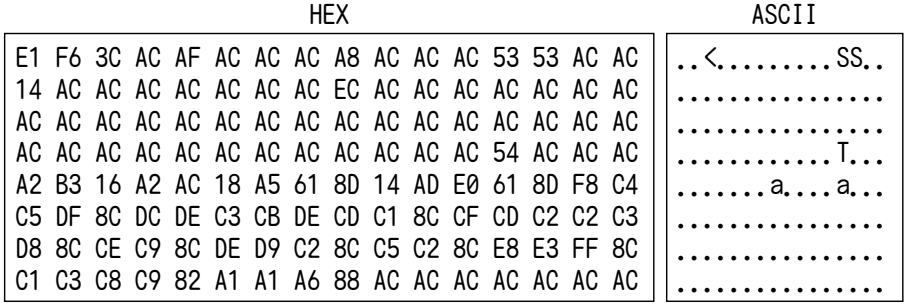

図 3.4 Example of an executable file encoded with XOR (AC).

Multibyte XOR

Multibyte XOR は，実行ファイルと任意の 2 byte 以上の鍵を XOR 演算するエンコード方式である．図 3.5 は Multibyte XOR でエンコードされた実行ファイルの例を示している．この方式もやはり，実行ファイルの NULL の領域に 2 byte 以上の繰返し文字列を検出することで，エンコードに用いる鍵を推定することが可能である．ただし，繰返し文字列を検出することで鍵を推定するためには，実行ファイルに鍵の長さの 2 倍以上の NULL の領域が存在する必要がある．

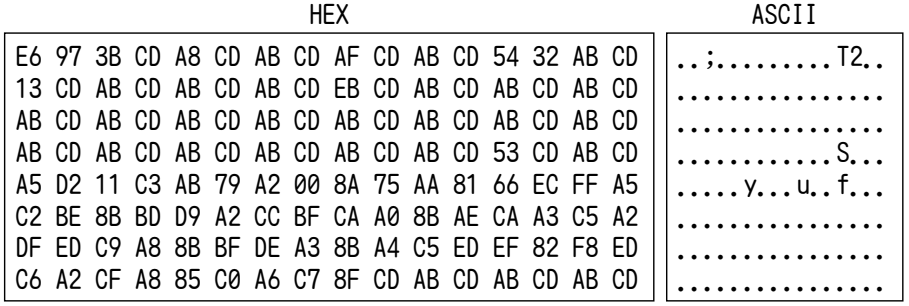

図 3.5 Example of an executable file encoded with Multibyte XOR (AB CD).

NULL-Preserving XOR

NULL-Preserving XOR は，エンコードに用いる任意の 1 byte の鍵の値と一致する場合，または NULL の場合は演算を実施せず，それ以外の場合のみ XOR 演算する方式である．図 3.6 は NULL-Preserving XOR でエンコードされた実行ファイルの例を示している．NULL-Preserving XOR では実行ファイルの NULL の部分は鍵の値とならず，NULL のままとなっているため，一見しただけでは鍵を推定することができない．したがって，鍵を特定するためには，すべての考えうる鍵を総当りで試す必要がある．

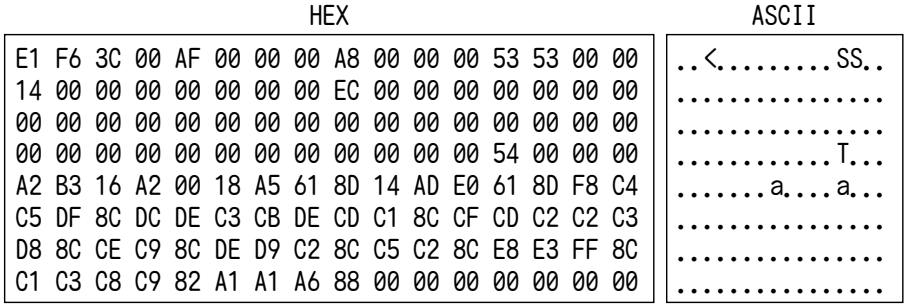

図 3.6 Example of an executable file encoded with NULL-Preserving XOR (AC).

Multibyte NULL-Preserving XOR

Multibyte NULL-Preserving XOR は，鍵と同じ長さの領域がすべて NULL の場合には演算を実施せず，それ以外の場合には任意の 2 byte 以上の鍵を XOR 演算するエンコード方式である．図 3.7 は Multibyte NULL-Preserving XOR でエンコードされた実行ファイルの例を示している．Multibyte NULL-Preserving XOR では，Multibyte XOR のように実行ファイルの NULL の領域に 2 byte 以上の鍵がそのまま表れることがない．したがって，2byte 以上の繰返し文字列を検出することで，エンコードに用いられた鍵を推定することができない．ただし，Multibyte NULL-Preserving XOR では，鍵と同じ長さの領域に 1 byte でも NULL 以外の値が含まれている場合には XOR 演算を実施するため，文字コードの頻度分析によって鍵を推定できる可能性がある．

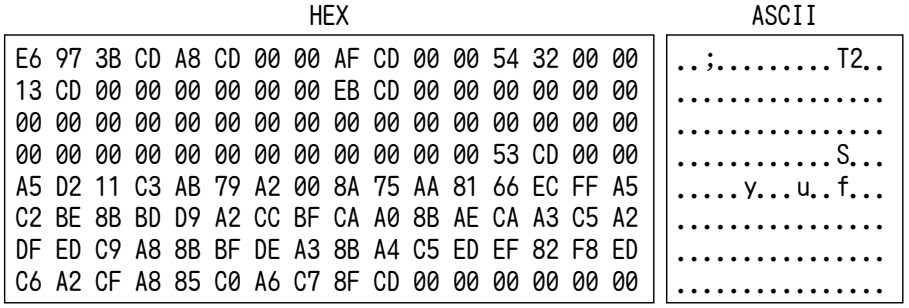

図 3.7 Example of an executable file encoded with Multibyte NULL-Preserving XOR (AB CD).

Rolling XOR

Rolling XOR は，1 byte 単位の鍵による XOR 演算と考えた場合，鍵の値がある周期で 1 byte ごとに変化するエンコード方式である．最も基本的な鍵の値の変化の方式は，1 ずつ加算するインクリメント方式および 1 ずつ減算するデクリメント方式である．図 3.8 は Rolling XOR でエンコードされた実行ファイルの例を示している．これらの方式の場合，256 byte ごとに鍵の値が元の値となるため，その周期は 256 となる．2 周目以降は，1 周目と同じ 256 byte の値が鍵としてエンコードに用いられる．そのエンコードの結果は，256 byte の鍵を用いた Multibyte XOR と同じである．つまり，Rolling XOR を 2 byte 以上の鍵による XOR と考えた場合には，Multibyte XOR と実質的に同義である．この場合，Rolling XOR の鍵の周期が Multibyte XOR の鍵の長さに相当する．したがって，Multibyte XOR と同様に実行ファイルの NULL 部分に着目し，2 byte 以上の繰返し文字列を検出することで，エンコードに用いられた鍵を推定することが可能である．

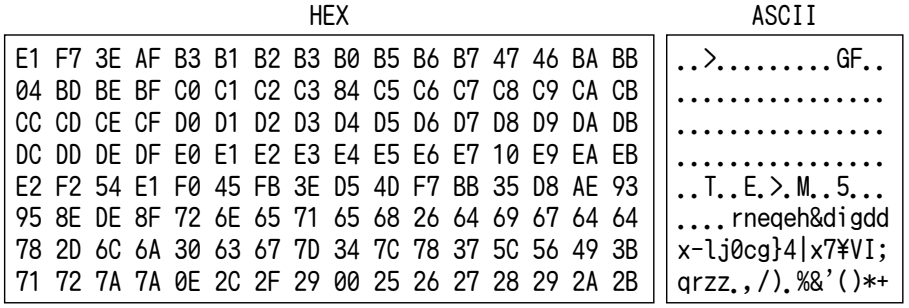

図 3.8 Example of an executable file encoded with Rolling (Incremental) XOR (AC AD AE ...).

Rolling XOR や Multibyte XOR によるエンコード結果は, Rolling XOR と XOR による多重エンコードの結果と同義となる．なぜならば，Rolling XOR と XOR による多重エンコード演算には結合法則が成り立つため，結果として生成される鍵の長さは元の Rolling XOR の周期となるからである．よって多重にエンコード方式が用いられている場合にも，2 byte 以上の繰返し文字列を検出することで，エンコードに用いられた鍵を推定できる可能性がある．

そのほかの方式

そのほかの換字によるエンコード方式としては，特定の周期のオフセットの場合にのみ演算を実施する方式や，独自のストリーム暗号を用いる方式等があげられる．また，少し変わった方式としては，ファイルの先頭から 1 byte ずつ順番に読み込み，次の 1 byte の値を鍵として XOR 演算を実施する方式も確認されている．

(2) 転置による方式

転置による方式は，何らかの方式で実行ファイルあるいは鍵の順序を並べ替える方式である．並べ替えの単位は，方式によって byte 単位の場合と bit 単位の場合がある．byte 単位の転置による方式では，元の実行ファイルのオフセットはエンコード後に変化する．

論理ローテート

論理ローテートは，対象とする 1 byte の値を，1〜4 bit 単位で左論理ローテートまたは右論理ローテートする方式である．図 3.9 は論理ローテートでエンコードされた実行ファイルの例を示している．4 bit 左ローテートした値と 4 bit 右ローテートした値は同一となる．論理ローテートは，実行ファイルに対して実施される場合と，Rolling XOR の鍵に対して実施される場合がある．周期が 256 の Rolling XOR の鍵に対して論理ローテートが実施された場合にも，結果として生

成される鍵の周期は変わらない．なぜならば，論理ローテートの周期である 8 は，256 の約数となるからである．よって，Rolling XOR に論理ローテートが併用された場合にも，2 byte 以上の繰返し文字列を検出することでエンコードに用いる鍵を推定できる可能性がある．

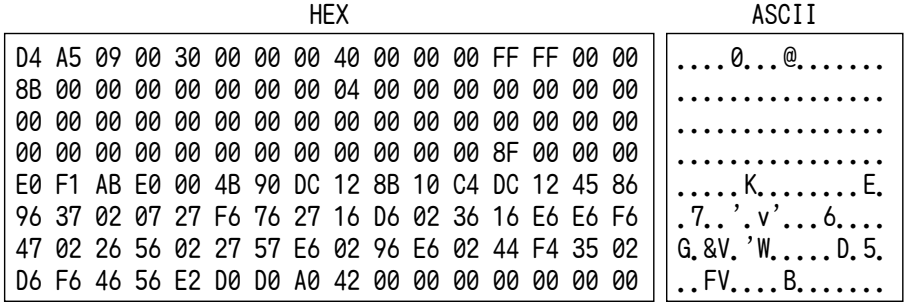

図 3.9　Example of an executable file encoded with 4 bit rotate.

SWAP

SWAP は，対象とする 2 byte の値を，1 byte 単位で順序を入れ換える方式である．図 3.10 は SWAP でエンコードされた実行ファイルの例を示している．SWAP 演算を実施した文字列に対し，オフセットを 1 byte ずらして再度 SWAP 演算を実施する Double SWAP 方式も確認されている．また，1 byte 単位ではなく，対象とする 1 byte を half byte 単位で順序を入れ換える Nibble SWAP 方式もある．なお，Nibble SWAP 方式は，4 bit 左論理ローテートおよび 4 bit 右論理ローテートと同義である．

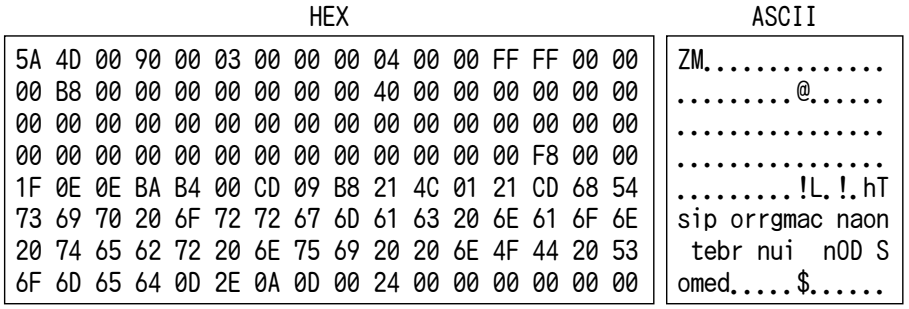

図 3.10　Example of an executable file encoded with SWAP.

(3) そのほかの方式

16 進数の文字列にエンコードする方式

ほとんどの RTF ファイルにおいては，実行ファイルの 1 byte を 2 byte の 0〜f の 16 進数の文字列にエンコードする方式が用いられている．図 3.11 は 16 進数の文字列にエンコードされた実行ファイルの例を示している．この方式は，デコードに鍵は不要であり，容易にデコードをすることができる．単独で用いられた場合，パターンマッチングによる検知が容易であるため，他の方式と組み合わせて使用されることが多い．この方式でエンコードされた実行ファイルは，元の実行ファイルの長さの 2 倍の長さの 0〜f の 16 進数の文字列にエンコードされているため，容易に発見することが可能である．

HEX	ASCII
34 64 35 61 39 30 30 30 30 33 30 30 30 30 30 30	4d5a900003000000
30 34 30 30 30 30 30 30 66 66 66 66 30 30 30 30	04000000ffff0000
62 38 30 30 30 30 30 30 30 30 30 30 30 30 30 30	b800000000000000
34 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	4000000000000000
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0000000000000000
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0000000000000000
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30	0000000000000000
30 30 30 30 30 30 30 30 66 38 30 30 30 30 30 30	00000000f8000000
30 65 31 66 62 61 30 65 30 30 62 34 30 39 63 64	0e1fba0e00b409cd
32 31 62 38 30 31 34 63 63 64 32 31 35 34 36 38	21b8014ccd215468
36 39 37 33 32 30 37 30 37 32 36 66 36 37 37 32	69732070726f6772
36 31 36 64 32 30 36 33 36 31 36 65 36 65 36 66	616d2063616e6e6f
37 34 32 30 36 32 36 35 32 30 37 32 37 35 36 65	742062652072756e
32 30 36 39 36 65 32 30 34 34 34 66 35 33 32 30	20696e20444f5320
36 64 36 66 36 34 36 35 32 65 30 64 30 64 30 61	6d6f64652e0d0d0a
32 34 30 30 30 30 30 30 30 30 30 30 30 30 30 30	2400000000000000

図 3.11 Example of an executable file encoded with HEX string.

Java Script を用いたエンコード方式

PDF ファイルにおいては，換字による様式および転置による方式で直接実行ファイルを埋め込む以外にも，Java Script を用いたエンコード方式が確認されている．Java Script には，最終的に実行ファイルの 1 byte を 2 byte の 0〜f の 16 進数の文字列にエンコードして埋め込まれていた．ただし，PDF の場合には，実行ファイルが埋め込まれた Java Script が，これまでに示した方式等で更にエンコードされることになるため，それらの方式も組み合わせて検索する必要がある．

ファイル形式固有の方式

ファイル形式固有の方式は，対象とする文書ファイルの形式に依存しており，特定のファイル形式のみで用いられているエンコード方式である．たとえば，VBScript や，Flash に使用されるプログラム言語である Action Script を用いて実行ファイルをエンコードする方式が確認されている．すなわち，これを用いた悪性文書ファイルでは，そのプログラム言語を用いて，エンコードされた実行ファイルのデータの記述とそのデータのデコード処理が記述されている．そのプログラム言語をサポートしている文書ファイルであれば，文書ファイルを開く際にプログラム言語で記述された命令が実行される．そのため，文書ファイルがサポートするプログラム言語を用いて，実行ファイルをエンコードすることが可能である．

3.6.2　エンコード方式の解読手法

3.6.1 で述べた様々な方式でエンコードされた実行ファイルを文書ファイルから取り出すためには，そのエンコード方式を解読する必要がある．エンコード方式を解読する手法としては，手法 1：総当たり，手法 2：繰返し文字列の検出および，手法 3：文字コードの頻度分析が考えられる．

(1)　手法 1：総当たり

何らかの方式で作成した鍵と XOR を計算する換字による方式は，実行ファイルを埋め込む場合に必ずといっていいほど常に用いられる方式である．換字による方式の最も単純な解読手法は，すべての考えられうる鍵を総当たりで試す手法である．任意の方式である転置による方式や 16 進数の文字列にエンコードする方式は，換字による方式の解読と並行して検索すれば効率が良い．総当たりによる手法は，XOR や NULL-Preserving XOR 等の鍵長が短い場合に有効である．特に，NULL-Preserving XOR は他の手法では解読できないため，唯一の解読手法であると考えられる．しかしながら，鍵長が長い Multibyte XOR や Multibyte NULL-Preserving XOR に対しては，現在のコンピュータでは実用的な時間内での解読は困難である．Rolling XOR については，単純なインクリメント方式やデクリメント方式であれば，実用的な時間内での解読は可能である．

(2)　手法 2：繰返し文字列の検出

鍵長が長いエンコード方式に対しては，総当たりでは実用的な時間内で解読することは困難である．そこで，実行ファイルの NULL の領域に着目し，2-byte 以上の繰返し文字列を検出することで，エンコードに用いる鍵を推定する手法が考えられる．この手法が機能するためには，実行ファイルに鍵の長さの 2 倍以上の NULL の領域が存在し，鍵がそのままの状態で悪性文書ファイルに含まれている必要がある．この手法は，XOR，Multibyte XOR，Rolling XOR および多重エンコードに対して有効であると考えれれる．同様に，転置による方式や 16 進数の文字列にエンコードする方式についても，並行して検索することで実用的な時間

内での解読は可能であると考えられる．しかしながら，鍵がそのままの状態で文書ファイルに含まれない NULL-Preserving XOR や Multibyte NULL-Preserving XOR に対しては，原理的に解読は不可能である．

(3)　手法 3：文字コードの頻度分析

鍵長が長い Multibyte NULL-Preserving XOR に対しては，総当りでも繰返し文字列の検出でも実用的な時間内で解読することは困難である．Multibyte NULL-Preserving XOR では，鍵と同じ長さの領域に 1 byte でも NULL 以外の値が含まれている場合には XOR 演算を実施する．ゆえに，実行ファイルのある程度以上の NULL の領域が含まれていれば，鍵はそのままの状態では含まれていないが，文字コードの頻度分析によって鍵を推定できる可能性が考えられる．この手法は，XOR，Multi XOR および Rolling XOR に対してもある程度有効であると考えられる．同様に，転置による方式や 16 進数の文字列にエンコードする方式についても，並行して検索することで実用的な時間内での解読は可能であると考えられる．しかしながら，ファイル形式固有で使用頻度が高い文字コードがある場合には，文字コードの頻度に偏りが生じてしまい，正しい鍵が推定できない可能性が考えられる．たとえば，RTF や PDF では，データに表示不可能な文字コードが多く含まれるため，その影響を緩和する必要がある．

3.6.3　実行ファイルの検知

文書ファイルに埋め込まれている実行ファイルを取り出すためには，解読したエンコード方式を用いてデコードしたデータに，実行ファイルが含まれていることを確認する必要がある．実行ファイルが含まれていることを確認する手法は，MS-DOS 用スタブプログラムを検索する方式と，実行ファイルのヘッダを検索する方式が考えられる．

(1)　MS-DOS 用のスタブプログラムを検索する方式

実行ファイルが含まれているか確認する方式は，実行ファイルの MS-DOS 用スタブプログラムに含まれる文字列を検索するのが基本である．MS-DOS スタブプログラムは，実行ファイルが MS-DOS 上で実行された場合に実行されるプログラムであり，一般的には “This program cannot be run in DOS mode” や “This program must be run under Win32” という文字列を表示して終了するプログラムである．しかしながら，MS-DOS 用スタブプログラムに含まれる文字列は任意に設定することが可能であるため，この文字列のみで実行ファイルを検知することはできない．我々が分析したマルウェアには，この方式による検知を回避するため，これらの文字列を 1 byte のみ変更した実行ファイルも含まれていた．また，“PE for Win32” や “Win32 only!” のようなまったく異なる文字列が含まれている場合や，あるいは空白となっている実行ファイルも確認された．

(2)　ヘッダを検索する方式

MS-DOS 用スタブプログラムを検索するよりも確実なのは，実行ファイルのヘッダを検索する方式である．実行ファイルの先頭には，MZ ヘッダおよび PE ヘッダがついているため，“MZ” や “PE” の文字列を削除した実行ファイルを埋め込んでいるマルウェアも確認された．“MZ” や “PE” の文字列は shellcode が実行された場合には復元されるため，実行ファイルは正常に動作する．このような場合には，MS-DOS 用スタブプログラムを検索する方式と合わせて総合的に検索する手法が有効である．実行ファイルの NULL が含まれるオフセット等も，RAT を検知するための有効な手がかりとなるものと考えられる．

3.6.4　試験プログラムの実装

これまでに示した 3 つの実行ファイルのエンコード方式の解読手法および 2 つの実行ファイルの検知方式を組み合わせた手法を，オープンソースのプログラミング言語である Python を用いて実装した．実装したプログラムの動作の概要を図 3.12 に示す．試験プログラムは，文書ファイルを引数として受け取り，埋め込まれた実行ファイルを検知し，取り出すコマンドラインプログラムである．STEP1 では，RTF ファイル固有である 2 byte の 0〜f の 16 進数の文字列を抽出し，バイナリコードに変換する．次に，変換したバイナリコードから，提案する手法 1〜3 を用いてエンコード方式を解読し，実行ファイルの検知を試みる．STEP1 で実行ファイルが検知できなかった場合には，STEP2 に進む．STEP2 では，入力したファイル全体から，手法 1〜3 を用いてエンコード方式を解読し，実行ファイルの検知を試みる．ファイルの検索を終了するか，実行ファイルを検知した場合には実行ファイルを取り出し，動作を終了する．各 STEP におけるエンコード方式の解読（手法 1〜3）および実行ファイル検知のアルゴリズムは同一である．以下，その詳細について説明する．

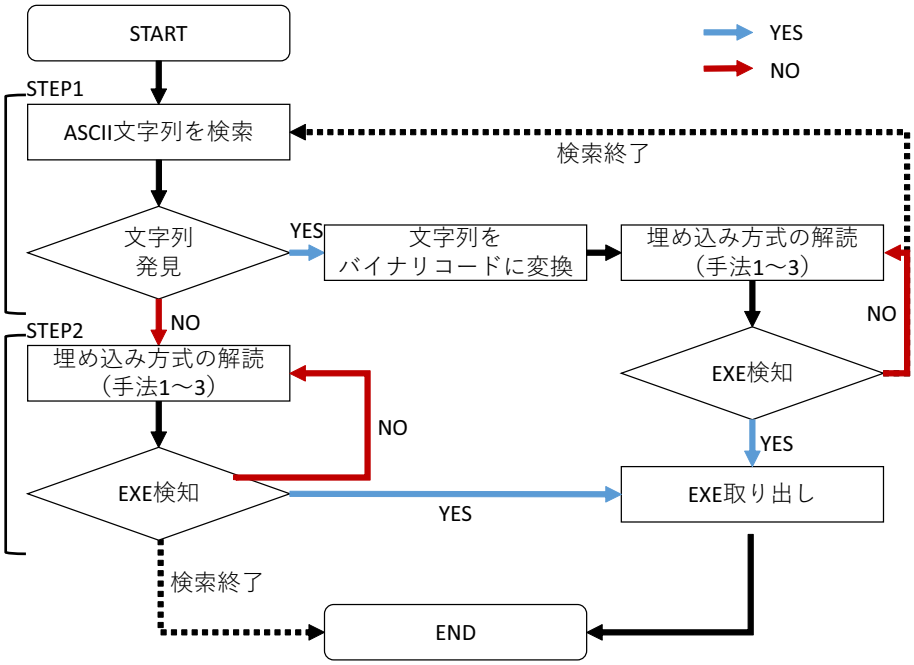

図 3.12　The algorithm of the test program.

(1)　埋め込み方式の解読手法

試験プログラムは，以下に示す 3 つの手法でエンコード方式の解読を試みる．実用性を考慮すると，いずれかの手法で実行ファイルのエンコード方式を解読した場合には，他の手法による解読は不要となる．しかしながら今回は，これらの 3 つの手法による解読の成功率を比較するため，最後まですべての手法により解読を試みるように実装した．

手法 1：総当たり

0x00 から 0xFF までの 256 通りの 1 byte の鍵候補を作成し，各鍵候補によりエンコードした RAT の検知のための固有のパターンを検索する．各鍵候補を用いて検索するエンコード方式を以下に示す．

必須である換字による方式については，XOR，NULL-Preserving XOR および Rolling XOR の一部であるインクリメント方式およびデクリメント方式を検索する．Multibyte XOR，NULL-Preserving XOR およびその他の Rolling XOR 方式については，実用的な時間内での解読は困難であるため検索しない．任意である転置による方式については，実行ファイルまたは鍵の論理ロテート，SWAP および Double SWAP 方式を同時に検索する．また，2 byte の 0～f の 16 進数の文字列にエンコードする方式を同時に検索する．

手法 2：繰返し文字列の検出

総当たりによる鍵の検索では，多バイト長の鍵の検索に時間を要してしまい，実行ファイルを短時間で検知することができない．したがって，それを緩和するために，文書ファイルから繰返し文字列を検出し，他バイト長の鍵の推定を試みる．以下，具体的な手順を示す．

まず，文書ファイルを 256 byte の長さで分割し，その各バイナリデータを順番に並べたリストを作成する．この 256 byte は，推定する鍵候補の長さである．次に，連続するリストのバイナリデータを比較し，一致するリストのバイナリデータを比較し，一致するリストのバイナリデータを鍵候補として蓄積する．以後，分割する文書ファイルの長さ（鍵候補の長さ）を半分とし，その長さが 2 byte になるまで同手順を繰り返す．この手順により抽出された鍵候補は，手順 1 における鍵候補に換字による方式を組み合わせた出力結果に相当するものである．任意である転置による方式については，実行ファイルまたは，鍵の論理ローテート，SWAP および Double SWAP 方式を同時に検索する．また，2 byte の 0〜f の 16 進数の文字列にエンコードする方式を同時に検索する．

手法 3：文字コードの頻度分析

鍵長が長い Multibyte NULL-Preserving XOR に対しては，総当たりでも繰り返し文字列の検出でも実用的な時間内で解読することは困難である．そこで，文字コードの頻度分析によって鍵の推定を試みる．以下，具体的な手順を示す．

まず，文書ファイルの文字コードを，そのオフセットの 256 の剰余ごとに配分する．ここで，256 は推定する鍵候補の長さを示す．次に，256 通りの各オフセットの剰余ごとに文字コードを集計し，最も使用頻度が高いコードを求める，この際に，文書ファイル固有の使用頻度が高い文字コードの影響を緩和するため，表示可能な文字コードが多く含まれる領域を集計の対象から除外する．次に，各オフセットごとの最頻出文字コードを順に連結し，鍵候補とする．以後，鍵候補の長さを半分とし，その長さが 2 byte になるまで同手順を繰り返す．任意である転置による方式については，実行ファイルまたは鍵の論理ローテート，SWAP および Double SWAP 方式を同時に検索する．また，2 byte の 0〜f の 16 進数の文字列にエンコードする方式を同時に検索する．

その他の方式への対応

試験プログラムには，3 つの埋め込み方式の解読手法のほかに，ファイルの先頭から 1 byte ずつ順番に読み込み，次の 1 byte の値を鍵として XOR 演算を実施する方式の検索機能を実装した．

(2)　実行ファイルの検知手法

各手法によるエンコード方式の解読の成否の判定は，以下に示す，実行ファイルの検知手法によって実施する．

まず，PE ヘッダのシグネチャおよび MS-DOS スタブプログラムの文字列の

一部を検索する．各々の文字列の一部を発見した場合には，PE ヘッダのシグネチャまたは MS-DOS スタブプログラムの文字列のすべてを検索する．PE ヘッダのシグネチャまたは MS-DOS スタブプログラムの文字列が見つかった場合には，そのオフセット直前の MZ ヘッダのシグネチャを検索する．発見した MZ シグネチャのオフセットが，PE ヘッダのシグネチャまたは MS-DOS スタブプログラムの文字列のオフセットから一定の範囲内であれば，実行ファイルを検知したものと判定する．このオフセット間の距離の製薬は，PE ヘッダのシグネチャまたは MS-DOS スタブプログラムの文字列と，MZ ヘッダのシグネチャが同一の実行ファイルのものであることを確認するために設定している．なお，今回の実験では，MS-DOS スタブプログラムの文字列には “This program” を用いることとする．

3.6.5 実験

提案するエンコード方式の解読手法および実行ファイルの検知手法の効果を確認するため，実装した試験プログラムを用いて実験を実施した．

(1) 実験内容

実験の対象となる文書ファイルは tar(09–12) である．これらのマルウェアを試験プログラムに入力し，実行ファイルの検知率と採取した当時の最新のパターンファイルを適用した大手ベンダのウイルス対策ソフトの検知率を比較する．実験を実施する環境は表 3.10 に示すとおりである．

表 3.10 Experimental environment.

CPU	Core i5-3450 3.1 GHz
Memory	8.0 GB
OS	Windows 7 SP1
Virtualization software	VMware Workstation 9
Memory (VM)	512 MB
OS (VM)	Windows XP SP3
Interpreter (VM)	Python 2.7.6

(2) 実験結果

検体の拡張子ごとの実行ファイルの検知率を表 3.11 に示す．表中の「Num.」欄の左側は検知できた検体数で，右側は入力した dropper の数を示している．370 個の検体のうち，実行ファイルが含まれていた検体は 362 個であり，この内の

96.1 %にあたる 348 個の検体を検知することができた．さらに，検知した検体から取り出したファイルは，実行ファイルとして動作することを確認した．この結果から，実行ファイルの判定は正しく動作したものと判断できる．

表 3.11 Detection rates of executable files.

Ext.	Num.	Rates (%)
rtf	97 / 98	99.0
doc	34 / 36	94.4
xls	45 / 48	93.8
jtd/jtdc	12 /17	70.6
pdf	160 / 170	94.1
Total	348 / 362	96.1

次に，ウイルス対策ソフトとの検知率の比較結果を表 3.12 に示す．試験プログラムの実行ファイルの検知数をマルウェアの検知率と見なすと，試験プログラムで検知した 348 個の検体は，370 個の検体のうちの 94.1 %にあたる．これに対し，その検体を採取した当時の最新のパターンファイルを適用した大手ベンダのウイルス対策ソフトでは，12.7 %から 19.5 %の低い確率でしかマルウェアを検知することができなかった．3 種類のウイルス対策ソフトを組み合わせた場合でも，検知率は 4 割未満にとどまった．

表 3.12 Comparing detection rates with antivirus softwares.

	Num.	Rates (%)
試験プログラム	348 / 370	94.1
S 社 AV	72 / 370	19.5
T 社 AV	58 / 370	15.7
M 社 AV	47 / 370	12.7
S，T，M 社 AV	131 / 370	35.4

3.6.6　考察

試験プログラムを用いた実験の結果を踏まえ，実行ファイルの検知に失敗した原因について考察する．さらに，エンコード方式ごとの検知率から，効率的な実行ファイルのエンコード方式の解読手法を導出し，最後に提案手法の効果と限界について考察する．

(1)　検知に失敗した原因

まず，試験プログラムが実行ファイルの検知に失敗した原因について考察する．

試験プログラムでは，370 個の検体のうち，実行ファイルが含まれていなかった 8 個の悪性文書ファイルを検知することができなかった．これらの悪性文書ファイルは，shellcode がネットワーク経由で実行ファイルをダウンロードするタイプのマルウェアであった．実行ファイルを悪性文書ファイルに埋め込まず，ネットワーク経由でダウンロードするマルウェアに対しては，OfficeMalScanner[9] のように shellcode を検知する手法が有効である．

実行ファイルが含まれていた 362 個の検体のうち，14 個の悪性文書ファイルの実行ファイルを検知することができなかった．これらの悪性文書ファイルには，VBScript，Action Script，独自のストリーム暗号等を用いて実行ファイルがエンコードされていた．VBScript，Action Script によるエンコード方式に対しては，実行ファイルの検知に用いるシグネチャを追加することで，対応することは可能である．しかしながら，独自のストリーム暗号を用いた方式に対しては，shellcode から初期化ベクトル（同じ暗号鍵で異なるストリームを生成するために用いるビット列）を特定し，キーストリームを生成してエンコードする必要があるため，対応は困難である．このような複雑なエンコード方式に対しては，固有のシグネチャを作成して対応せざるをえないものと考える．

(2)　提案手法の効果と限界

試験プログラムは，標的型攻撃に用いられる悪性文書ファイルのうち，dropper を検知することを目的としている．試験プログラムでは，downloader は検知することができない．しかしながら，今回検証した tar(09–12) は，2012 年に発生した標的型攻撃のメールに添付された悪性文書ファイルのほとんどを占めている [44][45]．また，2009 年から 2012 年の間に発生した標的型攻撃では，97.8 %の悪性文書ファイルが dropper であった．したがって，実行ファイルの検知率を向上させることができれば，多くの標的型メール攻撃に用いられる悪性文書ファイルを検知することが可能になるものと考えられる．

試験プログラムが対応することができる実行ファイルのエンコード方式を表 3.13 に示す．試験プログラムでは，論理ローテートや SWAP を組み合わせた様々な XOR を基本とするエンコード方式や，16 進数の文字列にエンコード方式に対応することが可能である．しかしながら，VBScript，Action Script，独自のストリーム暗号等を用いたエンコード方式や，その他の変則的なエンコード方式には対応

することはできない．これらの方式に対応するためには，固有のシグネチャ等を導入して試験プログラムを改良する必要がある．これらの方式を用いた悪性文書ファイルが増加すると，この検知方式は，未知のマルウェアを用いた攻撃に先行して対応することが困難となる．

表 3.13 Encoding methods that build test program can detect.

対応可能	対応不可能
XOR	VBScript
Multibyte XOR	Action Script
NULL-Preserving XOR	独自のストリーム暗号
Multibyte NULL-Preserving XOR	その他
Rolling XOR	
論理ローテート	
SWAP	
16 進数の文字列	

3.7 まとめ

本章では，標的型メール攻撃に使用された悪性文書ファイルを中心に予備的な調査を行った．その結果，2013 年までの標的型攻撃に利用されている文書ファイルは RTF，CFB および PDF の 3 種類で占められており，そのほとんどは dropper であることが分かった．さらに，データセットの tar(09–12) の使用する脆弱性は，国内で発生した標的型攻撃で使用された悪性文書ファイルの使用する脆弱性を包含していた．このことは，データセット tar(09–12) に対して有効な対策は，国内の標的型メール攻撃に対して有効な対策である可能性を示している．そこで，実行ファイルがどのように悪性文書ファイルに埋め込まれているのかを調査し，その方式を体系化して整理するとともに，各々の方式の特徴について考察した．そして，各々の特徴を考慮して悪性文書ファイルからの実行ファイルの埋め込み方式を解読し，実行ファイルを検知する手法を提案した．さらに，各々の提案手法を試験プログラムに実装し，実験によりその効果を定量的に評価した．最後にその実験結果から，実行ファイルに着目した検知手法の効果と限界を明らかにした．

第4章

ファイルフォーマットからの逸脱に着目した検知方式の提案

4.1 悪性文書ファイルの構造と閲覧ソフトの動作

悪性文書ファイルは，閲覧ソフトの脆弱性を突くため，ある程度文書ファイルとしての体裁を整えておかなければならない．ここでは，ある悪性文書ファイルを例に，このファイルを読み込む閲覧ソフトの動作を示すことで，攻撃手段として文書ファイル形式を選択することで受ける攻撃者の制約を明らかにする．

4.1.1 読み込む悪性文書ファイル

閲覧ソフトの動作を示す例として，閲覧ソフトに読み込ませる悪性文書ファイルの概要を表 4.1 に示す．この検体は 2013 年から 2014 年の間に VirusTotal[47] に登録された PDF ファイルで，“CVE” をキーワードに検索することで得られた検体である．

表 4.1 Summary of the specimen.

Type	PDF
File Size	5,566
Hash(SHA256)	3f52a11cfb979bc3e56eca785c2c56d5cd0583700b2b159798915b5f8a9bc376
Vulnerability	CVE-2010-2883
Exploit	JavaScript

この検体のファイル構造およびドキュメント構造を図 4.1 示す．

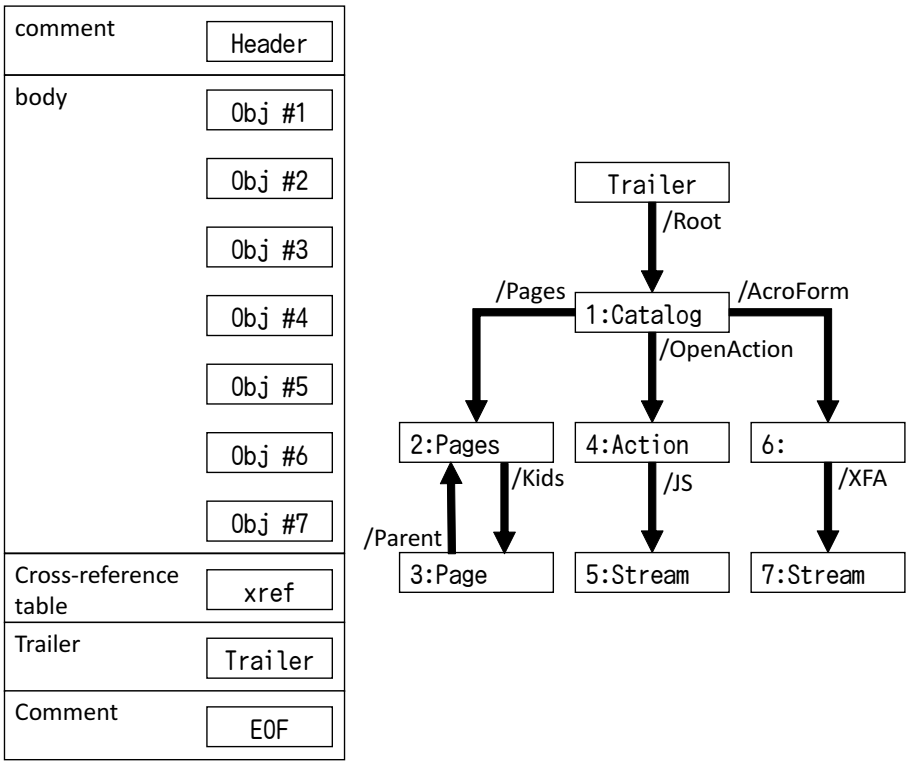

図 4.1 Structure of the specimen.

図 4.1 の左側がこの PDF ファイルのファイル構造である．このファイルは 5 つのセクションに分割され，body というセクションにオブジェクト番号 1 から 7 までの 7 個のオブジェクトが格納されている．

図 4.1 の右側がこの PDF ファイルのドキュメント構造である．四角は各オブジェクトを示しており，中の文字はオブジェクト番号とオブジェクトのドキュメントにおける役割を記載している．オブジェクトの役割は “/Type” タグを元に判断した．矢印はオブジェクトから別のオブジェクトに対する間接参照を示しており，矢印の根本の文字列は間接参照の元となったタグ名を記載している．

Exploit である JavaScript の本体はオブジェクト番号 5 の中に Flate 圧縮されて格納されている．

4.1.2 閲覧ソフトの典型的な動作

この検体を閲覧ソフトが読み込んだ際の典型的な動作を以下に示す．

(1) EOF マーカの検索

閲覧ソフトは，最初，文書を構成する各オブジェクトの位置がわからない．オブジェクトの位置を特定するためには，ファイルの最初から順番に構文解析してオブジェクトの位置を特定する方法がある．一方，多くの閲覧ソフトは効率化の

ため，ファイルの末端に記録されているメタデータを活用する．PDF ファイルの末端を示す EOF マーカ（%%EOF）は，メタデータを効率的に探索する要となるものである．

そこで，閲覧ソフトは，PDF ファイルを読み込んだ際に，まずは PDF ファイルの末端を示す EOF マーカ（%%EOF）を検索する．PDF ファイルは元のファイルの状態を保持したまま「追記」することで編集することが可能である．その場合，1 つの PDF ファイル中に複数の EOF マーカが存在することになる．同様にメタデータも複数の場所に格納されているが，PDF ファイル内の各オブジェクトの最新の状態を正しく記録しているのは，最も最後に追記されたメタデータとなる．したがって，最も最後に追記されたメタデータを探索するには最もファイルの後ろに現れる EOF マーカの位置を取得する必要がある．そのため，閲覧ソフトは，PDF ファイルの末端から先頭に向かって EOF マーカを検索していく．

(2) Trailer の読み込み

EOF マーカの位置の取得後，閲覧ソフトはメタデータである cross-reference table の位置を取得する．cross-reference table とは，各オブジェクトの位置を一覧化したもので，閲覧ソフトはこれを読み込むことで PDF 内のオブジェクトへのランダムアクセスが可能となる．EOF マーカの 2 行前に `startxref` というキーワードの行があり，その次の行に cross-reference table の先頭を示すオフセットアドレスが記載されている（図 4.2）．

また，`startxref` からファイルの先頭に向かって読み込んでいくと `trailer` というキーワードの行がある．この 2 つのキーワードに挟まれた部分に Trailer 辞書オブジェクトが格納されている．ここには，cross-reference table で定義されているオブジェクトの数とドキュメント構造のルートオブジェクトへの間接参照が記載されている．

```
⋮
xref
0 8
0000000000 65535 f
0000000015 00000 n
0000000135 00000 n
0000000213 00000 n
0000000271 00000 n
0000000340 00000 n
0000004880 00000 n
0000004913 00000 n
trailer
<</#53#69ze 8/#52#6f#6f#74 1 0 R>>
startxref
5341
%%EOF
```

図 4.2　Metadata of the specimen.

今回の例（図 4.2）の場合，cross-reference table の先頭のオフセットアドレスは 5341 である．Trailer 辞書オブジェクトは，単純な正規表現による検知を回避するため，単純な難読化が施されている．これは，PDF の標準機能を用いたものである．PDF の各辞書オブジェクトには，様々なタグが記載されている．このタグの “/” という制御文字を取り除いた名前部分の任意の文字について，“#” という制御文字とその文字に対応する 16 進数の文字列の組み合わせで表記することができる．例えば，“/Size” であれば，“S” と “i” の 2 文字だけ 16 進数の文字列で表記すると，“/#53#69ze” となる．難読化を解除すると “<</Size 8/Root 1 0 R>>” となり，cross-reference table で定義されているオブジェクトの数は合計 8 個であり，ルートオブジェクトのオブジェクト番号は 1 であることがわかる．

(3)　cross-reference table の読み込み

cross-reference table を読み込むと，最初の行には xref というキーワードがある．その次の行には，cross-reference table で定義されるオブジェクトの最初のオブジェクト番号とオブジェクトの量が記載される．

今回の例（図 4.2）の場合，オブジェクト番号 0 から 7 までの 8 個のオブジェクトのオフセットアドレスと使用状況が cross-reference table によって定義されていることがわかる．

cross-reference table を読み込んだ後，閲覧ソフトは PDF ファイル内の各オブジェクトに対するランダムアクセスが可能となる．

(4) ドキュメント構造の構築およびページの描画

次に，Trailer 辞書オブジェクトに定義されたルートオブジェクトを読み込む．今後，ルートオブジェクトを起点に参照されるオブジェクトを順番にたどっていき，ドキュメント構造を構築していく．これにより，各オブジェクトが文書を表示する上でどのような役割を担っているか明確化される．表示の高速化のため，閲覧ソフトは，描画の最初のページに関する情報等の表示に必要な部分を読み込んだ時点で，PDF の読み込み作業と並行して描画を開始する．この仕組みは，インターネット等で PDF ファイルを表示する際に，ファイル全体のダウンロードが完了する前に PDF の表示を可能とするために設けられた仕組みである．

今回の例の場合のルートオブジェクトを図 4.3 に示す．図の左側が元の定義のままであり，検知を回避するため簡単な難読化が施されている．図の右側は難読化を解除したものである．

```
1 0 obj
<<
/P#61g#65s 2 0 R
/Ty#70e /#43#61#74a#6c#6f#67
/#4f#70e#6eA#63t#69#6fn 4 0 R
/Ac#72oF#6f#72m 6 0 R
>>
endobj
```

```
1 0 obj
<<
/Pages 2 0 R
/Type /Catalog
/OpenAction 4 0 R
/AcroForm 6 0 R
>>
endobj
```

図 4.3 Root object of the specimen.

4.1.3 Exploit が動作するまでに最低限読み込むもの

今回の例における，Exploit が動作するまでの閲覧ソフトの動作を図 4.4 に示す．

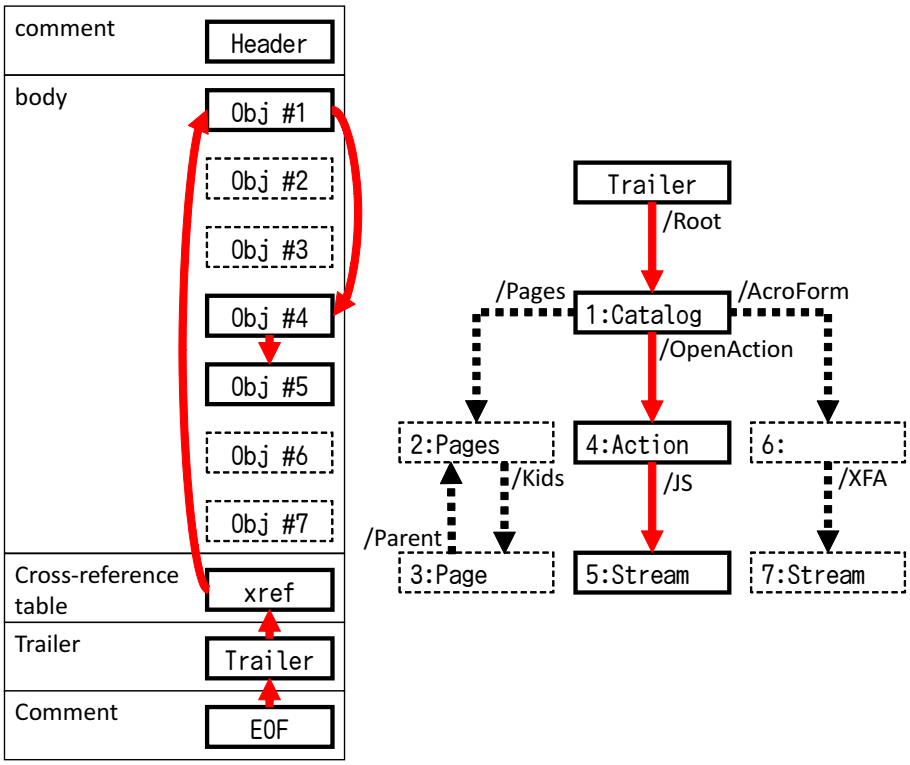

図 4.4 Structure of the specimen.

Exploit が動作するまでに，閲覧ソフトは「EOF」,「Trailer」,「xref」,「Obj #1」,「Obj #4」,「Obj #5」の順に処理をしていく．ルートオブジェクトに”`/OpenAction`“というファイルを開いた際に自動的に実行される命令を示すタグが定義されており，最終的に「Obj #5」に格納されている圧縮された JavaScript を展開し，実行することで Exploit が動作する．つまり，今回の場合，Obj #5 に Exploit が格納されていおり，Obj #5 が PDF ファイルを開いた際に自動実行される JavaScript であると閲覧ソフトが構文解析することによって，Exploit の動作が可能となる．したがって，悪性文書ファイルの動作には，ある程度 PDF の仕様を満たした PDF ファイルである必要がある．今回の例で示すと，図 4.4 のオブジェクトのうち，PDF の仕様を満たす必要のあるオブジェクトは実線で示している．

4.2 文書ファイルの構造に着目した検知方式と課題

このように，閲覧ソフトが，悪性文書ファイルを文書ファイルとして処理するためには，ある程度ファイルフォーマットの仕様に沿ったファイルでなければならない．つまり，その部分については，攻撃者にある種の制約がかかることになり，複雑な検知回避技術を適用することが難しくなってくる．

この点に着目した検知方式が文書ファイルの構造に着目した検知方式である．この方式は，exploit，shellcode，実行ファイル等の不正なコード以外の文書ファイルの構造部分に着目して検知を行う．

既存の研究は教師あり機械学習を用いたものが多く [24][25][26]．学習用のサンプルを集める必要があるだけでなく，その精度が学習用のサンプルに依存する．標的型攻撃では，以下の 2 つの理由により学習用のサンプルを集めることが困難な状況となっている．

4.2.1 検体そのものに機微な情報が含まれていること

標的型攻撃に使用されたマルウェアの収集の課題の 1 つは，マルウェアそのものに攻撃対象となった組織または関連組織のの内部情報が含まれていることがあるという点である．標的型メール攻撃に使用される dropper には 2 つのファイルが埋め込まれている．1 つは，おとりの文書ファイルであり，もう 1 つはマルウェアの本体となるプログラムである．それぞれのファイルに攻撃対象の機密情報が含まれていることがある．実際に機密情報が含まれるか否かは，検体を詳細に解析しなければ判断できず，共有のためだけにそのような解析を行うことは難しい．含まれる可能性のある機密情報の例を以下に示す．

(1) おとりの文書ファイル

おとりの文書ファイルは，攻撃の対象にマルウェアの感染に気づかせないようにするために使用される．悪性文書ファイルを開いた際に，閲覧ソフトに無害のおとりの文書ファイルを表示させ，バックグラウンドでマルウェア本体のプログラムを実行させる．これにより，閲覧者は攻撃に気づきにくくなる．この際に表示されるおとりの文書ファイルとして，攻撃対象の組織や関連組織の内部でやりとりしている情報が含まれた文書ファイルが使用されることがある．この文書ファイルは，インターネット上で公開されている情報をもとに作成されたり，別の攻撃により情報窃取した情報をもとに作成されたりする．この文書ファイルには，攻撃の対象となった組織を特定する情報や内部情報が含まれているため，情報共有がしづらくなる．

(2) マルウェア本体

マルウェア本体となるプログラムにも攻撃対象となった組織に関連する情報が記録されていることがある（図 4.5）．例えば，C2 サーバに接続を試みた端末をどの攻撃によるものなのか攻撃者が自ら識別できるようにするため，C2 サーバへ送信する情報に「キャンペーンコード」と呼ばれる識別子を埋め込むことがある．この識別子の中には，攻撃者が「いつ」「どの組織を対象に」攻撃したか容易に認識できるように，日付や攻撃対象の組織名が判別可能な名前が付けられることがある．例えば，識別子を見ただけで，2012 年 1 月 23 日に警察庁に対して送信した標的型メール攻撃により感染した端末が C2 サーバに接続を試みたことが分かるように，識別子を「20120123_npa」としたりする．このマルウェアを共

有した場合，攻撃対象となった組織や時期が特定されてしまうため，攻撃対象となった組織がマルウェアを情報共有する際の障壁となることがある．

また，ある一定以上の規模の組織では，内部ネットワークから外に出る際にプロキシサーバ経由でなければ外部にアクセスできないようにシステムを構築していることがある．このような場合，マルウェアはプロキシ経由で C2 サーバにアクセスする必要があり，その実装方法はいくつかある．その 1 つとして，攻撃対象組織のプロキシの情報をマルウェア本体に設定情報として直接埋め込むというものがある．このプロキシの情報は，別の攻撃により攻撃者が既に入手している情報と思われる．このプロキシに関する情報は，攻撃対象となった組織の内部ネットワークの設定を推測する一助となる．このような検体を共有することは，攻撃対象となった組織自らが，システムの弱点を晒すことにつながる．このため，攻撃対象となった組織がマルウェアを情報共有する際の障壁となることがある．

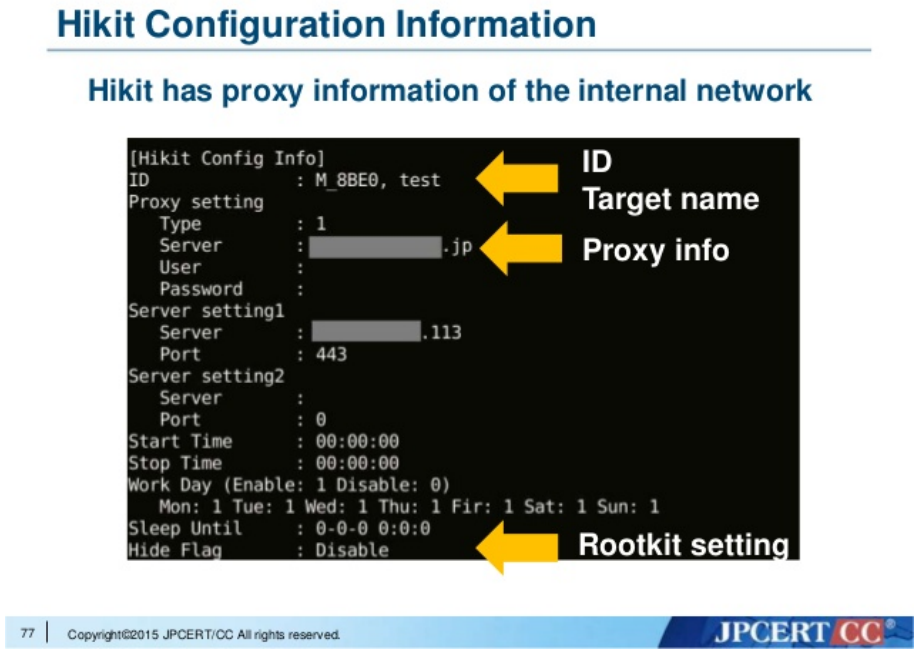

図 4.5　マルウェアに埋め込まれた組織の内部情報の例．朝長ら発表資料 [48]（CODE BLUE 2015[49]）より引用

4.2.2　標的型攻撃を受けている組織の特定

標的型攻撃に用いられたマルウェアは，既に述べたとおり，積極的には共有されない傾向がある．したがって，標的型攻撃に用いられたマルウェアを収集する

には，標的型攻撃の対象となっている特定の個人や組織へ個別に協力を得る必要がある．

標的型攻撃は，その名のとおり，攻撃者は特定の個人や組織を狙って攻撃を行う．不特定多数へ攻撃を行うバラマキ型の攻撃と比較して，攻撃の絶対数が少なく，攻撃の対象となる組織も限定されている．そのため，まずは，攻撃対象となっている特定の個人や組織の特定を行うこととなる．しかしながら，標的型攻撃を受けたとしても，不審さに気づきメールを開封しないなど攻撃が不発に終わった場合，企業は攻撃の被害を受けていないため，標的型攻撃を受けたことを公表することは稀である．また，攻撃に成功している標的型攻撃は，攻撃に気づかれないように巧妙に細工がされており，標的型攻撃の標的となった企業のネットワーク上で攻撃が発見されるまでの潜伏期間は平均で 200 日以上と言われている [50]．攻撃に気づかないため，当然攻撃を受けたことは公表されない．攻撃に気づいたとしても，積極的に公表するか否かは各組織の判断である．公表されている事例 [1] を見ると，企業の場合は顧客に実質的な被害があるか否かがひとつの判断基準で，顧客に被害がなければ公表していない企業が多いように思われる．

したがって，標的型攻撃を受けている個人・組織を把握することは困難な状況である．

4.3　ファイルフォーマットからの逸脱に着目した検知方式の提案

標的型攻撃に用いられたマルウェアを収集するには，4.2 で述べたようないくつかの課題を解決する必要がある．そこで，我々は，悪性文書ファイルの構造に着目しつつ学習用のサンプルを集める必要のない検知方式を提案する．

これまでに述べたとおり，悪性文書ファイルは，閲覧ソフトに文書ファイルとして処理される必要があることから，ある程度ファイルフォーマットの仕様に準拠している必要がある．

しかしながら，我々が実際の標的型メール攻撃に使用された文書ファイルを分析したところ，何らかの理由により文書ファイルのフォーマットの仕様から逸脱した構造をもつ悪性文書ファイルがほとんどであった．

そこで，仮にその逸脱構造が悪性文書ファイル特有なものであると仮定すると，逸脱構造を含む文書ファイルは悪性文書として検知できることになる．

我々が tar(09–12) の分析を行った結果，8 種類の逸脱構造（AS: Anomalous Structure）が明らかとなった．

- RTF: AS1
- CFB: AS2，AS3，AS4 および AS5
- PDF: AS6，AS7 および AS8

表 4.2 は，これら 8 種類の逸脱構造に基づいて tar(09–12) を分類した結果を示している．ほぼすべての検体を，これら 8 種類の逸脱構造のいずれかに分類することができる．

表 4.2 Rate of each anomalous structure.

Type	Anomalous structures	Num.	Rate
RTF	AS1	97 / 98	99.0 %
CFB	AS2	79 / 101	78.2 %
	AS3	92 / 101	91.1 %
	AS4	99 / 101	98.0 %
	AS5	98 / 101	97.0 %
	AS2, AS3, AS4, or AS5	100 / 101	99.0 %
PDF	AS6	81 / 163	49.7 %
	AS7	71 / 163	43.6 %
	AS8	102 / 163	62.6 %
	AS6, AS7, or AS8	162 / 163	99.4 %

次章以降では，この逸脱構造の詳細をファイルフォーマットごとに説明する．その後，この逸脱構造を検知するツールを実装し，この構造が，悪性文書ファイル特有のものであるか否か確かめる実験を行う．

第5章

悪性文書ファイルの仕様からの逸脱と検知

ここでは，悪性文書ファイルの仕様からの逸脱構造を示すとともに，その検知手法についても示す．

5.1 RTF

5.1.1 RTF ファイルの仕様の概要

悪性 RTF ファイルの仕様からの逸脱構造を明らかにするため，まず，RTF ファイルフォーマットの仕様の概要について示す．

RTF は Microsoft 社により開発された文書ファイルフォーマットである [7]．標準的な RTF ファイルは 7-bit ASCII 文字だけで記述されており，プレーンテキストに装飾やレイアウトのための制御用の文字列を付加した形式となっている．単純な RTF コードの例を図 5.1 に示す．

```
{¥rtf
Hello,¥par
{¥b world}!¥par
}
```

図 5.1 Example of an RTF file.

波括弧（“{” および “}”）はグループを定義し，グループは入れ子構造が可能である．バックスラッシュ “\”（日本語環境では円記号 “¥”）は，制御用の文字列の開始を意味する．通常，RTF ファイルの最初の文字は “{” であり，その後，RTF ファイルであることを示す制御文字列 “\rtf” が続く．ファイルの最後の文字は，ファイルの最初の “{” に対応する “}”（意味としては，EOF：End of File）となっている．閲覧ソフトは，EOF より後のデータについて，通常は何も処理しない．

5.1.2 AS1:EOF の後にデータの追記

(1) 逸脱の概要

仕様に沿った RTF ファイルでは，EOF がファイルの末尾となっていなければならない．しかしながら，分析した RTF ファイル 98 個のうち 97 個の RTF ファイルでは，EOF はファイルの末尾ではなく中間に位置していた．これらの RTF

ファイルでは，実行ファイルは，閲覧ソフトが処理を行わない EOF の後に埋め込まれていた．

(2) 検知手法

RTF ファイルを 1 byte ずつ読み込み，EOF に該当する “}” を読み込んだ時点で，まだ読み込まれていないデータがある場合に AS1 の検知とした．

5.2 CFB

5.2.1 CFB ファイルの仕様の概要

ここでは，CFB ファイルの仕様の概要について述べる．

CFB は Microsoft 社により開発された複合ファイルフォーマットの 1 つである [8]．CFB は，OLE（Object Linking and Embedding [51]）または COM （Component Object Model [52]）としても知られている．テキストや画像等様々なデータを階層構造を維持したまま 1 つの CFB ファイルに格納することが可能であり，CFB は様々な文書編集ソフト使用されている．

hwp 拡張子ファイルは，Hancom 社製の文書編集ソフトであるアレアハングルで使用されているファイル形式である．アレアハングルは韓国国内で広く使用されており，アレアハングルの脆弱性を狙った標的型攻撃も発生している [53]．OOXML [46] ファイル (docx，xlsx および pptx 拡張子ファイル) は通常，zip コンテナを使用しており，CFB フォーマットを使用していない．しかしながら，これらのファイルが暗号化されている場合には，CFB フォーマットが使用されることとなる．

以下に，CFB フォーマットを使用している代表的な文書編集ソフトを示す．

- Microsoft Word 2003 以前（doc 拡張子）
- Microsoft Excel 2003 以前（xls 拡張子）
- Microsoft PowerPoint 2003 以前（ppt 拡張子）
- 一太郎（jtd または jtdc 拡張子）
- アレアハングル（hwp 拡張子）
- Microsoft Word 2007 以降（docx 拡張子（暗号化されているものに限る））
- Microsoft Word 2007 以降（xlsx 拡張子（暗号化されているものに限る））
- Microsoft Word 2007 以降（pptx 拡張子（暗号化されているものに限る））

CFB ファイルの中には，storage と stream が階層構造を形成して格納されている．CFB の階層構造の例を図 5.2 に示す．CFB の階層構造は，ファイルシステ

ムの階層構造とよく似た構造となっており，ファイルに相当する stream とディレクトリに相当する storage の集合体となっている．

この階層構造を実現する CFB のファイル構造を図 5.3 に示す．CFB ファイルは，512 Byte のヘッダと sector と呼ばれる小さなブロックの集合で構成される．sector には 0 から始まる一連の番号が割り振られており，この数字は sector number と呼ばれる．stream のデータは sector に格納されるが，stream のサイズが sector のサイズより大きい場合，stream のデータは複数の sector に分割して格納される．

sector に分割して格納された stream は，sector chain により全体像を得ることができる．この sector chain は，論理的にバイト配列を形成するためのセクターのリンクリストである．sector chain の中では，sector number が，該当 sector に連結する sector を特定するために使用される．ただし，“−1” および “−2” は，特別な sector number として扱われる．“−1” は，該当 sector が free sector（未使用の sector）であることを示し，“−2” は，該当 sector が，sector chain の末端であることを示す．図 5.3 の左側は 2 つの sector chain を示している．ある sector chain（青）は，sector #2 から始まり，sector #8 で終わる．別の sector chain（赤）は，sector #5 から始まり，sector #7 で終わる．

sector chain は，FAT（File Allocation Table）において管理されている．各 sector に連結する sector の sector number は，32 bit（4 byte）整数で格納され，n 番目の sector に連結する sector の情報は，FAT の $n \times 4$ byte 目に格納されている．また，DE（Directory Entry）は，各 stream や storage について，名前，サイズおよび親子関係の情報を管理している．これら FAT や DE の位置についてはヘッダ内で定義されている．

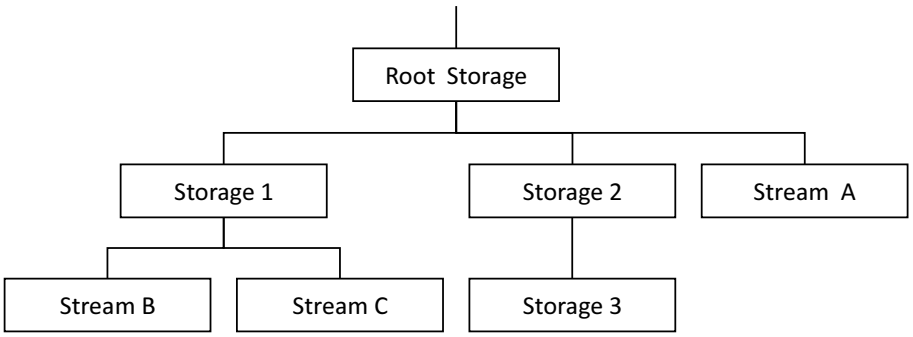

図 5.2　CFB hierarchy.

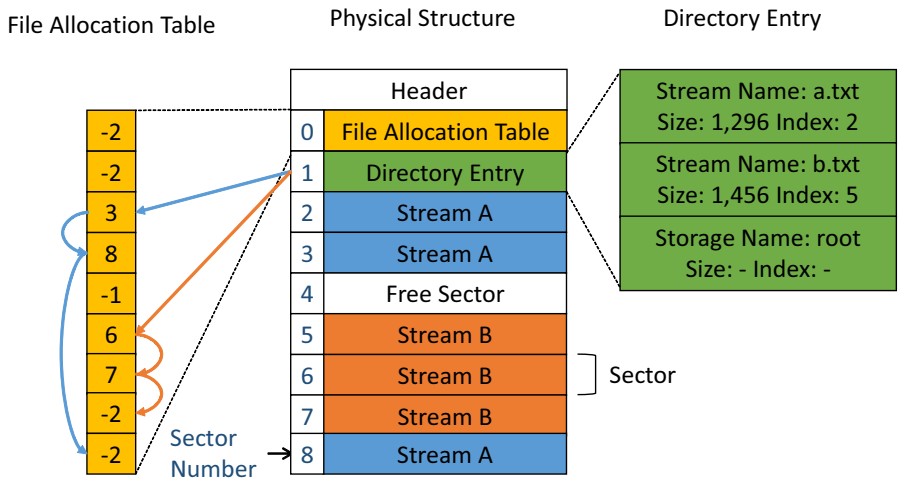

図 5.3　CFB structure.

文書編集ソフトが CFB ファイル内の a.txt という名前の stream を読み込む場合，その典型的な動作を図 5.3 を使って示す．まず，ヘッダの情報を元に FAT および DE の格納されている sector を特定する．次に，DE を読み込み a.txt という名前の stream を探す．DE には a.txt のデータが格納されている stream の先頭 sector の sector number が記録されており，当該 sector のデータを読み込む．読み込んでいる sector に対応する FAT 部分を読み込むと次に連結する sector があるか否か，ある場合はその sector number が分かり，連結する sector がなくなるまでデータの読み込みを続ける．このように，文書編集ソフトはヘッダ，FAT および DE の情報を元に文書ファイルの表示に必要な情報の読み込みを行う．

5.2.2　AS2: ファイルサイズの異常

(1)　逸脱の概要

仕様に沿った CFB ファイルのファイルサイズは，512（ヘッダサイズ）に sector サイズの倍数を合計したものとなる．ファイルサイズを $\mathrm{Size}_{\mathrm{file}}$，sector サイズを $\mathrm{Size}_{\mathrm{sector}}$ とし，整数 N を用いるとファイルサイズは，以下の数式で示せる．

$$\mathrm{Size}_{\mathrm{file}} = 512 + \mathrm{Size}_{\mathrm{sector}} \times N \tag{5.1}$$

したがって，以下の数式が成り立つ．

$$(\mathrm{Size}_{\mathrm{file}} - 512) \bmod \mathrm{Size}_{\mathrm{sector}} = 0. \tag{5.2}$$

しかしながら，分析した CFB ファイル 102 個のうち 79 個のファイルでは，式 (5.2) が成り立たなかった．この特徴を AS2 とする．図 5.4 は，AS2 の特徴を持つ CFB ファイルの例を示している．

AS2 の特徴を持つ原因として，埋め込まれている実行ファイルのファイルサイズが sector サイズの倍数となることは稀であり，CFB ファイルが secitor 単位で区切られているというファイル構造を無視して実行ファイルを CFB ファイルの末端に追記していることが考えられる．

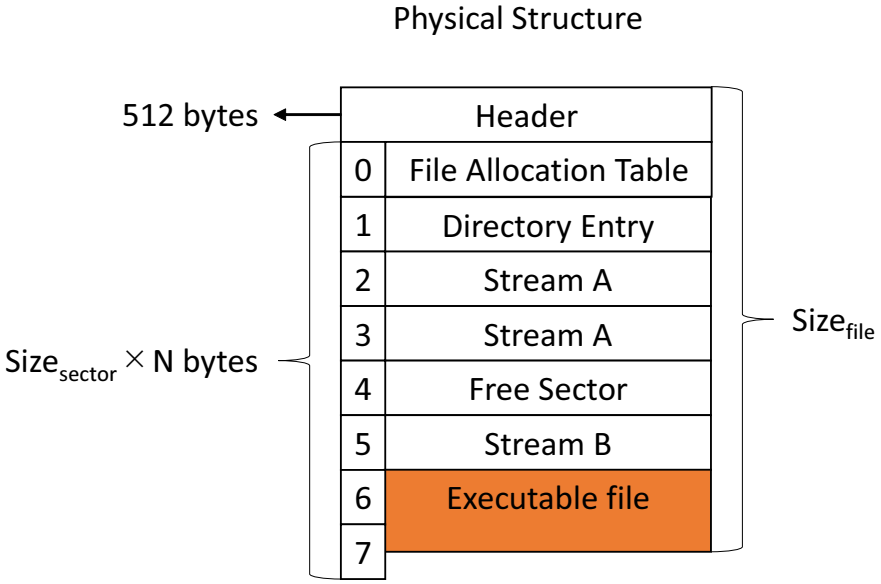

図 5.4　Example of a CFB file with an AS2 feature.

(2)　検知手法

ヘッダの 30 byte 目に sector サイズに関する情報が 2 byte の数値で格納されている．この値を SectorShift とすると，sector サイズ $\mathrm{Size}_{\mathrm{sector}}$ の値は $2^{\mathrm{SectorShift}}$ で表される．また，SectorShift の値は，0x0009 または 0x000C でなければならず，$\mathrm{Size}_{\mathrm{sector}}$ の値は 512 byte または 4096 byte となる．この値を用いて，式 (5.2) が成り立たない場合に AS2 の検知とした．

5.2.3　AS3: FAT で管理不可能領域のデータ

(1)　逸脱の概要

前に述べたとおり，FAT において各 sector に連結する sector の情報（sector number）は，32 bit（4 byte）の整数で管理されている．したがって，FAT 領域に割り当てられている sector1 個当たりで管理できる sector 数は，$\mathrm{Size}_{\mathrm{sector}} \div 4$ 個となる．FAT 領域に割り当てられている総 sector 数を $\mathrm{Count}_{\mathrm{FAT}}$ とし，その FAT 領域で管理できる領域の最大値を $\mathrm{Size}_{\mathrm{FAT}}$ とする．$\mathrm{Size}_{\mathrm{FAT}}$ は，以下の数式で表さ

れる．

$$\mathrm{Size}_{\mathrm{FAT}} = \mathrm{Size}_{\mathrm{sector}} \times \mathrm{Count}_{\mathrm{FAT}} \times (\mathrm{Size}_{\mathrm{sector}} \div 4). \tag{5.3}$$

文書編集ソフトは，通常，FAT により管理されていないデータを処理しない．文書ファイルという観点では，このような処理されないデータは無駄なデータとなる．このような無駄をなくすため，すべての sector は FAT により管理されるべきであるし，文書編集ソフトで作成された CFB ファイル内の sector はすべて FAT で管理されている．したがって，ファイルサイズからヘッダサイズの 512 を除いた値は，$\mathrm{Size}_{\mathrm{FAT}}$ 以下であるべきである．つまり，以下の数式が成り立つべきである．

$$\mathrm{Size}_{\mathrm{file}} - 512 \leq \mathrm{Size}_{\mathrm{FAT}}. \tag{5.4}$$

しかしながら，分析した CFB ファイル 102 個のうち 92 個のファイルで，FAT で管理できない領域にデータが含まれているものがあった．この特徴を AS3 とする．図 5.5 は，AS3 の特徴を持つ CFB ファイルの例を示している．

AS3 の特徴を持つ原因として，上記で述べた FAT に関するルールを無視して，CFB ファイルの末端に実行ファイルが追記された結果，そのファイルサイズは $\mathrm{Size}_{\mathrm{FAT}}$ を超えており，数式 (5.4) は成り立たなかったことが考えられる．

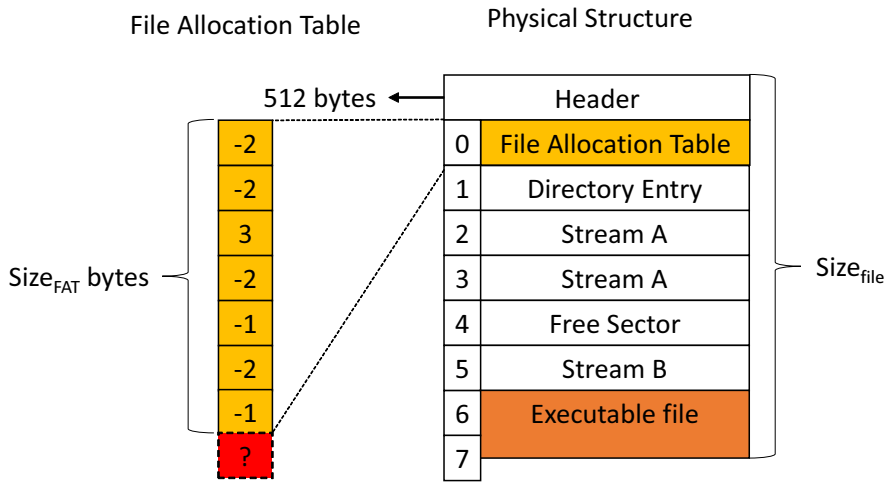

図 5.5　Example of a CFB file with an AS3 feature.

(2)　検知手法

ヘッダの 44 byte 目に FAT に割り当てられている sector の数が 4 byte で格納されている．この値を $\mathrm{Count}_{\mathrm{FAT}}$ とし，式 (5.3) から $\mathrm{Size}_{\mathrm{FAT}}$ を計算し，式 (5.4)

が成り立たない場合に AS3 の検知とした．

5.2.4 AS4: 末端の sector が free sector

(1) 逸脱の概要

前に述べたとおり，CFB の構造はファイルシステムによく似ている．例えば CFB によく似たファイルシステムとして FAT16 というファイルフォーマットがある．この FAT16 の場合，初期の状態では利用可能な空き領域は連続して確保されており，ファイルは連続した sector に格納される．しかし，ファイルの作成や削除を繰り返すことで，利用可能な空き領域は断片化して存在することとなる．同様な現象が CFB ファイルでも発生している．

CFB ファイルの場合も，文書の編集を繰り返す事で CFB ファイル内に未使用の領域（free sector）が点在してしまうことは一般的である．一般に，文書編集ソフトは free sector を処理しないため，free sector にどのようなデータを格納しても文書編集ソフトの動作に影響を与えない．したがって，free sector の中に秘匿したいデータを格納することが可能である．

攻撃者はこの仕組みを悪用し，free sector 内に実行ファイルを埋め込んでいる．分析した CFB ファイル 102 個のうち 99 個のファイルで，実行ファイルが free sector 内に埋め込まれていた．さらに，実行ファイルの埋め込まれている位置に着目すると，ファイルの末端であった．この特徴を AS4 とする．

図 5.6 は，AS4 の特徴を持つ CFB ファイルの例を示している．この場合，CFB ファイルのファイルサイズから逆算した末端の sector の sector number は 6 である．FAT を参照して，6 番目の sector の情報を調べると，“-1” となっており，ファイルの末端の sector が free sector であることが分かる．

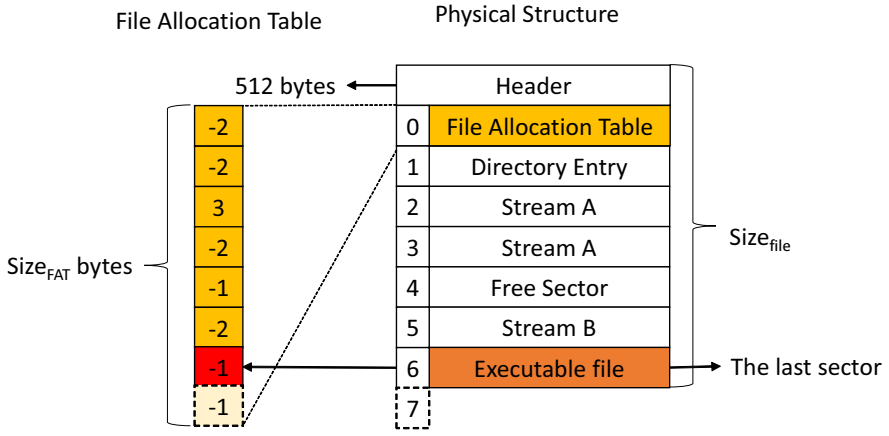

図 5.6 Example of a CFB file with an AS4 feature.

(2) ファイルの末端の free sector が不審な理由

ファイルの末端の free sector が不審な理由として，CFB ファイルの末端に free sector の追加や削除することは比較的容易にできるということがあげられる．一般に CFB ファイル内に free sector は点在しているものの，実行ファイルを埋め込めるほどの大きさの連続した空き領域があることは珍しい．実行ファイルを埋め込めるほどの大きな未使用領域のある CFB ファイルを一から作成することは，可能ではあるが，CFB の仕様を熟知する必要がある．既存の CFB ファイルを改変することにより，大きな未使用領域を確保する場合，未使用領域の位置が，CFB ファイルの途中か末端かで難易度が大きく異なる．

ファイルの途中に実行ファイルを埋め込めるほどの大きな未使用領域を確保しようとした場合，FAT が管理している sector chain に関するデータの大幅な並べ替えが必要である．さらに，DE で管理している各 steam の開始 sector の sector number 値の書き換えも必要となる．CFB ファイルの大幅な改変が必要であり，その難易度は一から CFB ファイルを作成することと大差ない．既存の CFB ファイルの構造分析が必要な分，場合によっては，一から CFB ファイルを作成するよりも難易度が高くなる．

一方，ファイルの末端に free sector を追加する場合，既存の CFB ファイルの改変は，ほぼ不要である．free sector を追加後のファイルサイズが，既に確保されている FAT 領域で管理できるサイズを超えた場合に，新たな FAT 領域の確保という CFB ファイルの改変が必要となるが，それ以外の場合は CFB ファイルの改変は不要である．

このように，ファイルの末端の free sector の追加することは比較的容易である，一方，標準的な文書編集ソフトで作成された CFB ファイルの末端は free sector ではない．この理由として，ファイルサイズの削減が考えられる．free sector は未使用領域であり，削除しても文書の中身には全く影響を与えない．一方，文書ファイルを編集するたびに，free sector をすべて削除した場合，編集箇所が文書の一部であっても，前に述べたとおり，既存の CFB ファイルの大幅な改変が必要となるため，CFB ファイル全体を更新する必要がある．小さな CFB ファイルであれば，問題にならないが，数十 MB にもなる CFB ファイルの場合，その作業にかかる時間は無視できなくなる．作業効率とファイルサイズの削減というバランスの中で，文書編集ソフトは，ファイルの末端の free sector を削除する一方，デフォルトのオプションによる保存方法ではファイルの途中の free sector は削除しないという実装を行っていると思われる．

(3) 検知手法

CFB ファイルのファイル末端に該当する sector の sector number を n とすると，CFB ファイルの中には 512 byte のヘッダと $n+1$ 個の sector がある．この

ため，$\mathrm{Size}_{\mathrm{file}}$ は以下の数式で表される．

$$\mathrm{Size}_{\mathrm{file}} = 512 + (n+1) \times \mathrm{Size}_{\mathrm{sector}}. \tag{5.5}$$

この式を n について解き，FAT を参照し，n 番目の sector の情報が “-1” (free sector) であった場合に AS4 の検知とした．

5.2.5 AS5: 使途不明の sector

(1) 逸脱の概要

CFB では，sector は，FAT，DIFAT（Double-Indirect FAT），miniFAT，DE，stream または free sector に分類される．ここでいう DIFAT は，FAT に割り当てられている sector を管理するための領域であり，miniFAT は，ヘッダで定義されたある一定サイズ未満の stream をまとめて管理するための領域である．

FAT, DIFAT, miniFAT および DE に関する情報はヘッダで定義され, 各 storage および stream に関する情報は DE で管理されている．また，free sector に関する情報は FAT で管理されている．したがって，一般に，CFB ファイルにあるすべての sector は，図 5.7 に示すようなヘッダを頂点とする階層構造を形成することができる．

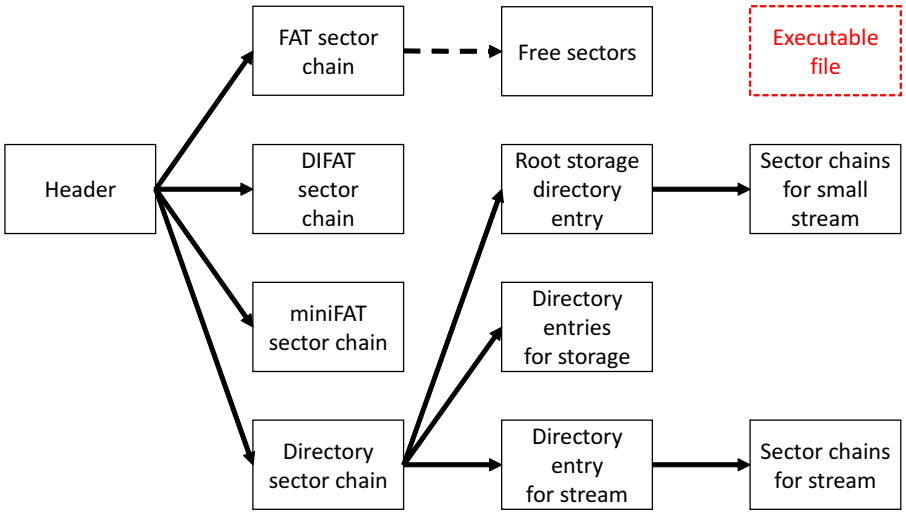

図 5.7 Example of a sector hierarchy.

しかしながら，分析した CFB ファイル 102 個のうち 98 個のファイルで，この階層構造を形成しない sector を含むファイルがあった．この特徴を AS5 とする．図 5.7 は，AS5 の特徴をもつ CFB ファイルの例を示している．ヘッダを起点に得られる情報を順番にたどる方法で sector を分類しようとした場合，実行ファイルを埋め込まれた領域に該当する sector の使途を明らかにすることはできない．

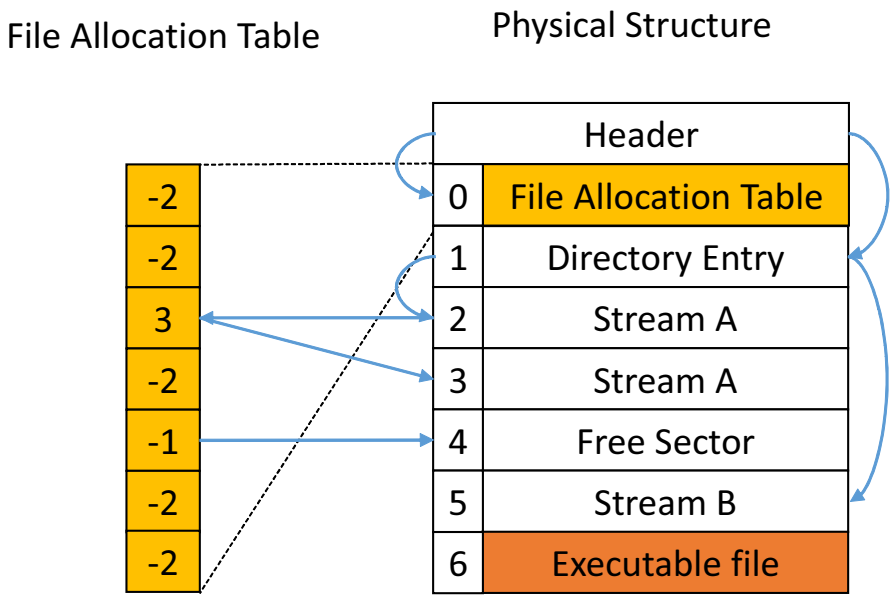

図 5.8 Example of a CFB file with an AS5 feature.

(2) 検知手法

使途不明の sector を検知する方法としては，ヘッダを起点として sector の階層構造を実際に形成し，その階層構造に使用されない sector を検知する方法がある．この方法では，CFB にある sector すべてについて，階層構造に使用されるか使用されないか調べることとなる．

一方，本論文では，実装がより単純で高速となるヘッダ等の情報を活用する方法を採用した．具体的には，CFB のヘッダには FAT 等に使用している sector 数が格納されており，この情報を活用して sector の種類ごとに数を数え，使途を明らかにできた sector 数とファイルサイズから逆算て求めた実際の sector 数を比較し，差異があった場合に AS5 の検知とした．以下，その詳細について述べる．

ファイルサイズから逆算して求めた実際の sector 数を $\mathrm{Count}_{\mathrm{real}}$ とする．CFB ファイルは，512 byte のヘッダと複数の sector の集合である．したがって，ファイルサイズは，512 と $\mathrm{Size}_{\mathrm{sector}}$ の倍数の合計で示され，以下の数式が成り立つ．

$$\mathrm{Size}_{\mathrm{file}} = 512 + \mathrm{Count}_{\mathrm{real}} \times \mathrm{Size}_{\mathrm{sector}}. \tag{5.6}$$

この数式を $\mathrm{Count}_{\mathrm{real}}$ について解くことで，$\mathrm{Count}_{\mathrm{real}}$ が得られる．

以下のとおり変数を定義する．

- $\mathrm{Count}_{\mathrm{FAT}}$：FAT に使用されている sector 数

- $\mathrm{Count}_{\mathrm{miniFAT}}$：miniFAT に使用されている sector 数
- $\mathrm{Count}_{\mathrm{DIFAT}}$：DIFAT に使用されている sector 数
- $\mathrm{Count}_{\mathrm{DE}}$：DE に使用されている sector 数
- $\mathrm{Count}_{\mathrm{Streams}}$：stream に使用している sector 数
- $\mathrm{Count}_{\mathrm{free}}$：free sector の数
- $\mathrm{Count}_{\mathrm{classified}}$：使途が明らかになった sector 数

仕様に沿った CFB ファイルは，以下の数式が成り立っていなければならない．

$$\begin{aligned}\mathrm{Count}_{\mathrm{classified}} = \mathrm{Count}_{\mathrm{FAT}} + \mathrm{Count}_{\mathrm{miniFAT}} + \mathrm{Count}_{\mathrm{DIFAT}} \\ +\mathrm{Count}_{\mathrm{DE}} + \mathrm{Count}_{\mathrm{Streams}} + \mathrm{Count}_{\mathrm{free}}.\end{aligned}$$

各変数の値は以下のようにして求める．

- $\mathrm{Count}_{\mathrm{FAT}}$ は，ヘッダの 44 byte 目に 4 byte の数値で格納されている．
- $\mathrm{Count}_{\mathrm{miniFAT}}$ は，ヘッダの 64 byte 目に 4 byte の数値で格納されている．
- $\mathrm{Count}_{\mathrm{DIFAT}}$ は，ヘッダの 72 byte 目に 4 byte の数値で格納されている．

DE が使用する sector のうち最初の sector の sector number（4 byte）が，ヘッダに格納されている．$\mathrm{Count}_{\mathrm{DE}}$ は以下の 3 ステップで求める．

(1) ヘッダの 48 byte 目から 4 byte の数値を読み込む

(2) 読み込んだ数値と FAT を元に，DE の sector chain の全体像を得る

(3) DE の sector chain が使用する sector 数を数える

$\mathrm{Count}_{\mathrm{Streams}}$ は，より複雑な手順で求める．

(1) 上記の手順で DE の sector chain の全体像を得る

(2) DE から全 stream のエントリを得る

(3) 各 stream について，stream の大きさから stream で使用する sector 数を求める

(4) 各 stream で使用する sector 数の合計を求める

上記 (3) の手順の詳細は以下のとおりである．n 番目の stream の大きさを Size_n，n 番目の stream が使用する sector 数を Count_n とする．Size_n は，n 番目のエントリの 120 byte 目に 4 byte の数字で格納されている．Size_n がある一定値より小さい場合，その stream は “root stream” と呼ばれる stream にまとめて格納され，Count_n は “0” となる．“root stream” に格納される stream の大きさの基準を $\mathrm{Size}_{\mathrm{mini}}$ とする．$\mathrm{Size}_{\mathrm{mini}}$ は，ヘッダの 56 byte 目に格納されている．Size_n が $\mathrm{Size}_{\mathrm{mini}}$ より大きい場合，Count_n は，Size_n を $\mathrm{Size}_{\mathrm{sector}}$ で割った値（小数点以下は切り上げ）と等しくなる．したがって，Count_n は次のように表される，

$$\mathrm{Count}_n = \begin{cases} 0 & (\mathrm{Size}_n < \mathrm{Size}_{\mathrm{mini}}) \\ \lceil \mathrm{Size}_n \div \mathrm{Size}_{\mathrm{sector}} \rceil & (\mathrm{Size}_n \geq \mathrm{Size}_{\mathrm{mini}}) \end{cases} . \tag{5.7}$$

$\mathrm{Count}_{\mathrm{free}}$ は，FAT で “−1” となっている sector の数を数えることで得られる．

仕様に沿った CFB ファイルの場合，$\mathrm{Count}_{\mathrm{real}}$ は，$\mathrm{Count}_{\mathrm{classified}}$ と等しくなければならない．$\mathrm{Count}_{\mathrm{real}}$ と $\mathrm{Count}_{\mathrm{classified}}$ の値に差異があった場合に AS5 の検知とした．

5.3 PDF

5.3.1 PDF ファイルの仕様の概要

ここでは，PDF ファイルの仕様の概要について述べる．

PDF は，Adobe 社によって開発された，文書を表示するためのファイルフォーマットである．PDF の仕様は，2008 年 7 月に ISO（国際標準化機構）によって標準化されている [54]．加えて，Adobe 社独自のバージョンアップも “Adobe Extensions” という形で行われている [55]．

(1) ファイル構造

PDF のファイル構造は 4 つのセクションで構成される（図 5.9）

- comment：ここに記述された情報はコメントとして扱われ，閲覧ソフトは特に処理を実施しない．ただし，`%PDF-n.m` は PDF のバージョンを示すヘッダとして使用され，`%%EOF` は PDF ファイルの終端を示すマーカとして使用される．
- body：ページコンテンツやグラフィックスコンテンツ等，多くの補助的な情報がオブジェクト一式としてエンコードされている．
- cross-reference table：body 中の各オブジェクトの位置を一覧化したもの．body に含まれるオブジェクトの数や各オブジェクトごとの開始オフセットおよびオブジェクトが閲覧ソフトの表示に使用されるものか否かが記述されている．

- trailer：トレーラ辞書というオブジェクトが格納されている．この中には PDF ファイル中に格納された様々なメタデータの位置が記述されている．

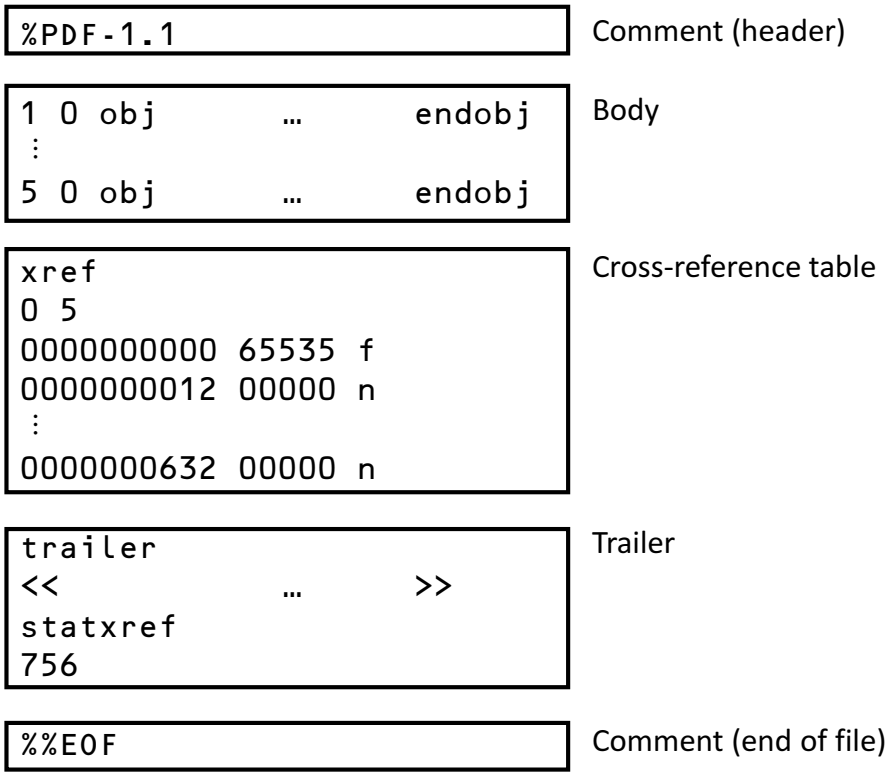

図 5.9 Structure of a PDF file.

trailer によって，閲覧ソフトは cross-reference table や特別な意味を持つオブジェクトを素早く見つけることができる．ファイルの最後の行は，ファイルの終了を示すマーカである`%%EOF` だけで構成される．その 2 行前は，`startxref` というキーワードだけの行で，その次の行で cross-reference table のセクションが始まるオフセットアドレスを示す数字が記述される．cross-reference table のセクションは，`xref` だけの行から始まる．トレーラ辞書は `startxref` 行の直前に配置され，`trailer` というキーワードだけの行から始まる．

(2) オブジェクト

PDF ファイルは，以下の 8 種類の基本的なオブジェクトで構成される．

- Boolean 値
- 数値（整数型および実数型）

- 文字列
- 名前
- 配列
- 辞書
- stream
- null

strean オブジェクトは，文字列オブジェクトに似ていて，バイナリデータを格納するために用いられる．stream は，辞書オブジェクトと 0 byte 以上のバイト列で構成される．このバイト列は，`stream` キーワード行と `endstream` キワード行に挟まれている（図 5.10）．stream には，バイト列を元のエンコードされていないデータにデコードするためのフィルタというオプションをつけることができる．フィルタは stream 内の辞書で定義することができる．標準的なフィルタの概要を表 5.1 に示す．

```
4 0 obj

<</Length 24 /Filter /ASCIIHexDecode>>

stream

48656C6C6F2C576F726C6421

endstream

endobj
```

図 5.10 Example of an object.

表 5.1 Examples of standard filters.

ASCII85 Decode	“{” から “z” までの印字可能文字を使用して表現した文字列をバイナリデータに変換するフィルタ
ASCIIHex Decode	2 桁の 16 進数の文字列をバイナリデータに変換するフィルタ
DCT Decode	JPEG による不可逆圧縮されたデータを展開するフィルタ
Flate Decode	オープンソースの zlib ライブラリで用いられている Flate 圧縮されたデータを展開するフィルタ
JBIG2 Decode	JBIG2 による圧縮されたデータを展開するフィルタ

オブジェクトは他のオブジェクトから参照できるようにラベルが付けられる．ラベルが付けられたオブジェクトは，間接参照オブジェクトと呼ばれる．間接参照オブジェクトは，そのオブジェクト番号，世代番号および `obj` キーワドを含む行と `endobj` キーワードだけの行に挟まれている（図 5.10）．

(3) ドキュメント構造

一般的な PDF ファイルのドキュメント構造を図 5.11 に示す．PDF のドキュメント構造はオブジェクトの階層構造となっている．ドキュメントカタログと名付けられたオブジェクトがドキュメント構造の頂点に位置するルートオブジェクトであり，その他すべてのオブジェクトはここからの間接参照を通じてアクセスされるようになっている．なお，ドキュメントカタログの位置はトレーラ辞書に格納されている．

図 5.11 中のドキュメント情報辞書（Document information）には，ファイルの作成日，更新日，著者名等が格納されている．また，ドキュメントカタログ（Document catalog）にはページツリーのルートオブジェクトへの間接参照，ドキュメントアウトライン（しおり）への間接参照等が格納されている．

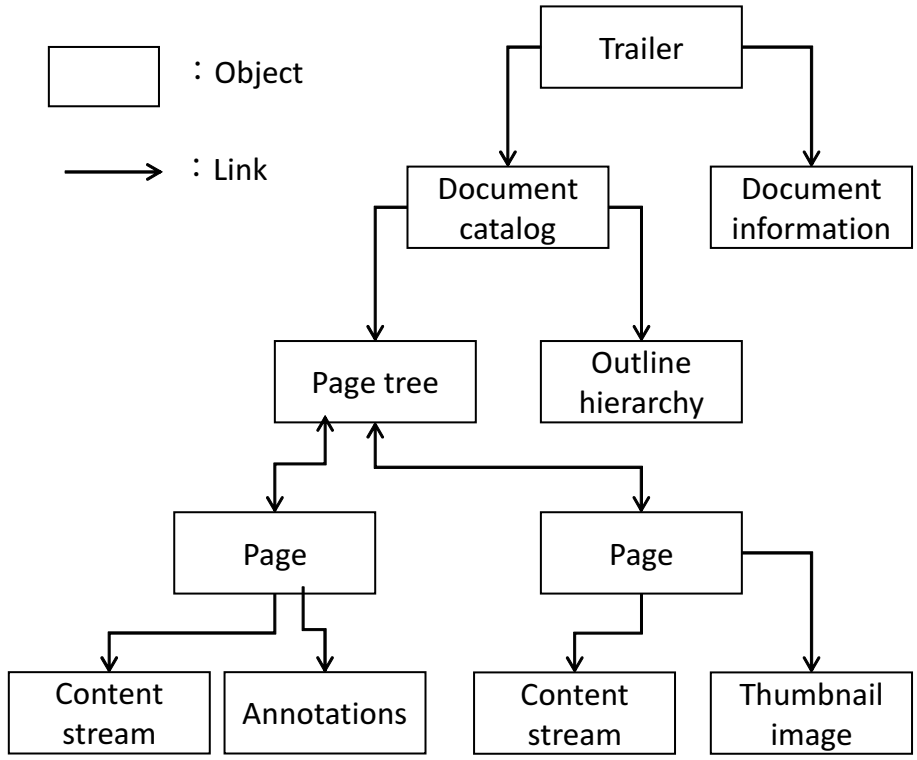

図 5.11　Structure of a PDF document.

(4)　ドキュメントの暗号化

PDF1.1 から，ドキュメントの暗号化機能がサポートされている．暗号化されている PDF ファイルには暗号化辞書というオブジェクトが格納されており，このオブジェクトへの間接参照がトレーラ辞書の “/Encrypt” キーに格納されている．暗号化辞書には，暗号化の方式等復号に必要な情報が格納されている．暗号化は基本的に，stream と文字列に適用され，数値やその他のオブジェクト型には適用されず，ファイル全体を暗号化することはない．したがって，復号しなくてもドキュメント構造にアクセスすることは可能である．

しかしながら，暗号化 PDF ファイルの中には，復号しなければドキュメント構造にアクセスすることができないものもある．PDF1.5 以降では，多くのオブジェクトを単一のオブジェクト stream という stream 内に格納し，その stream を圧縮することで PDF ファイルをさらにコンパクトなものにするという仕組みが導入されている．オブジェクト stream に格納されているオブジェクトは暗号化の対象となるため，オブジェクト stream のある暗号化 PDF ファイルの場合は復号化しなければ，ドキュメント構造にアクセスすることはできない．

暗号化された PDF ファイルを閲覧ソフトで開くと，パスワードの入力を求め

るプロンプトが表示される．一般に，暗号化されたPDFファイルは，ユーザパスワードおよびオーナパスワードの2種類のパスワードが設定されており，この2種類のパスワードのどちらを使用してもPDFファイルを復号することができる．ユーザパスワードが空の文字列（0 byte）である場合，暗号化されたPDFファイルを開いてもパスワードの入力を求めるプロンプトは表示されず，PDFファイルが表示される．そのため，印刷禁止の解除等のオーナパスワードの入力を求められる作業をしない限り，我々はそのファイルが暗号化されているか否か意識することはない．パターンマッチングによる検知を回避するため，悪性PDFファイルは，空のユーザパスワードを使用して暗号化されていることがしばしばある．

したがって，以下に示す悪性PDFファイルの検知手法については，暗号化への対応状況についても整理する．

5.3.2 AS6: 分類できないセクション

(1) 逸脱の概要

PDFファイル内のデータはすべてcomment，body，cross-reference tableおよびtrailerという4種類のセクションに分類される．

しかしながら，分析したPDFファイル164個のうち81個のファイルでは，この4種類のセクションに分類できない部分が存在するものがあった．この特徴をAS6とする．

(2) 検知手法

PDFファイルを1文字ずつ読み込み，4種類のどのセクションに該当するか分類を行う．各セクションの概要は以下に示す．

- commentは%文字で始まる行
- bodyは`obj`キーワードと`endobj`キーワードで挟まれた間接参照オブジェクトで構成されている．
- cross-reference tableは`xref`キーワード行で始まる．
- trailerは`trailer`キーワード行で始まる．

その結果，どのセクションにも分類できなかったデータがあった場合にAS6の検知とした．

(3) 検知手法の暗号化への対応状況

各セクションの分類に必要なキーワード行は，PDFの暗号化の対象となっていない．したがって，検査対象のPDFファイルが暗号化されていたとしても，この検知手法を用いて悪性文書ファイルか否かの検査をすることができる．

5.3.3 AS7: 参照されないオブジェクト

(1) 逸脱の概要

一般的なPDFファイルでは，cross-reference tableで未使用とされるオブジェクトやnullオブジェクトを除いたすべてのオブジェクトにドキュメントカタログからの間接参照を通じて，アクセスできるようになっている．

しかしながら，分析したPDFファイル164個のうち72個のファイルで，どのオブジェクトからも参照されていないオブジェクトが存在するものがあった．この特徴をAS7とする．

(2) 検知手法

PDFファイルの中にあるすべてのオブジェクトを読み込む．その後，すべての間接参照のリンク先を一覧にし，各オブジェクトがどのオブジェクトからリンクされているのかを調べる．その結果，どのオブジェクトからもリンクされていないオブジェクトがあった場合にAS7の検知とした．

ただし，物理的に実行ファイルを埋め込むことができない大きさのオブジェクトは検知対象から除いた．これはcross-reference tableで表示に使用するオブジェクトとなっているものの，どのオブジェクトからもリンクされていないオブジェクトが，いくつかの一般的なPDFファイルに見られたためである．実行ファイルのヘッダはMS-DOS用ヘッダ，MS-DOS用スタブプログラムおよびNULL領域で256 byte，NTヘッダおよびNULL領域で256 byteの合計512 byteの領域を使用する[56]ことから，検知対象から除くオブジェクトのサイズは512 byteとした．

(3) 検知手法の暗号化への対応状況

この検知手法は，検査対象のPDFファイルが暗号化されていたとしても，ほとんどの場合，PDFファイルの復号を行わずに検査を行うことができる．詳細を以下に示す．

PDFファイルの暗号化は，基本的にstreamおよび文字列に適用されるため，ドキュメント構造を検査するこの検知手法は，検査対象のPDFファイルが暗号化されていたとしても，ほとんどの場合，悪性文書ファイルか否かの検査を行うことができる．しかしながら，検査対象のPDFファイルの中にオブジェクトstreamが含まれていた場合，オブジェクトstreamは暗号化の対象であるため，復号化せずにオブジェクトstreamに格納されたオブジェクトへアクセスすることはできず，ドキュメント構造の全体像を把握することができない．そのため，ドキュメント構造の全体像を把握できなければ，どのオブジェクトからもリンクされていないオブジェクトがあるか否かも判断することはできない．

5.3.4 AS8: 偽装された stream

(1) 逸脱の概要

通常，PDF ファイルの stream は，stream とセットとなっている辞書に定義されたフィルタで問題なくデコードできる．また，デコードに使用するデータの大きさは stream 内のデータの大きさと等しい．

しかしながら，分析した PDF ファイル 164 個のうち 104 個のファイルでは，stream の中に実行ファイルが埋め込まれており，stream のデコード処理で通常と異なる挙動を示すものがあった．具体的には，以下のとおりである．

AS8-1: 偽装された filter

実行ファイルを埋め込まれた stream の中には実際の中身と定義されたフィルタが対応していないものがあった．攻撃者によりフィルタが偽装される原因として以下のことが考えられる．

実行ファイルはエントロピーが大きいという点で，圧縮されたデータと似た特徴を持つ．エントロピーは情報のランダムさを示す値で，小さいほど規則性があり，大きいほどランダムに近い状態であることを示す．エントロピーという点では，偽装されたストリームと圧縮されたデータを格納する stream を区別することは難しい．したがって，FlateDecode 等のデータが圧縮されていることを示すフィルタは，分析した PDF ファイルで広く使われていた．

この場合，stream の中身とデコード処理に使用するフィルタが対応していないため，stream のデコード処理は失敗する．仮に，この部分が閲覧ソフトに読み込まれると，閲覧ソフトが誤動作したり表示内容が文字化けする可能性が高い．それを防ぐため，フィルタ偽装をして実行ファイルを埋め込まれた stream は，どのオブジェクトからも参照されないようにすることが多い．したがって，AS7 の特徴を併せて持つことが多い．

AS8-2: EOD の後に追記されたデータ

stream 内に実行ファイルなどの余分なデータが含まれているものの，デコード処理がなんの問題もなく終了するものがある．その原因としてデータの末端を示す end-of-data（EOD）マーカが考えられる．

ほとんどのフィルタは，自らのデータを制限する情報が定義されている．この場合，stream のデータの末端には，データの末端を示す EOD マーカが記録されている．そのため，当該フィルタでデコードする stream の末端に別のデータを追加しても，追加したデータがデコード処理に使用されない以外は問題なく閲覧ソフトは動作する．この特性を利用して，攻撃者は stream 末端に実行ファイルを追記している．

(2)　検知手法

PDF ファイルの中にあるすべての stream を読み込む．FlateDecode，ASCIIHexDecode または ASCII85Decode がフィルタに指定されている場合は，デコードを試みる．この時，stream に格納されているデータが各種フィルタの形式に沿っていないと，デコードに失敗する．この場合を AS8 の検知とした．

また，FlateDecode，DCTDecode または JBIG2Decode がフィルタに指定されている場合は，データの終端を示すマーカの位置と stream の末端の位置を比較することにより，デコードに使用していないデータの有無について調べる．デコードに使用していないデータがあった場合についても AS8 の検知とした．

(3)　検知手法の暗号化への対応状況

この検知手法は，検査対象が暗号化される stream であるため，検査対象の PDF ファイルが暗号化されている場合，PDF ファイルの復号を行わなければ検査を行うことができない．

第6章

検知ツールo-checkerの開発

6.1 開発

文書ファイルの構造を構文解析する既存のツールはいくつかある．例えば, RTF ファイルおよび CFB ファイルを解析するものとして python-oletools[57] があり，PDF を解析するものとして PDFMiner[58] などがある．しかしながら，逸脱構造の検知と構文解析は密接に関わっており，文書ファイルのコンテンツを取り出すことを主な目的とした既存のツールを，逸脱構造の検知に活用することは困難であった．加えて，既存の PDF の構文解析ツールで AES を用いた暗号化方式に対応しているものが非常に限られていた．

そこで，これまでに述べた悪性文書ファイルの 8 種類の逸脱構造を検知するため，「o-checker」という検知ツールをオープンソースのプログラミング言語である Python[59] を用いて構文解析部分も含めて実装することとした．

o-checker はコマンドラインプログラムであり，引数として受け取った文書ファイルが悪性文書ファイルか否かを検査するプログラムである．o-checker による文書ファイルの検査の流れを図 6.1 に示す．

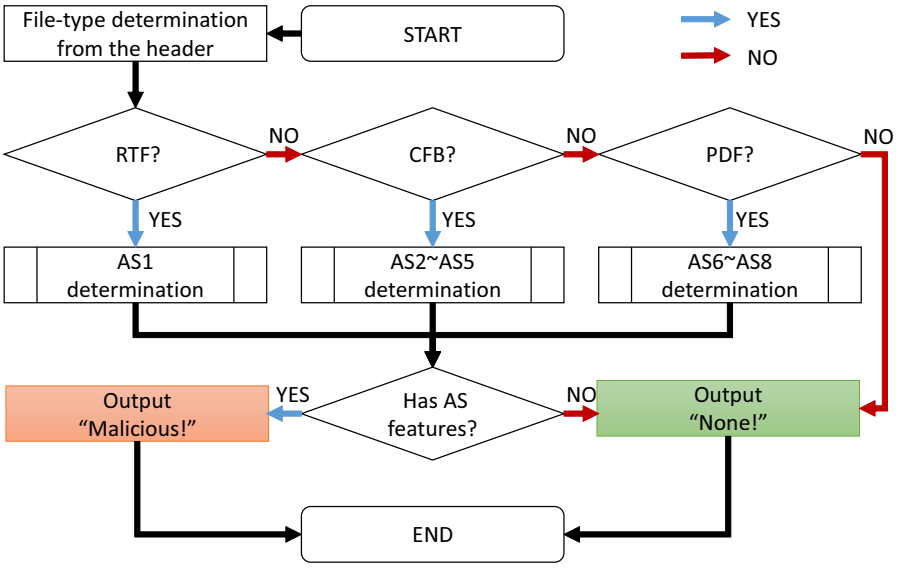

図 6.1 Flow of checking a document file with o-checker.

まず，文書ファイルのファイルパスを入力として受け取り，そのファイルヘッダ部分を読み込む．ヘッダの情報から文書ファイルの種類を特定し，その文書ファイルの種類に合った逸脱構造の検査を行う．最後に，入力した文書ファイルに逸脱構造があるか否かの出力を行う．

それぞれの逸脱構造の検知手法の詳細については既に述べたとおりである．AS1 から AS6 までの検知手法は，文書ファイルの暗号化の状態に依存せずに検査を実施することができる．しかしながら，AS7 および AS8 の検知手法は PDF の文書構造やコンテンツの検査を行うため，PDF ファイルの暗号化の状態により検査を実施することができない．

o-checker による PDF ファイルの検査の流れを図 6.2 に示す．

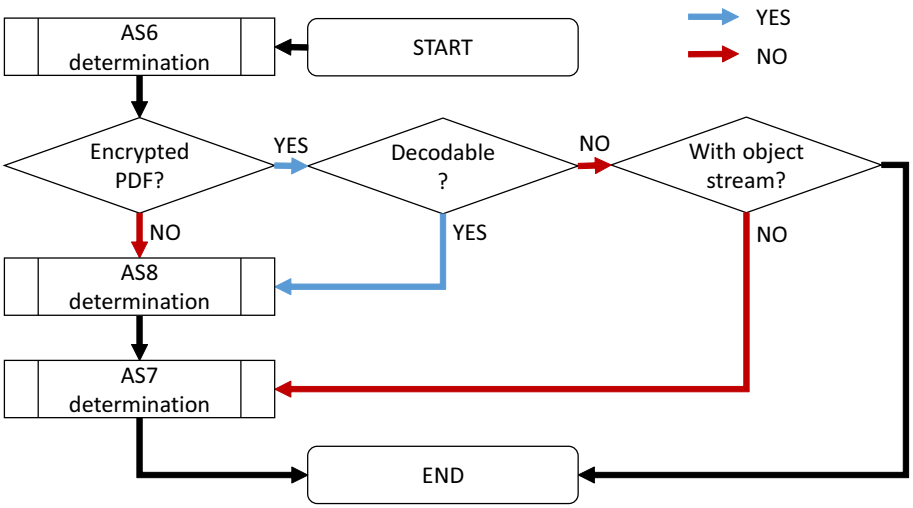

図 6.2 Flow of checking a PDF file with o-checker.

まず，入力ファイルが AS6 の特徴を持つか検査する．次に，trailer 辞書の情報を元に，PDF ファイルが暗号化されているか否か調べる．もし，PDF ファイルが暗号化されていない場合，AS7 および AS8 の特徴を持つか否か検査を行う．

もし，入力ファイルが暗号化されていた場合，o-checker は，ユーザパスワードを空の文字列に設定して PDF ファイルの復号を試みる．o-checker は，40-bit RC4，128-bit RC4，128-bit AES および 256-bit AES の 4 種類の暗号化方式の復号に対応している．PDF ファイルの復号に成功した場合，o-checker は AS7 および AS8 の特徴を持つか否か検査を行う．

PDF ファイルの復号に失敗した場合，o-checker は入力ファイルの中にオブジェクト stream があるか確認する．オブジェクト stream が入力ファイルから見つからなかった場合．o-checker は AS7 の特徴を持つか否か検査を行う．オブジェクト stream が入力ファイルから見つかった場合，PDF ファイルの検査は終了する．

6.2 使用方法

ここでは，o-checker の使用方法を簡単に示す．

6.2.1 インストール方法

o-checker は以下のサイトで公開している．

https://www.blackhat.com/us-16/briefings.html

o-checker の動作に必要な要件は以下のとおりである．

- Python 2.7.3 以降

- Python が動作する OS
- PyCrypto package for Python（暗号化された PDF ファイル用）

6.2.2 o-checker の構成

o-checker の構成を図 6.3 に示す．o-checker は 10 個のファイルから構成されており，ファイルサイズの総量は 93 KB でプログラムステップ数は約 2,600 行である．

	size	lines
o-checker.py	2KB	78
RTF/CFB		
├──msanalysis.py	4KB	153
│　└──docfileparser.py	7KB	241
PDF		
└──pdfanalysis.py	6KB	149
├──PDFFile.py	30KB	698
├──PDFCrypt.py	10KB	266
├──PDFObj.py	16KB	502
├──Stream.py	5KB	152
├──PDFFilter.py	9KB	223
└──JPEGCheck.py	4KB	126
Total	93KB	2,588

図 6.3　Components of o-checker.

各ファイルの役割について以下に示す．

(1)　o-checker.py

これは，o-checker のフロントエンドプログラムである．入力ファイルのヘッダからファイルの種類を特定し，RTF ファイルまたは CFB ファイルであれば `msanalysis.py` を，PDF ファイルであれば `pdfanalysis.py` を呼び出す．それぞれのプログラムで出力される解析ログは表示せず，`Malicious!` または `None!` という解析結果だけを表示する．

(2)　msanalysis.py

これは，`o-checcker.py` から呼び出される RTF ファイルおよび CFB ファイル用の解析ツールである．このプログラムに RTF ファイルおよび CFB ファイルを

入力することで，文書ファイルの構造を示す解析ログを出力することができる．

(3) docfileperser.py

これは，msanalysis.py から呼び出される，CFB ファイルの構文解析用プログラムである．

(4) pdfanalysis.py

これは，o-checcker.py から呼び出される PDF ファイル用の解析ツールである．このプログラムに PDF ファイルを入力することで，PDF ファイルの構造を示す解析ログを出力することができる．さらに，オブジェクト番号を指定することで，PDF 内の指定したオブジェクトの中身を平文で出力することもできる．

(5) PDFFile.py

これは，pdfanalysis.py が PDF ファイルを構文解析する際に呼び出すプログラムである．

(6) PDFCrypt.py

これは，pdfanalysis.py が暗号化 PDF ファイルを復号する際に呼び出すプログラムである．

(7) PDFObj.py

これは，pdfanalysis.py が PDF の各オブジェクトをクラス化して取り扱うために呼び出すプログラムである．

(8) Stream.py

これは，pdfanalysis.py が PDF の stream を取り扱うために呼び出すプログラムである．

(9) PDFFilter.py

これは，pdfanalysis.py が PDF の stream のデコード処理をする際に呼び出すプログラムである．

(10) JPEGCheck.py

これは，JPEG のデータに余分なデータが追加されていないか検証するプログラムであり，pdfanalysis.py が DCTDecode フィルタ（JPEG）を必要とする stream を取り扱う際に呼び出すプログラムである．

6.2.3 o-checker の基本的な使用方法

o-checker の基本的な使用方法は，o-checker.py に検査したいファイルをパラメータとして入力するだけである．o-checker の入出力の例を図 6.4 に示す．この場合，悪性 Microsoft doc ファイルに対し o-checker.py を実行し，その出力は Malicious!となっている．文書ファイルに検査対象の異常構造が見つからなかった場合，その出力は None!となる．

```
> python o-checker.py malware.doc
Malicious!
```

図 6.4 Example of o-checker input and output.

6.2.4 o-checker を使用した解析

o-checker は検知ツールとしてだけでなく，CFB ファイルや PDF ファイルの解析用のツールとしても使うことができる．

(1) CFB ファイルの解析

o-checker の中の msanalysis.py というプログラムを使用して，CFB ファイルを解析することができる．msanalysis.py は CFB ファイルを検査し，その解析ログを出力するとともに，Malicious! または None! という判定結果を出力する．

悪性 Microsoft doc ファイルに対し，判定オプション（-j）をつけて msanalysis.py を実行した例を図 6.5 に示す．判定オプション（-j）は，解析ログの結果を元に，入力ファイルが悪性文書ファイルか否かの判定結果を Malicious! または None! で出力するオプションである．この場合，入力ファイルを悪性文書ファイルと判定し，Malicious! と出力している．

```
> python msanalysis.py -j malware.doc
Compound File
1536
This is DocFile
Size of a sector: 512
Size of a short-sector: 64
Total number of sectors: 1
SecID of first sector of the dictionary stream 17
Minimum size of standard stream 4096
SecID of first sector of ssat 19
Total number of short-sectors: 1
0 Root Entry 20 stream size: 8064 composed size: 8192
1 U:Data 8 stream size: 4096 composed size: 4096
2 U:WordDocument 0 stream size: 4096 composed size: 4096
                                    ⋮
18 Empty -2 stream size: 0 composed size: 0
19 U:CompObj 124 stream size: 121 composed size: 128
suspicious file size!
00008800-000089FF:unused
00008A00-00008BFF:unused
                                    ⋮
0000FE00-0000FFFF:unused
00010000-000101FF:unused
suspicious unused sector!
file size: 140218
file size error!
header size: 1536
total composed size: 28672
Dictionary Stream size: 2560
unused sector 31232
unknown data: 107450
Null block size: 15360

Suspicious 2
Malicious!
run time: 0.0584909915924 sec
```

図 6.5 Example of msanalysis.py input and output.

出力結果の意味は以下のとおりである.

- この doc ファイルにはディレクトリエントリが 20 個含まれている（Nos. 0–19）.
- ファイルオフセット 0x8800〜0x101FF は，FAT で管理不可能領域（AS3）.
- `Suspicious unused sector` は，ファイルの末端が free secotr であることを意味する（AS4）.

- ファイルサイズは140,218 byte．$140218 \bmod 512 = 442 (\neq 0)$．`File size error!` は，ファイルサイズが異常であることを意味する（AS2）．
- 107,450 byte のデータが使途不明（AS5）．

以上のことから，この doc ファイルには AS2，AS3，AS4 および AS5 の逸脱構造があり，実行ファイルがファイルオフセット 0x8800 に蔵置されている可能性があるということが分かる．

(2) PDF ファイルの解析

o-checker の中の `pdfanalysis.py` を使用して，PDF ファイルを解析することができる．`pdfanalysis.py` は PDF ファイルを検査し，その解析ログを出力するとともに，`Malicious!` または `None!` という判定結果を出力する．

悪性 PDF ファイルに対し，判定オプション（`-j`）をつけて `pdfanalysis.py` を実行した例を図 6.6 に示す．判定オプション（`-j`）は，解析ログの結果を元に，入力ファイルが悪性文書ファイルか否かの判定結果を `Malicious!` または `None!` で出力するオプションである．この場合，入力ファイルを悪性文書ファイルと判定し，`Malicious!` と出力している．

```
> python pdfanalysis.py -j malware.pdf
00000000-00000008:comment,
00000009-000006E2:obj 1 0 xref from [(8 0 R)]
000006E3-00000721:obj 2 0 xref from [(3 0 R), (8 0 R)]
00000722-0000075E:obj 3 0 xref from [(2 0 R), (4 0 R)]
0000075F-0000083F:obj 4 0 xref from [(3 0 R)]
00000840-000008D9:obj 5 0 xref from [(4 0 R), (6 0 R)]
000008DA-0000090E:obj 6 0 xref from [(5 0 R), (7 0 R)]
0000090F-0000098B:obj 7 0 xref from [(-1 -1 R)]
0000098C-000009DA:obj 8 0 xref from [(7 0 R)]
000009DB-00010E04:obj 17 0 xref from None Suspicious
00010E05-00010E09:xref
00010E0A-00010E28:trailer
00010E29-00010E38:startxref 000039AD
00010E39-00010E3D:EOF,

obj 1 0 xml form
obj 17 0 zlib decompress error
Malicious!
run time: 0.133231163025 sec
```

図 6.6 Example of pdfanalysis.py input and output.

出力結果の意味は以下のとおりである．

- この PDF ファイルには間接参照オブジェクトが 9 個含まれている（Nos. 1–8

および 17）.

- 各間接参照オブジェクトを参照しているオブジェクトが列挙されている（例, No. 8 は No. 1 を参照している）.
- No. 17 は参照されないオブジェクト（AS7）.
- No. 1 には XML 形式のメタデータが含まれている（判定には使用しない参考情報）
- No. 17 の stream は FlateDecode フィルターによるデコード処理が必要であるが，デコード処理が失敗する（AS8-1）.

以上のことから，この PDF ファイルには，AS7 および AS8-1 の逸脱構造があり，ファイルオフセット 0x9DB に蔵置されている No. 17 の中に実行ファイルが蔵置されている可能性があることが分かる.

暗号化 PDF ファイルの解析

o-checker は，40-bit RC4，128-bit RC4，128-bit AES および 256-bit AES の 4 種類の暗号化方式に対応している．もし，入力ファイルが暗号化されていた場合，o-checker は，ユーザパスワードを空の文字列に設定して PDF ファイルの復号を試みる．また，`-p` オプションを使用することで，空の文字列以外をユーザパスワードに設定して PDF ファイルを復号することができる.

指定した間接参照オブジェクトまたは stream の取り出し

`-o` オプションまたは`-s` オプションを使用することで，指定した間接参照オブジェクトを取り出すことができる.

- `-o` オプションは，指定したオブジェクトの stream をデコードし，stream に付随する辞書オブジェクトを含めて JSON フォーマットで出力する.
- `-s` オプションは，指定したオブジェクトの stream をデコードし，その結果を出力する.

指定した stream を取り出すコマンドとその出力結果を図 6.7 に示す.

```
> python pdfanalysis.py -j malware.pdf -s 1
<?xml version="1.0" encoding="UTF-8" ?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
<config xmlns="http://www.xfa.org/schema/xci/1.0/">
<present>
<pdf>
<version>1.65</version>
<interactive>1</interactive>
<linearized>1</linearized>
</pdf>
                                        ⋮
<ImageField1 xfa:contentType="image/tif" href="">SUkqADggAACQ
kJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQ
kJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQ
                                        ⋮
FQAH/5CQkE0VAAcipwAHuxUAB////5BNFQAHMdcABy8RAAc=</ImageField1>
</topmostSubform>
</xfa:data>
</xfa:datasets>
```

図 6.7 Example of pdfanalysis.py input and output.

この例では，-s オプションを使用し，No. 1 の stream の中身を取り出している．No. 1 の中には XML データが含まれており，その中の tiff 画像の中に exploit コードが含まれていることが分かる．

第 7 章

実験

ここでは，o-checker の効果を評価するため，様々なデータセットを用いて実験を行う．

7.1 実験 1：dropper に対する o-checker の検知率（TPR）

第 3 章で述べたとおり，2009 年から 2012 年までの標的型攻撃に使用される悪性文書ファイルの多くが，自身に埋め込まれた実行ファイルを取り出して実行する dropper であった．そこで，標的型攻撃に対する o-checker の効果を評価するため，ここでは dropper 型悪性文書ファイルに対する o-checker の検知率（TPR）に関する実験を行う．

7.1.1 実験で使用する検体

実験の対象となる検体は，複数の組織において採取した文書ファイルで，分析により実行ファイルが埋め込まれている dropper であることをあらかじめ確認しているものである．特定の脆弱性，検知名，RAT の種類等について，同一のものが多数含まれるといった検体の偏りが生じるのを防ぐため，検体の採取期間は 2013 年 1 月から 2014 年 12 月までとし，その間に標的型攻撃に用いられたメールとして提供を受けたもの全てから添付ファイルを取り出し，特定の拡張子のものを機械的に選定した．そのうえで，同一のハッシュ値を持つものは取り除いた．また，拡張子は doc であるものの実際の中身が RTF であるものなど拡張子を偽装しているものが複数あったため，ファイルの種類は，拡張子ではなくファイルヘッダを元に分類した．これにより得られた検体を dro(13–14) とする．dro(13–14) の概要を表 7.1 に示す．

表 7.1 Summary of dro(13–14).

Type	Ext.	Num.	Avg. Size (KB)	Camouflaged
RTF	rtf	42	557.0	40
CFB	doc	35	193.8	2
	xls	15	277.3	0
	jtd	25	437.0	0
PDF	pdf	7	753.0	0
Total	-	124	407.5	42

dro(13–14) で悪用する脆弱性を表 7.2 に示す．最も悪用された脆弱性は MS12-027 であった．dro(13–14) の検体のうち 23 個の検体が使用する脆弱性は，メーカーから脆弱性を修正するプログラムが公開される前に攻撃が行われるゼロデイ攻撃であった．複数の脆弱性を攻撃する検体があったため，「Rates」欄（表 7.2 の右側）の合計は 100 %を超える．

表 7.2 Vulnerability used by dro(13–14).

Vulnerability	Num.	Rate
MS10-087[30]	2 / 124	1.6 %
MS12-027[31]	71 / 124	57.3 %
MS14-017[60]	3 / 124	2.4 %
JS13003[61]	15 / 124	12.1 %
JS14003[62]	19 / 124	15.3 %
APSB10-21[41]	4 / 124	3.2 %
APSB11-08[42]	4 / 124	3.2 %
APSB11-30[43]	4 / 124	3.2 %
APSB12-22[35]	1 / 124	0.8 %
APSB13-07[63]	3 / 124	2.4 %
Unknown	6 / 124	4.8 %

7.1.2 実験方法

実験で使用する環境は表 7.3 に示すとおりであり, 実験はすべて仮想マシン上で行った．表 7.3 の上方はホストマシンの環境を示しており，Core i5-3450 3.1 GHz の CPU と OS は Windows 7 SP1 で構成される．このホストマシン上の VMware Workstation 9 で仮想マシンを動作させる．表 7.3 の下方は仮想マシンの環境を示しており, デュアルコアの CPU と OS は Windows XP SP3 で構成される．Python のバージョンは 2.7.6 を使用した.

表 7.3 Experimental environment.

CPU	Core i5-3450 3.1 GHz
Memory	8.0 GB
OS	Windows 7 SP1
Virtualization software	VMware Workstation 9
Memory (VM)	512 MB
OS (VM)	Windows XP SP3
Interpreter (VM)	Python 2.7.6

この実験環境において，dro(13–14) を o-checker に入力し，検知の成功率および平均実行時間を求める．加えて，o-checker と大手ベンダのウイルス対策ソフトの性能を比較する．ほとんどのウイルス対策ソフトは，パターンファイルを更新することにより，既知のマルウェアを検知することができる．一方，標的型攻撃に使用されるマルウェアの多くは未知のマルウェアであり，標的型攻撃に対する性能を評価するためには，未知のマルウェアに対する検知率を求める必要がある．そこで，我々はウイルス対策ソフトを毎日最新の状態にアップデートし，検体を入手後速やかにウイルス対策ソフトで検知できるか実験を行った．

7.1.3 結果

dro(13–14) に対する o-checker の検知率（TPR）を表 7.4 に示す．この表の「Detection」欄の右側は，o-checker に入力した文書ファイルの数を，左側は o-checker で「Malicious!」と判定した文書ファイルの数を示している．o-checker は，検知率は全体で 98.4%であり，平均実行時間は 0.263 秒であった．

表 7.4 TPR of o-checker against dro(13–14).

Type	Ext.	Detection	TPR	Avg. Time
RTF	rtf	41 / 42	97.6 %	0.223 s
CFB	doc	34 / 35	97.1 %	0.170 s
	xls	15 / 15	100.0 %	0.184 s
	jtd	25 / 25	100.0 %	0.194 s
PDF	pdf	7 / 7	100.0%	1.382 s
Total	-	122 / 124	98.4 %	0.263 s

次に，o-checker の検知率と大手ベンダのウイルス対策ソフトの検知率との比較結果を表 7.5 に示す．この表の，「Detection」欄は，Table 7.4 と同様である．実験に用いた検体については，採取した当時の最新のパターンファイルを適用した大手ベンダのウイルス対策ソフトでも，半分も検知することができなかった．各ウイルス対策ソフトで検知できるマルウェアに特色があるので，3 種類のウイルス対策ソフトを組み合わせて，どれか 1 つでも検知した場合をとっても，検知率は 50.0%であった．

表 7.5 Comparing TPR with anti-virus software.

	detection	TPR
o-checker	122 / 124	98.4 %
T's AV	44 / 124	35.5 %
S's AV	28 / 124	22.6 %
M's AV	22 / 124	17.7 %
T&S&M's AV	62 / 124	50.0 %

7.1.4 検知に失敗した原因

o-checker が実験 1 で検知に失敗した検体 2 個を分析した結果，検知失敗の原因は o-checker の検査項目に合致しないように実行ファイルが文書ファイルに埋め込まれていたためであった．その埋め込み方式を以下に示す．

(1) shellcode の中に埋め込まれている

shellcode は閲覧ソフトが通常読み込む部分に埋め込まれることが多い．このため，shellcode が埋め込まれた部分には本論文で論じたような特徴が現れないことが多い．shellcode はサイズが制限されることが多いが，利用する脆弱性によっては実行ファイルを埋め込めるほど大きなサイズの shellcode も存在する．その場合，o-checker では検知することはできない．しかしながら，この方式は，使用できる脆弱性が限定されるため，悪性文書ファイル全体に占める割合は少なくなっている．

(2) 正規の Stream に偽装されている

実行ファイルが正規の Stream に偽装して CFB ファイルに埋め込まれた場合，CFB ファイルには本論文で論じた異常構造は現れない，このため，我々の提案手法では検知することはできない．ただし，偽装された Stream を閲覧ソフトが処理すると，閲覧ソフトが誤動作したり，表示される内容がいわゆる文字化け状態になったりしてしまう．そのため，偽装された Stream は閲覧ソフトに処理されないように，他のどの Stream からも参照されないことが多いと考えられる．したがって，o-checker の悪性 PDF ファイルの検知で行っているオブジェクト間の参照を検査する手法を悪性 CFB ファイルの検知に応用させれば，この方式の CFB ファイルは検知可能と思われる．

また，文書ファイルの構造を構文解釈して中身を書き換えるオープンソースのツールが，PDF の場合は複数普及しているが，CFB の場合はほとんど普及していない．CFB ファイルにこの方式で実行ファイルを埋め込むためには，攻撃者は CFB のファイル構造を習熟する必要がある．そのため，この方式の悪性文書ファイル全体に占める割合は少なくなっている．

7.2 実験 2：無害な文書ファイルに対する o-checker の誤検知率（FPR）

悪性文書ファイルに見られたファイルフォーマットからの逸脱について，それが悪性文書ファイル特有の特徴であるか明らかにするため，ここでは無害な文書ファイルに対する o-checker の誤検知率（FPR）に関する実験を行う．

7.2.1 実験で使用する検体

実験の対象となる文書ファイルを ben(con) とし，その概要を表 7.6 に示す．この検体は，マルウェアダンプサイト contagio でマルウェアでない（clean）とされ，研究用に公開された検体 [29] である．この検体は様々なオープンソースから収集されたとのことであるが，具体的な収集方法については不明である．いくつかの検体には，ファイルの先頭や末端に何らかの理由で HTML 形式のデータが追加されていた．文書ファイルのヘッダには，その文書ファイルの種類を示す情報（シグネチャ）が記録されている．ファイルの先頭にデータが追加された文書ファイルは，シグネチャを正しく読み込むことができないため，このファイルを

文書ファイルと認識できない閲覧ソフトが多い．このため我々はファイルの拡張子とシグネチャが一致しないファイルは取り除いた．

表 7.6 Summary of ben(con).

Type	Ext.	Num.	Avg. Size (KB)
RTF	rtf	199	516.2
CFB	doc	1,195	106.1
	xls	298	191.7
PDF	pdf	9,109	101.7
Total	-	10,801	112.3

7.2.2 実験方法

実験で使用する環境は表 7.7 に示すとおりであり，実験はすべて仮想マシン上で行った．表 7.7 の上方はホストマシンの環境を示しており，Core i5-3450 3.1 GHz の CPU と OS は Windows 7 SP1 で構成される．このホストマシン上の VMware Workstation 9 で仮想マシンを動作させる．表 7.7 の下方は仮想マシンの環境を示しており，デュアルコアの CPU と OS は Windows XP SP3 で構成される．Python のバージョンは 2.7.6 を使用した．

表 7.7 Experimental environment.

CPU	Core i5-3450 3.1 GHz
Memory	8.0 GB
OS	Windows 7 SP1
Virtualization software	VMware Workstation 9
Memory (VM)	512 MB
OS (VM)	Windows XP SP3
Interpreter (VM)	Python 2.7.6

この実験環境において，ben(con) を o-checker に入力する．o-checker で異常構造を検知した場合，この検知を誤検知とした．

7.2.3 結果

ben(con) に対する o-checker の誤検知率 (FPR) を表 7.8 に示す．この表の「Detection」欄の右側は，o-checker に入力した文書ファイルの数を，左側は o-checker で「Malicious!」と判定した文書ファイルの数を示している．o-checker の誤検知率は全体で 0.3 %であったが，xls ファイルに対する誤検知率は 4.7 %と比較的高い値であった．

表 7.8 FPR of o-checker against ben(con).

Type	Ext.	Detection	FPR
RTF	rtf	0 / 199	0.0 %
CFB	doc	2 / 1,195	0.2 %
	xls	14 / 298	4.7 %
PDF	pdf	19 / 9,109	0.2 %
Total	-	35 / 10,801	0.3 %

7.2.4 誤検知した原因

(1) CFB

o-checker が誤検知した CFB ファイルを分析した結果，誤検知の原因は以下の 2 点に集約された．

- ファイルの末尾に不要な html が付加されている．
- ファイルが途中で切れている．

まず最初に誤検知の原因としてあげられるのは，ファイルの末尾に不要な html が付加されている場合である．今回検知したデータは，すべて 4KByte 弱の同一の html データであった．これは，実験に使用した検体に埋め込まれていた実行ファイルの大きさの 10 分の 1 未満であることから，データのサイズでフィルタリングすることで当該誤検知を回避することは可能と考えられる．しかしながら，フィルタリングした大きさより小さなオブジェクトを用いた悪性 CFB ファイルがあった場合は，検知することができなくなってしまう．一方，ファイルの末尾に不要な html が付加されている CFB ファイルは，一般的な文書作成ソフトが作成することはないため，異常な CFB ファイルとして検知するという運用も考えられる．

次の原因としては，ファイルが途中で切れている場合である．ファイルが途中で切れているため，

- ヘッダを除いたファイルサイズがsectorサイズ単位になっていない（AS2）.
- ファイルサイズから求めたsector数とヘッダ情報等から計算したsector数が一致しない（AS3）.

などの特徴により，悪性CFBファイルの逸脱構造として誤検知していた．しかしながら，ファイルが途中で切れているCFBファイルは，閲覧ソフトで正しく内容を表示することができないため，一般的なCFBファイルとして使用されることはほぼないと考えて良いだろう.

したがって，今回誤検知した検体は，いずれも通常使用しないファイルであった．このようなファイルが実験で用いた検体に混在した理由の１つとして，ダウンロードの失敗が考えられる．しかしながら，今回の実験ではcontagioで配布されているzipファイルを展開した際に作成される文書ファイルを使用しており，zipにはCRCを用いたデータ破損を検査する仕組みがあるため，我々がダウンロードに失敗したとは考えづらい．つまり，contagioで配布されているzipファイルの中に通常使用しないファイルが元々格納されていたと考えられるが，その原因についてはcontagioでどのようにファイルを収集しているか明らかでないため推察することはできない.

(2) PDF

o-checkerが誤検知したPDFファイルを分析した結果，誤検知の原因は以下の3点であった.

- 間接参照されない通常のオブジェクト
- ファイルの一部が破損しているもの
- ファイルの途中に不要なデータが付加されているもの

まず最初に誤検知の原因としてあげられるのは，間接参照されない通常のオブジェクトである．今回誤検知した検体19個のうち12個はAS7のみに合致していた．どのオブジェクトからも間接参照でリンクされていないオブジェクトを調べたところ，ページの背景等を設定する管理情報に類似するデータであった．同種の管理情報でも間接参照でリンクされているものとされていないものがあり，間接参照されない原因については特定できなかった．当該オブジェクトのサイズは4KByte未満であり，悪性PDFファイルに埋め込まれていた実行ファイルの大きさの10分の1未満であることから，ある一定の大きさ未満のオブジェクトは検知の対象から外すことで当該誤検知を回避することは可能と考えられる．しかしながら，フィルタリングした大きさより小さなサイズのオブジェクトを用いた悪性PDFファイルがあった場合は，検知することができなくなってしまう.

次の原因としてあげられるのは，ファイルの一部が破損しているものである．誤検知した検体はファイルの末端に不要なデータが付加されているもの，ファイルが途中で切れているものおよびFlate圧縮されているデータが壊れているものであった．このようなPDFファイルは，通信回線の状況によっては発生しうるものであり，本論文の提案手法では誤検知してしまう．しかしながら，ファイルの末尾に不要なデータが付加されているPDFファイルは，PDFの仕様に準拠したPDF生成ソフトであれば作成することはないため，異常なPDFファイルとして検知するという運用も考えられる．また，ファイルが途中で切れているものやFlate圧縮されているデータが壊れているものは，閲覧ソフトで正しく内容を表示することができないため，一般的なPDFファイルとして使用されることはほぼないと考えて良いだろう．

最後の原因としてあげられるのは，ファイルの途中に不要なデータが付加されているものである．誤検知した検体は，どのオブジェクトにも対応しない“`endstream endobj`”という文字列が挿入されているものであった．これは，明らかにPDFの仕様に沿わない記述である．その他の構造に異常は見られなかったため，この構造はPDFファイルを生成したソフトの仕様（バグ）によるものであると考えられる．閲覧ソフトのエラー処理機能により問題が顕在化していないが，すべてのPDF生成ソフトがPDFの仕様に完璧に準拠しているわけではない．したがって，o-checkerは，一部のPDF生成ソフトで作成したPDFファイルを常に誤検知する可能性がある．

7.3 実験3：悪性文書ファイルに対するo-checkerの検知率（TPR）

標的型メール攻撃に使用される悪性文書ファイルの多くが，自身に埋め込まれた実行ファイルを取り出して実行するdropperであった．しかしながら，悪性文書ファイルの中にはdropper以外にも，外部のサーバに接続し実行ファイルをダウンロードして実行するdownloaderもある．悪性文書ファイル全般に対するo-checkerの効果を評価するため，悪性文書ファイルに対するo-checkerの検知率（TPR）に関する実験をdropperに限定せずに行う．

7.3.1 実験で使用する検体

実験の対象となる検体をmal(VT)とし，その概要を表7.9に示す．

表 7.9 Summary of mal(VT).

Type	Ext.	Num.	Avg. Size (KB)
RTF	rtf	69 (18)	487.7
CFB	doc	61 (22)	259.0
	xls	9 (2)	298.4
	ppt	2 (1)	480.5
PDF	pdf	86 (8)	653.5
Total	-	227 (51)	481.5

これらの検体は，VirusTotal[47] から収集を行った．検索の条件を以下に示す．

- 2013 年または 2014 年に登録されていること
- CVE 番号が登録されている（脆弱性を利用している）こと
- ファイルの拡張子が rtf，doc，xls，ppt または pdf であること

この条件に合致した文書ファイルを，実験 1 と同様にヘッダの情報を元に分類を行った．実験 1 で使用した dro(13–14) とは異なり，mal(VT) のうちいくつかの検体は実行ファイルを含まない downloader である．

7.3.2 実験方法

実験で使用する環境は表 7.10 に示すとおりであり，実験はすべて仮想マシン上で行った．表 7.10 の上方はホストマシンの環境を示しており，Core i5-3450 3.1 GHz の CPU と OS は Windows 7 SP1 で構成される．このホストマシン上の VMware Workstation 9 で仮想マシンを動作させる．表 7.10 の下方は仮想マシンの環境を示しており，デュアルコアの CPU と OS は Windows XP SP3 で構成される．Python のバージョンは 2.7.6 を使用した．

表 7.10 Experimental environment.

CPU	Core i5-3450 3.1 GHz
Memory	8.0 GB
OS	Windows 7 SP1
Virtualization software	VMware Workstation 9
Memory (VM)	512 MB
OS (VM)	Windows XP SP3
Interpreter (VM)	Python 2.7.6

この実験環境において, mal(VT) を o-checker に入力し, 検知の成功率を求めた.

われわれが mal(VT) の検体を入手したのは, VirusTotal にこれらの検体が登録されてから数ヶ月以上経過した後であった. そのため, 毎日最新の状態に更新しているウイルス対策ソフトを使用すると, ほとんどの検体は検知されてしまい, 未知のマルウェアに対する性能の評価を行うことができない.

そこで, われわれは o-checker の検知率と OfficeMalScanner[9] (OMS) の検知率との比較を行う. OMS は主に shellcode のパターンを検査するツールであり, RTF ファイルおよび CFB ファイルの両方に対し使用することができる. 実験に使用した OMS のバージョンは v0.58 である. CFB ファイルの場合, `OfficeMalScanner.exe` に `SCAN` オプションと `BRUTE` オプションを使用して実行した. RTF ファイルの場合は, OMS に付属する `RTFScan.exe` に `SCAN` オプションを使用して実行した. `SCAN` オプションは, 入力したファイルに対し, shellcode のパターン, CFB ファイルのシグネチャまたは実行ファイルを検索する. `BRUTE` オプションは, 1 byte 鍵で XOR されて暗号化された CFB ファイルや実行ファイルを総当りで検索するオプションである.

また, o-checker と OMS を組み合わせた場合の検知率も実験を行う. この場合の検知の流れを図 7.1 に示す. 要するに o-checker または OMS のいずれかで検知した場合に, 入力したファイルを悪性と判断する. OMS は PDF に対応していないことから, mal(VT) から PDF ファイルを取り除いたものを mal(VT)–ms とし, mal(VT)–ms に対する検知率を求めた.

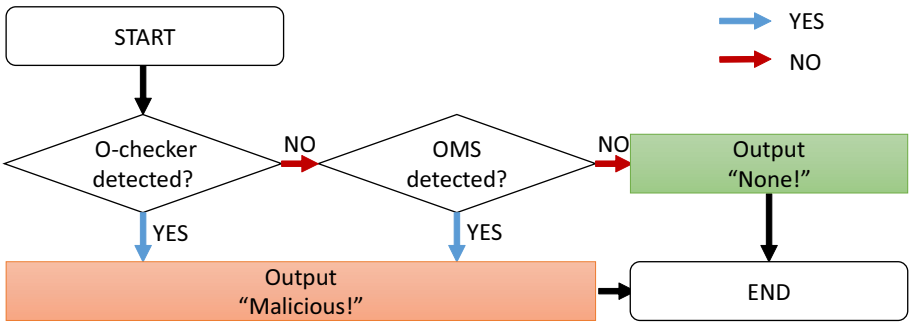

図 7.1 Flow of o-checker combined with OMS.

7.3.3 結果

mal(VT) に対する o-checker および OMS の検知率（TPR）を表 7.11 に示す．表の「OMS」欄は，`OfficeMalScanner.exe` または `RTFScan.exe` を使用した検知を示す．o-checker の検知率は 52.9 %であり，OMS の検知率は 32.6 %であった．

表 7.11 TPR of o-checker and OMS against mal(VT).

Type	Ext.	o-checker	OMS
RTF	rtf	34 / 69	12 / 69
CFB	doc	44 / 61	31 / 61
	xls	2 / 9	3 / 9
	ppt	0 / 2	0 / 2
PDF	pdf	40 / 86	-
Total	-	120 / 227	46 / 141
TPR		52.9 %	32.6 %

mal(VT)–ms に対する o-checker と OMS を組み合わせた手法の検知率（TPR）を表 7.12 に示す．表の「Combination」欄は，o-checker または OMS による検知を示す．o-checker の検知率は 56.7 %であり，OMS を組み合わせることにより検知率は 67.3 %に増加している．

表 7.12 TPR of o-checker combined with OMS against mal(VT)–ms.

Type	Ext.	o-checker	OMS	Combination
RTF	rtf	34 / 69	12 / 69	39 / 69
CFB	doc	44 / 61	31 / 61	51 / 61
	xls	2 / 9	3 / 9	5 / 9
	ppt	0 / 2	0 / 2	0 / 2
Total	-	80 / 141	46 / 141	95 / 141
TPR		56.7 %	32.6 %	67.3 %

7.4 実験 4：downloader に対する o-checker の検知率（TPR）

dropper に見られたファイルフォーマットからの逸脱について，downloader にも有効な特徴であるか明らかにするため，ここでは downloader に対する o-checker の検知率（TPR）に関する実験を行う．

7.4.1 実験で使用する検体

mal(con) は，その 98.8 %が downloader であることから，この検体を実験に用いる．その概要を表 7.13 に示す．この検体は，マルウェアダンプサイト contagio でマルウェア（malicious）とされ，研究用に公開された検体 [29] である．「Num.」中の括弧内の数値は dropper の内数を示している．

表 7.13 Summary of mal(con).

Type	Ext.	Num.	Avg. Size (KB)
PDF	pdf	11,101 (136)	26.5

7.4.2 実験方法

実験で使用する環境は表 7.14 に示すとおりであり，実験はすべて仮想マシン上で行った．表 7.14 の上方はホストマシンの環境を示しており，Core i5-3450 3.1 GHz の CPU と OS は Windows 7 SP1 で構成される．このホストマシン上の VMware Workstation 9 で仮想マシンを動作させる．表 7.14 の下方は仮想マシンの環境を示しており，デュアルコアの CPU と OS は Windows XP SP3 で構成される．Python のバージョンは 2.7.6 を使用した．

表 7.14 Experimental environment.

CPU	Core i5-3450 3.1 GHz
Memory	8.0 GB
OS	Windows 7 SP1
Virtualization software	VMware Workstation 9
Memory (VM)	512 MB
OS (VM)	Windows XP SP3
Interpreter (VM)	Python 2.7.6

この実験環境において, mal(con) を o-checker に入力し, 検知の成功率を求めた.

7.4.3 結果

mal(con) に対する o-checker の検知率（TPR）を表 7.15 に示す．o-checker の検知率は 0.9%であった.

表 7.15 TPR of o-checker against mal(con).

Type	Ext.	o-checker
PDF	pdf	96 / 11,101
TPR		0.9 %

第8章

提案方式の有効性に関する考察

この章では，提案方式の有効性を示し，研究分野への貢献内容について考察する．

8.1 仮説の検定

本論文では，4.3 においてファイルフォーマットからの逸脱構造が悪性文書ファイル特有なものであると仮定し，ファイルフォーマットからの逸脱に着目した検知方式を提案した．逸脱構造の有無について，調査および実験の結果を表 8.1 に示す．

表 8.1 各データセットと逸脱構造の有無.

dataset	検体数	逸脱構造あり	逸脱構造なし	備考
ben(con)	10,801	0.3 %	99.7 %	無害
tar(09–12)	370	97.0 %	3.0 %	97.8 %が dropper
dro(13–14)	124	98.4 %	1.6 %	100 %が dropper
mal(VT)	227	52.9 %	47.1 %	22.4 %が dropper
mal(con)	11,101	0.9 % (96/11,101)	99.1 %	1.2 %が dropper

実験 2 において，無害な文書ファイル 10,801 個を調べたところ，逸脱構造を含む文書ファイルが 0.3 %あった．しかしながら，逸脱構造を含む 35 個の文書ファイルのうち 22 個の文書ファイルは，破損等した通常使用することのない文書ファイルであり，contagio において収集する際に何らかの理由によりこのようなファイルが混在した可能性が考えられる．したがって，逸脱構造を含みかつ通常使用すると考えられるファイルは 13 個となり，全体の 1.2 %であった．以上のことから，無害な文書ファイルに逸脱構造が含まれることは，ほとんどないと言える．

一方, 悪性文書ファイルを調べたところ, tar(09–12) と dro(13–14) の文書ファイルのほとんどに逸脱構造があり, dropper のほとんどが逸脱構造を含む文書ファイルであることが明らかになった．さらに，VirusTotal に登録された悪性文書ファイルの半数が逸脱構造を含む文書ファイルであることも明らかとなった．

各データセットを併合し，逸脱構造の有無と悪性か否かで分類しなおした結果を表 8.2 に示す．

表 8.2　逸脱構造の有無と悪性文書ファイルの分類．

逸脱構造	悪性	無害
あり	95.2 % (697/732)	4.8 % (35/732)
なし	50.8 % (11,125/21,891)	49.2 % (10,766/21,891)

逸脱構造を有する文書ファイルの場合，そのほとんどは悪性文書ファイルであった．無害なものでも通常使用すると考えられる文書ファイルは 13 個であり，全体の 1.8 %であった．このことから，逸脱構造を有する文書ファイルは悪性文書ファイルであると言っても支障はないであろう．一方，逸脱構造を有しない文書ファイルの場合，その文書ファイルは悪性とも無害とも言えない．

8.2　本提案の効果

8.2.1　悪性文書ファイルに対する効果

(1)　dropper に対する効果

exploit は閲覧ソフトが通常読み込む部分に埋め込まれている．一方，実行ファイルやダミー表示用の文書ファイルを閲覧ソフトが通常読み込む部分に埋め込むと，閲覧ソフトが誤動作したり，表示される内容がいわゆる文字化け状態になってしまう．したがって，実行ファイルやダミー表示用の文書ファイルは閲覧ソフトが通常読み込まない部分に埋め込まれることが多い．その結果，dropper に仕様からの逸脱構造が含まれる．

また，実行ファイルは，多くの場合には，仕様を無視して文書ファイルの末尾に追加する形で埋め込まれる．仕様に沿って実行ファイルを埋め込む場合，インターネット上に公開されている仕様などを参照して仕様を理解するだけでなく，文書作成ソフトが生成する仕様に規定されていない文書作成ソフト固有の実装についても熟知する必要がある．加えて，一般的なファイル構造に沿おうとすると埋め込むファイルのエンコード方式や大きさに一定の制約を受ける場合がある．また，一般的なファイル構造に沿うために実行ファイルを複数に分割して埋め込むなど，実行ファイルの埋め込み方を複雑にするとデコーダ（文書ファイルに埋め込まれたファイルを取り出すコード）が複雑になり，デコードに必要なコードのサイズも大きくなってしまう．一方，shellcode のサイズは制限されることが多い．たとえば，プログラムが確保しているバッファ領域を超える大きさのデータの入力が可能という脆弱性を攻撃し，メモリ破壊を起こして当該プログラムの誤

動作を引き起こそうとした場合，攻撃手法によっては，プログラムの制御を奪うための入力データにNULLを含めざるをえない．その場合，バッファへのコピーがそこで終了してしまうため，入力するデータに含まれるshellcodeのサイズも制限されてしまう．このように，攻撃対象の脆弱性によっては攻撃手法が限定されるため，一般にshellcodeに複雑なデコーダを実装することは困難である．したがって，仕様に沿わない形で実行ファイルを文書ファイルに埋め込んだdropperがほとんどである．

2009年から2012年までの標的型攻撃に使用されたdropperを分析したところ，8つの逸脱構造を明らかにした．さらに，より後の時期に標的型攻撃で使用されたdropperで実験したところ，o-checkerは平均0.263 sという短い時間かつ98.4 %という高い確率でdropperを検知することができた．ファイル構造の仕様は攻撃者の意志で変更することが困難であることから，o-checkerは長期にわたり高い確率でdropperを検知できることが期待される．検知できた悪性文書ファイルの中にはゼロデイのもあり，最新の悪性文書ファイルが含まれていたが，文書ファイルに実行ファイルを埋め込む方式に大きな変化はなく，o-checkerで高い確率で検知することができた．

(2) downloaderに対する効果

本論文で検知対象としているAS1からAS8までの逸脱構造は，標的型メール攻撃に使われるdropperを対象に調査した結果判明した特徴であり，downloaderのことは考慮に入れていない．これらの逸脱構造は，マルウェアプログラムやおとりの文書ファイルが単純な方式で埋め込まれていることに伴って発生した仕様からの逸脱構造である．したがって，マルウェアプログラムやおとりの文書ファイルなどの余分なデータが含まれていないdownloaderには，本来，これらの逸脱構造が含まれないはずである．実際，ほぼdownloaderで構成されるmal(con)の場合，o-checkerはほぼ検知をすることができなかった．

しかしながら，実験3の結果は，downloaderの中にもこれらの逸脱構造を有しているものがあることを示している．mal(VT)の検体227個中dropperは51個であり，全体の22.5 %を占めている．AS1からAS8までの特徴がdropper特有の構造であると仮定すると，mal(VT)に対するo-checkerの検知率は，22.5 %以下でなければならない．しかしながら，mal(VT)に対するo-checkerの検知率は52.9 %であった．o-checkerがmal(VT)のdropperを100 %検知できていると仮定すると，52.9 %の検知率のうち，少なくとも，30.4 %はdownloaderの検知が寄与していることとなる．

検知できたdownloaderのいくつかを調査したところ，悪性文書ファイル中に，あえて仕様からの逸脱構造を埋め込んでいると思われるものがあった．文書ファイルの構造を正しく構文解析できないと検知ツールの検知率は低下するため，検知ツールの構文解析を誤動作させることを目的として逸脱構造を埋め込んでいる

と推察される．この種の検知回避技術は，Parser Confusion Attak と呼ばれている [64].

図 8.1 は，Parser Confusion Attack を行う文書ファイルを概念的に示したものである．図の実線より内側は閲覧ソフトが構文解析することができる文書ファイルを，破線より内側は文書ファイルの仕様を満たしている文書ファイルを，二重線の内側は検知ツールが正しく構文解析することができる文書ファイルを示している．各グループに差が出ている理由は，文書ファイルの仕様の複雑さによるものである．文書ファイルの仕様は非常に複雑であり，その全てに対応した構文解析ツールの開発には高い技術力と膨大なコストが必要となる．したがって，検知ツールに組み込まれている構文解析ツールが文書ファイルの仕様の全てに対応していることは稀である．同様に，サードパーティ製の文書編集ソフトも文書ファイルの仕様に完璧に対応することは困難であり，文書ファイルの仕様から逸脱した文書ファイルを生成してしまうことがある．閲覧ソフトは，互換性を高めるため, 仕様からの逸脱をある程度許容し, 仕様からの逸脱構造を訂正する機能を実装している．Parser Confusion Attack は，閲覧ソフトに組み込まれている構文解析ツールと検知ツールに組み込まれている構文解析ツールの構文解析能力のギャップを利用したものであり，図の実線と二重線に挟まれた範囲が Parser Confusion Attack を行える文書ファイルである．

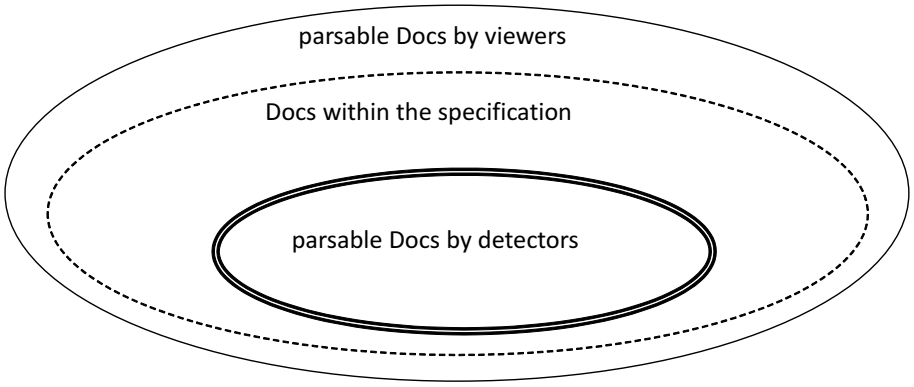

図 8.1　Conceptual diagram of Parser Confusion Attacks.

図 8.1 に示したとおり，Parser Confusion Attack を行う文書ファイルは，文書ファイルの仕様から逸脱したものが含まれる．つまり，検知から回避するため Parser Confusion Attack をしたもののうち，文書ファイルの仕様から逸脱してしまったものが，o-checker により検知されたものと考えられる．

したがって，既存の構文解析を使用する検知手法を使用する前に，本論文の手法でフィルタリングすることで，Parser Confusion Attack による検知回避の効果

を低減させることができると思われる．

(3)　悪性文書ファイル全体に対する効果

mal(VT) は，VirusTotal に 2013 年から 2014 年までに登録された悪性文書ファイルから無作為に抽出して作成したデータセットである．このため，mal(VT) は 2013 年から 2014 年の悪性文書ファイルの概ねの傾向を示しているとみなすことができる．o-checker の mal(VT) に対する検知率は，52.9 %であり，約半数は検知することができない．本提案は，ファイルフォーマットからの逸脱構造のない悪性文書ファイルは原理的に検知することができない．検知率の向上のためには，実行ファイルに着目した手法，exploit に着目した手法等の他の手法と組み合わせる必要がある．実験 3 では，主にシェルコードに着目した OMS を o-checker に組み合わせて検知率を比較した．OMS 単体では，検知率は 32.6 %と o-checker と比較すると低い値であったが，o-checker と組み合わせることで，検知率が 67.3 %に向上した．

8.2.2　標的型攻撃に対する効果

標的型メール攻撃に使用された悪性文書ファイルを調査した結果，その 97.8 %は dropper であった．標的型メール攻撃に使用される悪性文書ファイルのほとんどが dropper である理由は様々なものが考えられるが，最も大きな理由の 1 つとして，利用者に明示的に文書ファイルを開かせるという攻撃手法を攻撃者が選択したことによると思われる．標的型メール攻撃は，受信者が興味を引くような件名，本文の電子メールにマルウェアを添付する．受信者が添付ファイルを開くとマルウェアに感染する．この際，マルウェアに感染するだけで，閲覧ソフトがフリーズしたり，何も表示しなかったりすると，受信者が，意識的に文書ファイルを開いているため，受信者が不審に感じてしまう．このため，添付ファイルを開くと同時におとりの文書ファイルを表示するように攻撃者は細工をする必要がある．

一方，ドライブバイダウンロード攻撃の場合，閲覧者は意識的に文書ファイルを開くことはない．例えば，改ざんされた WEB サイトを表示した際に，ブラウザが自動的に悪性文書ファイルを開いてしまい，マルウェアに感染する．この際は，おとりの文書ファイルを表示すると逆に，意図しない文書ファイルを開いていることが明らかになり，攻撃が気づかれやすくなる．そのため，攻撃者がおとりの文書ファイルを表示させることは，まずしない．

このように，標的型メール攻撃の場合はおとりの文書ファイルを表示することがほとんどであるが，おとりの文書ファイルを悪性文書ファイルの内部に格納するか外部に格納するかという問題がある．おとりの文書ファイルを悪性文書ファイルの外部に格納した場合，おとりの文書ファイルをダウンロードできない可能性があったり，端末上に保存したファイルを開くのと比較してダウンロードに時間がかかるという問題がある．そのため，高速かつ確実におとりの文書ファイルを取り出すことができるように，悪性文書ファイルの内部におとりの文書ファイ

ルを格納する場合がほとんどである．マルウェア本体のプログラムの格納先は，悪性文書ファイルの内部・外部いずれでも構わないが，おとりの文書ファイルの格納先に引っ張られる形で，悪性文書ファイルの内部に格納されていると思われる．

このような理由から，標的型メール攻撃に使用される悪性文書は，引き続き dropper であることが考えられ，標的型メール攻撃に添付された悪性文書ファイルを o-checker が高い確率で検知できることが期待される．

一方，攻撃対象がよく利用する WEB サイトを改ざんしてマルウェアを仕込む，いわゆる「水飲み場型攻撃」の場合，使用される悪性文書ファイルの傾向としては，ドライブバイダウンロード型で downloader になると思われる，その場合，o-checker 単独での検知はほとんど期待することができないため，他の手法と組み合わせた検知手法を検討する必要がある．

8.2.3 o-checker の応用

o-checker は, パターンファイルや学習用のデータの収集をすることなく 98.4 %という高い確率で，実行ファイルが埋め込まれた悪性文書ファイルを検知することに成功した．一方，多くのウイルス対策ソフトは，日々新たに出現するマルウェアに対応するために，毎日最新のものに更新する必要がある．o-checker は，更新することなく高い検知率で悪性文書ファイルを検知することができるため，スタンドアロンで動作する端末や，ソフトウェアのアップデートを運用上の問題で頻繁にアップデートすることができないシステムに導入した場合でも，高い効果を期待できる．

さらに，検査時間の平均値はわずか 0.263s であった．一般的に，サンドボックス型の標的型メール対策ソフトの検査時間は分単位で時間がかかることを考えると，o-checker は高速に検査が完了する．o-checker の検査は，ヘッダ等の一部の情報だけで完了することもあり，場合によってはファイル全体を読み込む必要のあるハッシュ値の計算や，シグネチャの検索より高速に検査が終了する．したがって，o-checker を組織内のメールサーバ等で自動実行させれば，組織内に到達するメールを高速で検査することが可能である．

加えて，o-checker は基本的に Python が動作すれば，どのような OS でも動作することが可能である．o-checker は，静的解析を用いたツールであるため，解析環境に検知率は依存しない．このため，他の検知手法と容易に組み合わせて運用することができる．o-checker の動作は高速であることから，他の検知手法のパフォーマンスへの影響を抑えた上で，検知率の向上に寄与することができる．

例えば，組織内に到達する悪性文書ファイルは実行ファイルが埋め込まれた文書ファイルだけではないことから，悪性文書ファイル全般を想定し，実験 3 において，VirusTotal に登録された悪性文書ファイルで実験を行った．その結果，検知率は 52.9 %であった．実行ファイルが埋め込まれていない悪性文書ファイル

には，o-checker の検査対象であるファイル構造の不整合が生じないことが多い．このため，検知率の向上には，o-checker 単体の運用ではなく，実行ファイルが埋め込まれていない悪性文書ファイルを検知できる手法と組み合わせた運用について検討する必要があった．OMS の場合，検知率は o-checker と比較すると低くなっているが，o-checker で検知できなかった悪性文書ファイルを検知することができたため，o-checker と OMS を組み合わせて，いずれかで検知した場合をとると検知率の向上が見込める．

8.3 本提案の限界および課題

8.3.1 ファイルフォーマットへの依存

o-checker は，ファイルフォーマットに依存しており，RTF，CFB および PDF にのみに対応している．o-checker は OOXML（Office Open XML[46]）（docx，xlsx，pptx の拡張子）の文書ファイルは検知対象としていない．しかしながら，正規のオブジェクトとして実行ファイルを埋め込む以外の方法で実行ファイルが埋め込まれた OOXML の文書ファイルはほとんど確認できていない．その原因として 2 つの理由が考えられる．

1 つ目は，OOXML の文書ファイルに埋め込まれた不正なファイルの検知を容易にする仕組みが導入されていることが考えられる．OOXML の文書ファイルの中には複数のファイルが zip 圧縮されて格納されている．格納されている各ファイル相互の関連性を指定するリレーションシップという仕組みがあり，この仕組みを利用することで不正なファイルが埋め込まれた文書ファイルは閲覧ソフトで開くことができない．

2 つ目は，仮にリレーションシップの仕組みを突破して実行ファイルを OOXML の文書ファイルに埋め込んだとしても，サイズが制限されることが多い shellcode に zip 圧縮をデコードするコードを組み込むことは困難であることが考えられる．このことから，実行ファイルが埋め込まれた OOXML の文書ファイルを利用した標的型攻撃もほとんど確認できていない．

しかしながら，今後 OOXML の悪性文書ファイルを用いた標的型攻撃が増加した場合には，その対応についても検討する必要がある．

8.3.2 脆弱性を使わない攻撃への対策

本論文では，脆弱性を攻撃する悪性文書ファイルを対象としている．しかしながら，文書ファイルを利用し，脆弱性を攻撃することなくマルウェアに感染させる手法はいくつかある．例えば，実行ファイルを OLE オブジェクトとして文書ファイルに埋め込み，利用者に埋め込んだオブジェクトをダブルクリックさせることでマルウェアに感染させることができる．それ以外に，文書ファイル標準のマクロ機能を悪用し，利用者にマクロを有効化させることでマルウェアに感染させることもできる．これらは，利用者の明示的な操作が伴わない限り，マルウェ

アに感染することはない．しかしながら，攻撃者は言葉巧みに利用者の行動を誘導し，マルウェアに感染させようとする．このような攻撃には，利用者の権限ではマクロを有効化できなくするなど，検知とは別の手段で防御策を考える必要がある．

8.3.3 日本を対象とした標的型メール攻撃以外への有効性

o-checker は，日本国内の組織を対象とした標的型メール攻撃に使用された悪性文書ファイルを非常に高い確率で検知することができた．実験に使用したデータセットは，脆弱性の観点から見ると，日本国内で発生した標的型メール攻撃に使用された悪性文書ファイルの傾向を概ね反映させたものとみなすことができたため，o-checker は日本国内の組織を対象とした標的型メール攻撃には有効であると言えた．

しかしながら，悪性文書ファイル全般の傾向を見るものとして，VirusTotal に登録された悪性文書ファイルに対する o-checker の性能を評価したが，検知できたものは半数程度であり，o-cheker 単体では十分な性能とは評価できず，別の検知手法と検討する必要がある．

さらに，標的型メール攻撃対策に対象を絞った場合でも，標的型メール攻撃に使用される悪性文書ファイルが dropper である理由については地域性はないものの，日本国外の組織を対象とした標的型メール攻撃は，攻撃主体や悪性文書ファイルの傾向が異なる可能性が否定できない．したがって，今後，日本国外を対象とした標的型メール攻撃に使用された悪性文書ファイルを収集し，o-checker の性能を評価する必要がある．

8.4 利便性のための逸脱の許容とセキュリティ

閲覧ソフト開発ベンダーは，これまで，利便性向上のため，いかに文書ファイルを正しく表示するかという観点で開発を行ってきた．その結果，仕様からの逸脱構造を有する文書ファイルであっても，その逸脱構造を許容し，強力なエラー修正機能を実装することで文書ファイルを表示できるように閲覧ソフトを開発している．

エラー修正機能は利便性の向上に寄与する一方, 攻撃者に付け込まれる隙ともなっている．これまでに述べた, dropper へのデータの埋め込みや, Parser Confusion Attack は，このエラー修正機能を悪用したものである．

利便性の観点から，エラー修正機能を完全に除去するということは困難であると思われる．しかしながら，ほとんどの良性の文書ファイルは，文書ファイルの仕様に沿った構造となっていることから，標準の設定では仕様に沿っていない文書ファイルを表示しないようにするため，エラー修正機能を無効化するなど，セキュリティと利便性のバランスを考慮して開発することが望まれる．

第9章

結　論

本論文では，標的型攻撃における被害を抑止し，情報システムのセキュリティを向上させることを目的とし，標的型攻撃における初期段階のメールを用いた攻撃に着目した．

標的型メールに添付されるマルウェアの約 6 割は，文書ファイル形式のマルウェアである悪性文書ファイルであり，そのほとんどが，実行ファイルを自身に内包する dropper であった．標的型メール攻撃の場合，受信者がメールの件名や本文に騙されて悪性文書ファイルを閲覧することから，攻撃に気づきにくくするためには，ダミー表示用の文書ファイルが必要である．素早く確実にダミー表示用の文書ファイルを表示するためには，ダミー表示用の文書ファイルを悪性文書ファイル中に埋め込む必要があり，このことが標的型メール攻撃に使用される悪性文書ファイルのほとんどが dropper である主たる要因となっていると思われる．

本論文では，ファイルフォーマットからの逸脱に着目することにより，悪性文書ファイルの特徴として 8 つの逸脱構造を明らかにした．また，これらの逸脱構造を検知することにより悪性文書ファイルを検知する方式を提案した．さらに，提案方式を o-checker という検知・解析ツールに実装し，提案方式の有効性を検証する実験を行った．標的型メール攻撃に使用された悪性文書ファイルは，最新のパターンファイルを適用したウイルス対策ソフトでも 17.7 %から 35.5 %しか検知することができなかった．一方，o-checker は，平均 0.263 s という短い時間かつ 98.4 %という高い確率で検知することができた．ファイル構造の仕様は攻撃者の意志で変更することが困難であることから，o-checker は長期にわたり高い確率で標的型メール攻撃に使用された悪性文書ファイルを検知できることが期待される．

また，閲覧ソフト開発ベンダーは，利便性向上のため，いかに文書ファイルを表示するかという観点で，文書ファイルのフォーマットからの逸脱を許容する実装を行ってきた．しかしながら，この実装が攻撃者に付け込まれる隙となっていることを踏まえ，閲覧ソフト開発ベンダーがセキュリティと利便性のバランスを考慮した開発を行うことを期待する．

謝　　辞

本論文を執筆するにあたり，担当指導教官ならびに主査である田中英彦教授から多大な御指導と御鞭撻を賜り，論文の構成を検討するにあたり貴重な助言をいただいたことに心から御礼申し上げます．

また，副査を担当していただいた後藤厚宏教授，湯浅墾道教授，大久保隆夫教授ならびに，橋本正樹准教授に厚く御礼申し上げます．

さらに，研究を通じて活発な議論にお付き合いいただくとともに博士後期課程への進学を勧めていただいた三村守氏に深く感謝いたします．

最後に，研究の進捗にあたり貴重な助言をいただいた情報セキュリティ大学院大学の皆様，博士後期課程への進学に深い理解を示し勤務環境に配慮いただいた警察庁の皆様に深く感謝いたします．

参 考 文 献

[1] トレンドマイクロ：TrendLabs 2015 年 年間セキュリティラウンドアップ (2016).

[2] 警察庁：サイバーインテリジェンスにかかる最近の情勢（平成 23 年 4 月〜9 月）別添資料 (2011). https://www.npa.go.jp/keibi/biki7/231014shiryou.pdf.

[3] トレンドマイクロ：国内標的型サイバー攻撃分析レポート 2015 年板〜「気付けない攻撃」の高度化が進む〜(2015).

[4] Micro, T.: Targeted Attack Campaigns and Trends: 2014 Annual Report (2015). http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-targeted-attack-trends-annual-2014-report.pdf.

[5] 総務省：脆弱性（ぜいじゃくせい）とは？ ｜どんな危険があるの？ ｜基礎知識｜国民のための情報セキュリティサイト (2013). http://www.soumu.go.jp/main_sosiki/joho_tsusin/security/basic/risk/11.html.

[6] Kaspersky: The New Daily Malware Count from Kaspersky Lab Decreases by 15,000 in 2015 (2015). http://usa.kaspersky.com/about-us/press-center/press-releases/2015/new-daily-malware-count-kaspersky-lab-decreases-15000-2015.

[7] Microsoft: Rich Text Format (RTF) Specification, version 1.9.1. https://www.microsoft.com/en-us/download/details.aspx?id=10725.

[8] Microsoft: [MS-CFB]: Compound File Binary File Format. https://msdn.microsoft.com/ja-jp/library/dd942138.aspx.

[9] Boldewin, F.: Analyzing MSOffice malware with OfficeMalScanner. http://www.reconstructer.org/papers/Analyzing\%20MSOffice\%20malware\%20with\%20OfficeMalScanner.zip.

[10] Laskov, P. and Šrndić, N.: Static Detection of Malicious JavaScript-bearing PDF Documents, *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, New York, NY, USA, ACM, pp. 373–382 (2011).

[11] ISO: ISO32000–1:2008 Document management – Portable document format – Part 1 : PDF1.7. http://www.iso.org/iso/iso_catalogue/catalogue_tc/\\catalogue_detail.htm?csnumber=51502.

[12] 神薗雅紀，西田雅太，小島恵美，星澤裕二：抽象構文解析木による不正なJavaScriptの特徴点抽出手法の提案，情報処理学会論文誌，Vol. 54, No. 1, pp. 349–356 (2013).

[13] CVE Details: Microsoft Word : CVE security vulnerabilities, versions and detailed reports, https://www.cvedetails.com/product/529/Microsoft-Word.html?vendor_id=26.

[14] CVE Details: Microsoft Excel : CVE security vulnerabilities, versions and detailed reports, https://www.cvedetails.com/product/410/Microsoft-Excel.html?vendor_id=26.

[15] CVE Details: Microsoft Powerpoint : CVE security vulnerabilities, versions and detailed reports, https://www.cvedetails.com/product/623/Microsoft-Powerpoint.html?vendor_id=26.

[16] CVE Details: Adobe Acrobat Reader : CVE security vulnerabilities, versions and detailed reports, http://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor_id=53.

[17] CVE Details: Adobe Flash Player : CVE security vulnerabilities, versions and detailed reports, https://www.cvedetails.com/product/6761/Adobe-Flash-Player.html?vendor_id=53.

[18] Li, W., Stolfo, S. J., Stavrou, A., Androulaki, E. and Keromytis, A. D.: A Study of Malcode-Bearing Documents, *Detection of Intrusions and Malware, and Vulnerability Assessment, 4th International Conference, DIMVA 2007, Lucerne, Switzerland, July 12-13, 2007, Proceedings*, pp. 231–250 (2007).

[19] 岩本一樹，和﨑克己：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，情報処理学会論文誌，Vol. 56, No. 3, pp. 892–902 (2015).

[20] Tzermias, Z., Sykiotakis, G., Polychronakis, M. and Markatos, E. P.: Combining static and dynamic analysis for the detection of malicious documents, *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11, April 10, 2011, Salzburg, Austria*, p. 4 (2011).

[21] Stolfo, S. J., Wang, K. and Li, W.: Towards Stealthy Malware Detection, *Malware Detection*, pp. 231–249 (2007).

[22] 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).

[23] 三村　守，大坪雄平，田中英彦：悪性文書ファイルに埋め込まれた RAT の検知手法，情報処理学会論文誌，Vol. 55, No. 2, pp. 1089–1099 (2014).

[24] Xu, W., Wang, X., Xie, H. and Zhang, Y.: A Fast and Precise Malicious PDF Filter, *Proceedings of the 22nd Virus Bulletin International Conference*, pp. 14–19 (2012).

[25] Smutz, C. and Stavrou, A.: Malicious PDF detection using metadata and structural features, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pp. 239–248 (2012).

[26] Srndic, N. and Laskov, P.: Detection of Malicious PDF Files Based on Hierarchical Document Structure, *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013* (2013).

[27] Kwon, H., Kim, Y., Lee, S. and Lim, J.: A Tool for the Detection of Hidden Data in Microsoft Compound Document File Format, *Information Science and Security, 2008. ICISS. International Conference on*, pp. 141–146 (2008).

[28] Microsoft: マイクロソフト セキュリティ アドバイザリ 2501584：Microsoft Office 向けの Microsoft Office ファイル検証機能の公開 (2011). `https://technet.microsoft.com/library/security/2501584`.

[29] contagio: 16,800 clean and 11,960 malicious files for signature testing and research (2013). `contagiodump.blogspot.jp/2013/03/16800-clean-and-11960-malicious-files.html`.

[30] Microsoft: マイクロソフト セキュリティ情報 MS10-087 - 緊急 (2010). `https://technet.microsoft.com/ja-jp/library/security/ms10-087.aspx`.

[31] Microsoft: マイクロソフト セキュリティ情報 MS12-087 - 緊急 (2012). `https://technet.microsoft.com/ja-jp/library/security/ms12-027.aspx`.

[32] Adobe: Adobe - Security Bulletins: APSB11-07 - Security update available for Adobe Flash Player (2011). http://www.adobe.com/support/security/bulletins/apsb11-07.html.

[33] Adobe: Adobe - Security Bulletins: APSB12-03 - Security update available for Adobe Flash Player (2012). http://www.adobe.com/support/security/bulletins/apsb12-03.html.

[34] Adobe: Adobe - Security Bulletins: APSB12-18 - Security update available for Adobe Flash Player (2012). http://www.adobe.com/support/security/bulletins/apsb12-18.html.

[35] Adobe: Adobe - Security Bulletins: APSB12-22 - Security update available for Adobe Flash Player (2012). http://www.adobe.com/support/security/bulletins/apsb12-22.html.

[36] Microsoft: マイクロソフト セキュリティ情報 MS09-067 - 緊急 (2009). https://technet.microsoft.com/ja-jp/library/security/ms09-067.aspx.

[37] Microsoft: マイクロソフト セキュリティ情報 MS11-021 - 重要 (2011). https://technet.microsoft.com/ja-jp/library/security/ms11-021.aspx.

[38] Adobe: Adobe - Security Advisories : APSB09-04 - Security Updates available for Adobe Reader and Acrobat (2009). http://www.adobe.com/support/security/bulletins/apsb09-04.html.

[39] Adobe: Adobe - Security Advisories : APSB10-02 - Security Updates available for Adobe Reader and Acrobat (2010). http://www.adobe.com/support/security/bulletins/apsb10-02.html.

[40] Adobe: Adobe - Security Advisories : APSB10-02 - Security Updates available for Adobe Reader and Acrobat (2010). http://www.adobe.com/support/security/bulletins/apsb10-07.html.

[41] Adobe: Adobe - Security Advisories : APSB10-21 - Security Updates available for Adobe Reader and Acrobat (2010). http://www.adobe.com/support/security/bulletins/apsb10-21.html.

[42] Adobe: Adobe-Security Bulletins: APSB11-08 - Security update available for Adobe Reader and Acrobat (2011). http://www.adobe.com/support/security/bulletins/apsb11-08.html.

[43] Adobe: Adobe-Security Bulletins: APSB11-30 - Security update available for Adobe Reader and Acrobat (2011). http://www.adobe.com/support/security/bulletins/apsb11-30.html.

[44] 日本アイ・ビー・エム株式会社：2012 年上半期 Tokyo SOC 情報分析レポート (2012). http://www-935.ibm.com/services/jp/its/pdf/tokyo_soc_report2012_h1.pdf.

[45] 日本アイ・ビー・エム株式会社：2012 年下半期 Tokyo SOC 情報分析レポート (2013). http://www-935.ibm.com/services/jp/its/pdf/tokyo_soc_report2012_h2.pdf.

[46] ISO: ISO/IEC 29500:2012:Information technology – Document description and processing languages – Office Open XML File Formats (2012).

[47] VirusTotal: VirusTotal, https://www.virustotal.com/.

[48] Shusei Tomonaga and Yuu Nakamura: Revealing the Attack Operations Targeting Japan, https://www.jpcert.or.jp/present/2015/20151028_codeblue_apt-en.pdf (2015).

[49] BLUE Inc: CODE BLUE : International Security Conference in Tokyo where Global Security Trends, Top Notch Professionals, and participants intersect, http://codeblue.jp/2015/en/ (2015).

[50] FireEye: M-Trends: セキュリティ侵害の最新動向 2014 (2015).

[51] Microsoft: [MS-OLEDS]: Object Linking and Embedding (OLE) Data Structures, https://msdn.microsoft.com/en-us/library/dd942265.aspx.

[52] Microsoft: Component Object Model (COM), https://msdn.microsoft.com/en-us/library/ms680573.aspx.

[53] トレンドマイクロ株式会社：韓国の水力原子力発電所を狙った「マスター・ブート・レコード」を破壊する不正プログラム攻撃, http://blog.trendmicro.co.jp/archives/10641.

[54] ISO: ISO 32000-1:2008 Document management - Portable document format - Part 1：PDF1.7, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502.

[55] Adobe: PDF Reference and Adobe Extensions to the PDF Specification, http://www.adobe.com/jp/devnet/pdf/pdf_reference.html.

[56] Microsoft: Microsoft PE and COFF Specification, https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.

[57] Decalage: python-oletools - python tools to analyze OLE files, http://www.decalage.info/python/oletools.

[58] 新山祐介：GitHub - euske/pdfminer: Python PDF Parser, https://github.com/euske/pdfminer.

[59] python: https://www.python.org/.

[60] Microsoft: マイクロソフト セキュリティ情報 MS14-017 - 緊急 (2014). https://technet.microsoft.com/ja-jp/library/security/ms14-017.aspx.

[61] ジャストシステム：一太郎の脆弱性を悪用した不正なプログラムの実行危険性について (2014). https://www.justsystems.com/jp/info/js14003.html.

[62] ジャストシステム：一太郎の脆弱性を悪用した不正なプログラムの実行危険性について (2013). https://www.justsystems.com/jp/info/js13003.html.

[63] Adobe: Adobe-Security Bulletins: APSB13-07 - Security update available for Adobe Reader and Acrobat (2013). http://www.adobe.com/support/security/bulletins/apsb13-07.html.

[64] Carmony, C., Hu, X., Yin, H. and Xu, A. V. M. Z.: Extract Me If You Can: Abusing PDF Parsers in Malware Detectors, *Proceedings of the Network and Distributed System Security Symposium (NDSS) 2016* (2016).

研究発表

1．学術論文

(1) 三村 守，大坪 雄平，田中 英彦, “悪性文書ファイルに埋め込まれた RAT の検知手法,” 情報処理学会論文誌, Vol. 55, No. 2, pp.1089–1099 (2014).

(2) 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).

(3) 大坪 雄平，三村 守，田中 英彦, “ファイル構造検査の悪性 PDF ファイル検知への応用,” 情報処理学会論文誌, Vol. 55, No. 10, pp.2281–2289 (2014).

2．国際会議

(1) 大坪 雄平, “o-checker:悪性文書ファイル検知ツール〜ファイルサイズからにじみ出る悪意,” CODE BLUE 2013 (2014).

(2) Yuhei Otsubo, “Document Malware Detection Tool ‘o-checker’ introduction,” アジア大洋州地域サイバー犯罪捜査技術会議 (2015).

(3) Yuhei Otsubo, “O-checker: Detection of Malicious Documents through Deviation from File Format Specifications,” Black Hat USA 2016 (2016).

3．学会報告（口頭発表）

(1) 三村 守, 大坪 雄平, 田中 英彦, “悪性文書ファイルへの RAT の埋め込み方式の調査,” 研究報告マルチメディア通信と分散処理 (DPS) 2013-DPS-154(21), pp.1–6 (2013).

(2) 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,” 研究報告インターネットと運用技術 (IOT) 2013-IOT-22(16), pp.1–6 (2013).

(3) 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 文書ファイル検知手法の評価,” コンピュータセキュリティシンポジウム 2013(CSS2013) 論文集, pp.642–648 (2013).

(4) 大坪 雄平，三村 守，田中 英彦, “PDF の構造検査による悪性 PDF ファイルの検知,” コンピュータセキュリティシンポジウム 2013 (CSS2013) 論文集, pp.649–656 (2013).

(5) 大坪 雄平，三村 守，田中 英彦, “悪性文書ファイル検知のためのファイル構造検査の長期有効性,” コンピュータセキュリティシンポジウム 2015 (CSS2015) 論文集, pp.955–962 (2013).