信州大学審査学位論文

静的解析に基づくマルウェア分類システムに関する研究

2015年3月

岩本 一樹

目次

第 1 章

序論

マルウェアとは，計算機上において不正で有害な動作を行う「悪意のある」ソフトウェアやコードの総称である．マルウェアは計算機が提供するサービスやセキュリティ上での脅威であることが多く，感染防止や除去対策援用プログラムの開発上，マルウェアの解析が重要な課題である．そこで本論文ではマルウェア解析のために，標的型攻撃に用いられる文書型マルウェアからシェルコードを抽出する方法，圧縮・暗号化されたマルウェアを復号する方法，大量のマルウェアを分類する方法について述べる．

マルウェアが作成される動機は，2000 年代初頭までは技術を誇示することや世の中を騒がせることなどが目的であった．インターネットが一般的になる前は実行可能ファイルやディスクのブートセクタへの感染が主流であり，実行可能ファイルを実行したときや，ディスクを起動したときに感染を広めるという動作をする．インターネットが一般的になった 1990 年代後半には，文書ファイルのマクロ機能を利用したマルウェア（マクロウイルス）が現れた．マクロウイルスは文書ファイルを開いたときに PC が感染し，他の文書ファイルに感染を広める．また，同時に感染を広める手段としてインターネットを利用するマルウェアが現れた．メールで自身を拡散させることで，実行可能ファイルやディスクのブートセクタへの感染により広まるマルウェアに比べて短期間で広範囲に感染を広めるマルウェアも現れた．これらのマルウェアでは一定の条件でメッセージの表示やファイルやディスクの削除・破壊などの発病を行うマルウェアも多く見られた．

一方，2000 年代初頭以降は金銭や経済的な目的でマルウェアが作成されるようになり，マルウェアは単なる愉快犯の人騒がせなプログラムから，経済的な被害や信用の失墜を引き起こす脅威へと変化した．マルウェアの動作・機能としては無差別に感染を広めたり，目立つ発病が起こるようなマルウェアは少なくなり，マルウェアは情報の窃取や感染 PC を Bot 化することが目的となった．そのような状況の中でマルウェアに対する対策も進み，1 つのマルウェアが大規模に感染する事例は少なくなったものの，マルウェアはアンチウイルスの検出を回避して感染を広めるために大量の亜種が作られるようになった．

しかし大量に公開されるマルウェアの中で本当に新しいと言えるマルウェアは少数であり，ほとんどのマルウェアは既存のマルウェアを改変したものである．ゆえに本論文では大量に作られるマルウェアを分類する方法を提案する．マルウェアを分類することで既に解析済みのマルウェアと類似していることがわかれば，解析済みのマルウェアの情報から機能を推定することができる．自動的な機能の推定は初動のマルウェア対策として役立つと考えられる．また既存のマルウェアとの類似度は解析者による詳細な解析を行う必要があるマルウェアを絞り込み，その優先順位を決めるための情報となる．さらにマルウェアの分類できれば，過去に同種のマルウェアを解析した経験がある解析者を選定することで，効率よくマルウェアの解析を行うことができる可能性がある．

既存のマルウェアの改変に加えて，実行コードはパッカーと呼ばれる圧縮・暗号化プログラムを用いて難読化（パック）した後に配布されることがあり，同じマルウェアであっても異なるバイナリイメージが作られる．初めてパッカーでパックされたマルウェアは 1990 年代後半に公開された PrettyPark であり，このときは通常のプログラムを圧縮するために開発された既存のパッカーが用いられた．マルウェアのために作られたパッカーではないが，それでもアンチ

ウイルス製品の検知を回避するには十分であった．その後，アンチウイルス製品が既存のパッカーでパックされたプログラムを復号するアンパッカーを内包するなどの対応もあり，既存のパッカーをそのまま使うような方法ではアンチウイルス製品の検知を回避できなくなる．それにともないマルウェアの方もパッカーを発展させることになり，パッカーに対応する専用のアンパッカーを個別に作成する方法は困難になっている．

さらにマルウェアの配布においては脆弱性が利用される場合がある．単純に実行可能ファイルをメール等で送りつけるような手法も多くあるが，文書ファイルを開くアプリケーションソフトウェアの脆弱性を利用して，脆弱性を攻撃する文書ファイルを開いたときに PC をマルウェアに感染させる手法がある．ユーザはメールに添付されていたり Web 等で配布される実行可能ファイルを開くことに対しては警戒するが，文書ファイルを開くことは一般的に行われていることなので，実行可能ファイルを開くことに比べると抵抗が少ない．ゆえに，文書ファイルを開くアプリケーションの脆弱性を攻撃することはマルウェアに感染させるための手段として有効な方法の 1 つである．しかし文書型マルウェアの場合，脆弱性をもつアプリケーションソフトウェアの存在が前提であり，アプリケーションソフトウェアのバージョンや実行環境が文書型マルウェアが想定する環境に適合していなければならないという弱点もある．

文書型マルウェアでは，文書ファイル内部に脆弱性への攻撃が成功したときに実行されるプログラムコード（シェルコード）が内包されている．このシェルコードが文書ファイル内部にあるデータから実行可能ファイルを作成して実行したり，あるいはインターネットから実行可能ファイルをダウンロードして実行する．マルウェアの最終的な目的となるような動作はシェルコードにはなく，この実行可能ファイルにある．しかし実行可能ファイルを入手するためには，何らかの方法で文書型マルウェアを解析する必要がある．特に文書型マルウェアが想定する環境に適合するアプリケーションソフトウェアを準備できなければ，動的解析により実行可能ファイルを入手することはできない．

これらの問題のために，マルウェア解析および分類は困難になっている．文書型マルウェアとしてマルウェアが配布される場合には，文書ファイルに含まれる実行可能ファイルを取得する必要がある．実行可能ファイルが圧縮・暗号化されているならば，それを復号して元のコードを得る必要がある．その後，元のコードから特徴を抽出し，検体間の類似度を求めることで分類を行う．よって，本論文では文書型マルウェアからシェルコードを抽出する方法，パックされたマルウェアを展開してオリジナルのコードを抽出する方法，大量のマルウェアを分類する方法を提案する．

本論文が提案するシステム（本システム）の全体像を図 1.1 に示す．本システムは投入された検体が文書ファイルのときにはシェルコードの抽出して実行可能ファイルを取得し，以後は取得した実行可能ファイルを対象とする．本システムは実行可能ファイルがパックされているときにはアンパックを行う．最後に本システムは実行可能ファイルを分類する．

第 4 章では Microsoft Office の文書ファイル（doc，xls，ppt，rtf）から文書ファイルを開くアプリケーションソフトウェアを用いずにシェルコードを特定する方法について述べる．文書ファイルのエントロピーに基づいて，シェルコードの候補となるデータ列をエミュレータで実行することで効率よくシェルコードを特定することを試みた．特定したシェルコードを実行するための実行可能ファイルを作成することで，文書ファイルを開くアプリケーションソフトウェアで脆弱性が攻撃されてシェルコードが実行された状態を再現する．これにより，シェルコードによって作成される実行可能ファイルを得ることができる．本論文では実際の文書型マルウェアを用いて，シェルコードの特定を試みて成否を確認した．また特定したシェルコードを実行して，その動作を観測した．

第 5 章ではパックされた実行可能ファイルをアンパックする方法について述べる．独自に作成したエミュレータにより，実行可能ファイルの復号コードを実行することで元のバイナリイメージを取得する．エミュレータは完全にソフトウェアで動作するので，CPU や Operation System（OS）には依存しない．アンパックではコードの復号にあわせて，エントリーポイントの特定と Import Address Table（IAT）の再構築を行う．自身が書き換えたメモリの実行を検知および既知のエントリーポイントのコードと比較することでエントリーポイントを特定を試みる．本論文では実際のマルウェアの中でパックされた検体に対してコードの復号とエントリーポイントの特定，IAT の再構築を試みた．また既知のプログラムをパックして，本システムでアンパックができるか調査した．

第 6 章ではマルウェアを分類する方法について述べる．マルウェアの分類のために，逆アセンブルした結果を制御フ

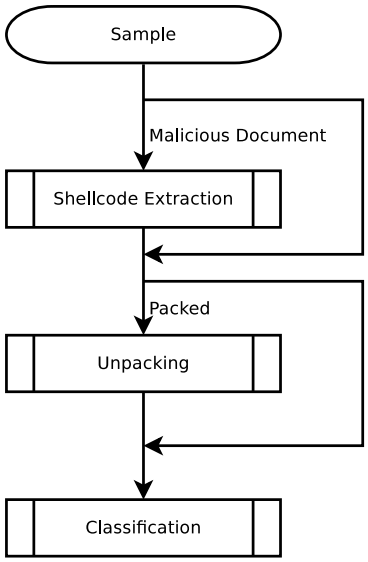

図 1.1　システムの全体像

ロー解析し，ある Application Programming Interface（API）が呼び出された後に呼び出される可能性のある API の対（API 推移）を抽出した．API 推移の推移を比較することで，マルウェア間の類似度を算出して階層型クラスタ解析を行いマルウェアを分類する．本論文では実際のマルウェアの分類することでパフォーマンス評価を行い，マルウェアのソースコードをコンパイラおよびコンパイラのオプションを変更してコンパイルしたバイナリから類似度を求めることで性能評価を行った．

その他，第 2 章では関連研究について述べ，第 7 章では結論を示す．また，実際のマルウェア「ZeuS」の亜種の分類および本論文で使用したプログラムの解説を付加する．

1.1　マルウェア解析の方法

本節では，主なマルウェア解析の方法とその問題点，およびマルウェアが行う解析を妨害するための方法について述べる．

マルウェア解析とは，マルウェアの動作・機能を明らかにすることである．アンチウイルス製品やその他のセキュリティ関連製品を作成やマルウェアに感染した PC の修復等を行うためには，マルウェアにどのような動作・機能があるのかを知る必要がある．ゆえに，マルウェア解析はマルウェア対策の基本的な部分である．マルウェア解析の方法は主に動的解析と静的解析に分けることができる．動的解析と静的解析はどちらか一方を行うというものではなく，それぞれの長所を生かし短所を補うために平行して行われる．

1.1.1　動的解析

動的解析とはマルウェアを実行させて動作（システムコールや API 呼び出しなど）を見ることで，マルウェアを解析する方法である．マルウェアを動作させるための方法には複数ある．最も確実にマルウェアを動作させる方法は，実際にマルウェアが動作する物理環境（CPU や OS など）を用いる方法である．一方，仮想環境のゲスト OS としてマルウェアを動作させる環境を構築する方法もある．仮想環境の場合，実際の環境（CPU や OS など）が必要な場合と，エミュレータにより CPU なしで動作が可能な場合もある．仮想環境を用いることでマルウェアに感染した解析環境の復旧が容易であったり，ホスト OS 側からの制御が可能などの利点がある．

動的解析では後述の 1.1.2 節に示す難読化の影響を受けず，また逆アセンブルや逆コンパイル結果を読み解く能力は必要とされない．しかし動的解析を行うためにはマルウェアが動作する条件にあう環境を準備する必要がある．また実行時に動作しなかった機能については知ることができない．

動的解析を妨害するために解析環境（仮想環境やデバッガ，解析ツールなど）を検出する方法がある．マルウェアは解析環境を検出したときには，動作を停止させたり，本来の動作とは異なるダミーの動作をする．

1.1.2 静的解析

静的解析とはマルウェアを逆アセンブル・逆コンパイルなどの方法で，マルウェアを動作させずに解析を行う方法である．JavaScript などのスクリプト言語の場合には，直接そのソースコードを読むことで動作や機能を調べる．静的解析では実際にマルウェアが動作する環境を準備する必要はない．しかしそのマルウェアに対応する逆アセンブラや逆コンパイラは必要であり，また逆アセンブルや逆コンパイル結果を読み解く能力が必要になる．マルウェアの動作を見る動的解析に比べて時間がかかる．

静的解析を困難にするための手法として難読化があり，その 1 つに検体を圧縮・暗号化する（パック）がある．パックされた検体はコンパイラが出力したコードがファイルに圧縮・暗号化された状態で格納されているので，逆アセンブルや逆コンパイルでマルウェアのコードを可読な形式に変換できない．また他にもコードそのものを改変する方法もある．元のコードの動作が変わらないように無意味なコードを挿入したり，コードを異なる命令に変換するなどの方法がある．たとえば $a+5$ という命令を $a+5-0$ や $a+2+3$ に変換することでコードを読み難くする．コードそのものの難読化は解析者が逆アセンブルされた結果を読み難くしたり，逆コンパイルを妨害するために行われる．

1.1.2.1 表層解析

表層解析もマルウェアを実行させずに解析を行う方法であるので，静的解析の一種である．1.1.2 節の静的解析とは異なり，表層解析では実行されるコードまたはそれを逆アセンブル・逆コンパイルした結果を用いない．表層解析ではファイル内の文字列や IAT から使われる API を抽出するなどの，間接的にマルウェアの動作に関係がある部分を扱う．コードを参照する静的解析に比べて短時間で解析でき，また動的解析のように環境を準備する必要もない．しかし表層解析では，マルウェアの動作そのものを扱わないので，マルウェアの動作や機能を正確に解析することはできない．動的解析や静的解析を行う前に，マルウェアの概要を知るために表層解析が行われることがある．

1.1.3 本論文の提案と解析手法の関係

本論文の第 4 章ではシェルコードを特定する方法について述べる．文書型マルウェアを動的解析するためには，文書型マルウェアが想定する環境に適合するアプリケーションソフトウェアが必要である．本システムはアプリケーションソフトウェアを必要とせず，文書型マルウェアの内部のシェルコードを特定して実行可能ファイルを出力する．本システムはエミュレータを用いてシェルコードの特定を試みるが，これは動的解析の一種である．本システムは CPU と OS の一部をエミュレートするので，物理環境は全く必要ない．

本システムが出力した 32 ビット Windows 実行可能ファイルを動作させる環境は本論文の提案の範囲ではない．ゆえに解析環境の検出などの動的解析を妨害する方法に対応する方法は本論文では扱わず，他の提案によって解決されることを期待している．本論文では表 3.1 の実験環境を用いたが，32 ビット Windows 実行可能ファイルを動作させる環境は他の提案を用いることができる．本システムが特定したシェルコードを動作させた結果，作成される実行可能ファイルの解析は，本論文では第 5 章のアンパックを経て第 6 章の分類を行うことを想定している．しかし本論文の提案以外の方法も用いることができる．

1.1.2 節で挙げた静的解析の問題点を解決するために，第 5 章ではマルウェアをアンパックする方法について述べる．

アンパックでもエミュレータを用いており，第 4 章のシェルコードの特定と同様に本システムは CPU と OS の一部をエミュレートするので物理環境は全く必要ない．

第 5 章では Microsoft Visual Basic で作成された 32 ビット Windows 実行可能ファイルを判別し除外している．これは 1.1.2.1 節の表層解析の一種であると言える．

第 6 章では，マルウェアの検体を静的解析することで特徴を抽出し，マルウェアを分類している．第 5 章の提案で静的解析の問題点は解決済みであることが前提となっているが，本システム以外のアンパッカーを用いることも可能である．静的解析により特徴を抽出するので，動的解析の問題点である実行時に動作しなかった機能についても分類の対象とすることができる．

本論文はコードそのものを改変する難読化を解除し，解析者がコードを読み易くする方法は提案していない．マルウェアの分類では解析者が逆アセンブルされた結果を扱うことはなく，静的解析による API 推移の抽出は自動的に行われる．ゆえにマルウェアの分類ではこの種の難読化の影響は受けないので，本システムはアンパックのみを行っている．

第 2 章

関連研究

2.1 シェルコード抽出

2.1.1 ネットワーク通信解析

Polychronakis らはネットワークのパケットからシェルコードを検出する方法 [1] を提案している．文献 [1] ではシェルコードが自身のアドレスを取得するためのコードに注目してシェルコードの候補を決定している．また本システムと同様にコードをエミュレータで実行し，自身が書き換えたメモリを実行するという特徴にも注目している．しかし Process Environment Block（PEB）の取得や API 呼び出しといった他のシェルコードの特徴には対応していないので，自己書き換えがない場合には検出できない．

藤井らは自己書き換えがない場合にも対応する方法 [2] を提案している．また神保らは藤井らの提案する方法を用いてシェルコードを検知し，実行可能ファイルを作成して解析を行っている [3]．

本システムは文献 [1, 2, 3] を文書型マルウェアに応用したものであると言える．

2.1.2 文書ファイルの構造解析

Li らは統計的に分析することで Microsoft Word の文書ファイルがマルウェアであるか判定する方法 [4] を提案している．文献 [4] ではエントロピーについて言及されているが，この研究はシェルコードを特定するものではない．また文献 [4] では，実際に文書ファイルを開くアプリケーションの実装を多数準備して動的解析を行っており，アプリケーションを必要としない本論文のシステムとは異なる．

大坪らは文書ファイルの構造を調べることで文書型マルウェアを検知する方法 [5] を提案している．文献 [4] とは異なり，文書ファイルを開くアプリケーションを必要とせず，シェルコードの有無は問わず静的な解析のみで検知する．しかし文献 [5] の目的は検知であり，本システムのシェルコードの特定とは異なる．

2.1.3 シェルコード解析

Cova らは悪意のある JavaScript を検出する方法 [6] を提案しており，この中でシェルコードの抽出も行っている．JavaScript が生成するコードを抽出するので，バイナリデータの中から実行可能なコードを探すことでシェルコードを特定する本システムとは異なる．また Fratantonio らは文献 [6] で抽出したシェルコードを動的解析を行うためのツール [7] を提案している．本システムはシェルコードの特定を行い実行可能ファイルを出力することで，他の動的解析環境を利用することを前提としている．そのため，文献 [7] のように API のリストを出力するような機能はなく，文献 [7] で挙げられているような動的解析を行う上での問題点は，本システムでは他の動的解析環境が解決することを期待している．しかし本システムが出力する実行可能ファイルは，文献 [7] と同様に GetCommandLine と

GetModuleFileName をフックして実行可能ファイルの名称を標準で文書ファイルを開くアプリケーションに偽装している．本システムは文献 [8, 9, 10] のような動的解析環境を想定しているが，本システムが特定したシェルコードを文献 [7] のツールで動的解析することができる可能性はある．

2.1.4 シェルコード特定

Boldewin は文書型マルウェアを解析する OfficeMalScanner[11] を提案している．OfficeMalScanner ではシェルコードに特徴的なコードである GetEIP などのパターンを探すことでシェルコードを特定しているので，文献 [11] で想定されていない同等の動作をするコードが使われているときにはシェルコードを特定できない．パターンを探す方法ではパターンに柔軟性をもたせたならば，検知できる可能性は高くなるが誤認の可能性も生じる．さらにシェルコードが暗号化されているときには PEB へのアクセスや API 呼び出しなど，GetEIP 以外のパターンを見つけることはできない．本システムでは，コードをエミュレータで実行しているので，コードのパターンには依存しない．また暗号化されたシェルコードもエミュレータによる実行で復号できる．ゆえに本システムは文献 [11] で特定できないシェルコードも特定できる可能性がある．しかしエミュレータで実行しているため，その実効速度は文献 [11] よりも遅くなる．

OfficeMalScanner に含まれる MalHost-Setup は，本システムの 32 ビット Windows 実行可能ファイルを出力する機能と同様の機能を提供する．本システムが特定したシェルコードのアドレス情報を用いて MalHost-Setup で 32 ビット Windows 実行可能ファイルを出力することは可能であり，逆に OfficeMalScanner が特定したシェルコードのアドレス情報を用いて本システムで 32 ビット Windows 実行可能ファイルを出力することも可能である．ただし文献 [11] では，4.1.4 節で挙げられているような実際に文書ファイルを開くアプリケーションの状態を再現する方法については言及されていない．

2.1.5 文書型マルウェア内部の実行可能ファイル抽出

Boldewin が提案する OfficeMalScanner[11] には文書ファイルに埋め込まれた実行可能ファイルを探す機能がある．三村らは文書ファイルに埋め込まれた実行可能ファイルを抽出する方法 [12] を提案している．これらの方法ではシェルコードで用いられているエンコード方式を試すことで実行可能ファイルを特定する．本システムと同様に脆弱性をもつアプリケーションを準備する必要はなく，4.1.1 節で示した本システムが対応できないマルウェアにも対応できる．しかし提案する方法が想定していないエンコード方式が使われている場合には実行可能ファイルを特定できない．また，これらの方法ではシェルコードそのものの解析を行うことはできないので，ファイルをダウンロードする場合はその URL および作成されるファイルのパスや名前などを知ることができない．

2.2 アンパック

汎用的なアンパックを行う方法は，実機もしくはエミュレータ等でアンパックの対象となる実行可能ファイルを開き，ある条件で実行を打ち切ってメモリの内容をファイル化することである．この条件の違いが各アンパックの研究の違いであり，この条件の違いによりアンパックの精度が決定する．またコードの復号に付随して，オリジナルのエントリーポイントの特定や IAT の再構築が行われる．

2.2.1 整合性による検出

Josse らの研究 [13] では，実行可能コードの整合性（元のファイルイメージとメモリ上のコード）を検査するという自動的アンパックのアルゴリズムを提案している．API フック，オリジナルのエントリーポイントの検出，IAT の再構築などのアンパッカーとしての基本的な提案がされている．

Royal らの PolyUnpack[14] では，初めにコードを静的に逆アセンブルし，デバッカで実行時にその逆アセンブルしたコードではないコードが実行されようとしたときにアンパックを終了する方法を提案している．Kang らの Renovo[15] はエミュレータを用いてコードを実行する方法であり，書き変えられたメモリが実行されたときに，そこをオリジナルのエントリーポイントとしている．これらの方法では多重にパックされているときにはオリジナルのコードを検出できない可能性がある．

2.2.2 システムコールによる検出

Martignoni らの OmniUnpack[16] では危険なシステムコールを定義し，これらが呼ばれたときをマルウェア検出の条件としている．しかしアンパック中に危険なシステムコールが呼ばれた場合には，アンパックが不完全な状態になる問題がある．

LUNGU らの CJ-Unpack[17] では API 推移を元にコンパイラを認識している．マルウェアにおけるコンパイラの占める割合を調べており，コンパイラが作成したコードによる API 推移が現れた付近にオリジナルのエントリーポイントがあり，アンパックが終了したと判断している．

2.2.3 統計的手法

Sharif らの Eureka[18] はマルウェアの静的解析のためのフレームワークであり，ヒューリスティックな方法と統計的な方法を含んでいる．前者ではプロセス終了のシステムコールが呼ばれたときにメモリの内容を取る．これは多重にパックされている場合に有効だが，再パックには弱い．後者では IA-32 にありがちな機械語のパターン（push および call 命令）を検出することでアンパックを行っている．また織井らの研究では [19] では典型的な機械語が頻出する部分をオリジナルのコードセクションとしている．

岩村らの研究 [20] では確率モデルを用いてメモリ上のバイナリの尤度を求めて判定を行っている．また岩村らの研究 [21] ではコード領域の識別も行っている．

Kim らの研究 [22] では書き換え後に実行されたコードの量が増加したところをオリジナルのエントリーポイントとしている．また川古谷らの研究 [23] ではメモリアクセス傾向の変化が大きい場合にはオリジナルのエントリーポイントが近いと判断している．

塩治らの研究 [24] ではパックされたデータは均一であることに基づき，メモリに読み書きされる値のエントロピー値の変化を見てアンパックを行っている．

2.3 マルウェア分類

マルウェアの分類を行う研究は，主に動的解析と静的解析に分かれる．多くの研究では，まず始めに何らかの方法で各検体から特徴を抽出している．その特徴の抽出で実際に検体を実行させるケースを動的解析，そうでないならば静的解析と言える．その後，その特徴を比較することで各検体間がどの程度同じか（あるいはどの程度違うのか）を数値化する．

2.3.1 動的解析

動的解析の場合にはアンパックの必要がなく，パッカーが付加したバイナリコードも含めて動的解析を行う．Bailey らの研究 [25] では，マルウェアの動作から正規化圧縮距離を求め階層型クラスタ分析を行うことでマルウェアの分類を行っている．

Christodorescu らは，システムコールのログの引数の値からシステムコールの依存関係を求めて，グラフにして比

較する方法 [26] を提案している．

2.3.2 静的解析

静的解析の研究としては，Flake はマルウェアのコールグラフや制御フロー解析の結果のグラフを比較することで，マルウェアの分類を行う提案を行っている [27]．この方法はコンパイラの設定を変えるなどで，見かけのバイナリコードが大きく異なるような場合に対して耐性がある．

Kruegel らが提案するネットワークの通信からマルウェアを見つける方法 [28] では，マルウェアのバイナリコードを静的に制御フロー解析することで特徴をグラフとして表し，その部分グラフを比較を行うことで，マルウェアの検出を試みている．この方法では，グラフのノードに対応するバイナリコードの種類をグラフの属性（色）として与え，それを含めて一致するグラフを見つけることでマルウェアの検出を行っている．

Zhang らはメタモーフィック型のマルウェアを検出する方法 [29] を提案している．この提案では，逆アセンブルして得られた機械語の命令を，その関数呼び出しからブロックに分けて一致する部分を探すことで，マルウェアのパターンと検体の類似度を求めている．

Han らはマルウェア検体から API 呼び出し関係グラフを抽出し，Jaccard 指数を用いて類似度を求めることでマルウェアを検出する方法 [30] を提案している．また Han らはマルウェアが実行されるときに呼び出される API の順序と IAT が類似するという仮定に基づき，IAT から API リストを作成し，無害なプログラムから作成した API リスト（White List）の API を除いた上，重複した半順序関係を再帰的に除去することで検体間の類似度を求める方法 [31] を提案している．

Altaher らはマルウェアの検体がインポートしている API を抽出し，Evolving Clustering Method（ECM）による分類する方法 [32] を提案している．

文献 [27, 30, 31, 32] では何らかの方法で検体はアンパックされ，マルウェアのパックされる前のバイナリイメージが入手できていることを前提としている．

2.3.3 アンパックと静的解析

アンパックと特徴抽出・分類を組み合わせた提案がある．岩村らの研究 [33] では，動的なアンパックの後，マルウェアの検体を逆アセンブルし，その機械語の命令の並びを Jaccard 係数を用いて比較することでマルウェアの類似度を求めている．

Ye らの研究 [34, 35] ではマルウェアの検体から API を対応する数値（ID）に変換して抽出している．文献 [34] では Objective-Oriented Association（OOA）による分類を行っており，他の分類方法との比較している．文献 [35] では Jaccard 係数を用いて階層型クラスタ分析を行っており，教師ありの学習により実際の名前付けを反映させている．またこの研究では他者の名前付けとの比較も行っている．

第 3 章

実験環境・検体セット

3.1 実験環境

本論文の各種実験のために表 3.1 の実験環境を準備した．ただし 4.3.4 節のシェルコードを特定した検体で出力された実行可能ファイルは表 3.1 の仮想環境で実行した．

表 3.1 実験環境

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

3.2 検体セット

3.2.1 文書ファイル

第 4 章では事前調査および実験のために文書型マルウェアの検体を用いている．他の章では実行可能ファイル形式の検体を用いているため，共通する検体はない．表 4.6 のとおり，文書型マルウェアの内部にあるシェルコードが生成した実行可能ファイルが 50 種類ある．シェルコードが生成した実行可能ファイルを必要に応じてアンパックし分類することを本論文の提案は想定している．しかし 4.3.4 節で得られた実行可能ファイルは他の章の実験では用いられていない．

また 4.3.2 節では正常な文書ファイルで実験を行っている．この正常な文書ファイルも同様に他の章では用いられていない．

3.2.2 実行可能ファイル

本論文でいう「実行可能ファイル」とは 32 ビットの Windows を対象とする，Portable Executable（PE）形式で IMAGE_FILE_HEADER の Machine メンバの値が IMAGE_FILE_MACHINE_I386（14Ch）のファイルである．ただし Microsoft .NET Framework の Common Intermediate Language（CIL）が含まれている実行可能ファイルは除外する．

一般にマルウェアの名称は，科名（Family Name）と亜種名（Variant Name）で構成される．たとえばW32/Mydoom.A ならば Mydoom が科名，A が亜種名である．本論文では世界各国のマルウェア研究者の報告に基づいて作成されている WildList[*1]による命名を採用する．本論文で名称が与えられている検体はすべて 32 ビット Windows の実行可能ファイルなので接頭辞の W32 は以後省略する．

なお，第 6 章で sdbot v0.4b，sdbot v0.5a，sdbot v0.5b，rxBot v0.7.7 Sass のソースコードからコンパイラ・最適化オプションを変えてコンパイルして作成した実行可能ファイルの名称は存在しない．また付録 A の ZeuS の検体は WildList による名称がないため，一般的に用いられている呼び名を記した．

第 5 章および第 6 章では実行可能ファイル形式のマルウェア検体を用いている．第 5 章の 5,092 種類の検体は実行可能ファイル形式のマルウェアであることはわかっている．これらの検体にはパックされている検体とパックされていない検体があり，Microsoft Visual Basic で作成された検体も含まれている．

一方，第 6 章の 4,684 種類の検体は第 5 章で提案する手法でアンパックの処理が行われている．これらの検体はアンパックの処理の中で Microsoft Visual Basic で作成されていないことが判明している．4,684 種類の検体には第 5 章で下記の結果になった 4,461 種類の検体が含まれている．

- パックされていない検体（1,272 種類）
- アンパックに成功しかつエントリーポイントが既知のコードであった検体（312 種類）
- アンパックに成功しかつ検体内部に既知のエントリーポイントのコードが含まれていた検体（283 種類）
- アンパックの成否が不明な検体（2,594 種類）

残りの 223 種類の検体は第 5 章の実験の後に追加した検体であり，同様に上記の条件に一致した検体である．第 6 章で提案する分類ではアンパックが完全に成功している必要はなく，API 推移が抽出できれば十分である．そのためアンパックの成否が不明な検体も検体セットに含めた．

第 5 章ではマルウェア検体とは別にランタイムライブラリのコードを抽出するために 3,545 種類のマルウェアではない実行可能ファイルを用意した．またアンパッカーの性能評価のために既知のプログラムを 107 種類のパッカーでパックしたファイルも用意した．

第 6 章では 4,684 種類の検体とは別に sdbot v0.4b，sdbot v0.5a，sdbot v0.5b，rxBot v0.7.7 Sass のソースコードからコンパイラ・最適化オプションを変えてコンパイルして作成した実行可能ファイルを実験に用いた．本論文の実験のために作成した検体なので，他の検体セットとの重複はない．また，第 5 章および第 6 章の検体には付録 A の ZeuS は含まれていない．

[*1] The WildList Organization International http://www.wildlist.org/

第 4 章

シェルコード抽出

標的となる組織からの情報窃取を目的とした標的型攻撃の導入として，標的型メールにより不正なプログラムが送り込まれることが報告されている [36, 37]．標的型メールとは攻撃対象を特定の者に狙い定めて送られるもので，このメールには不正なプログラムが組み込まれた文書ファイルが添付されている場合がある．標的となった者はこの文書ファイルが攻撃目的で作成されたことを知らずに開封し，利用しているコンピュータで不正なプログラムを実行させてしまう．標的型攻撃対策の一環としては，このような文書ファイルを動的解析することで不正なプログラムの挙動を分析する．

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

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

4.1　提案手法

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

シェルコードの特定手順を図 4.1 に示す．

4.1.1　対象とする環境

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

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

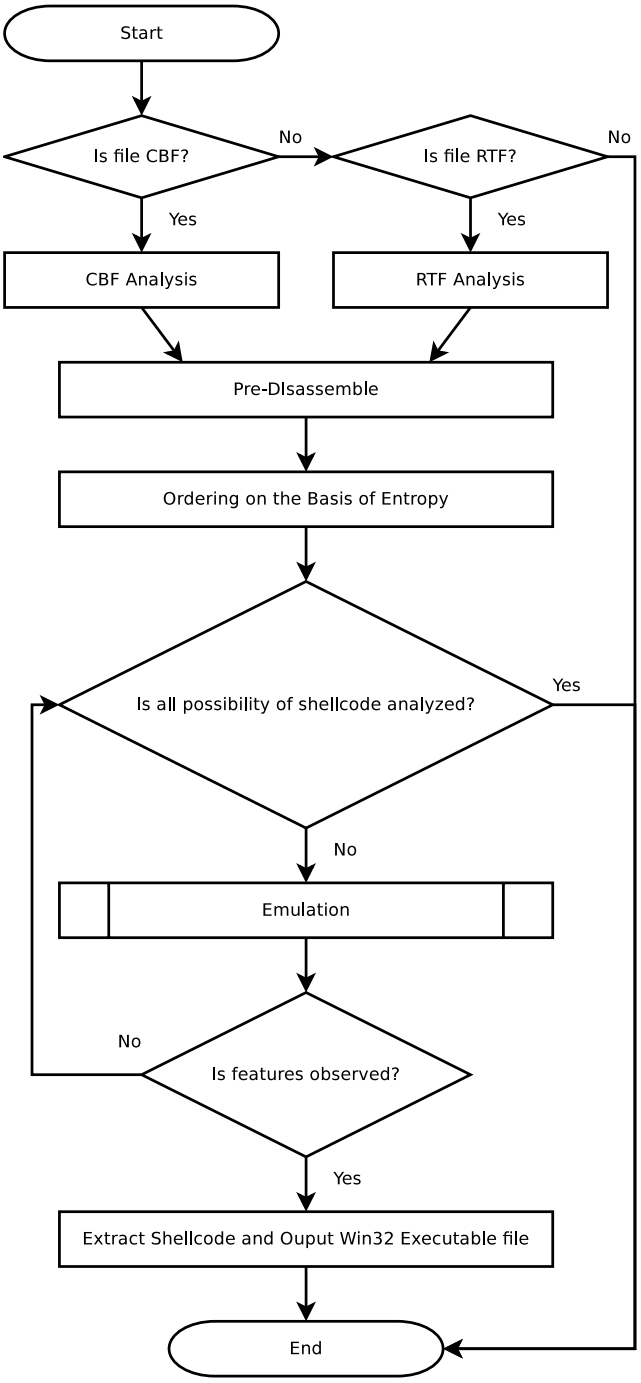

図 4.1 シェルコード特定手順

- Rich Text Format（rtf）

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

Return Oriented Programming（ROP）[38]

スタックに格納された戻り値に基づいてアプリケーションまたは OS 等のコードが実行される ROP では，文書

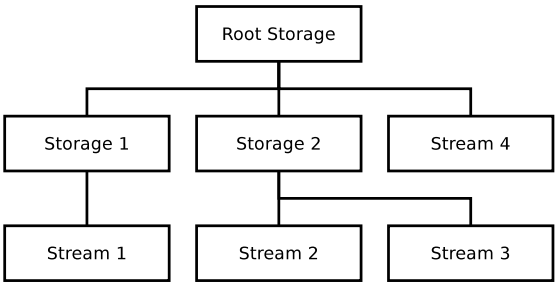

図 4.2　CFB 形式の構造

ファイル内部にシェルコードが含まれていない場合があり，本システムでは対応しない．シェルコードがある場合でも，特定のアドレスにシェルコードが読み込まれることが前提なっている場合には本システムでは対応できない．

文書ファイル内部にシェルコードがないマルウェア

脆弱性が攻撃されることで，文書ファイル以外のファイルのコードが実行される場合には，文書ファイル内部に存在しないので本システムでは対応しない．たとえば CVE-2011-1980[39] では文書ファイルと同じフォルダにある Dynamic Link Library（DLL）が実行される．

環境に強く依存する文書型マルウェア

シェルコードに汎用性がなく，メモリの確保やシェルコードのアドレスなど特定の条件が前提なっている場合には本システムでは対応しない．

4.1.2　候補の絞り込みと優先順位

本システムは次の方法でシェルコードの候補を絞り込み，優先順位を決定する．CFB 解析および RTF 解析による絞り込みはファイルを対象とするシステムであるから可能であり，ネットワークのパケットからシェルコードを抽出する研究 [1, 2, 3] には見られない．ファイルの構造を解析する研究 [4, 5] も提案されているが，それらはシェルコードの抽出を目的とはしていないので本論文の提案とは異なる．

4.1.2.1　CFB 解析

4.1.1 節のファイル形式は RTF を除いて Compound File Binary（CFB）形式 [40, 41] である．CFB 形式のファイルは図 4.2 のようにファイルシステムを模した構造になっており，Header，DiFAT，FAT，Mini FAT，Directory，Stream，Mini Stream，Free の各要素に分類できる．Header はファイルの先頭の情報領域である．DiFAT と FAT，Mini FAT はファイルシステムの FAT，Directory はディレクトリエントリ，Stream と Mini Stream はファイルに対応する．Free は未使用の領域である．本システムでは CFB を解析してファイル内をこれらの領域に分け，その中の特定の領域をシェルコードの候補とする．

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

4.1.2.2　RTF 解析

RTF はテキスト形式であるが，内部にはテキストにエンコードされたバイナリデータ（文字列を含む）がある [42]．シェルコードはバイナリデータの中にあることは明らかなので，RTF の場合にはこのバイナリデータを対象とする．また少なくとも 4.1.3 節で挙げた特徴をコード内に含む必要があるので，小さなバイナリデータの中にシェルコードを

収めることはできない．本システムでは 4.1.3 節の特徴をもつ最小のコードは 128 バイト程度であると見積り，128 バイト未満のデータは対象としない．

4.1.2.3 事前逆アセンブル

本システムではエミュレーションを行う前に，対象となるバイト列を逆アセンブルする．逆アセンブルが正しく行えないときにはエミュレーションを行わずに，そのバイト列にはシェルコードが存在しないとみなす．この処理はエミュレータの起動に時間がかかるため，エミュレータを起動する回数を減らすために行う．本システムでは逆アセンブルの可否を絞り込みのみに利用しており，逆アセンブル結果を用いる他のシェルコードを抽出する方法 [1, 2, 3] とは異なる．

4.1.2.4 エントロピーによる優先順位

仮にファイルの先頭から順番にシェルコード判定を行った場合，明らかにシェルコードではない箇所もエミュレーションされることになり効率が悪い．そこで他のシェルコードを抽出する方法 [1, 2, 3] とは異なり，エントロピーを用いてシェルコードの候補の順番を決定する．

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

$$H(X) = \sum_{i=0}^{255} -P_i \log_2 P_i \tag{4.1}$$

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

$$P_i = \frac{\displaystyle\sum_{j=1}^{n} \begin{cases} 1 & (a_j = i) \\ 0 & (a_j \neq i) \end{cases}}{n} \tag{4.2}$$

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

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

たとえば表 4.1 ではファイルのアドレスの 5E00 からシェルコードが開始される．シェルコードの手前は 00 が連続するのに対して，シェルコード部分は意味のある実行可能なコードなので値の分布に大きな偏りはない．

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

著者がツール類に頼らずに文書型マルウェアからシェルコードを特定するときには，表 4.1 のような文書ファイル中のシェルコードが存在しそうな場所を，バイナリエディタで探索して逆アセンブルを行っていた．本システムはこの作業をエントロピーを利用した定量的な評価により自動化する．

4.1.3 シェルコード判定

本システムはシェルコードが存在する可能性が高いファイル内部のバイト列を 32 ビット Windows で動作するコードとみなしてエミュレータで実行する．仮にそのバイト列からシェルコードが開始するならば，継続的にエミュレータで実行が可能であり，エミュレータの実行中にシェルコードに特徴的な動作が観測できる．エミュレーションが継続ができなくなったとき，または一定のステップ数の実行後に特徴的な動作が観測できないときには，本システムはエミュ

表 4.1　シェルコード付近のバイナリイメージ

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

レーションを打ち切り，次のシェルコードの候補に対して同様の処理を行う．すべてのシェルコードの候補で，特徴的な動作が観測できないときにはシェルコードが存在しないと判断する．

本システムでは下記の動作をシェルコードの特徴とする．

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

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

32 ビット Windows では CPU のレジスタの 1 つである FS レジスタに実行中のスレッドに関する情報を格納する Thread Environment Block（TEB）のアドレスが設定されている．図 4.3 のように TEB から PEB, PEB_LDR_DATA へと構造体内のポインタをたどると LDR_MODULE への連結リストがあり，LDR_MODULE から DLL のアドレスを取得できる．シェルコードは図 4.3 の構造体から DLL のアドレスを取得することで必要とされる API のアドレスを取得するので（2）の動作をシェルコードの特徴とする．

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

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

4.1.4　実行可能ファイル

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

動的解析のために実行可能ファイルを出力するという方法は，文書型マルウェアが対象ではないが他にも既に存在する [3, 11]．API をフックしてアプリケーションを偽装する方法もシェルコードを解析するツール [7] として既に存在する．

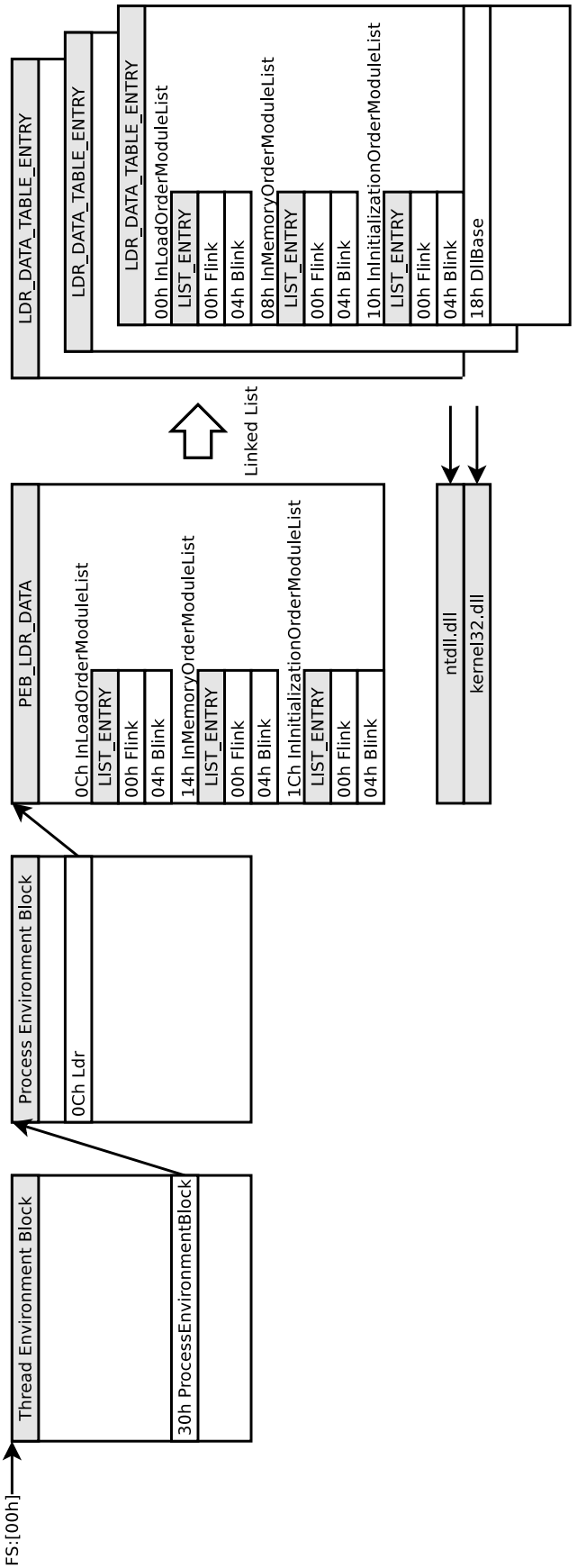

図 4.3 DLL のアドレスを取得するための構造体

表 4.2　ファイルの種類と脆弱性

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

4.2　事前調査

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

4.2.1　検体セット

著者は本システムの事前調査と実験のために，入手したマルウェアの検体の中から 4.1.1 節の条件に合致すると推測できる検体をランダムに選び，静的解析を行いシェルコードの存在が確認できた検体から検体セットを作成した．したがって検体セットの検体はシェルコードのアドレスが確認できている．CFB 形式の文書型マルウェアではランダムに選んだ 125 種類の検体の中から 73 種類が条件に合致し，RTF の文書型マルウェアではランダムに選んだ 25 種類の検体の中から 15 種類が条件に合致した．検体セットのファイル形式別の脆弱性の内訳は表 4.2 である．脆弱性は表 4.2 のとおり 15 種類あり，3 つのファイルは脆弱性の詳細等が不明である．脆弱性が異なるかシェルコードのファイル内のアドレスが異なるユニークな検体は 42 種類ある．

CFB 形式のファイルを一括してランダムで選んだため表 4.2 では各ファイル形式の比率に偏りがある．検体の入手の段階ではファイル形式の比率が均等になることは意図しておらず，表 4.2 の比率が入手できた検体のファイル形式のおおよその比率であり，本論文が対象としている攻撃のファイル形式の比率を反映していると思われる．表 4.2 の脆弱性にも偏りがあるが，同様に攻撃で使われる脆弱性の比率を反映していると思われる．

表 4.3 観測された特徴

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

表 4.4 最大ステップ数

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

4.2.2 シェルコードが存在する CFB の要素

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

4.2.3 ステップ数測定

エミュレータで実行する際に 4.1.3 節のシェルコードの特徴を観測するために必要なステップ数を決定する．そのため解析済みの検体のシェルコードの先頭からエミュレータで実行し，4.1.3 節の特徴が観測できるまでのステップ数を測定したところ表 4.3 と表 4.4 の結果になった．

4.2.4 エントロピー算出対象のバイト数とアルゴリズム

エントロピーを求める場合，ファイルの中のどの程度の長さのバイト列からエントロピーを算出するのが問題となる．そこで最も適切なバイト数，数式 (4.2) の n を求めるために 128 バイトから 2,048 バイトで検体セットのエントロピーを算出した．エントロピーの差は求めるバイト列の前のバイト列との差とする．実装上はファイルの範囲を超えてしまう場合にはファイルの先頭または末尾からエントロピーを求めることとする．また，すべてのバイト列のエントロピーを求めると時間がかかりすぎるので，16 バイト毎にエントロピーを算出した．

表 4.5 では，検体セットの検体に対して「エントロピーが高い順」と「エントロピーの差が大きい順」にシェルコードを探索した場合に，エミュレーションの試行回数と期待値（ランダムにバイト列を選びシェルコードの特定を試みた場合の試行回数）の比率の平均値をエントロピー算出対象のバイト数ごとにまとめた．

エントロピー算出対象のバイト数が 384 バイトで「エントロピーの差が大きい順」のときに最も試行回数が少なかった．このときシェルコードを探索した場合の試行回数と期待値の比率の分布を図 4.4 に示す．比率が小さいほど効率が良く，比率が 1 よりも小さければランダムに選ぶよりも効率が良いと言える．

また参考のため，ファイルの先頭から順番にシェルコードを探索した場合の試行回数と期待値の比率の分布を図 4.5 に示す．図 4.5 には示していないが，比率が 3 を超える検体は 2 つあった．

表 4.5　エミュレーション回数の比率の平均

Size	Entropy Order	Delta Order
128	0.561	0.330
192	0.581	0.317
256	0.578	0.288
384	0.554	0.268
512	0.593	0.270
1,024	0.715	0.305
1,536	0.817	0.403
2,048	0.882	0.550

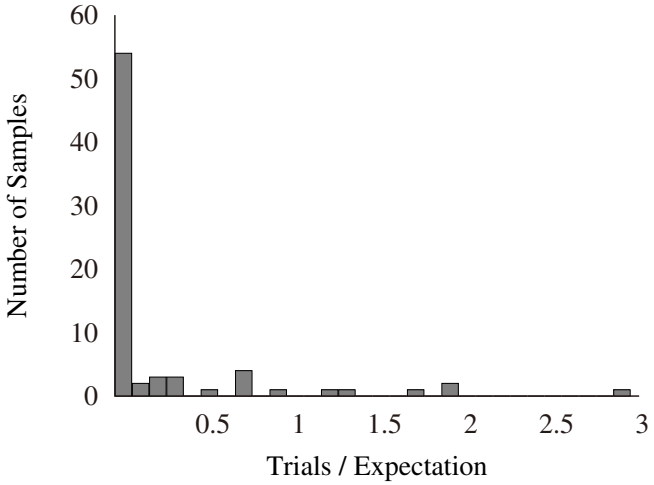

図 4.4　エントロピーが高い順の試行回数の比率

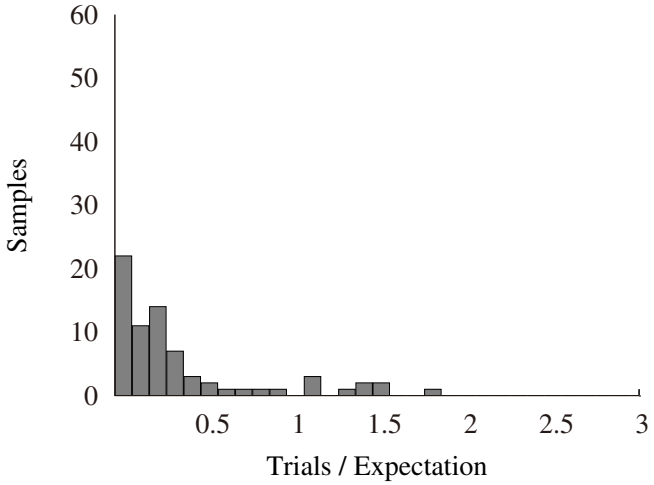

図 4.5　ファイルの先頭から順の試行回数の比率

4.3　実験

4.2.2 節の調査結果より，本システムは Stream 領域だけを探索する．4.2.3 節の調査結果より，本システムではエミュレータによる実行で書き換えたメモリの実行が観測されたときには 16,384 ステップ延長し，PEB へのアクセスが

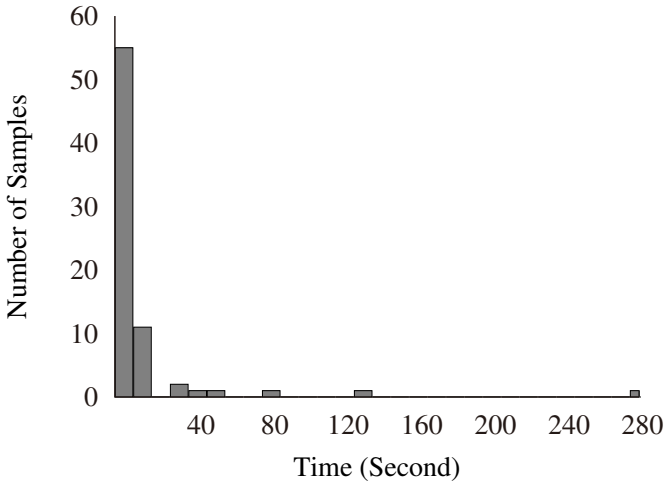

図 4.6 シェルコード抽出時間

あったときには最大 4,194,304 ステップ先まで実行して API 呼び出しを観測する．また 4.2.4 節の調査結果より，本システムではエントロピーを求める範囲のバイト数は 384 バイトで「エントロピーの差が大きい順」にシェルコードを探索する方法を実装する．

4.3.1 CFB・RTF 解析と逆アセンブルによる絞り込み

検体セットの検体に対して CFB 解析を行ったところ Stream 領域はファイル全体の約 31%，RTF 解析ではファイルのバイトサイズに対してバイナリデータは約 12% であった．またすべてのファイルに対して逆アセンブルできたのは約 96% であった．

4.3.2 False Positive 検査

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

4.3.3 シェルコード特定

表 3.1 の実験環境で本システムで検体セットの 88 種類の検体に対してシェルコードの抽出を試みたところ，74 種類の検体でシェルコードを特定することができた．全体では 2 時間 37 分 48 秒，シェルコードが特定できた検体だけでは 1 時間 29 分 49 秒かかった．しかし 1 時間 29 分 49 秒のうち 1 つの検体で 1 時間 13 分 19 秒かかっていた．エミュレータを 1 回実行するのにかかった時間は平均で約 1.638 ミリ秒であった．1 時間 13 分 19 秒かかった検体を除くシェルコードが特定できたときにかかった時間の分布を図 4.6 に示す．

なお，ファイルの先頭から順番にシェルコードを探したときには，シェルコードを特定できた 74 検体の中からラン

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

表 4.6　出力されたファイルの実行結果

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

ダムに選んだ 57 検体で約 6,804 秒かかった．エントロピーによってシェルコードの候補を決定する方法と性能を比較できれば十分なので 5,389 秒を超えたところで実験を打ち切った．シェルコードが特定できた検体 74 検体で約 2 時間以上，全体では約 3 時間以上はかかると推計できる．

4.3.4　動的解析の結果

本システムでシェルコードを特定した 74 種類の検体で出力された実行可能ファイルを表 3.1 に示す仮想環境で実行したところ，結果は表 4.6 になった．シェルコードがファイルを書き出して実行を試みた検体が 50 あり，1 検体は壊れたファイルが作成された．残りの 49 検体では 32 ビット Windows 実行可能ファイルが作成されて実行された．この 49 検体のハッシュ値を Virus Total で調べたところ，27 検体は Virus Total に登録されており，26 検体はマルウェアとして検出されていた．ネットワークに接続を試みた検体が 1 つあったが，サーバからの応答が得られずに通信は失敗した．一方，マルウェアとしての動作の前に無効な命令の実行や無効なメモリのアクセスなどが発生して継続して実行できなくなった検体が合計で 10 検体あった．またループから抜け出せなくなっている検体もあった．

4.4　考察

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

4.4.1　エミュレーションの絞り込み

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

4.4.2　エミュレーションの試行順番

本システムではエントロピーを用いて候補となるバイト列の優先順位を決定した．図 4.4 より多くの検体では期待値に対して十分に少ないエミュレーションの試行回数でシェルコードを特定できることがわかった．しかし 74 検体中 6 検体は期待値を超える回数のエミュレーションが必要な（ランダムに選ぶよりも効率が悪い）検体であった．

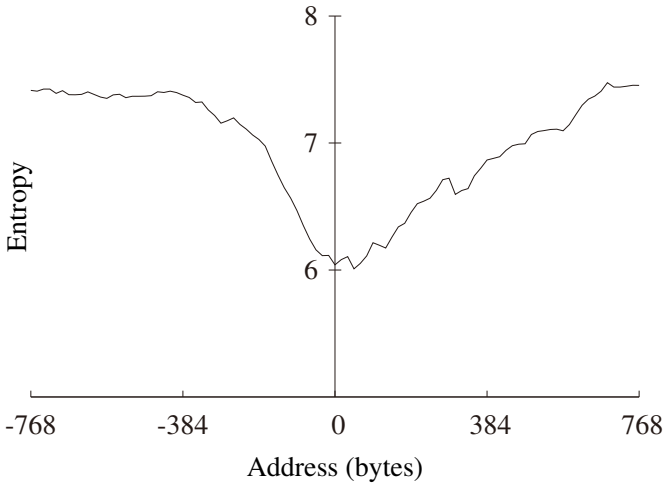

図 4.7 シェルコード付近のエントロピー

図 4.5 から，シェルコードの開始位置はファイルの先頭付近に偏っており，ランダムに候補を選ぶよりは効率が良いことがわかる．しかし図 4.4 と比べると図 4.5 は効率が良いとは言えない．4.3.3 節の実験では，57 検体で約 6,804 秒かかっており実際の実行結果でもエントロピーを用いて候補を求めるより効率が悪かった．ランダムに候補を選ぶ場合の実験は行っていないが，図 4.4 および図 4.5 から，ランダムに候補を選ぶ方法が良いとは推測できない．

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

検体のシェルコードのエントリーポイント付近における，エントロピー値の変化を図 4.7 に示す．図 4.7 の横軸はシェルコードのアドレスからの差であり，0 はシェルコードのアドレスになる．縦軸はエントロピーを示す．この検体のシェルコードの開始位置のエントロピーが約 6.041 であるのに対して，その前の文書ファイルのデータのエントロピーの方が高く約 7.377 であった．そのためエントロピーの差が負になったことでシェルコードの開始位置の優先順位が下がり効率が悪くなった．

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

4.4.3 動的解析の結果

シェルコード特定のためのエミュレータによる実行は最初の API 呼び出しで打ち切られるが，本システムが出力した実行可能ファイルではその後のコードも実行される．そのため表 4.6 の Failure に示すように，シェルコードが特定できても実際には実行できない検体もあった．シェルコードがロードされるメモリのアドレスやアプリケーションが確保しているメモリ等の再現や偽装が不十分であったことが原因であると考えられる．

一方，ファイルの作成と実行や通信の発生というようなシェルコードにおけるマルウェアとしての動作を確認できた検体もあった．たとえ作成されたファイルが実行できなかったりマルウェアとしての動作が観測できなくても，あるい

は通信に失敗しても，表 4.6 の Success の結果は脆弱性があるアプリケーションでシェルコードが動作したときと同じ動作であると思われる．対象となる脆弱性があるアプリケーションを準備せずにマルウェアを動的解析するという本システムの目的は達成しているので，これらの検体については成功したとみなした．

4.4.4　破損した検体

表 4.6 ではシェルコードが想定するファイルのサイズよりも実際のファイルサイズが小さいために無限ループに陥った検体があった．これらの検体は仮に脆弱性があるアプリケーションでシェルコードが動作しても，同様に無限ループに陥ってしまいマルウェアとしては成立しないと考えられる．

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

4.5　まとめ

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

本システムは文書型マルウェアの中でも 4.1.1 節で示すファイル形式だけが対象であり，またその中でシェルコードが存在してアプリケーションなしでも実行が可能であることを前提条件としている．本システムが対象とする文書型マルウェアが，文書型マルウェア全体のどの程度の割合になるのかは正確にはわからないが，本システムが限定的であることは間違いない．ROP をどのように自動解析するか，あるいはアプリケーションに依存する文書型マルウェアをどのように扱うかは課題の 1 つである．

一方，対応するファイル形式を増やすことで対象となる文書型マルウェアを増やすことも可能である．CFB 形式は他のアプリケーションでも使われているので，対応することは難しくないかもしれない．また Microsoft Office 2007 以降の Office Open XML 形式にも応用できる可能性がある．その他，圧縮またはエンコードされていても，元のバイナリイメージが取り出せるならば，本システムの方法は有効であると考えられる．

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

少数ではあるがランダムに選ぶよりも効率が悪い検体があった．エントロピーからシェルコードの候補を求める方法では，図 4.4 より多くは 10% 程度以下のエミュレーションでシェルコードを見つけることができる．ゆえに 10% 程度を探索してもシェルコードを見つけることができないときには，ファイルの先頭から順番にシェルコードを探すなど，異なる方法に切り替えることも検討できるかもしれない．

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

第 5 章

アンパック

コードを静的に自動解析するためには，コンパイラなどが出力したオリジナルのマルウェアのコードが必要である．しかし多くのマルウェアはパッカーと呼ばれるコードを圧縮し難読化するプログラムでパックされているため，同一のソースコードから派生したマルウェアであってもバイナリは異なる．そのため，パックされたマルウェアのファイルをそのまま静的に自動解析することができない．パックされたプログラムは，オリジナルのコードが隠蔽されており，パックされたプログラムが実行されるときにはパッカーが付加したコードが先に実行されてオリジナルのコードがメモリに展開されてからオリジナルのコードが実行される．この過程は図 5.1 に示す．

パックされたプログラムからオリジナルのコードを抽出する方法としては，パッカーが付加したコードを実行することでオリジナルのコードを取り出す方法がある．

手動でオリジナルのコードを抽出するためには，マルウェア解析者がデバッカを使ってマルウェアを実行し，オリジナルのコードがメモリに展開したと思われるところで実行を中断してメモリの内容をファイル化する．しかしこれは手間のかかる作業であり，大量にマルウェアが公開されている現状には対応できない．

そこでパックされたプログラムから自動的にオリジナルのコードを抽出するアンパッカーが研究されている．このとき，パッカーが付加したコードの実行を実際の実行環境（CPU，OS など）で行うのか，あるいはエミュレーションエンジンで行うのかの大きく分けて 2 つの方法がある．

実際の実行環境を用いる方法は多く研究されており，この方法ではメモリへの読み書きや実行を監視してオリジナルのコードの取得を試みる．その実装が不十分だとプログラムが監視下から離れて実行されてしまったり，あるいは実行不能になってしまう．また対象となるプログラム自身に監視下でプログラムが動作していることを検知され，処理が中断される可能性もある．

エミュレーションエンジンでコードを実行する場合には，エミュレーションの精度が問題となる．CPU の命令や API の実装などが十分でなければアンパックに成功しない．また実際にプログラムを実行するよりも時間がかかる．しかし実際の実行環境を用意する必要がないという利点もある．

本論文では著者が作成したエミュレーションエンジンを用いてパックされたマルウェアを展開し，ランタイムライブラリのパターンを用いてマルウェアのパックされる前のオリジナルのコードを取り出すことに成功したかを確認してオリジナルのエントリーポイントを特定する方法を提案する．

5.1　提案手法

コンパイラによって作成されたプログラムは多くの場合にランタイムライブラリをリンクするため，エントリーポイントのコードはプログラムの内容に関係なく共通のコードになると考えられる．本システムではこの考えに基づき，エントリーポイントのコードがランタイムライブラリのコードに一致するかどうかを比較することでオリジナルのコードの抽出を試みる．このコンパイラごとに共通する部分があるという考え方は LUNGU らの CJ-Unpack[17] で API 推

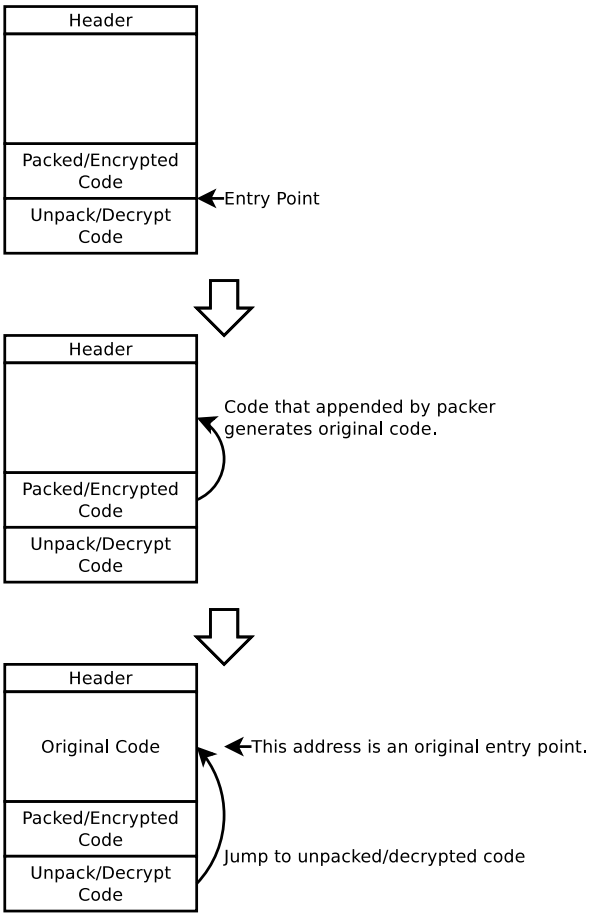

図 5.1　パックされたプログラムの実行

移を比較している方法でも用いられている．

まず一般に流通しているパックされていないと思われるプログラムからエントリーポイントのコードを抽出する．次にアンパックを行う前に，マルウェアのエントリーポイントのコードと抽出したランタイムライブラリのコードの比較を行う．ここで一致した場合にはパックされていないとみなす．パックされているときにはアンパッカーでアンパックを試みる．アンパッカーの出力結果からアンパッカーが新たに設定したエントリーポイントのコードと抽出したランタイムライブラリのコードの比較を行う．ここで一致した場合にはアンパックが成功したとみなす．さらにアンパックの成功が確認できなかった検体について，アンパッカーが出力したコード全体からランタイムライブラリのコードを探す．見つかった場合にはアンパックには成功したもののアンパッカーが新たに設定したエントリーポイントのアドレスが間違っていたということになる．

本論文では著者が作成したアンパッカーを用いてマルウェアのオリジナルのコードの抽出を試みる．アンパックの対象となるプログラムは 32 ビット Windows の実行可能ファイルであるが，このアンパッカーは CPU の動作および Windows の API をエミュレーションすることでコードを実行してオリジナルのコードをメモリに展開する．この方法は Josse らの研究 [13] による定義では Complete Software Interpreter Machine（CSIM）に相当する．著者はアンパックに必要な Windows の API だけを実装した．これは Martignoni らの OmniUnpack[16] で危険なシステムコールを定義した方法とは逆の方法だと言える．

表 5.1 逆アセンブルリスト

```
4010d5 push 58
4010d7 push 402218
(abbr.)
40111b jz   401135
40111d cmp  eax,esi
40111f jnz  401128
401121 xor  esi,esi
(abbr.)
4012ae call 4017b1
4012b3 ret
; Entry Point
401395 call 401818
40139a jmp  4010d5
```

5.1.1 エントリーポイントのコード抽出

コードの抽出はエントリーポイントからプログラムの流れに沿って逆アセンブルすることで行い，条件分岐では分岐しない方向に進める．また関数の呼び出しが起こった場合には，その呼び出される関数は逆アセンブルせずに次の命令に進む．このときコードのアドレスを示す部分はプログラムごとに変化するので無視してオペコードだけを見る．ret 命令や halt 命令などのプログラムの流れが打ち切られる命令が現れたとき，無効な命令が現れたとき，無効なメモリが参照されたときには終了する．エントリーポイントのコードを抽出する目的は，あるエントリーポイントのコードが既知のランタイムライブラリのエントリーポイントのコードと一致するか比べることが目的なので，一定の長さがあれば十分である．ゆえに本システムでは 24 ステップで抽出を打ち切る．また十分な長さがないと偶然に一致してしまうことがありうるので，本システムでは 8 ステップ未満で抽出を終了する原因が発生したときにはコードの抽出に失敗したとみなす．

表 5.1 はあるプログラムのエントリーポイント付近のコードである．表 5.1 ではエントリーポイントの 401395 では call 命令で 401818 が呼ばれているが，401818 の解析は行わず次の命令に進む．40139a では jmp 命令があるのでその流れに沿って 4010d5 へと進む．40111b の条件分岐命令では分岐先ではなく，その次の命令へと進む．表 5.1 から抽出されるではエントリーポイントのコードのパターンは call, jmp, push, push,...（途中省略）..., cmp, jnz となる．

5.1.2 アンパック

著者が作成したアンパッカーはエミュレーションエンジンでコードを実行する．エミュレーションエンジンは実行可能ファイルを確保したメモリに読み込み，CPU のレジスタやスタック，Process Environment Block，Thread Environment Block，例外ハンドラ，その他 Windows 7 で用意されている環境を準備する．また対象となるプログラムがインポートする DLL をメモリ上に作成して IAT を解決する．

エミュレーションエンジンではメモリに「レベル」という概念を導入する．レベルは図 5.2 のようにメモリのアドレス毎に存在し，すべての初期値は 0 とする．コードが実行されることでメモリの内容が書き変えられたときには，そのメモリのレベルをコードがあるメモリのレベルに 1 加えた値にする．既にレベルがそれより大きい値のときには変更しない．これは Kang らの Renovo[15] の Shadow Memory を拡張したものだと言える．

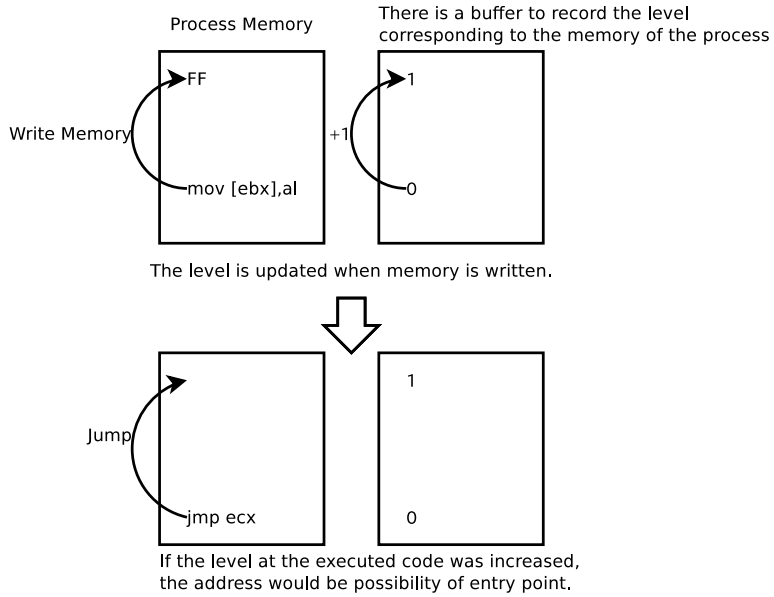

図 5.2　レベルの動作とエントリーポイント特定

実行中のコードがあるメモリのレベルが今までに実行したコードのあるメモリのレベルよりも大きいときには，そのアドレスをオリジナルのエントリーポイントの候補とする．より大きいレベルのメモリにあるコードが初めて実行される毎に，オリジナルのエントリーポイントの候補は更新される．

アンパッカーは ExitProcess の呼び出しなどの通常の終了手続き，無効な命令，確保されていないメモリの参照，スタックオーバフロー，実装されていない API の実行が行われたときにはエミュレーションを終了する．また所定の時間が経過したときには強制終了する．本システムではこれを 30 秒とした．アンパッカーはヘッダの設定および IAT を再構築して終了時のメモリの状態をファイル化する．このときにオリジナルのエントリーポイントの候補があるならばヘッダに設定する．

5.2　実験

5.2.1　実際のマルウェア検体による実験

本システムを検証するめに表 3.1 の実験環境を用意した．まず 3,545 種類のプログラムからランタイムライブラリのコードを抽出した．これらは主に OS とともにインストールされたファイルと，インターネットで入手可能なファイルである．これらのファイルから抽出したコードを比較し，同一のコードを除いたユニークなコードのパターン数は 306 種類であった．図 5.3 は同一のコードが見つかったファイルの数の上位 10 種類が全体に占める割合を示している．図 5.3 の上位 10 種類だけで 2,229 種類のファイルに対応しており，全体の約 63% を上位 10 種類のコードで占める結果になった．

本システムを検証するために 5,092 種類のマルウェアの検体を準備した．これらは 32 ビット Windows の実行可能ファイルであるが，それらがパックされているか否かはわからない．また Microsoft Visual Basic で作成された実行可能ファイルも含まれている．Microsoft Visual Basic によって作成される実行可能ファイルは 5.1.1 節のエントリーポイントのコード抽出で提案した方法は利用できないので，これらは除外する．Microsoft Visual Basic によって作成されたことが明らかな検体は 408 種類あったので残りの 4,684 種類について分析を行った．

この 4,684 種類のうちパックされていない検体を特定するために，これらの検体のエントリーポイントをランタイムライブラリのコードと比較した．その結果，本システムで抽出したランタイムライブラリのコードに一致したマルウェ

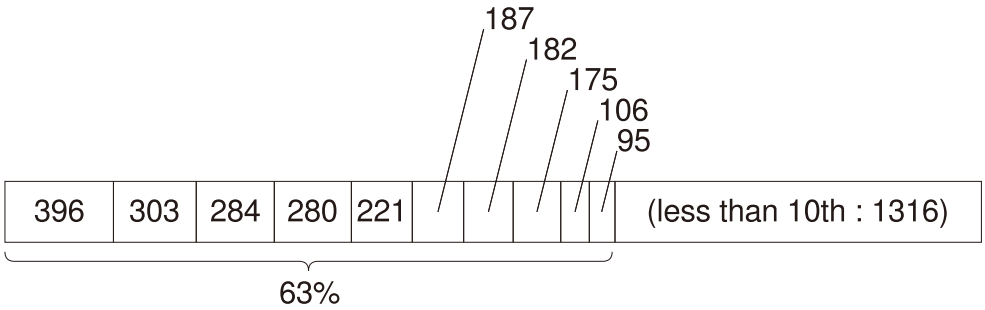

図 5.3 同一エントリーポイントをもつファイルの数の上位 10 種類が全体に占める割合

表 5.2 エントリーポイントのレベル

Level	Number	Level	Number
1	948	8	21
2	478	10	1
3	279	12	2
4	9	16	1
5	9	18	1
6	2	19	1
7	24	21	1

アの検体は 1,272 種類あった．これらを除く 3,412 種類は

- パックされている
- 未知のランタイムライブラリがリンクされている
- ランタイムライブラリがリンクされていない

のいずれかと考えられるので，これらの検体についてアンパックを試みた．

アンパックを試みたうち 175 種類はアンパッカーが API をサポートしていなかったか，エミュレーションエンジンが終了したときにヘッダの再構築ができなかったためにメモリダンプを作成することができなかった．ファイルを出力した 3,237 種類のうち 48 種類はアンパックした結果 Microsoft Visual Basic によって作成されたマルウェアであることがわかった．残り 3,189 種類のうち 1,777 種類はアンパッカーによってオリジナルのエントリーポイントが設定された．この 1,777 種類のうち 312 種類は本システムで抽出したランタイムライブラリのコードに一致した．表 5.2 はアンパッカーによって設定されたオリジナルのエントリーポイントのメモリのレベル毎の検体数である．

アンパッカーによってオリジナルのエントリーポイントが設定されたもののランタイムライブラリのコードに一致しなかった 1,465 種類とアンパッカーによってオリジナルのエントリーポイントが設定されなかった 1,412 種類の合計 2,877 種類について，アンパッカーが出力したファイルの先頭から 1 バイトづつそのアドレスにあるバイナリがランタイムライブラリのコードに一致するか比較した．アンパッカーによってオリジナルのエントリーポイントが設定されたときには 227 種類，アンパッカーによってオリジナルのエントリーポイントが設定されなかったときには 56 種類，合計で 283 種類についてファイルの中からランタイムライブラリのコードを見つけることができた．

図 5.4 はこの実験の手順をフローチャートで示している．アンパックの前後に行ったエントリーポイントとランタイムライブラリのコードと比較したときに要した時間は合計で 196.890 秒，1 ファイルあたり 0.025 秒であった．図 5.5 はアンパックにかかった時間である．475 種類の検体はアンパック中に 30 秒経過して強制終了した．また 2,877 種類のアンパックした検体からエントリーポイントを探す処理には 17 時間 19 分 25.051 秒かかった．

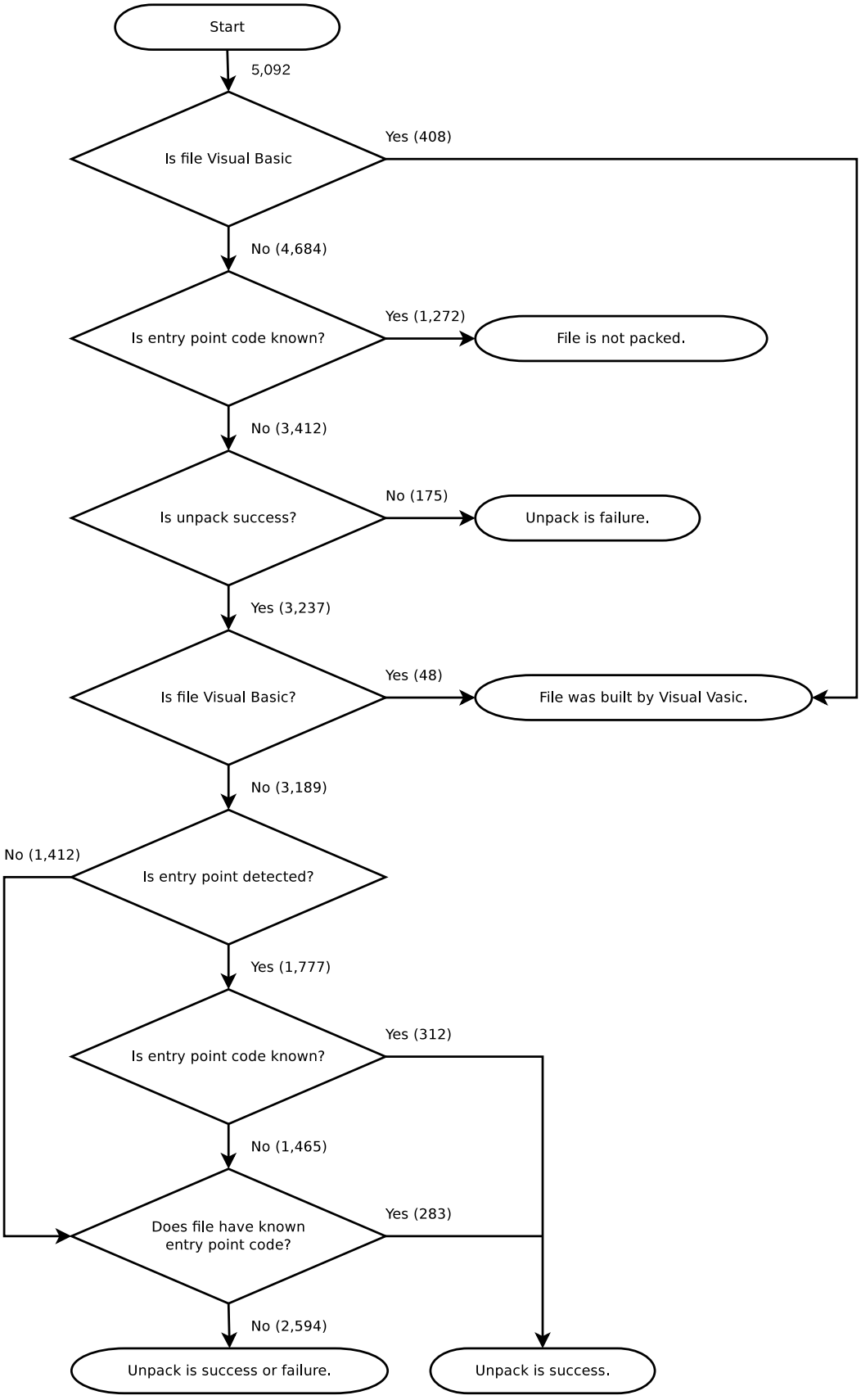
Start
5,092
Is file Visual Basic
Yes (408)
No (4,684)
Is entry point code known?
Yes (1,272)
File is not packed.
No (3,412)
Is unpack success?
No (175)
Unpack is failure.
Yes (3,237)
Is file Visual Basic?
Yes (48)
File was built by Visual Vasic.
No (3,189)
No (1,412)
Is entry point detected?
Yes (1,777)
Is entry point code known?
Yes (312)
No (1,465)
Does file have known
entry point code?
Yes (283)
No (2,594)
Unpack is success or failure.
Unpack is success.

図 5.4　アンパック手順

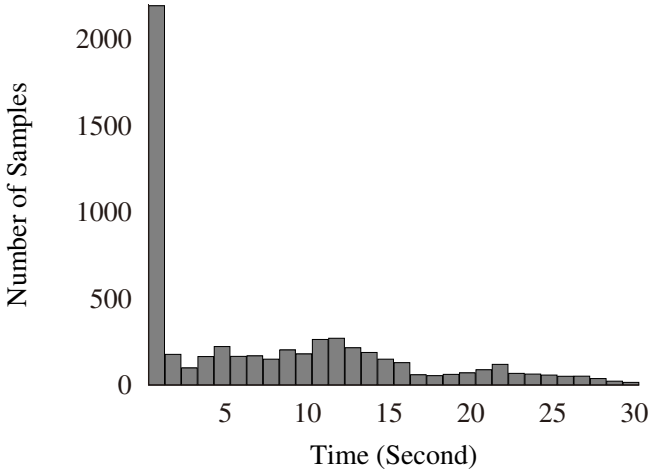

図 5.5 アンパックの所要時間

5.2.2 パックされた既知のプログラムを用いた実験

本システムを検証するために 107 種類のパッカーを準備し，既知のプログラム（calc.exe または regedt32，vim.exe など）をパックした．パックされた既知のプログラムを本システムでアンパックし，コードの復号とエントリーポイントの特定，IAT の再構築を試みた．結果を元のプログラムと比較することで，アンパックの成否を検証した．なお，ヘッダや IAT は元のプログラムと異なっていても，プログラムの実行に支障をきたさないならば，アンパックが成功したとみなす．

既知のプログラムをアンパックしたので，コードを復号できたときにはファイルの中からランタイムライブラリのコードを見つけることができる．ゆえに 5.2.1 節の実験とは異なり，アンパッカーが出力したコード全体からランタイムライブラリのコードは探していない．仮にランタイムライブラリのコードを探すことでエントリーポイントを特定したならば，表 5.3 の Unpack が有効な項目は必ず Entry Point の項目も有効になる．

表 5.3 は各パッカーでパックされたプログラムが，本システムでコードの復号，エントリーポイントの特定，IAT の再構築が成功したかを示す．コードのアンパックと IAT 再構築，エントリーポイントの特定のすべてに成功しているパッカーは 33 種類，コードのアンパックと IAT 再構築に成功したがエントリーポイントの特定に失敗しているパッカーは 24 種類，コードのアンパックだけが成功したパッカーは 6 種類であった．コードのアンパックとエントリーポイントの特定に成功したが IAT 再構築に失敗した事例はなかった．結果，合計で 63 種類のパッカーで成功し，44 種類では失敗であった．

表 5.3: パッカーの検証

Packer	Unpack	IAT	Entry Point
acpr pro 1.32			
AKALA v3.20.31122			
alloy	✓	✓	
armadillo softwarepassport v2.0.0			
armadillo v3.60			
Armadillov4.00.0053			
Armadillo v4.20			

ARMProtector v0.2			
ASPack 1.02b	✓	✓	✓
ASPack 1.05b	✓	✓	✓
ASPack 1.061b	✓	✓	
ASPack 1.07b	✓	✓	
ASPack 1.08.02	✓	✓	✓
ASPack 1.08.03	✓	✓	✓
ASPack 1.08.04	✓	✓	
ASPack 2.000	✓	✓	
ASPack 2.001	✓	✓	
ASPack 2.1	✓	✓	
aspack2.11	✓	✓	
ASPack2.11	✓	✓	
ASPack2.12	✓	✓	
AsProtect v1.2			
ASProtect v1.2 rc4 build 08.07			
ASProtect v1.23 rc1			
asprotect2.1			
CEXE 1.0a			
CryptX 1.0 build0.1	✓	✓	
DalKrypt 1.0	✓	✓	✓
EnigmaProtector 1.16			
epprotector 0.3 privatespecialbuild	✓	✓	✓
exe32pack 1.4.2	✓	✓	✓
exe32pack1.4.2	✓	✓	✓
EXECryptor 1.3			
EXEFog 1.1			
ExePack 1.0	✓	✓	✓
ExePack 1.4 lite final	✓	✓	✓
ExeStealth 2.73			
eXPressor1.5.0.1	✓		
Ezip 1.0			
Ezip1 1.0			
FakeNinja 2.8 ASPack2.xx			
FileXPack 1.0b			
FSG 1.2	✓	✓	✓
FSG 1.33	✓	✓	✓
fsg2.0	✓	✓	✓
FuckUnpacked 0.15			
HidePX 1.4	✓	✓	
JDPack 2.00	✓		

KillFlower 1.2	✓	✓	
MEW11 SEv1.1	✓	✓	✓
Mew11SEv1.2	✓	✓	✓
MoleboxPro2.6.4			
Morphine 1.5			
morphine v1.6			
Morphine 1.7			
muckiprotector II 5.1	✓	✓	✓
nPackv1.1.300	✓	✓	✓
NsPack v3.4	✓	✓	✓
NsPackv3.7	✓	✓	✓
Obsidium 1.2.5.0			
obsidium1.3.5.4			
obsidium1.4.5			
Packman 0.0.0.1	✓		
Packman 1.0	✓	✓	
PE optimizer 1.4	✓	✓	
PEBundle 2.30	✓		
PECompactv2.64	✓	✓	✓
PECompact2 v2.79 Engine20907	✓	✓	✓
PEDiminisher 0.1	✓	✓	✓
PELOCKNT 2.04			
PEncrypt v3.1	✓	✓	✓
PEncrypt v4.0	✓	✓	✓
PEPACK 1.0	✓	✓	
PEShield 0.25			
PESHIELD 0.26			
PESpinv1.33			
Pestil 1.0	✓	✓	
PETITE 1.2	✓	✓	✓
PETITE 1.3	✓	✓	
PETITE 1.4	✓		
PETITE2.2	✓		
PKLITE32	✓	✓	✓
RLPack 1.16 FullEdition	✓	✓	✓
RLPack1.20	✓	✓	
ScofieldPrivate 1.1	✓	✓	
Scramble.Upx 1.07w calc			
Shrinker 3.4			
SimplePack			
StonesPE-EXEEncrypter 1.13	✓	✓	✓

SVK-Protector v1.32 demo			
SVK-Protector v1.43			
teLock 0.96			
tElock 0.99			
teLock0.98			
tElockv0.98			
Themidav1.8.5.5			
unkOwnCrypter 1.0	✓	✓	
Upack0.39	✓	✓	✓
UPX3.08	✓	✓	✓
VGCrypt 0.75	✓	✓	
WinKript 1.0	✓	✓	✓
WinUpack0.31beta	✓	✓	✓
WWPack32.1.20	✓	✓	✓
XCR 0.13	✓	✓	
yodacrypter v1.3			
yodaprotector v1.02			

5.3　考察

5.3.1　実際のマルウェア検体による実験の評価

実験の結果，アンパッカーがエントリーポイントを特定しかつ既知のランタイムライブラリのコードと一致した 312 種類と，ファイル全体からランタイムライブラリのコードを探して一致するパターンを見つけた 283 種類の合計 595 種類の検体のオリジナルのエントリーポイントを特定できた．またパックされていないと特定した 1,272 種類と合わせると 1,867 種類の検体について，エントリーポイントのコードの比較によって検体の状態（パックされているか否か，アンパックが成功したか否か，オリジナルのエントリーポイント）を特定できた．これに Microsoft Visual Basic で作成された検体を加えると 2,323 種類となり，全体の約 46% に相当する検体について状態を特定できた．

一方で 175 種類はアンパッカーがバイナリイメージを作成できなかった．このうち 141 種類が RegisterServiceProcess などのアンパッカーで実装されていない API が IAT に含まれていたためである．アンパッカーは Windows 7 を基準にしているため，Windows 7 で実装されていない古い API は実装されていない．32 種類はアンパッカーが終了するときにヘッダ部分が破壊されていたためメモリの内容をファイル化できなかった．

アンパッカーがエントリーポイントを特定したときのメモリのレベルは約 53% が 1 であった．これは単純にパッカーが付加したコードが生成したコードを初めて実行したところをオリジナルのエントリーポイントとしており，多重にパックされていないことを意味する．しかし少数であるが，大変多く多重にパックされた検体もあった．レベル 21 の Mytob.FC が最大である．

アンパックに要した時間は 3,412 種類中 2,190 種類（約 64%）の検体が 1 秒未満であった．図 5.5 を見ると時間が増えたとしても，終了する検体数は大きく増加していない．仮にアンパックの作業を強制終了させる時間を延ばしても，終了条件に達する検体は少ないと思われる．30 秒以上経過しても終了しなかった検体は無限ループに陥っている可能性が高いと思われる．

5.3.2 パックされた既知のプログラムを用いた実験の評価

完全にアンパックに成功したパッカーは 107 種類中 33 種類（約 31%）であり，既知のパッカーでは失敗した事例の方が多かった．失敗の原因の多くはエミュレータが実際の OS の環境を完全に再現できていないため，アンパックが完了する前にエラーが発生したことであった．パッカーによってはアンチデバッグの機能が含まれているものがある．本システムでは既存のエミュレータを用いていないので，パッカーによる仮想環境の検知には対応できる．しかし通常は発生しないような例外を用いたり，エミュレータがサポートしていない命令を使うなどのには対応できなかった．

ASPack や PETITE のように同じ種類のパッカーであってもバージョンにより異なる結果になることもあった．新しいバージョンで何らかのアンチデバッグが加わったために失敗したということはなく，古いパッカーで失敗していても新しいパッカーでは成功しているという事例もあった．わずかなエミュレータの環境の差が，成否を分けたと思われる．

元のコードが復号できたパッカーでは 107 種類中 63 種類（約 59%）であり，コードの復号だけを考えれば半数以上では成功であった．

5.4 まとめ

今回の実験ではアンパッカーはランタイムライブラリのコードのパターンは扱わず，アンパッカーが出力した結果を検証するためだけに，ランタイムライブラリのコードのパターンを用いた．これを統合し，アンパッカーが検体のコードを実行している最中にコードの比較を行い，ランタイムライブラリのコードが実行された時点で終了してオリジナルのエントリーポイントを特定すれば，アンパッカーの実行効率は改善すると考えられる．

著者が作成したアンパッカーは Windows 7 を想定したので古い API が実装されていなかった．しかしこのアンパッカーは実際の OS を必要としないので，古い API と新しい API が混在するような実際にはない OS の構成も可能である．今後もアンパッカーが実装する API の数を増やしていく必要がある．

実験の中で最も時間がかかったのはアンパックした検体からエントリーポイントを探す処理であり，その時間は 17 時間 19 分 25.051 秒であった．時間がかかった理由はエントリーポイントのコードを比較するプログラムを利用して，単純にファイルの先頭から 1 バイトづつ比較を行ったためである．比較のアルゴリズムを見直したならば，実行時間は改善できる可能性がある．

登録しているランタイムライブラリのコードのパターンを増やしたならば，オリジナルのエントリーポイントを特定できる比率は増加すると思われる．しかしランタイムライブラリを使わずにコンパイラでプログラムを作成することも可能なので 100% に達することはない．またアンチデバックの手法の 1 つに Stolen Bytes がある．これはオリジナルのコードの一部を別のメモリに移して実行し，その後に本来のコードの位置にジャンプするという方法である．ランタイムライブラリのコードの一部が盗まれた場合には，そのパターンを検出できなくなる．

マルウェアの作者はランタイムライブラリのコードを同じ意味の異なる命令に書き変えることでも本論文で提案に対抗できる．ランタイムライブラリのコードと同じパターンになるコードをパッカーが付加すれば，オリジナルのエントリーポイントを誤検出することになる．マルウェアの作者が本論文の提案を知ったならば対抗することは難しくない．

以上のことから本システムだけで，オリジナルのエントリーポイントを完全に特定することはできない．しかし本システムで特定したオリジナルのエントリーポイントは，パターンを比較するという方法を用いているので誤検出の可能性は低いと思われる．

第 6 章

マルウェア分類

未知のマルウェアの検体が，既に解析されているどのマルウェアに似ているかを知ることができれば，検体の動作や構造を推定できるため，マルウェアの解析を行う上で手がかりとなる．

マルウェアを手動あるいは自動分類する上で障害となっている主な原因は，マルウェアの元になるソースコードが共通の基盤として広く流通し，配布バイナリイメージは異なっているが，似た機能・動作を有するマルウェアが存在することである．具体的には，ソースコードの改変やコンパイル環境の違いなどが原因で実行コードが異なった亜種が制作されている．

本論文ではマルウェアの解析のために，対象とする多数の検体を静的解析することで特徴を抽出し，元となったソースコードの構成に基づいた，精度の高いマルウェアの自動分類法を提案する．

本論文で提案する手法では，アンパックされた検体に対して逆アセンブルを行った後，制御フロー解析を行い全ての関数のグラフを取得する．制御フロー解析の結果は巨大なグラフとなるため，現実的な時間で比較することはできない．そこで提案する手法では，API が呼び出されたときに，次に呼び出される可能性のある API を探査することにより，巨大なグラフを API 推移依存グラフに収縮する．次に，マルウェアの検体から抽出した API 推移（特徴量）を比較することで，検体間の類似度を求める．類似度の算出ではグラフの比較は行わず，Dice 係数を適用することで実行時間の短縮を図る．最後に，特徴が似ている検体群の可視化のため，抽出した特徴量に基づいた階層型クラスタ分析を行う．

提案手法を評価するため，著者は，第 5 章のアンパッカーに加え，逆アセンブラ，制御フロー解析器，API 推移特徴抽出器，Dice 係数生成器，ならびに階層型クラスタ分析器の処理プログラムを制作し，自動マルウェア静的解析システムを構築した．耐性評価実験として，ソースコードが公開されている，あるマルウェアに対して，コンパイラと最適化オプションを変化させた場合の類似度抽出に関する耐性を調査した．性能評価実験としては，32 ビット Windows を攻撃対象とした 4,684 種類のマルウェアの検体を用意し，本システムにより分類した．結果，逆アセンブルとフロー解析に成功し，API 推移を抽出できた検体は 1,821 種類，科名では 113 種類の，類似度比較可能な検体を短い実行時間内で得ることができた．このデータセットからランダムに 500 種類を取り出し，階層型クラスタ分析を行い，係数 0.8 以上の類似度を有する検体でクラスタを作成したところ，有意な 56 クラスタを得た．しかしながら，検体の科名（名付け）との関係においては，マルウェアの実際の動作に基づく命名が為されている現状があり，静的解析のみに頼る手法の限界や課題も明らかになった．

表 6.1 ソースコード

```
#include <windows.h>
int WINAPI
WinMain (HINSTANCE hInstance,
         HINSTANCE hPrevInstance,
         LPSTR     lpszCmdLine,
         int       nCmdShow)
{
  int i, s = 0;

  for (i = 1; i < 10; i++)
    s += i;
  return 0;
}
```

6.1 提案手法

6.1.1 特徴抽出

コンパイラのバイナリコード生成オプションを変更した場合，生成されるバイナリコードが変化するため，バイナリコードを見ただけでは同じソースコードから作られたマルウェアであると推測することが困難である．たとえば表 6.1 のソースコードをコンパイラのオプションを変えてコンパイルした場合，

最適化なし 55 8B EC 83 EC 08 C7 45 FC 00 00 00 00 C7 45 F8 01 00 00 00 EB 09 8B 45 F8 83 C0 01 89 45 F8 83 7D F8 0A 7D 0B 8B 4D FC 03 4D F8 89 4D FC EB E6 33 C0 8B E5 5D C2 10 00
サイズで最適化 6A 01 33 C9 58 03 C8 40 83 F8 0A 7C F8 33 C0 C2 10 00
速度で最適化 33 C9 B8 01 00 00 00 03 C8 40 83 F8 0A 7C F8 33 C0 C2 10 00

となる．生成されたコードを見るとコンパイラのオプションを変えただけで，まったく別のコードになってしまう．

本システムによるマルウェアの分類の目的は同じソースコードから作られたマルウェアに対して高い類似度を与えることである．ゆえにバイナリコード生成時の環境変化に影響されない特徴を抽出する必要がある．

本システムではマルウェアの特徴抽出として，制御フロー解析の結果から API 推移の抽出を行い，マルウェアの検体間の類似度を求めて分類を試みる．既存のソースコードを利用して新しいマルウェアが作成されたときには，新たに改変されたバイナリコードの割合が多いほど，類似度は小さくなる．また 2 つ以上のマルウェアのソースコードを組み合わせて 1 つのマルウェアが作成されたときには，元になったマルウェアのコードが新しいマルウェアに占める割合が多いほど，類似度は大きくなる．

6.1.1.1 逆アセンブル

多くのマルウェアではソースコードが公開されていない．そのためマルウェアの検体を逆アセンブルすることで静的解析を行う必要がある．本論文では著者が作成した逆アセンブラを使用する．逆アセンブルの手法は主に線形掃引法（Linear sweep method）と再帰走査法（Recursive traversal method）に分かれる [43]．高い精度の結果を得るために実行時間は長くなるが，本システムの逆アセンブラでは再帰走査法を用いる．

表 6.2　逆アセンブルリスト

```
401000 xor  eax,eax
401002 cmp  [esp+04],eax
401006 jz   401012
401008 push [esp+04]
40100c call [402040] ; lstrlenA
401012 inc  eax
401013 ret
;
401014 call [402038] ; GetVersion
40101a test eax,eax
40101c jns  401022
40101e or   eax,ff
401021 ret
401022 cmp  [esp+04],01
(abbr.)
401075 xor  eax,eax
401077 pop  esi
401078 ret
```

6.1.1.2　制御フロー解析

6.1.1.1 節で取得した逆アセンブルリストに対して制御フロー解析を行う．エントリーポイントから制御フロー解析を行い，呼び出されている関数があるならば，その内部も再帰的に制御フロー解析を行う．

たとえば表 6.2 の逆アセンブルリストに示す命令列の場合，エントリーポイントに相当する 401014 から再帰的に制御フロー解析を行う．制御フロー解析においては，jmp や jnz，loop，call，ret などの分岐命令が必要である．例として，表 6.2 から分岐命令を取り出すと表 6.3 となり，これを制御フロー解析した場合には図 6.1 のグラフを得る．

6.1.1.3　API 推移の抽出

制御フロー解析の結果グラフからは，ある API が呼び出された後に呼び出される可能性のある API を判別可能である．制御フロー解析の結果グラフから API の推移を求めるために，グラフの中の API 呼び出しを行わないノードを，API 呼び出しを行うノードに統合する．

たとえば図 6.1 の場合には図 6.2 で点線で示したノードとエッジは消滅し，太線で示したエッジが作られる．図 6.3 は API 呼び出しを行わないノードを削除した結果である．さらに 40102b と 401052 は同一の API を呼び出しているので統合され，最終的には図 6.4 になる．

6.1.2　分類

マルウェア検体は 6.1.1.3 節で抽出した API 推移を要素とする集合とみなすことができる．この集合を各検体の特徴とし，各検体間の類似度を決める方法を定義する．本論文では同一の科名（マルウェアの名付け）の検体間の類似度および異なる科名の検体間の類似度を求め，これを比べることで定義した類似度について検証する．その後，検体間の類

表 6.3 分岐命令リスト

```
401006 jz   401012
40100c call [402040] ; lstrlenA
401013 ret
;
401014 call [402038] ; GetVersion
40101c jns  401022
401021 ret
401029 jg   40103a
40102b call [40203c] ; GetTickCount
401033 jz   401067
401038 jmp  40106b
401041 jle  401067
40104a call 401000
401052 call [40203c] ; GetTickCount
40105e jz   401035
401065 jl   401043
40106e call [4020ac] ; MessageBoxA
401078 ret
```

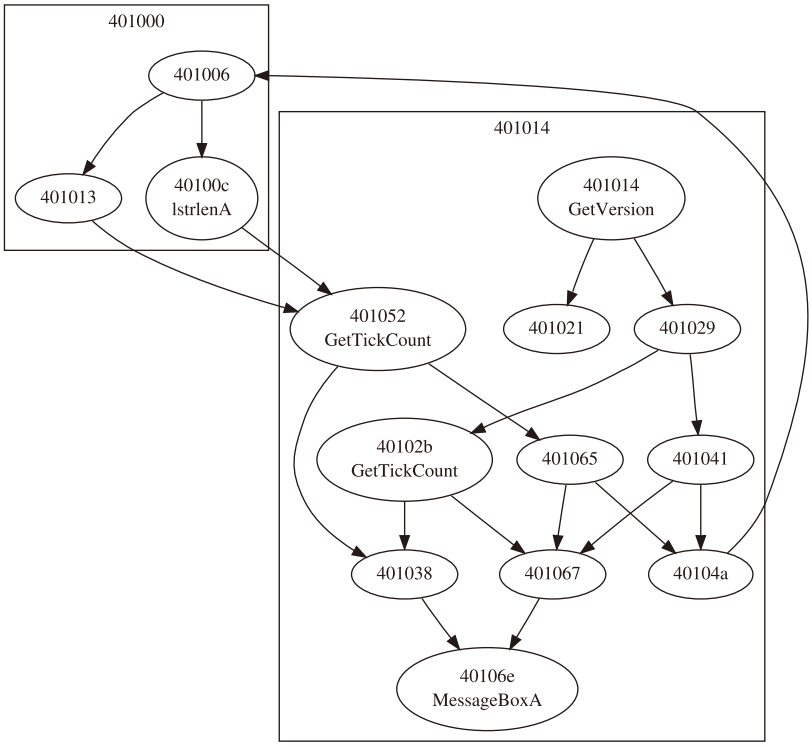

図 6.1 制御フローグラフ

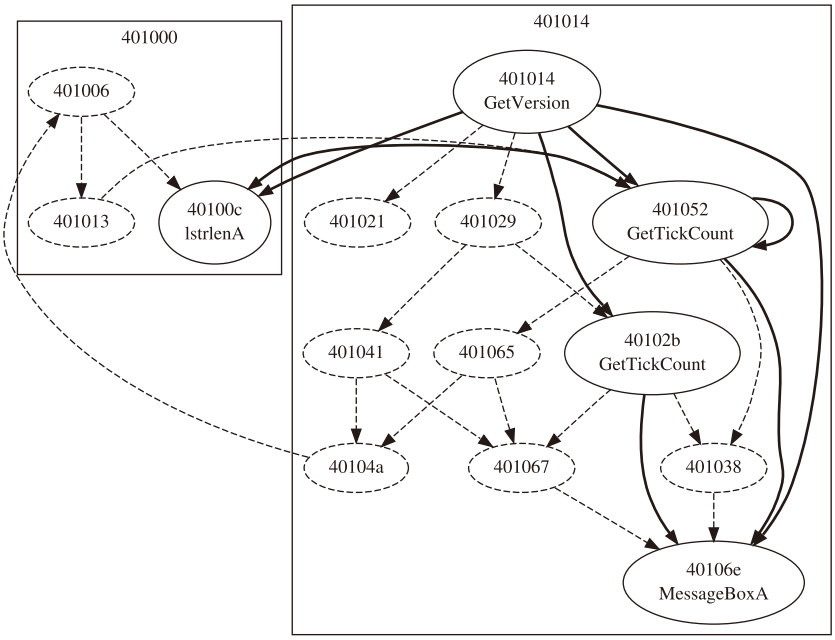

図 6.2　削除されるノードと追加されるエッジ

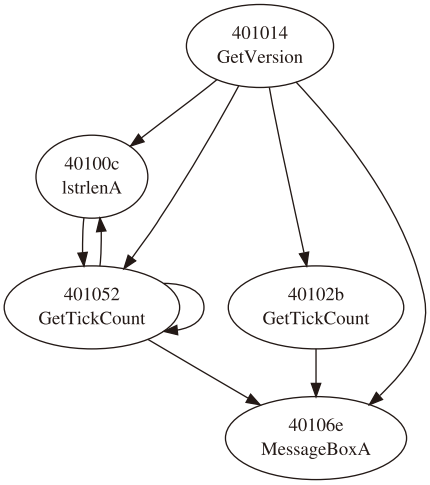

図 6.3　ノード統合前

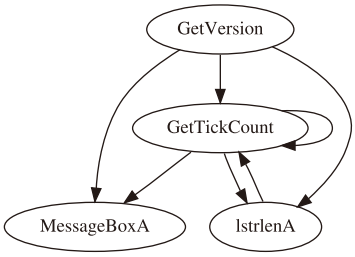

図 6.4　API 推移

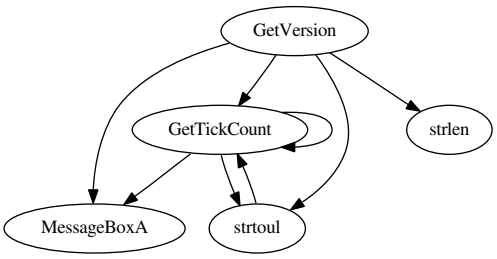

図 6.5 API 推移の比較例

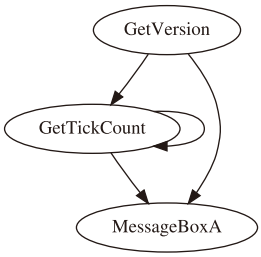

図 6.6 共通する API 推移

似度を基づいて階層型クラスタ分析を行う．

6.1.2.1 API 推移における検体間の類似度の定義

検体から抽出した API 推移の集合を X，Y とすると，これらの集合の類似度 S は Dice 係数を用いて

$$S = \frac{|X \cap Y| \times 2}{|X| + |Y|} \tag{6.1}$$

と定義する．$|X|$ は集合 X の要素数である．完全に一致するならば類似度は 1 になり，まったく一致しないならば類似度は 0 になる．

たとえば，図 6.4 と図 6.5 に共通するエッジは図 6.6 になる．図 6.4 の API 推移の数の合計が 7，図 6.5 の API 推移の数の合計が 8，共通する API 推移が 4 なので，2 つの検体の類似度は 0.53 となる．

6.1.2.2 階層型クラスタ分析による分類

検体間の類似度を求めた結果は，類似度の数値が並ぶだけの大きな表になるため，全体像を把握することが困難になる．そこで類似度に基づき階層型クラスタ分析を行い検体間の関係の可視化を試みる．

6.1.3 提案手法と関連研究の比較

本論文で提案する分類手法は，上で述べた既存研究と同様に，検体から特徴抽出を行い類似度を求めるという方法である．

静的解析で必要となるマルウェアの，パック前のバイナリイメージを得るアンパックの手法については，本論文の提案を含めて様々な研究が行われている [13, 14, 15, 16, 22, 17]．本論文のマルウェアの分類では，第 5 章でアンパックを試みた検体を用いるが，他のアンパッカーに関する研究で得られた成果を用いることも可能である．

文献 [26] では API（システムコール）の前後関係あるいは依存関係を動的解析を用いて抽出しているが，本論文の提案はこれを静的解析により API 推移として抽出するものである．

文献 [27, 28] と同様に本システムでも制御フロー解析を行いグラフを取得しているが，本システムでは API を呼び

表 6.4　プログラム

Name	Language	Description
disw32	C	Disassembler
ctlflw.pl	Perl	Control flow analysis
apiseq.pl	Perl	Extract API sequence
ascomp	C	Calculate similarity
dendro	C	Draw dendrogram with Graphviz
setup.pl	Perl	Classification

表 6.5　評価用コンパイラ

Compiler (abbr.)	Version	Optimization Option
Microsoft Visual C++ (msvc)	16.00.40219.01	/Od (Disable Optimization) /Ox (Maximum Optimization)
MinGW (mingw)	4.6.1	-O0 (Disable Optimization) -O3 (Maximum Optimization)

出すに至るバイナリコードを削除して，制御フロー解析のグラフを API 推移のグラフに収縮させている．本システムでは同じソースコードから作られたマルウェアであれば，コンパイラなどの違いにより変わることはない API の呼び出しに注目する．文献 [27] では API を扱わないので，API を実行時に取得する場合にも対応できるが，API を呼び出すに至るバイナリコードの影響をより大きく受ける可能性がある．

また文献 [30] は本システムと同様に API 呼び出しをグラフ化しており，またグラフを直接比較していないところも本システムとも類似している．しかし本システムとは異なり文献 [30] では API に関するブロックとエッジの数を比べることで類似度を算出している．

一方，文献 [28, 29, 33] はバイナリコードの違いも特徴として捉えて分類している．これらの方法では同じソースコードから作られたマルウェアであっても，コンパイラや最適化法の違いなどにより異なるバイナリコードが生成されたときには，類似度が小さくなる可能性がある．そのため，ソースコードを基に分類を行うためには，本論文の提案する方法が有効である．しかしバイナリコードそのものの違いを重要視するならば，文献 [28, 29, 33] などの方法が有効である．

文献 [31, 32, 34, 35] ではグラフを扱わないので本システムよりも高速な処理が期待できるが，API 呼び出しをグラフ化する本システムは API 呼び出しの前後関係をより正確に捉えることができる．

本システムでは文献 [25, 33, 35] と同様に類似度を定義して階層型クラスタ分析を行う．

6.2　実験

6.2.1　実験環境とマルウェア検体

提案手法の評価実験のため表 3.1 の実験環境と表 6.4 の著者が作成したプログラム，耐性評価実験のためにバージョンの異なる同種のマルウェアのソースコード，性能評価実験のために 4,684 種類のマルウェアの検体を準備した．

表 6.4 の disw32[*1]は 6.1.1.1 節で説明した逆アセンブラである．ctlflw.pl は制御フロー解析を行いグラフを

[*1] IWM Utilities http://gtklab.sourceforge.jp/iwmutlis/

表 6.6 評価用マルウェア

Name	Supported Compiler
sdbot v0.4b	msvc, lcc-win32
sdbot v0.5a	msvc, lcc-win32
sdbot v0.5b	msvc, lcc-win32, mingw
rxBot v0.7.7 Sass	msvc

表 6.7 sdbot のソースコード行数

Name	LOC
sdbot v0.4b	1,601
sdbot v0.5a	1,902
sdbot v0.5b	2,173

表 6.8 共通する行数

	sdbot v0.5a	sdbot v0.5b
sdbot v0.4b	1,309	1,188
sdbot v0.5a		1,661

Graphviz[*2]の DOT 言語で出力する．apiseq.pl は ctlflw.pl の出力結果から API 推移を表すグラフを DOT 言語で出力する．ascomp は 2 つ以上の apiseq.pl の出力結果から，それぞれの検体間の類似度を求める．dendro は apiseq.pl の出力結果に対して階層型クラスタ分析を行いグラフを DOT 言語で出力する．setup.pl はこれらのプログラムを検体に対して実行し，その出力結果から指定した類似度を閾値としてクラスタを生成する．

Microsoft Visual C++ 2010 の SDK に含まれる，インポートライブラリで定義されている 18,095 種類の API を対象とする．ただし API 呼び出しの間に挿入される可能性がある GetLastError，Sleep，SleepEx は対象としない．

6.2.1.1 耐性評価実験の検体

耐性評価実験のために，表 6.5 の 2 種類のコンパイラと，それぞれ 2 種類の最適化オプションを指定した合計 4 種類のコンパイル方法で，表 6.6 のソースコード[*3] をコンパイルしたバイナリイメージを準備した．これらのバイナリイメージを用いて，下記の 2 点の類似度を取得することで，提案手法の類似度の精度を評価する．

1. 同じコンパイラ・最適化オプションでコンパイルされた，別種・亜種のマルウェア間の類似度
2. 異なるコンパイラ・最適化オプションでコンパイルされた，同じマルウェア間の類似度

表 6.7 は 3 種類の科名 sdbot のソースコードの行数，表 6.8 は各ソースコードの共通する行数である．表 6.9 はソースコードの行数を集合の要素数，共通行の行数を積集合の要素数と見なし，Dice 係数に基づいて算出した各ソースコード間の類似度である．これらの検体は文献 [33] の 5.4 節の考察[*4] に相当する．

[*2] Graphviz - Graph Visualization Software http://www.graphviz.org/

[*3] rxBot v0.7.7 Sass のソースコードは for 文ループ変数に関する古い C++ の仕様に基づいて記述されているなどの問題があり，表 6.5 のコンパイラでコンパイルできない．そのため本システムの検証では元の意味を変えることが無い範囲でソースコードを修正してコンパイルした．

[*4] sdbot v0.4b には msvc 版と lcc-win32 版の 2 つの異なるソースコードがあり，文献 [33] では lcc-win32 版のソースコードの行数が提示されている．しかし本論文と文献 [33] では msvc で作成されたバイナリイメージが使われているので，本論文では msvc 版のソースコードの行数を表 6.7 に提示している．

表 6.9　diff コマンドに基づく類似度

	sdbot v0.5a	sdbot v0.5b
sdbot v0.4b	0.75	0.63
sdbot v0.5a		0.82

表 6.10　類似度行列，最適化オプション ‘Od’

	sdbot v0.5a	sdbot v0.5b	rxBot v0.7.7 Sass
sdbot v0.4b	0.67	0.61	0.05
sdbot v0.5a		0.81	0.08
sdbot v0.5b			0.08

表 6.11　類似度行列，最適化オプション ‘Ox’

	sdbot v0.5a	sdbot v0.5b	rxBot v0.7.7 Sass
sdbot v0.4b	0.71	0.67	0.08
sdbot v0.5a		0.79	0.09
sdbot v0.5b			0.11

表 6.12　‘sdbot v0.5b’ に関する類似度

	mingw (O3)	msvc (Od)	msvc (Ox)
mingw (O0)	0.85	0.80	0.79
mingw (O3)		0.74	0.75
msvc (Od)			0.85

6.2.1.2　性能評価実験の検体

性能評価実験のための 4,684 種類の検体は，第 5 章で提案する方法でアンパックを試みた検体である．検体は 32 ビット Windows の実行可能ファイルであり，第 5 章で提案する方法により Microsoft Visual Basic で作成されていないことが確認されている．

6.2.2　実験結果

6.2.2.1　耐性評価実験の結果

表 6.10，表 6.11 は表 6.5 に示す同じコンパイラ・最適化オプションで，表 6.6 のソースコードをコンパイルしたバイナリイメージの間の類似度である．表 6.12 は表 6.5 に示す異なるコンパイラ・最適化オプションで表 6.6 の亜種 sdbot v0.5b のソースコードをコンパイルしたバイナリイメージの間の類似度である．

6.2.2.2　性能評価実験の結果

4,684 種類の検体に対して，逆アセンブル，制御フロー解析，API 推移抽出を行ったところ，API 推移を抽出できた検体数は 1,821 種類，科名では 113 種類であった．1,821 種類について表 3.1 に示す計算機上で分類プログラムを実行した結果，逆アセンブルに要した時間は 5 時間 26 分 3 秒，制御フロー解析に要した時間は 43 分 38 秒，API 推移の抽出に要した時間は 3 時間 7 分 15 秒であった．図 6.7，図 6.8，図 6.9 はそれぞれ逆アセンブル，制御フロー解析，API

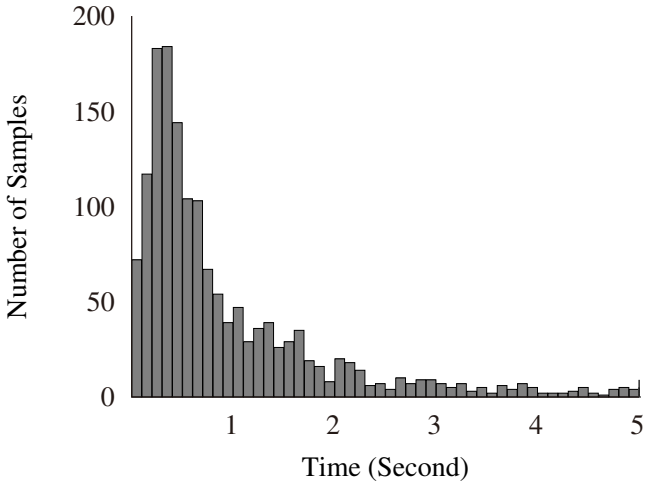

図 6.7 逆アセンブルの時間と検体数

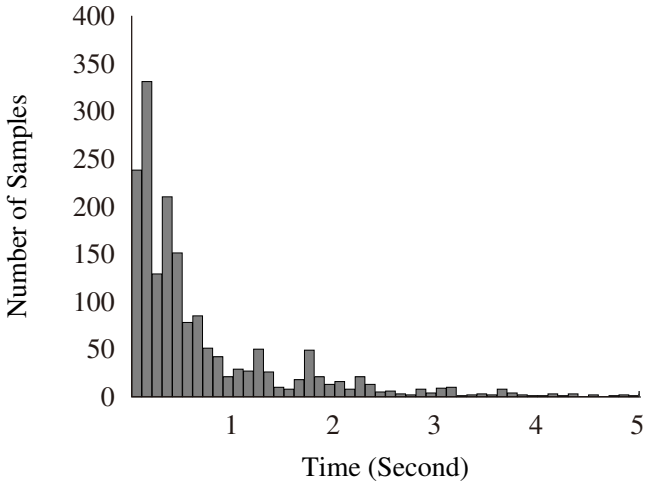

図 6.8 制御フロー解析の時間と検体数

推移抽出に要した時間と検体数の関係を表すグラフである．グラフは横軸が 0.1 秒毎の時間，縦軸が検体数を示している．時間が 5 秒以上であった検体は逆アセンブルでは 231 種類，制御フロー解析では 92 種類，API 推移抽出では 55 種類であった．最も時間がかかった検体は，逆アセンブルでは 41 分 9 秒，制御フロー解析では 50 秒，API 推移抽出では 1 時間 3 分であった．図 6.10 は横軸は抽出できた API 推移の数，縦軸は検体数である．136 種類の検体は抽出できた API 推移の数が 2,000 を超えていた．

検体から抽出した API 推移に基づいて検体間の類似度を求めた．検体間の類似度を求めるのに要した時間は 1 時間 41 分 36 秒であった．図 6.11 は，検体間全体の類似度および表 6.13 に示す科名に属する検体数が多い上位 4 種類について，検体間の類似度の分布を示すグラフである．横軸は 0.1 毎の類似度，縦軸はその範囲の類似度が対象に占める割合を表す．検体数が多いためすべての類似度を表にすることはできないので，それらの中から検体を選びその検体間の類似度を表 6.14 に示す．

図 6.12 は 1,821 種類の検体のうちランダムに選んだ 500 種類について，階層型クラスタ分析による可視化を行った結果である．この樹形図ではノードは検体を表し，類似度が高いノードは近くに配置される．表 6.13 の検体はそれぞれ固有の形のノードで表し，それ以外の検体は同じ形のノードで表す．今回のケースでは，類似度 0.8 以上の検体でク

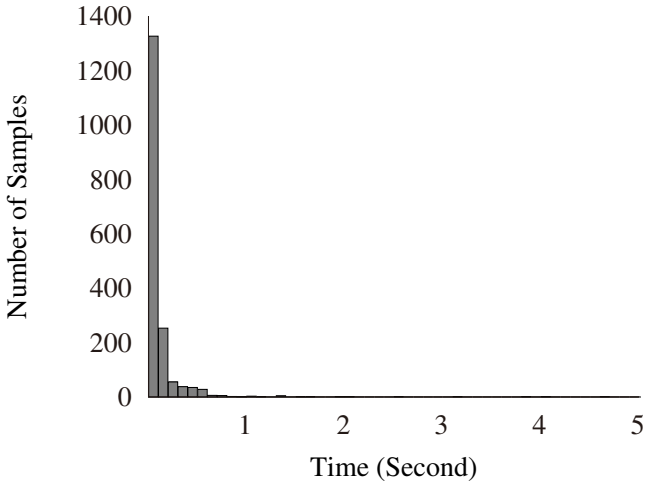

図 6.9　API 推移の抽出の時間と検体数

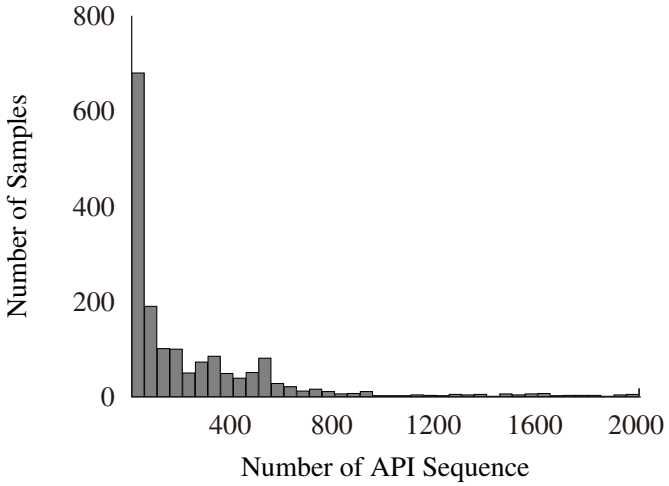

図 6.10　API 推移の数と検体数

ラスタを作成したとき，56 のクラスタが作成され，172 種類の検体がクラスタに属さなかった．表 6.15 はクラスタに属する検体数が多い 10 のクラスタの，そのクラスタで最も多い科名，その科名の検体数，クラスタの検体数を示す．同様に類似度 0.6 以上では表 6.16 になり，40 のクラスタが作成され，119 種類の検体がクラスタに属さなかった．

6.3　考察

6.3.1　耐性評価実験の評価

表 6.10 と表 6.11 の双方で 3 種類の sdbot 間の類似度は高い値になっている．一方，sdbot と rxbot の類似度は低いことから，亜種と別科の関係が類似度に反映されている．また表 6.9 では sdbot v0.5a と sdbot v0.5b の類似度が最も高く，次に sdbot v0.4b と sdbot v0.5a，sdbot v0.4b と sdbot v0.5b の類似度が最も低い．表 6.10 と表 6.11 でもこの大小関係は変化していない．

表 6.12 では，同じソースコードであってもコンパイラや最適化オプションによって制御フローが変化すること，コ

表 6.13 科名の上位 10 種類

	Family	Number
1	Koobface	208
2	Palevo	129
3	Mytob	92
4	Autorun	79
5	Ircbot	42
6	Kolab	29
7	Sdbot	27
8	Yahos	24
9	Areses	21
10	Agent	20

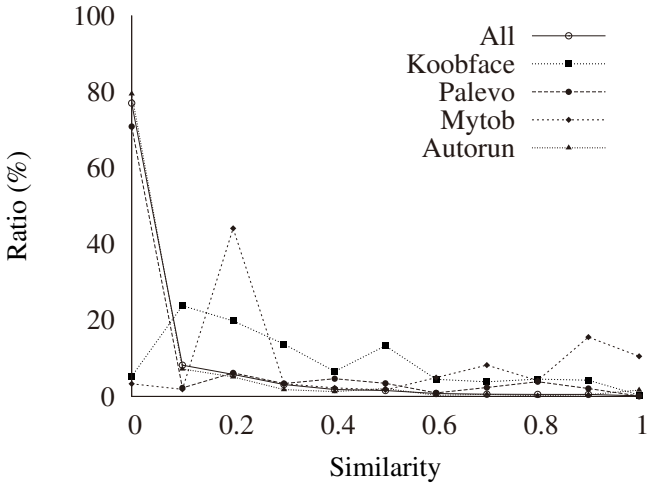

図 6.11 類似度の分布

ンパイラがランタイムライブラリの初期化やスタックのチェックのために，固有のコードをバイナリイメージに付け加えること，などが原因で類似度は 1 にはならなかった．しかし文献 [33] とは異なり類似度は十分に高い値になっており，ソースコードに基づいてマルウェアの分類を行うという，本システムの目的を達成している．

表 6.10 と表 6.11，表 6.12 を比べると，同じソースコードで異なるコンパイラにおける類似度よりも，異なるソースコードで同じコンパイラ・最適化オプションにおける類似度の方が高くなっている事例がある．通常はソースコードの改変はその部分にのみ影響を与えるが，コンパイラの変更はバイナリコード全体に影響を与えることが原因である．

6.3.2 性能評価実験の評価

6.3.2.1 API 推移抽出が困難なマルウェア

今回のケースで，API 推移を抽出できた検体は 4,684 種類中 1,821 種類（約 39%）であった．API 推移を抽出できない原因は 2 つ考えられる．1 つは本システムにおけるマルウェアの分類では検体がアンパック済みであることが求められており，それが十分でなかったことである．実際に著者がいくつかの検体を解析したところ，IAT の再構築が不完全であったり，アンパックの途中で終了していた検体があった．この場合，研究の前段階である第 5 章のアンパッカー

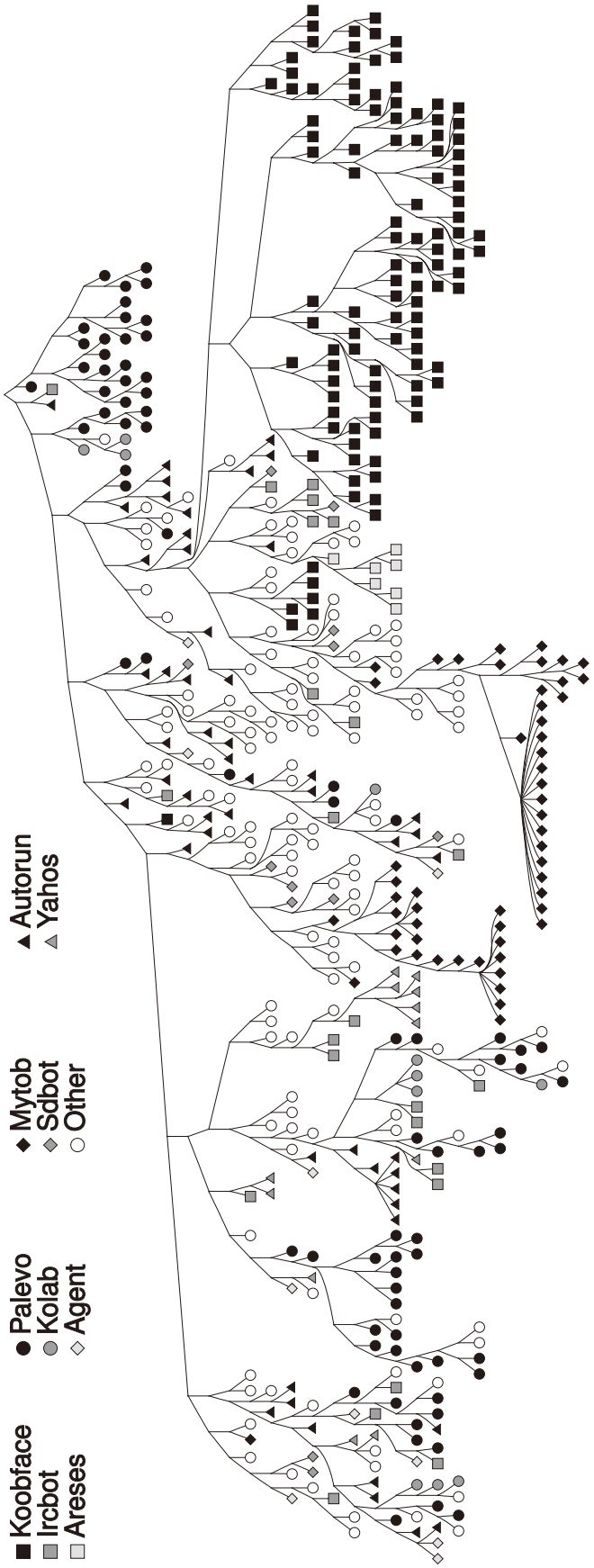

図 6.12　階層型クラスタ分析木

表 6.14 マルウェアの検体間の類似度

	Koobface.026	Koobface.064	Mydoom.60	Mytob.120	Netsky.AA	Palevo.186	Palevo.241	Scar.04	Slenfbot.47
Autorun.019	0.10	0.10	0.07	0.08	0.03	0.02	0.02	0.01	0.01
Koobface.026		0.99	0.03	0.07	0.02	0.02	0.02	0.00	0.02
Koobface.064			0.03	0.07	0.02	0.02	0.02	0.00	0.02
Mydoom.60				0.63	0.09	0.02	0.02	0.00	0.03
Mytob.120					0.08	0.01	0.01	0.00	0.03
Netsky.AA						0.37	0.52	0.16	0.19
Palevo.186							0.45	0.25	0.26
Palevo.241								0.19	0.23
Scar.04									0.47

表 6.15 類似度 0.8 以上のときの上位 10 クラスタ

	Family	Number	Total	Ratio(%)
1	Koobface	34	34	100.0
2	Palevo	13	30	43.3
3	Mytob	28	28	100.0
4	Koobface	26	26	100.0
5	Palevo	18	21	85.7
6	Autorun	4	18	22.3
7	Koobface	12	12	100.0
7	Mytob	12	12	100.0
9	Sdbot	2	8	25.0
10	Koobface	8	8	100.0

を改善するか，文献 [25, 26] のような動的解析で分類する必要がある．

もう 1 つの原因は実行時に API のアドレスが取得されることである．通常，Windows では API のアドレスはプログラムが実行される前に IAT の記述に基づいて OS によって解決される．しかし Stration などの一部のマルウェアは，マルウェアが実行されているときに必要な API のアドレスを随時取得している．そのため本システムのようにマルウェア検体を実行しない静的解析では，実行時に API のアドレスが取得される API の呼び出しを抽出できない．この API 抽出に関する問題は静的解析のみでは解決できないため，別途動的解析の結果に基づいた API 呼び出しアドレスの特定等，他の手法の併用を検討するか，文献 [27, 33] のように API に依存しない別の特徴を抽出する方法を検討する必要がある．

6.3.2.2 実行時間

図 6.7，図 6.8，図 6.9 より，多くの検体は数秒で解析を終えているものの，一部のバイナリコードのサイズが大きい検体では，解析に時間がかかっていることがわかった．図 6.10 において，API 推移を抽出できた検体の中では，その

表 6.16　類似度 0.6 以上のときの上位 10 クラスタ

	Family	Number	Total	Ratio(%)
1	Koobface	74	74	100.0
2	Palevo	13	56	23.3
3	Palevo	10	36	27.8
4	Mytob	28	34	82.4
5	Palevo	18	25	72.0
6	Mytob	23	23	100.0
7	Koobface	20	20	100.0
8	Sdbot	2	9	22.3
9	Areses	7	7	100.0
10	Looked	7	7	100.0

数が 100 未満が多数を占めていた．一方で少数ではあるが 2,000 を超える検体があった．これは Downloader のよう単一の目的に特化したマルウェアと，Bot のような複数の機能を持つ汎用的なマルウェアの違いであると考えられる．分類プログラムの実行時間の期待値は，対象とするマルウェアの検体数に比例して増大するので，並列実行可能なプロセッサあるいはクラスタリング分散処理によって，必要な実行速度が得られる．

6.3.2.3　類似度の分布

図 6.11 において，全体を見ると求めた類似度の 80% 以上は 0.1 未満であった．113 種類の科名であることから考えると，同じ科名の検体間で類似度が求められるのは 1% に満たないので，この結果は妥当である．

同一の科名の中では，Koobface と Mytob は全体の傾向とは異なる結果となった．Koobface の類似度の分布は 0.1 以上 0.2 未満，Mytob の類似度の分布は 0.2 以上 0.3 未満が最大となっている．さらにそれ以上に大きな類似度の分布も多くなっており，同一の科名では類似度が高くなることを示している．

一方，同一の科名であるが Palevo は全体よりも若干類似度が高くなっており，Autorun は全体の傾向とほとんど違いがなかった．これは Autorun はソースコードの関連性による命名ではなく，メディアの自動実行機能を利用して広まるマルウェアに対して Autorun という科名が与えられていることが原因である．著者が Autorun の検体をいくつか解析したところ，異なる C 言語のソースコードから作成された検体や，本論文の実験で用いた検体ではないが Microsoft Visual Basic で作成された検体もあった．また他の科名でもこのようにソースコードの特徴以外の基準で命名されることがあるため，マルウェアの動作として命名した科名と，静的解析を行った結果としての類似度分布には，差異が認められる．

6.3.2.4　検体間の類似度と検体の関連性

表 6.14 では多くの組み合わせでは類似度が 0.1 以下であり，これらの検体は関連性がないと推測が可能である．科名が同じ Koobface.026 と Koobface.064，Palevo.186 と Palevo.241 の類似度は高くなった．Koobface.026 と Koobface.064 は共に Facebook を利用して広まるマルウェアであり，Palevo.186 と Palevo.241 は共にメディアの自動実行機能や P2P ファイル共有，インスタントメッセンジャーで広まるマルウェアである．また Mydoom.60 と Mytob.120 の類似度は 0.63 であった．図 6.11 より類似度が 0.2 以上となるのは稀であるので，これらの検体は何らかの関連性がある．著者が Mydoom や Mytob の亜種を解析したとき，制御フローや使われている API，関数の呼び出し関係の類似性から Mytob は Mydoom のソースコードを流用した可能性が高いと推測していた．特にメールの送信エンジンは共通しており，両者とも DNS に問い合わせて送信先の SMTP サーバに直接接続してメールを送信するが，

それが失敗したときにはレジストリから標準の SMTP サーバを取得してメールを送信する．この実験の結果により，解析者の推測に根拠が与えられた．同様に科名が異なっているが Scar.04 と Slenfbot.47 は共にメディアの自動実行機能で広まるマルウェアであり，先頭部分のコードが類似していた．

著者は準備した検体のすべてを解析してはいないが，Netsky.AA と Palevo.241 のように類似度が高くなっているならば，何らかの関連性があると推測が可能である．

6.3.2.5 階層型クラスタ分析

図 6.12 の樹形図では Koobface や Mytob などは 1 つまたは複数の枝の下にグループを形成した．これらの亜種は同一の起源をもつソースコードから作成されているため，同じ科名の検体間での類似度が高くなったことが原因である．一方，Autorun はグループを形成せず全体に散らばった．表 6.15 と表 6.16 では Koobface と Mytob，Areses，Looked のクラスタでは同一の科名で占められた．一方，Autorun や Palevo，Sdbot が多数を占めるクラスタでは他の科名の検体がクラスタに含まれた．これらは前述の命名の方法や基準のあいまいさが原因である．

6.4 まとめ

本論文で提案した方法でソースコードの特徴からマルウェアを分類することは可能であった．類似度からマルウェアの機能を推定し，マルウェア解析や対策に寄与すると考えられる．しかし必ずしも既存のマルウェアの分類と同じ分類結果ではなかった．これは Autorun のようにソースコードの特徴以外の理由で命名が行われている検体があることが原因である．

また本論文では共通する API 推移の割合を類似度として定義した．しかし既存のソースコードを元に新たなマルウェアが作られるならば，あるマルウェアの API 推移を別のマルウェアがどれだけ含んでいるかを見ることで，マルウェアの包含関係を数値化することが可能である．この場合，元のマルウェアに機能を追加したならば，新たなマルウェアは元のマルウェアの特徴をすべて持っていることになる．複数のマルウェアのソースコードから新たなマルウェアが作られたならば，それぞれの元のマルウェアに対して，本論文で定義した類似度よりも高い値となる．類似度をどのように定義するかも検討する必要がある．

本論文の提案は Microsoft Visual Basic で作成されたマルウェアを除く 32 ビット Windows で動作するマルウェアを対象としている．しかし制御フロー解析を行い API 呼び出し（あるいは何らかの API に相当する機能）に注目するという方法自体は異なる環境にも適用可能である．

本システムの実験には 10 時間以上の時間を要した．この中で時間がかかった処理は逆アセンブルと API 推移の抽出であった．特に API 推移の抽出は Perl のスクリプトで行っているため，この処理を C 言語で再実装し，ネイティブコンパイラで生成したバイナリプログラムに置き換えれば，所要時間は改善すると考えられる．また一部の検体において，解析に長い時間を要した原因を分析する必要がある．

第 7 章

結論

第 4 章では文書型マルウェアの中にあるシェルコードを特定する方法を提案した．提案する方法でシェルコードが作成する実行可能ファイルを得ることができた．これらの実行可能ファイルは，パックされているならば第 5 章で提案する方法でアンパックし，第 6 章で提案する方法で分類できる可能性がある．シェルコードを特定によりマルウェア感染の最初の過程の 1 つである文書型マルウェアからマルウェア本体への解析へと繋げることができ，マルウェア感染の全体像を知ることができる．

しかし第 4 章の提案では，合計 150 種類のマルウェア検体の中で 4.1.1 節の条件に一致する検体は 88 種類，さらにその中で実行可能ファイルが得られた検体は表 4.6 より 50 種類であった．その比率は約 33% であり，全体の一部にしか対応できていない．主に原因は 4.1.1 節の条件に合致しない検体があること，およびシェルコードの特定の精度上の問題の 2 点である．前者に対しては，特に ROP に対しては本論文の提案とは異なる新たな方法が必要になる．後者に対してはエミュレータの精度を向上させるなどの方法で成功率を高めることができると考えられる．

さらに本論文では対象としなかったファイル形式への応用も考えられる．Microsoft Office 2007 以降の Office Open XML 形式や圧縮またはエンコードされたファイルにも対応できる可能性はある．一方，PDF や Flash などで JavaScript によりシェルコードがメモリに生成される文書型マルウェア，および JavaScript そのものに対しては本論文の提案とは異なる方法を用いる必要がある．

第 5 章ではパックされたマルウェアをアンパックする方法を提案した．実際のマルウェア検体を用いた実験と，既知のプログラムをパックした実行可能ファイルを用いた実験を行った．

準備したマルウェア検体のうちアンパックの対象となった検体は 3,189 種類であり，実際のマルウェア検体では確実にアンパックが成功したと判断した検体は 595 種類であった．約 19% 程度しか成功していないが実際のマルウェア検体では元のイメージが不明なためアンパックの失敗を断言することは難しい．そのため成功したと判断できなかった残りの 2,594 種類のすべてが失敗したとはいえない．

一方，既知のプログラムを用いた実験では 107 種類中 33 種類（約 31%）でアンパックが成功した．エントリーポイントの特定は，書き換えられたメモリの実行を検知する方法と既知のエントリーポイントのパターンと照合する 2 つの方法を実装したが，この実験ではエントリーポイントのパターンとの照合は行っていない．実際にはエントリーポイントのパターンとの照合でエントリーポイントを特定できるパッカーの数が増えることが期待できる．元のコードの復号だけが成功したパッカーも含めると 107 種類中 63 種類（約 59%）に上昇する．

マルウェアを解析する場合には．完全なアンパックの成功が求められない場合もある．事実，第 6 章の実験では復号されたバイナリイメージから API 推移が抽出できれば十分であるので，アンパックの成否が不明となった検体も実験に用いている．

第 6 章ではパックされたマルウェアを分類する方法を提案した．マルウェアの分類は特徴の抽出と抽出した特徴から類似度を求める過程に分かれる．本論文ではマルウェアのバイナリイメージの違いではなく，元になったソースコードの違いによって分類するために，マルウェアの特徴として API 推移を抽出した．

マルウェアのソースコードを異なるコンパイラ・最適化オプションで作成した検体を用いた実験と実際のマルウェア検体を用いた実験を行った．ソースコードの違いは作成した検体の類似度に反映され，同一のソースコードから作成された検体は類似度が高くなることが確認された．

実際のマルウェア検体を用いた実験では 4,684 種類中 1,821 種類（約 39%）の検体で API 推移の抽出に成功し分類ができた．API 推移の抽出に失敗した原因は主の 1 つは第 5 章のアンパッカーの精度にある．アンパッカーの精度が向上すれば，分類の精度も向上することになる．

もう 1 つの原因は実行時に API のアドレスが取得されることである．この場合，本論文が提案する方法では解決できないので，別途動的解析を用いるか API 推移以外の特徴を用いる必要がある．

本論文はマルウェアの感染の最初の過程の 1 つである文書型マルウェアのシェルコード特定から，実行可能ファイルのアンパック，特徴抽出と分類というマルウェアを静的解析するための方法を提案した．その目的を達することはできたが，本論文のシステムを構築する上で，対象外としたファイル形式や実行可能ファイルも多くあり，また必ずしもすべての検体で成功する訳ではない．精度の向上と対象となるファイル形式や実行可能ファイルを増やすことは今後の課題である．

参考文献

[1] Michalis Polychronakis, Kostas G. Anagnostakis, Evangelos P. Markatos : Network-Level Polymorphic Shellcode Detection Using Emulation ; *Journal in Computer Virology*, Vol.2, No.4, pp.257–274, (2007).

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

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

[4] Wei-Jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, Angelos D. Keromytis : A Study of Malcode-Bearing Documents ; *Detection of Intrusions and Malware, and Vulnerability Assessment*, Volume 4579 of *Lecture Notes in Computer Science*, Pages 231–250, (2007).

[5] 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, (May 2014).

[6] Marco Cova, Christopher Kruegel, Giovanni Vigna : Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code ; *Proceedings of the World Wide Web Conference (WWW)*, Raleigh, NC, (April 2010).

[7] Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna : Shellzer: a tool for the dynamic analysis of malicious shellcode ; *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, S. Francisco, CA, (2011).

[8] Daisuke Inoue, Katsunari Yoshioka, Masashi Eto, Yuji Hoshizawa and Koji Nakao : Automated Malware Analysis System and Its Sandbox for Revealing Malware's Internal and External Activities ; *IEICE transactions on information and systems*, Vol.92, No.5 pp.945–954, (May 2009).

[9] Daisuke Inoue, Katsunari Yoshioka, Masashi Eto, Yuji Hoshizawa and Koji Nakao : Malware Behavior Analysis in Isolated Miniature Network for Revealing Malware's Network Activity ; *Communications, 2008. ICC '08. IEEE International Conference*, pp.1715–1721, (May 2008).

[10] Katsunari Yoshioka, Daisuke Inoue, Masashi Eto, Yuji Hoshizawa, Hiroki Nogawa, Koji Nakao : Malware Sandbox Analysis for Secure Observation of Vulnerability Exploitation ; *IEICE transactions on information and systems*, Vol.92, No.5 pp.955–966, (May 2009).

[11] Frank Boldewin : Analyzing MSOffice malware with OfficeMalScanner ; http://www.reconstructer.org/papers/Analyzing%20MSOffice%20malware%20with%20OfficeMalScanner.zip

[12] 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, (March 2013).

[13] Sébastien Josse : Secure and advanced unpacking using computer emulation ; *AVAR 2006 Conference*,

pp.174–190, Auckland, New Zealand, (2006).

[14] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee : PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware ; *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp.289–300, (December 2006).

[15] Min Gyung Kang, Pongsin Poosankam, Heng Yin : Renovo: A Hidden Code Extractor for Packed Executables ; *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, pp.46–53, New York, NY, USA, (2007).

[16] Lorenzo Martignoni, Mihai Christodorescu, Somesh Jha : OmniUnpack: Fast, Generic, and Safe Unpacking of Malware ; *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp.431–441, (December 2007).

[17] Cristian Lungu, Marius Botis : CJ-Unpack: Efficient Runtime Unpacking System ; *19th EICAR Annual Conference*, pp.235–253, Paris, France, (2010).

[18] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, Wenke Lee : Eureka: A Framework for Enabling Static Malware Analysis ; *Computer Security - ESORICS 2008*, Volume 5283 of *Lecture Notes in Computer Science*, pp.481–500, Berlin Heidelberg, (2008).

[19] 織井達憲, 吉岡克成, 四方順司, 松本勉, 金亨燦, 井上大介, 中尾康二 : パッキングされたマルウェアの類似度算出手法とその評価 ; 電子情報通信学会技術研究報告. ICSS, 情報通信システムセキュリティ, Vol.109, No.285, pp.7–12, (November 2009).

[20] 岩村誠, 伊藤光恭, 村岡洋一 : コンパイラ出力コードモデルの尤度に基づくアンパッキング手法 ; *MWS2008*, pp.103–108, (October 2008).

[21] 岩村誠, 伊藤光恭, 村岡洋一 : マルウェアのエントリポイント検出後におけるコード領域識別手法 ; 電子情報通信学会技術研究報告. ICSS, 情報通信システムセキュリティ, Vol.110, No.79, pp.19–24, (June 2010).

[22] Hyung Chan Kim, Daisuke Inoue, Masashi Eto, Yaichiro Takagi, Koji Nakao : Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation ; *Joint Workshop on Information Security 2009*, Kaohsiung, Taiwan, (2009).

[23] 川古谷裕平, 岩村誠, 伊藤光恭 : OEP 自動検出によるマルウェアアンパック手法 ; 電子情報通信学会技術研究報告. ICSS, 情報通信システムセキュリティ, Vol.110, No.79, pp.13–18, (June 2010).

[24] 塩治榮太朗, 川古谷裕平, 岩村誠, 伊藤光恭 : メモリアクセス値のエントロピーに基づく自動アンパッキング ; 電子情報通信学会技術研究報告. ICSS, 情報通信システムセキュリティ, Vol.110, No.475, pp.41–46, (March 2011).

[25] Michael Bailey, Jon Oberheide, Jon Andersen, Z.Morley Mao, Farnam Jahanian, Jose Nazario : Automated Classification and Analysis of Internet Malware ; *Recent Advances in Intrusion Detection*, Volume 4637 of *Lecture Notes in Computer Science*, pp.178–197, Berlin Heidelberg, (2007).

[26] Mihai Christodorescu, Somesh Jha, Christopher Kruegel : Mining Specifications of Malicious Behavior ; *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pp.5–14, New York, NY, USA, (2007).

[27] Halvar Flake : Automated Unpacking and Malware Classification ; *Black Hat Japan*, pp.61–88, Tokyo, Japan, (2007).

[28] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, Giovanni Vigna : Polymorphic Worm Detection Using Structural Information of Executables ; *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pp.207–226, Berlin, Heidelberg, (2006).

[29] Qinghua Zhang, Douglas S. Reeves : MetaAware: Identifying Metamorphic Malware ; *Computer Security*

Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, pp.411–420, (December 2007).

[30] Kyoung-Soo Han, In-Kyoung Kim, EulGyu Im : Detection Methods for Malware Variant Using API Call Related Graphs ; *Proceedings of the International Conference on IT Convergence and Security 2011*, Volume 120 of *Lecture Notes in Electrical Engineering*, pp.607–611, Netherlands, (2012).

[31] Kyoung-Soo Han, In-Kyoung Kim, EulGyu Im : Malware Classification Methods Using API Sequence Characteristics ; *Proceedings of the International Conference on IT Convergence and Security 2011*, Volume 120 of *Lecture Notes in Electrical Engineering*, pp.613–626, Netherlands, (2012).

[32] Altyeb Altaher, Supriyanto, Ammar ALmomani, Mohammed Anbarm, Sureswaran Ramadass : Malware detection based on evolving clustering method for classification ; *Scientific Research and Essays*, Vol.7, No.22, pp.2031–2036, (2012).

[33] 岩村誠, 伊藤光恭, 村岡洋一 : 機械語命令列の類似性に基づく自動マルウェア分類システム ; 情報処理学会論文誌, Vol.51, No.9, pp.1622–1632, (September 2010).

[34] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye : IMDS: Intelligent Malware Detection System ; *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pp.1043–1047, New York, NY, USA, (2007).

[35] Yanfang Ye, Yinming Mei, Rencheng Peng : MCNS: Intelligent Malware Categorization and Naming System ; *AVAR 2009 Conference*, pp.15–25, Kyoto, Japan, (2009).

[36] 独立行政法人 情報処理推進機構 : 標的型攻撃メールの傾向と事例分析 ＜2013 年＞ ; http://www.ipa.go.jp/files/000036584.pdf

[37] Mandiant : Mandiant APT1: Exposing One of China's Cyber Espionage Units ; http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf

[38] Marco Prandini, Marco Ramilli : Return-Oriented Programming ; *Security Privacy, IEEE*, Vol.10, No.6, pp.84–87, (November 2012).

[39] Microsoft : Vulnerabilities in Microsoft Office Could Allow Remote Code Execution (2587634) ; https://technet.microsoft.com/library/security/ms11-073

[40] Daniel Rentz : The Microsoft Compound Document File Format ; http://www.openoffice.org/sc/compdocfileformat.pdf

[41] Microsoft : Compound File Binary File Format ; http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMS-CFB%5D.pdf

[42] Microsoft : Word 2007: Rich Text Format (RTF) Specification, version 1.9.1 ; http://www.microsoft.com/en-us/download/details.aspx?id=10725

[43] Benjamin Schwarz, Saumya Debray, Gregory Andrews : Disassembly of Executable Code Revisited ; *Reverse Engineering, 2002. Proceedings. Ninth Working Conference*, pp.45–54, (2002).

研究業績

審査付発表論文

（レフェリー制のある学術雑誌）

[1] 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).

[2] 岩本一樹, 和﨑克己：静的解析により抽出された API 推移に基づくマルウェアの分類；情報処理学会論文誌, Vol.54, No.3, pp.1199-1210, (2013).

審査付発表論文

（レフェリー制のある国際会議発表論文）

[1] Kazuki Iwamoto, Katsumi Wasaki : A Method for Shellcode Extraction from Malicious Document Files using Entropy and Emulation ; *Proceedings of the 4th International Conference on Security Science and Technology (ICSST2015), (ST005)*, 7pages, (2015).

[2] Kazuki Iwamoto, Katsumi Wasaki : Malware Classification based on Extracted API Sequences using Static Analysis ; *Proceedings of the 8th Asian Internet Engineering Conference (AINTEC2012), ACM Conference Proceedings*, pp.31-38, (2012).

審査なし発表論文

（レフェリー制のない学術雑誌，プロシーディング，総説，解説，口頭発表等）

[1] 岩本一樹, 神薗雅紀, 津田侑, 遠峰隆史, 井上大介, 中尾康二：電子文書型マルウェアからシェルコードを抽出する方法の提案；情報処理学会研究報告（コンピュータセキュリティ）, 2014-CSEC-65, No.13, pp.1-6, (2014).

[2] 岩本一樹, 和﨑克己：マルウェアアンパッキングにおけるランタイムライブラリのコード比較によるオリジナルエントリーポイント検出；電子情報通信学会技術研究報告（情報通信システムセキュリティ）, ICSS2011-10, pp.57-62, (2011).

[3] 岩本一樹, 和﨑克己：静的解析によるマルウェアの分類と結果の検討；マルチメディア・分散・協調とモバイル (DICOMO2010) シンポジウム論文集, 情報処理学会, pp.477-491, (2010).

[4] Kazuki Iwamoto : Feature Extraction, Classification and Learning for Malware Codes ; *12th Association of anti-Virus Asia Researchers International Conference (AVAR2009)*, 14 pages, CD-ROM publishing, (2009).

[5] 岩本一樹, 和﨑克己：静的解析によるマルウェアの API 推移の抽出とクラスタ解析；コンピュータセキュリティ

シンポジウム 2009 (CSS2009) 論文集, 情報処理学会, pp.361-366, (2009).

[6] 岩本一樹, 和﨑克己 : コールグラフと制御フロー解析によるマルウェアの静的解析と分類 ; マルチメディア・分散・協調とモバイル (DICOMO2008) シンポジウム論文集, 情報処理学会, pp.1-14, (2008).

[7] 岩本一樹, 和﨑克己 : コンピュータウイルスのコード静的解析による特徴抽出と分類について ; 電子情報通信学会技術研究報告（情報セキュリティ）, ISEC2007-127, pp.107-113, (2007).

謝辞

本論文で提案するシステムを作成するにあたり，信州大学学術研究院（工学系） 和﨑克己 教授には，多大なる御指導・御助言を賜りました．ここに深く感謝の意を表し，御礼申し上げます．

本論文をご精読頂き有用な御意見を頂きました信州大学学術研究院（工学系） 師玉康成 教授，山本博章 教授，丸山稔 教授，静岡大学大学院情報学研究科 西垣正勝 教授に深謝致します．

株式会社セキュアブレイン プリンシパル ソフトウェア エンジニア 遠藤 基 氏には前職より本研究に御協力を賜りました．ここに深く感謝致します．また，同社の方々には本研究のために多くの御意見・御支援を頂きましたことを，心より御礼申し上げます．

本論文の第 4 章は 2013 年度に情報通信研究機構から委託を受け実施した「環境に依存しない電子文書型マルウェア解析システムの開発」の成果の一部です．ご協力頂いた皆様に，謹んで感謝の意を表します．

付録 A

ZeuS の分類

実際のマルウェアの分類事例として，ZeuS の亜種を扱う．オリジナルの ZeuS はソースコードが流出しており，流出したソースコードを元にして様々な亜種が作成されている．本論文では著者によって解析済みの 17 種類の ZeuS の亜種を準備した．検体は 0 から 16 の通し番号で区別する．検体の別名は表 A.1 の Alias に示される．表 A.1 の Alias が空白の検体はオリジナルの ZeuS とほぼ同じか，またはオリジナルの ZeuS とは異なるが特に別名が定められていない検体である．

ZeuS を制御フロー解析した結果のグラフのノード数は 6,000 を超えるため，すべてを表示することはできない．たとえば 0 番の検体では制御フロー解析の結果ではエッジは 11,494，ノードは 6,801 あり，API 推移の結果ではエッジは 11,155，ノードは 328 ある．ゆえに，エントリーポイント付近に絞って図示する．ZeuS の亜種の 1 つのエントリーポイント付近の制御フロー解析の結果は図 A.1 になる．図 A.1 から API 推移を求めた結果は図 A.2 になる．

17 種類の ZeuS の亜種間の類似度を表 A.1 に示す．また，図 A.3 は表 A.1 を階層型クラスタ分析による可視化を行った結果である．

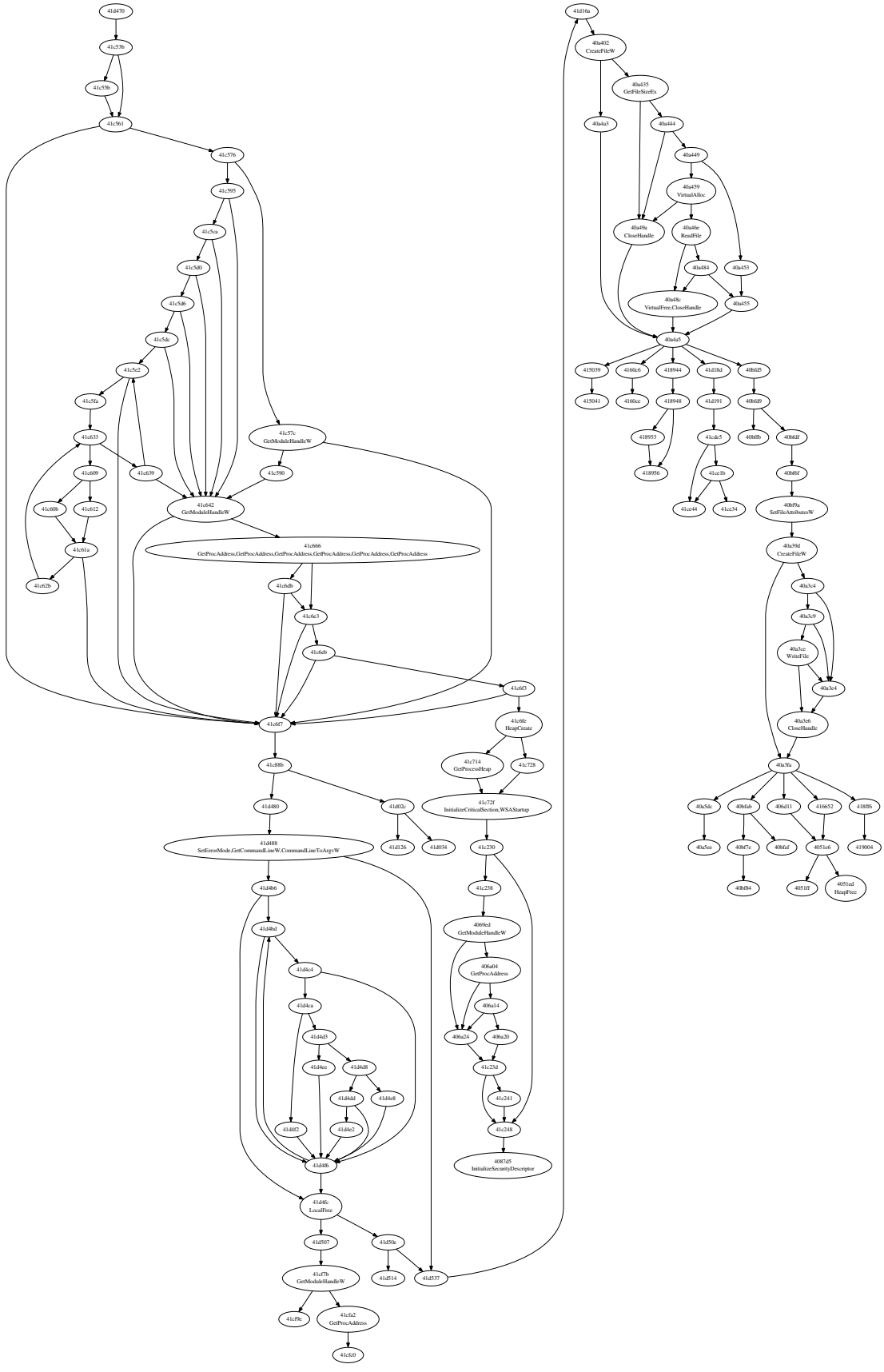

図 A.1　ZeuS の制御フロー解析

図 A.2 ZeuS の API 推移

表 A.1　ZeuS の各亜種間の類似度

	Alias	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0		0.90	0.28	0.72	0.91	0.30	0.72	0.77	0.93	0.31	0.31	0.30	0.31	0.30	0.77	0.79	0.83
1	Ice9		0.29	0.73	0.90	0.31	0.73	0.81	0.87	0.32	0.32	0.32	0.32	0.31	0.81	0.79	0.82
2	P2P ZeuS			0.29	0.28	0.60	0.29	0.30	0.28	0.56	0.56	0.58	0.56	0.59	0.30	0.28	0.30
3	Citadel				0.75	0.32	1.00	0.87	0.74	0.34	0.33	0.33	0.33	0.33	0.87	0.69	0.71
4	Ice9					0.31	0.75	0.77	0.96	0.32	0.32	0.31	0.32	0.31	0.77	0.79	0.82
5	P2P ZeuS						0.32	0.33	0.30	0.84	0.81	0.82	0.81	0.78	0.33	0.33	0.33
6	Citadel							0.87	0.74	0.34	0.33	0.33	0.33	0.33	0.87	0.69	0.71
7	Citadel								0.76	0.34	0.33	0.33	0.33	0.33	1.00	0.72	0.73
8										0.32	0.32	0.32	0.32	0.31	0.76	0.77	0.81
9	P2P ZeuS										0.84	0.83	0.84	0.79	0.34	0.32	0.32
10	P2P ZeuS											0.80	1.00	0.83	0.33	0.32	0.32
11	P2P ZeuS												0.80	0.81	0.33	0.32	0.32
12	P2P ZeuS													0.83	0.33	0.32	0.32
13	P2P ZeuS														0.33	0.31	0.32
14	Citadel															0.72	0.73
15	ZeusVM																0.76
16																	

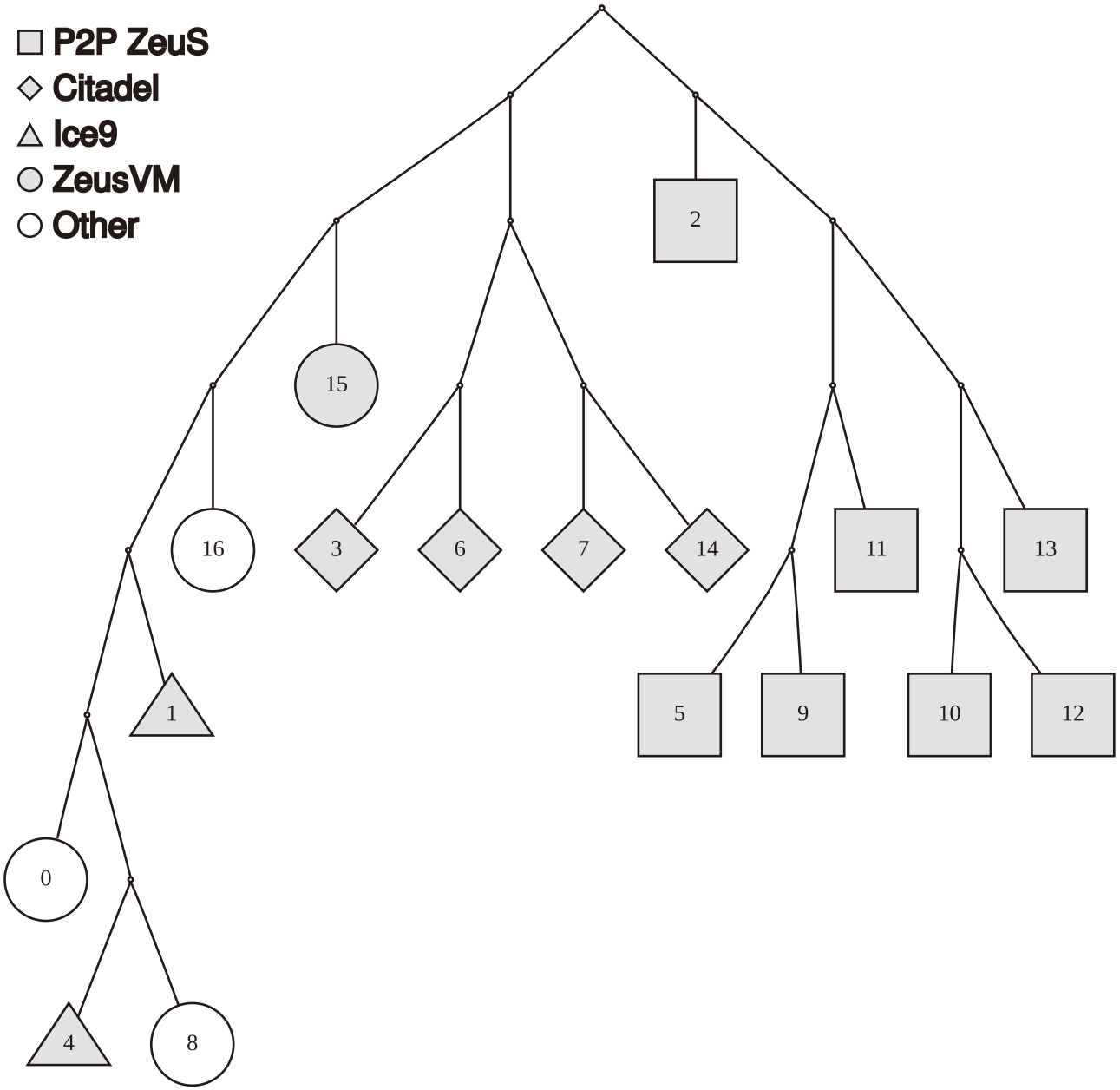

図 A.3　ZeuS の各亜種間の類似度の可視化

付録 B

プログラム解説

B.1 apiseq.pl

書式

apiseq.pl [infile] [outfile]

説明

ctlflw.pl の制御フロー解析の結果を標準入力またはファイルから受け取り Graphviz の dot 形式で API 推移を標準出力またはファイルに出力する．

例

apiseq.pl ctlflw_sample.dot apiseq_sample.dot

ノードのラベルは API の名前になる．エッジの数値はその API 推移が発生する回数になる．

B.2 ascomp

書式

ascomp [option] dotfile...

説明

apiseq.pl の出力結果から検体間の類似度を求めて標準出力またはファイルに出力する．

- -o, --outfile=FILE 出力ファイル

例

ascomp -o ascomp.txt apiseq_sample1.dot apiseq_sample2.dot

1 行に 1 つの距離が出力される．各行はカンマ（,）で 3 つに区切られ，最初と次の要素は入力した dot ファイルの名前，最後の要素は距離になる．距離は 0 から 10,000 の値になり，値が小さいほど類似している（0 ならば全く同じ，10,000 ならば完全に異なる）．距離は本論文の類似度とは値が異なるが意味は同じである．距離を d，類似度を s とすると $s = (10000 - d) \div 10000$ となる．

B.3 ctlflw.pl

書式

ctlflw.pl [infile] [outfile]

説明

disw32 の逆アセンブル結果を標準入力またはファイルから受け取り Graphviz の dot 形式で制御フロー解析の結果を標準出力またはファイルに出力する．

例

ctlflw.pl sample.asm ctlflw_sample.dot

実行にはカレントディレクトリに allow.txt が必要である．allow.txt は 1 行に 1 つの API が列挙されており，ctlflw.pl は allow.txt に含まれている API だけを対象とする．ただし GetLastError と Sleep，SleepEx は allow.txt に含まれていても無効である．

カレントディレクトリに forward.txt があるならば，ctlflw.pl は forward.txt を読み込む．forward.txt は 1 行に 2 つの API が列挙されており，2 つの API は空白で区切られる．forward.txt にある API は各行の最初の API は 2 番め目の API と見なされる．たとえば「lstrlenA lstrlen」という記述があるならば，lstrlenA は lstrlen として扱われる．

出力結果では，関数単位でサブグラフが作られる．ノードのラベルの 1 行目はアドレスになる．ノードが API を呼び出すときにはラベルの 2 行名は API の名前になる．複数の API を呼び出すときにはカンマ（,）で区切られる．

B.4　dendro

書式

dendro [option...] dotfile...

説明

apiseq.pl の出力結果から検体間の類似度を求めて階層型クラスタ解析を行う．

- -d, --directory=DIR 出力ディレクトリ
- -o, --outfile=FILE 出力ファイル

例

dendro -o dendro.txt -d out apiseq_sample1.dot apiseq_sample2.dot apiseq_sample3.dot

検体間の類似度を求めて階層型クラスタ解析を行う．その結果はクラスタを作成するごとに Graphviz の dot ファイルとしてディレクトリの中に保存される．オプションでディレクトリを指定しなければカレントディレクトリの「dendrogram」に出力する．これらのファイルの名前はクラスタ数を表す数値になる．（たとえば，10 の dot ファイルを入力したときには，9 の dot ファイルが出力される）

オプションで出力ファイルを指定したときには，ファイルに各クラスタ数で分割したときのベイズ情報量規準を出力する．

B.5　disw32

書式

disw32 [option...] exefile [asmfile]

説明

32 ビット Windows 実行可能プログラムを逆アセンブルして標準出力またはファイルに出力する．

- --analyse-linear 線形解析を行う
- --analyse-referer 参照されているサブルーチンを解析する
- --analyse-data データ領域を解析する
- -a, --analyse-all 線形解析と参照されているサブルーチンを解析する
- --display-export エクスポートテーブルを表示する

- --display-import インポートテーブルを表示する
- --display-resource リソース領域を表示する
- --display-reloc リロケーションテーブルを表示する
- -d, --display-all すべて表示する

例

disw32 sample.exe sample.asm

標準ではエントリーポイントおよびエクスポートされているサブルーチンをエクスポートテーブルの順番で強制逆アセンブルをする．強制逆アセンブルでは，たとえそのサブルーチンで逆アセンブルが正常に終了しなくても逆アセンブルを行う．次にここでサブルーチンと思われるコードの逆アセンブルを試みる．相対アドレス指定は分岐命令しかありえないので，相対アドレス指定の場合には強制逆アセンブルをする．即値アドレス指定の場合には (正常終了した場合のみ) 逆アセンブルをする．オフセット値指定の場合にはそのアドレスがまだ未解決のときに，そのアドレスにある値が示すアドレスに対してサブルーチンの call ならば強制逆アセンブル，それ以外ならば逆アセンブルをする．32 ビットの配列指定の場合にはそのアドレスにある値が示すアドレスに対して逆アセンブルが可能な限り繰り返す．逆アセンブルの優先順位はこの順番であり，同じならばアドレスが小さい方を優先する．強制逆アセンブルを除いて，逆アセンブルができなかった場合にはデータ参照として扱われる．

データ参照では即値アドレス指定の場合には，そのアドレスが文字列として有効かどうか調べる．オフセット値指定の場合にはそのアドレスに 8 から 32 ビットのデータがあると仮定する．また 32 ビットのときにはその値が示すアドレスが文字列として有効かどうか調べる．32 ビットの配列指定の場合には，そのアドレスにある値が示すアドレスが文字列として有効かどうか調べる．これを文字列が有効な限り繰り返す．

--analyse-linear が有効ならば，アドレスの下位から順番に逆アセンブルを試みる．逆アセンブルが可能ならば，そのアドレスをエントリーポイントやエクスポートと同様に扱う．

--analyse-referer が有効ならば，アドレスの下位から順番にその 32 ビットの値が示すアドレスの逆アセンブルを試みる．逆アセンブルが可能ならば，そのアドレスを 32 ビットのデータとし，値が示すアドレスをエントリーポイントやエクスポートと同様に扱う．

--analyse-data が有効ならば，未定義の領域をデータとして扱う．

B.6 findep

書式

findep command [option...] [file or string...]

説明

バイナリからエントリーポイントを探す．次のコマンドを指定する．

- c, check エントリーポイントが登録されているか確認する
- d, display 登録されているエントリーポイントを表示する
- e, entry エントリーポイントを登録する
- f, find エントリーポイントを探す
- i, ignore 無視するエントリーポイントを登録する
- r, remove 登録されているエントリーポイントを削除する

次のオプションを指定できる．

- -a, --align=BYTE セクションのアライメントのバイト数
- -b, --base=ADDRESS イメージのベースアドレス

- -d, --dump ファイルはメモリダンプ
- -r, --raw ファイルはバイナリ
- --dynamic 動的解析を行う
- --static 静的解析を行う

例

findep c sample.exe
findep d
findep e sample.exe
findep f sample.exe
findep i sample.exe
findep r call,jmp,mov
findep r n0001

バイナリの解析方法は静的解析と動的解析がある．静的解析では逆アセンブルを行ってニーモニックを得る．動的解析では仮想環境で実行してニーモニックを得る．デフォルトは静的解析である．静的解析と動的解析は同一のデータベースに登録される．

entry コマンドでは引数で与えられた複数の実行可能ファイルのエントリーポイントのニーモニックを登録する．最低でも 8 ステップのニーモニックが必要であり，最大で 24 ステップのニーモニックを登録する．

ignore コマンドでは引数で与えられた複数の実行可能ファイルのエントリーポイントのニーモニックを無視するために登録する．登録されたニーモニックは entry コマンドで登録できない．

display コマンドは登録されているエントリーポイントのニーモニックを表示する．

remove コマンドは登録されているニーモニックを削除する．引数で先頭から一致した場合には，該当するニーモニックをすべて削除する．

check コマンドでは引数で与えられた複数の実行可能ファイルのエントリーポイントのニーモニックが登録されているか確認する．

find コマンドでは引数で与えられたファイルの中に登録されているエントリーポイントがあるか確認する．ファイルは実行可能ファイルに限らない．また引数で 2 つ目のファイルを指定したときには，見つけたエントリーポイントに変更した実行可能ファイルを書き出す．入力ファイルが実行可能ファイルではないときにはヘッダが先頭に付加される．--dump および--raw オプションは入力ファイルが実行可能ファイルか任意のバイナリかを明示的に指定する．どちらのオプションもないときには自動的に判定する．デフォルトのセクションアライメントは 1000h，ベースアドレスは 400000h である．

find コマンドはホームディレクトリの.maid.org/iwmutlis/findep.ini にデータを保存する．このファイルは Windows では
C:\Users\<ユーザ名>\AppData\Local\maid.org\iwmutlis\findep.ini
になる．

B.7　nictract

書式

nictract [option...] infile [outfile...]

説明

CFB 形式または RTF の文書ファイル，バイナリファイルからシェルコードを探す．

- -n, --number=N 探すシェルコードの数（default:1）
- -t, --time=SECOND 最大秒数（default:無制限）
- -c, --count=N 最大エミュレーション回数（default:無制限）
- -r, --range=START[-END] 有効ファイルオフセット
- -i, --ignore=START[-END] 無視ファイルオフセット
- -q, --quiet メッセージを減らす
- -v, --verbose メッセージを増やす
- --step-first=N 書き換えメモリの実行と PEB アクセスを検出するまでのステップ数（default:16384）
- --step-second=N API 呼び出しを検出するまでのステップ数（default:4194304）
- --base=ADDRESS イメージのベースアドレス （default:400000h）
- --ole-structure CFB のストリーム領域だけを対象とする（default）
- --ole-format CFB のファイルの末尾に付加されたデータを無視する
- --ole-header CFB のヘッダを確認する
- --ole-ignore ファイル全体からシェルコードを探す
- --ole-allow-header CFB をパースできないときにはヘッダの確認だけを行う
- --ole-allow-ignore ヘッダがないときにはファイル全体からシェルコードを探す
- --rtf-binary RTF の復号したバイナリからシェルコードを探す（default）
- --rtf-full RTF の復号したバイナリと元のファイルからシェルコードを探す
- --rtf-header RTF のヘッダを確認する
- --rtf-ignore ファイル全体からシェルコードを探す
- --rtf-allow-header RTF をパースできないときにはヘッダの確認だけを行う
- --rtf-allow-ignore ヘッダがないときにはファイル全体からシェルコードを探す
- --entropy-simple エントロピーが高いバイト列から順番にシェルコードを探す
- --entropy-size=N エントロピーを求めるバイト列のサイズ（default:384）
- --entropy-step=N エントロピーを求めるバイト列の間隔（default:16）
- --enable-all すべて有効（default）
- --disable-all すべて無効
- --enable-entropy エントロピーを用いて探す（default）
- --disable-entropy ファイルの先頭から探す
- --enable-disasm 逆アセンブル有効（default）
- --disable-disasm 逆アセンブル無効
- --enable-emulation エミュレータ有効（default）
- --disable-emulation エミュレータ無効
- --map-ole CFB の構造を表示
- --map-rtf RTF の構造を表示
- --map-entropy ファイルのエントロピーを表示
- --map-info ファイルの情報を表示

例

nictract sample.doc sample.exe

--number オプションでは見つけるシェルコードの数を指定する．何も指定しないときには outfile の数または 1 になる．--quiet と--verbose は複数指定できる．

--range または--ignore では有効または無効なファイルの範囲を 16 進数で指定する．--range で有効な範囲であったも，--ignore オプションや CFB および RTF の解析により無効となる場合がある．たとえば「–range=1000」や「–range=1000-2000」とする．RTF のときにはバイナリの先頭オフセット値を指定する．「+」記号で区切ることで，その中のオフセット値で範囲を指定できる．たとえば「–range=1000+」や「–range=1000+2000」，「–range=1000+2000-3000」とする．

--step-first，--step-second ではシェルコードの特徴を観測するまでのステップ数を指定する．

--ole-*および--rtf-*の各オプションはシェルコードを探すファイルの領域を指定する．またヘッダの確認だけを行い，ファイル全体を対象とすることもできる．--ole-ignore と--rtf-ignore は同じ意味であり，ファイルの構造を無視する．--ole-allow-*および--rtf-allow-*はファイルの検査に失敗したときには，--*-header または--ole-ignore と同様の動作になる．

B.8　setup.pl

書式

setup.pl dir

説明

disw32，ctlflw.pl，apiseq.pl，ascomp を起動して検体間の類似度を求める．

例

setup.pl dir

setup.pl の実行には Graphviz（neato）と disw32 が実行可能でなければならない．setup.pl と同じディレクトリに ctlflw.pl，apiseq.pl，ascomp がなければ実行できない．また ctlflw.pl が必要とする allow.txt も必要である．カレントディレクトリに forward.txt があるならば，ctlflw.pl は forward.txt を読み込む．dir で指定されたディレクトリには sample という名前のサブディレクトリが必要である．このディレクトリには disw32 で逆アセンブルされる実行可能ファイルを格納する．

setup.pl を実行すると，dir で指定されたディレクトリに disw32，ctlflw，apiseq という名前のサブディレクトリが作られる．これらのディレクトリには逆アセンブル（disw32），制御フロー解析（ctlflw.pl），API 推移（apiseq.pl）の結果が格納される．ファイル名には元の拡張子に加えて，新たに拡張子（.asm または.dot）が加えられる．

apiseq の内容を元に，検体間の類似度のファイル ascomp.txt がディレクトリ内に作られる．また ascomp.txt から similarity.html，similarity.dot が作られ，Graphviz（neato）により similarity.dot から similarity.png が作られる．similarity.html は検体間の類似度の表であり，similarity.png は検体間の類似度を多次元尺度構成法により可視化した結果である．

サブディレクトリ sample に新たにファイルを加えて setup.pl を実行したときには，必要なファイルだけが作成，更新される．サブディレクトリ sample のファイルを更新したときには，ファイルのタイムスタンプにより，古いファイルだけが更新される．

B.9　unpk

書式

unpk [option...] exefile [argument...]

説明

暗号化された実行可能プログラムを復号する．

- -b, --break=ADDRESS ブレイクポイント
- -d, --dynamic 動的に確保したメモリを出力ファイルに含める
- -e, --entry ファイルの出力には新しいエントリーポイントが必要
- -i, --import インポートテーブル再構築
- -l, --logfile=FILE ログをファイルに出力する
- -o, --outfile=FILE 出力ファイル
- -s, --step=STEP 最大ステップ数
- -t, --timer=SECOND 最大秒数
- -m, --trace-mnemonic ニーモニックをトレースする
- -r, --trace-register レジスタをトレースする
- -f, --trace-tail=STEP 最後から指定されただけトレースする
- -v, --verbose 詳細出力

例

unpk -o unpacked.exe packed.exe

仮想環境でプログラムを実行することで，Unpack されたイメージの取得を行う．実装されていない命令や API の呼び出しなど，継続して実行することができなくなったときには，その時点で終了する．仮想環境のメモリにはレベルという概念があり，プログラムによって書き換えられたメモリのレベルは，そのプログラムコードがあるメモリのレベルよりも 1 つ大きくなる． またレベルが高いメモリにプログラムの制御が移ったときには，そのアドレスをオリジナルのエントリーポイントの候補とする．レベルの初期値は 0 である．

--dynamic オプションではプログラムが終了するときに確保されているメモリを出力ファイルに含める．

--import オプションが指定されると，プログラムが終了するとき状態でインポートテーブルを再構築して出力ファイルに加える．

--break オプションではプログラムを終了させるアドレスを 16 進数で指定する．「-b 401000」では 401000 を実行するときに終了する．

また「-b 401000,402000-403000」というようにカンマで区切り複数指定することや，一定の範囲の指定も可能である．

--entry オプションが指定されるときには，新たなエントリーポイントを見つけることができなかったときにはファイルに出力しない．