効率的な解析を目的とした
自動マルウェア分類に関する研究

Automatic malware classification for efficient analysis

2012 年 2 月

岩村　誠

効率的な解析を目的とした
自動マルウェア分類に関する研究

Automatic malware classification for efficient analysis

2012 年 2 月

早稲田大学大学院基幹理工学研究科

情報理工学専攻 情報構造研究

岩村 誠

目次

図目次

表目次

第 1 章

序論

本章では，本研究の目的を示し，続いて本研究を取り巻く背景とその位置付けについて概説する．最後に本論文の構成について説明する．

1.1　本研究の目的

本研究は，多数のマルウェアを効率的に解析する仕組みを構築することを目的とする．これによりマルウェアが備える脅威の全容解明を可能にし，駆除ツールの作成やネットワークでの攻撃遮断といった対処を促進することを目指す．

1.2　本研究の背景

かつてのコンピュータウィルスは，自己顕示欲や好奇心を満たすために開発されてきた．これらは現在，悪意あるソフトウェア（Malicious Software）を意味するマルウェアと呼ばれ，情報漏えいやスパムメール配信，DDoS（Distributed Denial of Service）攻撃等，多くのセキュリティ侵害の基盤ツールとして暗躍するに至り，社会問題化している [1].

一方，マルウェアの種類数は増加の一途を辿り，流行しているマルウェアやその脅威の把握が困難になっている [2–4]．こうしたマルウェアの種類数増加の原因は，その用途の多様化に加え，次のようなマルウェアの開発プロセスにあると考えられている [5].

昨今の多くのマルウェアは，一般的なソフトウェアと同様，バグ改修や機能改良といった工程を円滑に実施するために，C や C++ といった高級言語で開発されており，これが，亜種の効率的な開発につながっている．一方でマルウェアの作者は，アンチウィルスソフ

トウェアによる検出から逃れるために，マルウェアの特徴的なパターンを消そうとする．こうしたプログラムコードの多様化には，一般的にメタモーフィックエンジンやポリモーフィックエンジンと呼ばれるツールが用いられる [6]．メタモーフィックエンジンは，機械語命令におけるレジスタの役割を変更したり，同じ意味を持つ違う命令に変異させるため，対象となるプログラムの機械語命令を事前に解釈しておく必要がある．このため，メタモーフィックエンジンはアセンブリ言語で記述されたプログラムに適用されることが多い．これとは対照的にポリモーフィックエンジンは，機械語命令を意識することなくプログラム全体を暗号化し，その復号モジュールと暗号化されたプログラムをつなぎ合わせることで，新たなプログラムを出力する．高級言語で開発される昨今の多くのマルウェアにとって，この機械語命令列を意識する必要のないポリモーフィックエンジンは，適用が容易であり都合がよい．実際，ポリモーフィックエンジンの一種であるランタイムパッカーは，実に 90% 以上ものマルウェアで利用されているとの報告もある [7]．こうしたマルウェアの効率的な亜種開発とランタイムパッカーによるポリモーフィズムの両立が，マルウェアの増加を引き起こしていると考えられる．

1.3 関連研究

マルウェアの増加に対処すべく，マルウェアを分類することで，そのトレンドを把握する研究が進んでいる．マルウェアの分類技術には，大きく分けて挙動により分類する方法 [8–14] と，プログラムコードに基づき分類する方法 [15–28] が存在する．

マルウェアの挙動により分類する方法では，たとえば，システムコールの呼び出し履歴や，ネットワーク・ファイルシステム等のシステムリソースへのアクセス履歴を収集できる環境上でマルウェアを動作させ，得られた動作履歴を元にマルウェアを分類する．この方法は，マルウェアの挙動を把握できる実行環境さえ用意できればよく，リバースエンジニアリング等の高度な作業を必要としない．しかしながら，ボットのような攻撃者からの指令無しには動作しないマルウェアや，ある決まった時刻にしか動作しないマルウェアのように，その挙動を網羅的に抽出することが困難なケースも存在する．

こうした課題を解決するために，マルウェアのプログラムコードから抽出した特徴に基づきマルウェア間の類似度を算出し，分類する手法も存在する．これは挙動による分類とは対照的に，マルウェアが潜在的に備える機能を踏まえつつ分類できる．本研究でも，こ

うしたマルウェアのプログラムコードの特徴に着目したアプローチに焦点を当てる．マルウェアのプログラムコードに基づく分類に関する従来研究は，プログラムコードの特徴として何に着目しているかで整理することができる．

N-gram に基づくアプローチ [16] では，まずマルウェアの逆アセンブル結果からオペレーションコード列を抽出し，連続する N 個のオペレーションコード列（N-gram）がそれぞれ何回出現したかを特徴ベクトルとする．この特徴ベクトルをマルウェアの特徴とし，マルウェアの非類似度をコサイン距離として定義する．

他にもプログラムのベーシックブロックに着目した手法 [17] も提案されている．ベーシックブロックとは，分岐命令・分岐先命令を含まない連続した命令列である．まずマルウェアの逆アセンブル結果から，プログラムコードをベーシックブロックに分割する．そしてベーシックブロック単位でレーベンシュタイン距離を算出し，それをマルウェアの非類似度とする．

一方でプログラムのコールツリーを特徴とし，類似性を求める手法 [18] も存在する．コールツリーは，ソースコード上のプログラム構造が反映されるため，コンパイラの変更に強いと言われている．

このようにプログラムコードに基づく分類技術に関して様々な研究がなされており，N-gram やベーシックブロック，コールツリー等の各々の着眼点に関する類似度を算出することができる．しかし分類の全自動化や，マルウェアが備える脅威の全容を把握するという観点では，大きく分けて以下の三つの課題が残る．

一つ目の課題は，マルウェアのプログラムコードの抽出についてである．前述したように多くのマルウェアはパックされており，そのプログラムコードは隠蔽されている．このためプログラムコードに基づいたマルウェアの類似度算出を自動化するには，まずアンパックを自動化する手法が必要となる．

二つ目は，プログラム構造の再構築についてである．多くのアンパックされたマルウェアのメモリダンプには，逆アセンブルのヒントとなる実行ファイルのヘッダ情報が存在しない．当然，マルウェアのデバッグシンボル情報等も一般的には入手できないため，こうしたヒント無しに機械語命令とデータを区別する必要がある．加えて，例えばベーシックブロックに依る手法では，マルウェアのプログラムコードをベーシックブロック単位に区切る必要がある．一方，C++ のようなポリモーフィズムを備えるプログラミング言語によ

り開発されたプログラムでは，クラス継承を実現するために関数ポインタを利用した関数呼び出しが使われる [29]．このように動的に呼び出し先が変化する命令を多用されると，分岐命令の宛先を特定することができずベーシックブロックに区切ることも困難になる．コールツリーの構築もまた同様であり，現状では人手により正確な情報を再構築する必要がある．

三つ目の課題は，従来技術における類似度がプログラムコードの特徴の一部しか捉えていない点にある．例えば N-gram/N-perm やコールツリーに依る手法では，マルウェア間の距離が近かったとしても，偶然 N-gram/N-perm やコールツリーが似ていただけで，全く異なるマルウェアである可能性は否定できない．つまりマルウェアが備える脅威を完全に解明するには，結局全てのマルウェアの解析が必要となってしまう．他にも N-gram/N-perm による手法では，N-gram/N-perm の出現回数という一種の統計情報により類似度が定義されるため，実際に変化のあった場所を抽出することが難しいといった側面もあり，算出される類似度がプログラムコードのどの箇所に依るものかを特定できない．

本研究では，まず上述の各課題に対し以下の手法を提案し，プログラムコードに基づく自動マルウェア分類システムを構築する．

- アンパック

  マルウェアに適用されるランタイムパッカーの種類数は非常に多いため，パッカー毎にアンパックツールを開発するには多大なコストがかかる．加えて，公になっていないランタイムパッカーも存在するため，アンパックツールの開発そのものが難しい状況も存在する．こうした状況に対し，本研究ではパッカーの種類に依存しない汎用的なアンパック手法を提案する．これにより自動的にマルウェアのプログラムコードを抽出し，これを後述の逆アセンブル対象とする．
- 逆アセンブル

  インテルアーキテクチャ（IA-32）の機械語命令は可変長であり，加えて Windows 向けコンパイラは，実行バイナリのサイズを削減するために，機械語命令とデータを混在させる傾向にある．このため，メモリダンプからその機械語命令とデータを正確に区別することは非常に難しい．一方，前述のようにマルウェアの開発には高級言語が用いられる．本研究では，マルウェア開発に用いられるコンパイラの特徴

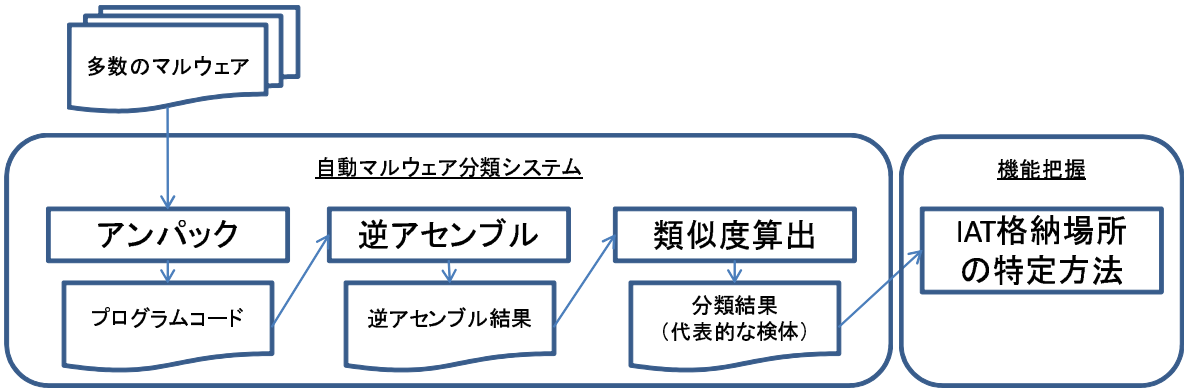

図 1.1　提案技術の関連性

を学習し，その結果に基づき，最も尤もらしい逆アセンブル結果を得る手法を提案する．この逆アセンブル結果は，後述の類似度算出手法にそのまま利用することができる．

- 類似度算出

  マルウェアが備える脅威の全容解明に要する解析作業を削減するべく，本研究では機械語命令列の LCS（Longest Common Subsequence）に基づく類似度算出手法を提案する．この過程で算出される LCS は，比較対象のプログラムコード間で共通している機械語命令列であり，同時に差分となる機械語命令列も容易に算出できる．つまり解析済みのマルウェアと新たなマルウェアを比較することで，必要な個所のみを解析することが可能になる．加えて，この手法により算出された類似度は，比較対象のプログラムコード同士で共通する命令数を表しており，その解析作業も容易に見積もることが可能となる．

さらに本研究では，個々のマルウェアの詳細な解析を支援するべく，IAT（Import Address Table）格納場所を特定する新たな手法を提案する．IAT はマルウェアが利用する外部関数のアドレスを格納するテーブルであり，この特定はマルウェアの機能を把握するために，必須のプロセスである．

本研究における提案手法の関連性は図 1.1 に示す通りである．本研究では，こうした提案手法の連携により，マルウェアの分類プロセスおよび個々の検体の解析を自動化することで，マルウェアが備える機能の全容解明に要する解析コストの削減を目指す．

1.4　本論文の構成

本論文は 7 章から構成される．第 2 章では，マルウェアのプログラムコード抽出に要するアンパックについて従来技術とその課題を整理し，新たな自動アンパック手法を提案する．第 3 章では，アンパックされたマルウェアのメモリダンプを逆アセンブルする手法について説明する．第 4 章でこうして得られた逆アセンブル結果に基づき類似度を算出する手法について説明する．さらに第 5 章では，第 2 章，第 3 章，第 4 章における提案手法を組み合わせて構築する自動マルウェア分類システムについて述べ，実際のインターネット環境で収集されたマルウェアを分類することで，本システムによるマルウェア解析作業の削減効果を示す．第 6 章では，マルウェアを詳細に解析する際に必須となる，IAT 格納場所の特定方法を提案し，その有効性を示す．最後に，第 7 章にて本論文をまとめる．

第 2 章

アンパック

2.1　はじめに

多くのマルウェアは，ランタイムパッカーとよばれる一種のポリモーフィックエンジンにより，そのプログラムコードが隠ぺいされている．本章では，ランタイムパッカーにより隠蔽されているプログラムコードを抽出することを目的とし，まず，その背景・関連研究および課題を整理する．その後に従来の課題を解決するアンパック手法を提案し，その有効性を示す．

2.2　ランタイムパッカー

まず，ランタイムパッカー [7] の仕組みについて概説する（図 2.1）．ランタイムパッカーは，実行ファイルを入力として受け取り，入力となった実行ファイルもしくはそこに含まれるプログラムコード（以下，オリジナルコードと呼ぶ）に対し何らかのエンコード

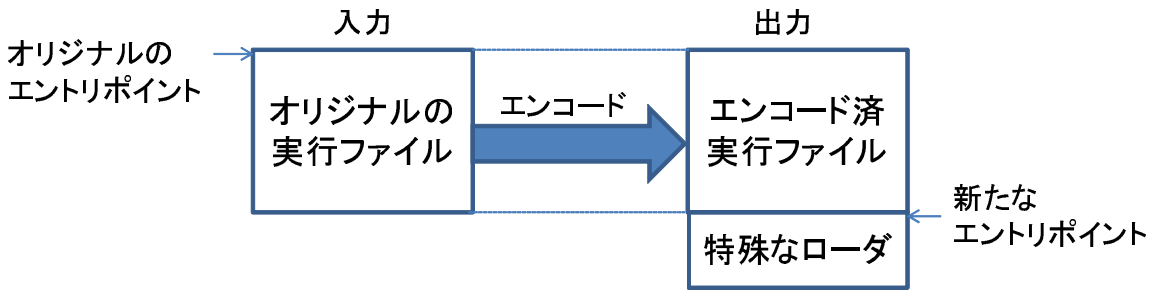

図 2.1　ランタイムパッカーによる処理

を行う．そして，特殊なローダにこのエンコードされたオリジナルコードを付け加え，新たな実行ファイルを出力する．この特殊なローダが新たな実行ファイルのエントリポイントになり，実行された際には図 2.2 のように以下の処理を行う．

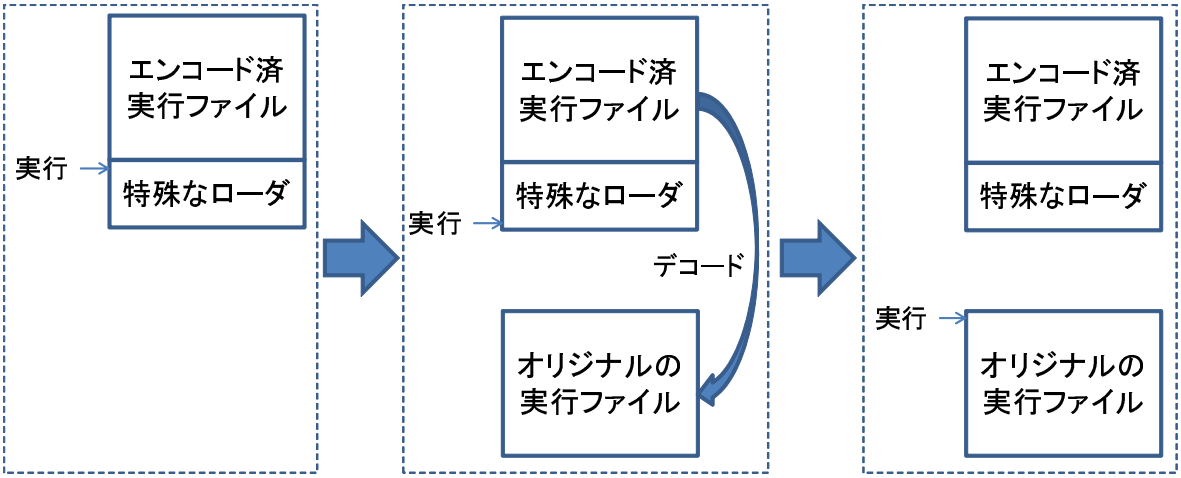

図 2.2　パックされた実行ファイルの動作

1. オリジナルコードをデコードし，メモリ上に展開する
2. オリジナルコードが呼び出す外部関数アドレスを解決し， IAT(Import Address Table) [30] に当該アドレスを書き込む
3. OEP（Original Entry Point，オリジナルコードのエントリポイント）に制御を渡す

通常の実行ファイルの場合，これらの処理は OS が備えるローダにより行われるが，パックされたマルウェアでは，エンコードされたプログラムコードを適切にロードするために，これらの処理を独自の特殊なローダが担っている．この一連のメカニズムにより，オリジナルコードは隠蔽されつつも，その機能は維持されることになる．

2.3　従来のアンパック手法

ここでは従来のアンパック手法について述べる．

実行ファイルがどのパッカーでパッキングされたかを識別し，隠蔽されたオリジナルコードを抽出するツールはいくつも存在する．例えば，PEiD [31] は単純なパターンマッチングによりパッカーを識別することができる．こうしてパッカーを特定することができ

れば，個別に開発されたアンパッカー [32] を利用し隠蔽されたオリジナルコードを抽出することができる．このアプローチは非常に高速かつ正確にアンパックが可能であるが，パッカーを識別でき，尚且つ対応するアンパッカーが存在している必要があるため，新しいアルゴリズムを備えるパッカーが利用された場合には適用できない．

一方，パッカー毎のアルゴリズムに依存しない動的なアンパック手法 [33–39] も提案されている．ランタイムパッカーでパッキングされたプログラムは，実行されるとオリジナルコードがメモリ上に展開され，通常 OS に備わるローダが行う処理（動的ライブラリのリンク，リロケーションなど）を行った後に，オリジナルコードが実行される．ほとんどの動的なアンパック手法はこの事実を前提として開発されている．OllyBonE [33] や Universal PE Unpacker [34] は事前にオリジナルコードのエントリポイントが含まれるであろう領域に対して，実行ブレークポイントを張り対象プログラムを実行することで，オリジナルコードのエントリポイントを発見する．ただしオリジナルコードのエントリポイントはマルウェアをリンクする際に任意に設定可能であるため，広大なアドレス空間に対してこれを事前に予測することは非常に難しい．この問題を解決するために Renovo [35], Pandora's Bochs [36] や Saffron [37] は，書き込みが生じた領域がその後実行された場合に，当該領域をオリジナルコードとして出力する．Renovo は TEMU [40] と呼ばれる一種のエミュレーションエンジン上に構築されており，マルウェアを安全に解析することができるが，その解析時間は実機を用いた場合と比較すると長くなる．Pandora's Bochs もまた Renovo と同じくエミュレーションエンジンを用いたアプローチである．一方，Saffron は OS のページフォールトハンドラを独自のハンドラに置き換えることで，解析速度の向上を果たしている．具体的には解析対象プロセス空間内の全メモリページに関して PTE (Page Table Entry) 内の User accessible ビットをクリアすることで，全メモリアクセスをフックし，メモリに対する書き込みおよび実行アクセスを監視している．メモリアクセスをフックした後に処理を続行させるためには，PTE の専用キャッシュである TLB (Translation Lookaside Buffer) を用いることで解決している．IA-32 [41] における TLB と PTE の同期処理は，システムプログラマに任されており，Saffron ではこの同期処理を敢えて省略することで PTE には User accessible ビットをクリアしつつも，TLB では User accessible ビットが設定された状態を作り出し，メモリアクセスをフックした後の処理を続行させている．

しかしながら，これらの手法にも二つの課題が残っている．

一つ目の課題は，従来技術が対象にしているのは OEP の発見のみであり，その OEP を含むプログラムコード領域が，どこからどこまでであるかを，簡単には決定できないことである．また二つ目の課題は，多重にパックされたバイナリを解析するときに，各層のアンパックが完了するたびにオリジナルコード（の候補）が抽出されてしまうことにある．このため事前にタイムアウトを設定しておく等の対処も提案されているが，オリジナルコードを特定するための根本的な対処には至っていない．

次節以降では，各課題を解決する提案手法について説明する．

2.4　プログラムコード領域特定へのアプローチ

アンパックの際に抽出すべき領域は，ランタイムパッカーが隠ぺいするコード領域，すなわちひとつの実行ファイルに含まれるプログラムコード領域である．本章での目標は，このプログラムコード領域全体を，他のプログラムコード領域を含むことなく，取り出すことにある．通常の実行ファイルの場合，プログラムコード領域の始点・終点アドレスは，実行ファイル中の PE ヘッダから特定可能である．しかしながら，PE ヘッダを削除してしまうパッカーも存在するため，パッキングされた実行ファイルに関してオリジナルのプログラムコード領域を特定するには，別の手段が必要となる．

2.4.1　プログラムコード領域全体の抽出

マルウェアのプログラムコード領域全体をカバーするには，単純に対象プロセスのメモリ空間で，使用中の領域を全て抽出すればよい．ただしこの方法だと，その後のリバースエンジニアリングに多大な負担がかかってしまう．ここでは，可能な限り範囲を絞ってプログラムコード領域全体を抽出する方法を考察する．IA-32 [41] 上の Windows は，主に以下の単位でメモリを管理している [42].

1. アロケーションレンジ
2. メモリリージョン
3. メモリページ

アロケーションレンジとは，VirtualAlloc [43] 等により一度に確保された領域であり，1 つ以上のメモリリージョンから構成される．メモリリージョンは 1 つ以上のページから構成され，メモリ属性（state, protect, type）が同一で，かつ同一のアロケーションレンジに収まる連続するメモリ領域である．また，メモリページは CPU における物理メモリと仮想メモリをマッピングする単位であり，IA-32 では 4KB もしくは 2MB ごとに 1 ページとなっている．

一般的なアプリケーションの場合，プログラムコード領域全体は，OEP を含むひとつのメモリリージョンとなっている．しかし，VirtualProtect [44] 等によりあるメモリリージョンの一部のメモリ属性が変更されると，複数のメモリリージョンに分割されてしまう．例えば，ひとつのコード領域の前半部分のメモリ属性 [45] を PAGE_EXECUTE_READ，後半部分を PAGE_EXECUTE_READWRITE，などとするだけで，二つのメモリリージョンに分割される（図 2.3）．つまり OEP を含むひとつのメモリリージョンを抽出しただけで

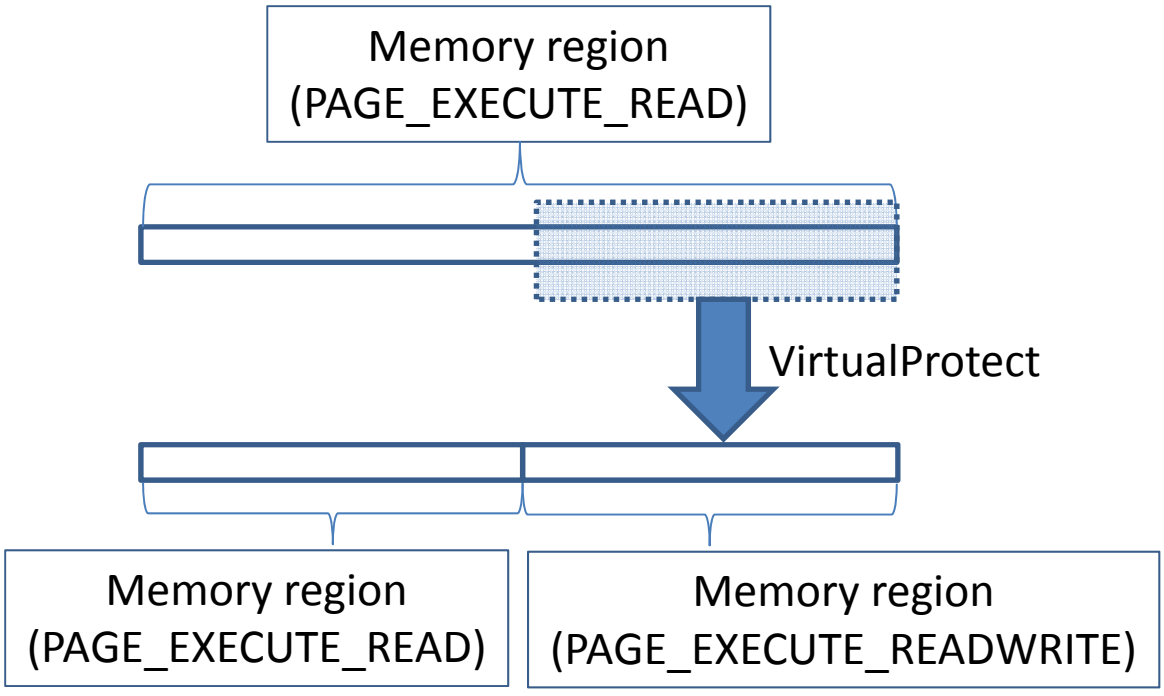

図 2.3　VirtualProtect によるメモリリージョンの分割

は，不十分となるケースが存在する．同様に，VirtualAlloc を複数回呼び出すことで複数の連続するアロケーションレンジを確保し，そこにオリジナルコードを展開することも可能である．このため，OEP を含むひとつのアロケーションレンジを抽出しても，プログラ

ムコード領域全体をカバーできないケースが存在する．

ところで，ランタイムパッカーへの入力が実行ファイルであることは先に述べた．一般的に，シンボル情報等がない実行ファイルを完全に逆アセンブルすることは難しい．これは，多くの Windows 向けコンパイラがメモリ空間を有効活用するために，機械語命令とデータを混在させたバイナリを出力することが要因である．つまり，ランタイムパッカーはオリジナルのプログラムコード領域をバイト列としてしか扱うことができず，基本的にこれを分割することはない．よって，プログラムコード領域全体を抽出するには，OEP を含む連続するコミット済みメモリ領域を抽出すればよいと考えられる．

ただ，オリジナルのプログラムコード領域の前後に動的リンクライブラリが接する場合，連続するコミット済みメモリ領域を抽出すると，その動的リンクライブラリも併せて抽出される．当然，アンパック結果として他のプログラムコード領域が混在すると，リバースエンジニアリングの作業量増加や，マルウェア分類の精度低下などを招くことになる．次節では，この問題を解決するため相対分岐命令に着目したプログラムコード領域の特定方法を提案する．

2.4.2　相対分岐命令に着目したプログラムコード領域の特定方法

IA-32 には，相対アドレス指定と絶対アドレス指定の 2 種類の分岐命令が存在する．一般的に，これらはコンパイラによって次のように使い分けられている．ある実行ファイルもしくは動的リンクライブラリが，別の動的リンクライブラリ内の関数を呼ぶ場合，その呼び出し先はコンパイル・リンク時に決定できない．そのため，通常はメモリ参照を用いた絶対アドレス指定の分岐命令が用いられ，実行時にローダが分岐先アドレスを解決する [46]．一方，同じ実行ファイル内もしくは動的リンクライブラリ内で閉じる関数呼び出しは，コンパイル・リンク時に呼び出し元から呼び出し対象までのオフセットが確定しているため，相対アドレス指定の分岐命令が用いられる．

こうした特徴から，相対アドレス指定の分岐命令とその分岐先は，通常ひとつの実行ファイルに収まっている．言い換えると，相対アドレス指定の分岐命令と分岐先の間（以下，分岐区域）で，実行モジュールが分断されることはない，と仮定できる（図 2.4）．本提案手法は，OEP を含む連続するコミット済みメモリ領域上の各バイトが，いくつの分岐区域に属するか（以下，分岐区域数）を算出することで，その分岐区域数が十分に少ない

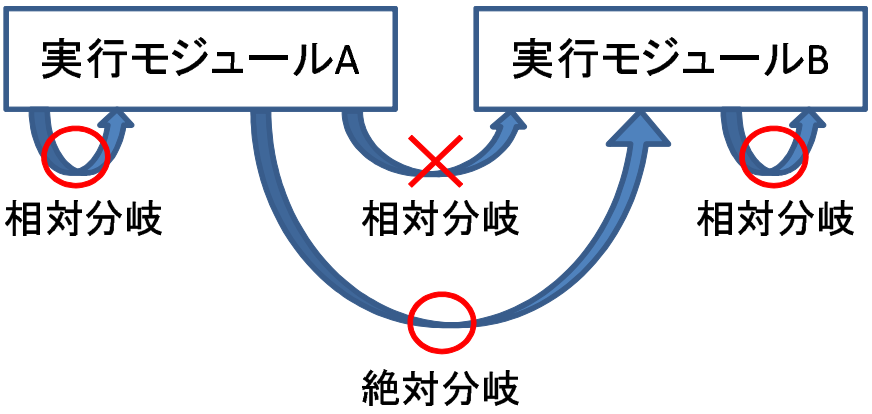

図 2.4　相対分岐と絶対分岐

箇所をプログラムコード領域の境界の候補とする．

しかしながら，パックされたマルウェアのオリジナルコードで利用される分岐命令を正確に特定する（正確な逆アセンブル結果を得る）ことは非常に難しい．これはパックされたマルウェアや IA-32 ならでは事情に依る．まず，IA-32 の機械語命令は可変長である．加えて Windows 向けコンパイラは，バイナリ全体のサイズ削減やキャッシュヒット率向上の観点から，機械語命令とデータが混在するバイナリを出力する傾向にある．さらに，パックされたマルウェアから抽出されるオリジナルコードには，逆アセンブルのヒントとなる実行ファイルのヘッダ情報[*1]が存在しない状況もある．これは，実行ファイルのヘッダ情報は OS が備えるローダのための情報であり，パックされたマルウェアのオリジナルコードをロードするのは，専用のローダが担っており，必ずしもオリジナルの実行ファイルのヘッダ情報を持っておく必要がないためである．

こうした逆アセンブルのヒント無しに，オリジナルコード内の分岐命令を探すために，本提案手法では，隠れマルコフモデル（HMM: Hidden Markov Model）[47] に従い，実行ファイルを確率モデルで表現する．HMM は確率モデルの一つであり，観測可能な情報からその背後に存在する未知の状態を推定することができる．また未知の状態（隠れ状態）間の遷移確率は，その名が示す通りマルコフ過程であることを前提としている．HMM は連続的な信号列におけるパターン抽出やラベル付与に効果を発揮することから，その応用

[*1] コードセクション/データセクションの区分や，インポートしている外部関数の情報が逆アセンブルのヒントになりうる．

分野は，形態素解析，音声認識，手書き認識，ジェスチャ認識，画像認識，ゲノム配列の解析など多岐に渡る．

実行ファイルの確率モデル

多くのマルウェアは，効率的なバグ改修や新機能の追加のために，Visual C++ や Delphi などの一般的なコンパイラが用いられる．本手法では，こうしたコンパイラが出力するバイナリの特徴を事前に学習しておくことで，オリジナルコード内の分岐命令らしさの算出を試みる．本手法における実行ファイルの確率モデルは，図 2.5 の 3 つの状態とトポロジを持つ．Start 状態は対象バイナリの開始を意味し，何も出力しない．Data 状態と

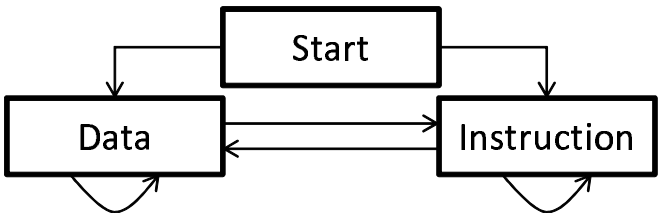

図 2.5　3 つの状態とトポロジ

Instruction 状態は，それぞれ 1 バイトのデータと 1 つの機械語命令を，各出力確率に基づき出力する．また Data 状態と Instruction 状態の間の遷移や自己遷移についても，各々の遷移確率に基づき遷移する．

Data 状態における出力確率は，出力される 1 バイトの値（0〜255）によって決定される．一方で，Instruction 状態における出力確率は少し複雑である．IA-32 の機械語命令は次の 6 つのパートから構成される．

- Prefix（0〜4 バイト）

  機械語命令はオプションとなる Prefix（最大で 4 バイト）から始まることがある．この役割は，デフォルトの機械語命令の振る舞いを変更するために用いられる．例えば，命令セットが使うデフォルト・セグメントを強制的に切り替えたり，オペランドのデフォルト・サイズを強制的に 16 ビットや 32 ビットに切り替えるために用いられる．Prefix は 4 つのグループに分かれており，4 つのグループからひとつずつ利用することができる．また Prefix の順序に制限はない．
- Opcode（1〜3 バイト）

Opcode は機械語命令の動作および機械語命令全体のフォーマットを決定づける．

- ModR/M（0〜1 バイト）

  ModR/M はオペランドのアドレッシング・モードを決定づける情報を含む．またオペランド中でレジスタが使われる場合は，そのレジスタ種別に関する情報も含む．Opcode によって存在しない場合もある．

- SIB（0〜1 バイト）

  SIB（Scale Index Base）は，アドレッシング・モードに関する情報を含む．Opcode や Mod R/M によって存在しない場合もある．

- Displacement（0〜4 バイト）

  Displacement には機械語命令がアクセスするアドレス情報が格納される．Opcode によって存在しない場合もある．

- Immediate（0〜4 バイト）

  Immediate には機械語命令が使う即値が格納される．Opcode によって存在しない場合もある．

これらの各パートにおいて出力される値の傾向は大きく異なる．例えば，Prefix として出力される値はわずか 11 種類しか存在しないが，Opcode は 200 種類以上の値を取りうる．そこで本手法では命令状態における出力確率を，図 2.6 のマルコフ連鎖モデルに従い定義する．これにより，データと機械語命令の差をはっきりさせることが見込める．対象となるバイト列における各バイト値は，逆アセンブルにより，図 2.6 におけるどの状態（パート）になるかを決定することができる．こうして得られた状態（パート）系列を用いることで，図 2.6 のモデルは，各命令パートの出力確率とパート間の状態遷移確率に従い，機械語命令を出力する．

この実行ファイルの確率モデルを利用するには，まずモデルのパラメータとなる各々の出力確率と遷移確率を事前に学習しておく必要がある．この学習データは，マルウェア開発によく用いられる Visual C++ や Delphi により作成すればよい．また，これまでの実験から Visual C++ は Delphi 等の他のコンパイラと比較し，多様な機械語命令を出力する傾向にあることが分かっている．このため，Visual C++ により出力された実行ファイルだけを学習データとして利用することでも，十分に汎用的なモデルパラメータを構築することができる．

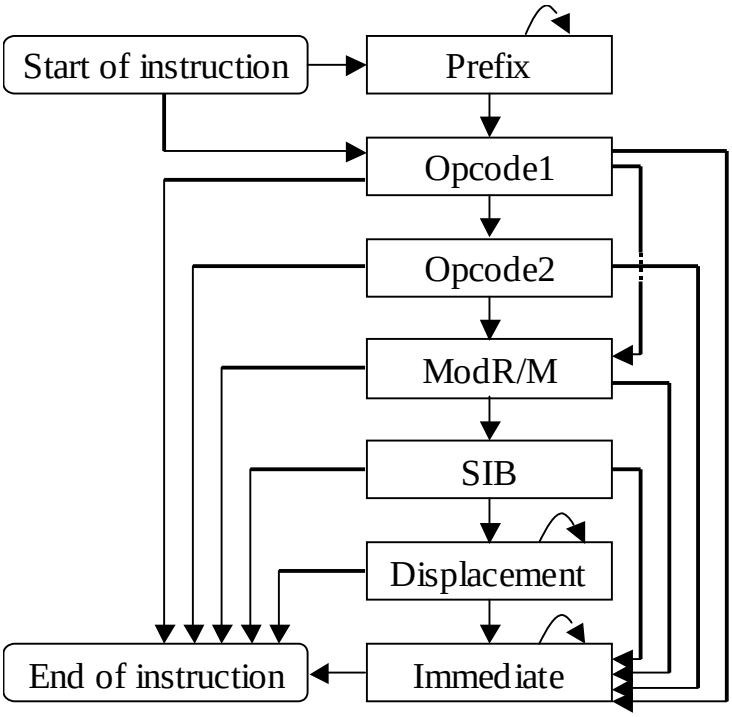

図 2.6　機械語命令用のマルコフ連鎖モデル

次に，上述のモデルに基づき Forward/Backword アルゴリズム [47] を用いることで，バイト列中の各バイト値が機械語命令である確率を算出する．このとき，コンパイラが出力する機械語命令は，オーバーラップしない[*2]ことに注意する必要がある．例えば以下のバイト列が与えられたとする．

0xEB, 0x01, 0xFF, 0xC3[*3]

もし最初のバイト値 0xEB が機械語命令の先頭として解釈されると，0xEB,0x01 の 2 バイトがひとつの機械語命令 ADD EDI,EDI となる．このとき，ほとんどのコンパイラ出力では，2 バイト目の 0x01 が機械語命令の先頭となることはない．つまり上述のモデルにおいて，0xEB が機械語命令の先頭として解釈された時点で，2 バイト目の 0x01 はデータ状態に遷移しないことを意味する．機械語命令のオーバーラップを排除するためには，このバイト列に対して，事前に図 2.7 のラティス構造を生成する．このラティス構造は，各バイト値を命令の先頭として解釈し，それらの命令長を算出することで構築できる．ラ

[*2] 意図的に機械語命令をオーバーラップさせることも難しいとされている [48].
[*3] 本論文で 16 進数を表現する際は，プレフィックスとして 0x を付ける．

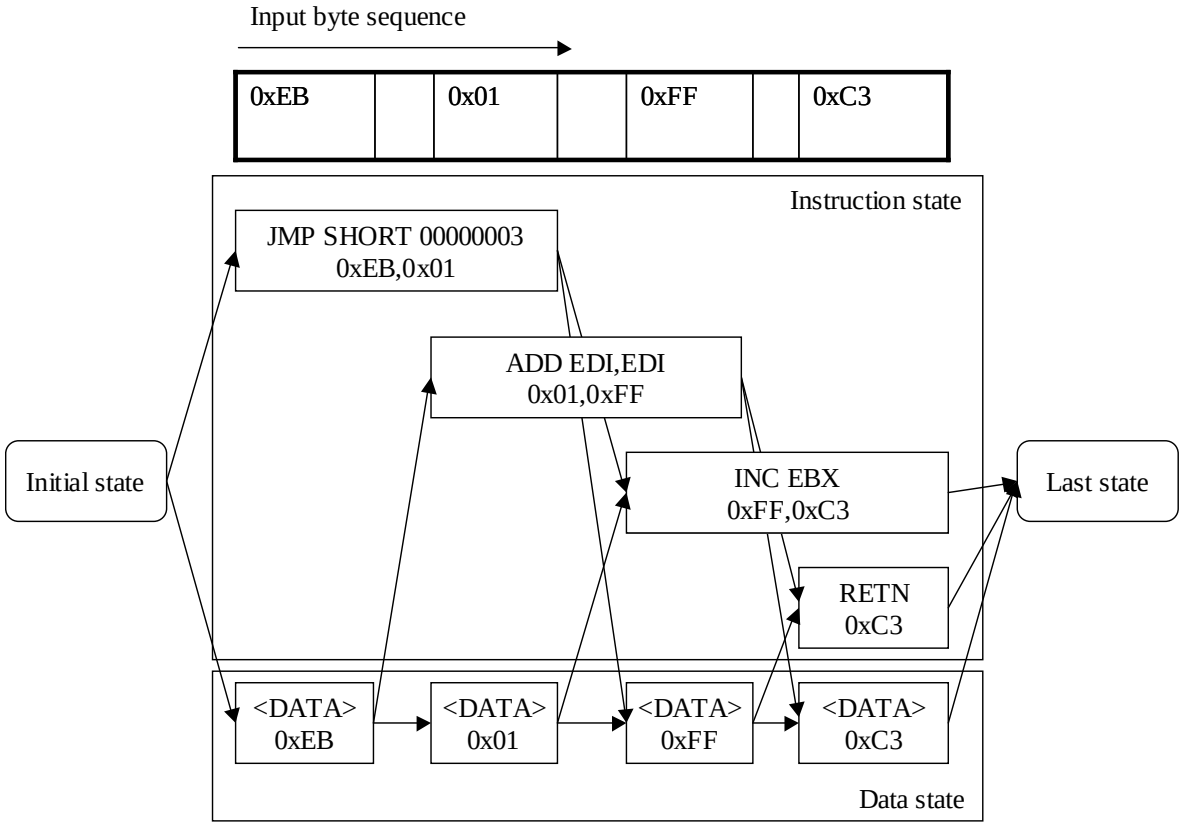

図 2.7　ラティス構造

ティス構造生成のための疑似コードを図 2.8 に示す．こうして得られたラティス構造に従い Forward/Backward アルゴリズムを用いることで，機械語命令のオーバーラップを排除しながら，各バイト値が機械語命令として解釈される確率を算出することができる．

ここではまず，Forward アルゴリズムを説明するために，いくつか記号を定義しておく．

- N: プログラムコードのバイト数.
- $X = x_1 \cdots x_N$: プログラムコードのバイト列．ここでは $x_1 \cdots x_n$ を x_1^n とも表記する．
- $S = \{S_B, S_I, S_D, S_F, S_M\}$: HMM における状態の集合． S_B はプログラムコードの開始，S_I は機械語命令の先頭，S_D はデータ，S_F はプログラムコードの終端，S_M は機械語命令の 2 バイト目以降を表す．
- $L = l_1 \cdots l_N$: $x_i (1 \leq i \leq N)$ に割り当てられる状態系列． 便宜上，$l_0 = S_B, l_{N+1} = S_F$ とする．

```
const InsnState = 0;
const DataState = 1;
var matrix : array[number of states,number of symbols];
var lastState;  {special state for last instructions or last data}
var startAddr;  {first symbol address}
var endAddr;    {last symbol address}
var insn;       {decoded instruction}
var addr;       {counter}
procedure generateMatrix();
begin
    for addr := startAddr to endAddr do
    begin
    insn := decodeInstruction(addr); {decode instruction at addr}
    if isValid(insn) = true then     {insn is a valid instruction.}
        if addr + length(insn) <= endAddr then
        begin
            matrix[InsnState, addr + length(insn)].previous.push(
                InsnState,addr);
            matrix[DataState, addr + length(insn)].previous.push(
                InsnState,addr);
        end
        else
            lastState.previous.push(InsnState,addr);
        end;
    end;
    if addr + 1 <= endAddr then
    begin
        matrix[InsnState, addr + 1].previous.push(DataState,addr);
        matrix[DataState, addr + 1].previous.push(DataState,addr);
    end
    else
        lastState.previous.push(DataState,addr);
    end;
    end;
end.
```

図 2.8 ラティス構造の生成

- $w_{s,n}$: 状態 s において x_n を先頭とする単語． $w_{s,n}$ の長さ（バイト数）を $|w_{s,n}|$ とすると，$w_{s,n}$ はバイト列 $x_n^{n+|w_{s,n}|-1}$ を意味する．
- $a_{s,s'}$: 状態 s から状態 s' へ遷移する確率．
- $b_{s,n}$: 状態 s で $w_{s,n}$ を出力する確率．

状態遷移確率 $a_{s,s'}$ と単語出力確率 $b_{s,n}$ は，事前に正確な逆アセンブル結果から学習して

おく．また，この $a_{s,s'}$ と $b_{s,n}$ をあわせてモデルパラメータと呼び θ と表記する．Forward アルゴリズムは，このモデルパラメータ θ が与えられたときの X の出力確率 $P(X|\theta)$ を効率よく算出するためのアルゴリズムである．まず単純な $P(X|\theta)$ の算出方法を考えてみる．$P(X|\theta)$ は X を出力する際に，全ての状態遷移系列を考慮した確率として表現できる．

$$P(X|\theta) = \sum_{allL} P(X, L|\theta) \tag{2.1}$$

$$= \sum_{allL} P(L|\theta)P(X|L, \theta) \tag{2.2}$$

ここで以下の仮説を利用する．

- ある状態に遷移する確率は，遷移するひとつ前の状態にのみ依存する．
- ある単語が出力される確率は，そのときの状態にのみ依存する．

この仮説のもとでは，$P(L|\theta)$ と $P(X|L, \theta)$ を次のように近似できる．

$$P(L|\theta) \approx \prod_{1 \leq n \leq N, l_n \neq S_M} P(l_n|l_{n-1}, \theta) \tag{2.3}$$

$$P(X|L, \theta) \approx \prod_{1 \leq n \leq N, l_n \neq S_M} P(w_{l_n,n}|l_n, \theta) \tag{2.4}$$

よって $P(X|\theta)$ を求めるには，以下を計算すればよい．

$$P(X|\theta) \approx \sum_{allL} \prod_{1 \leq n \leq N, l_n \neq C} P(l_n|l_{n-1}, \theta)P(w_{l_n,n}|l_n, \theta) \tag{2.5}$$

ただ，すべての状態遷移系列 L に関する確率の総和を求めようとすると，$O(N2^N)$ の計算量を要する．

この計算量を動的計画法を用いることで，$O(N)$ とするのが Forward アルゴリズムである．ここで，x_1^{n-1} を出力し l_n が s となる確率を $F(n, s)$，$F(1, S_I) = a_{S_B,S_I}$，

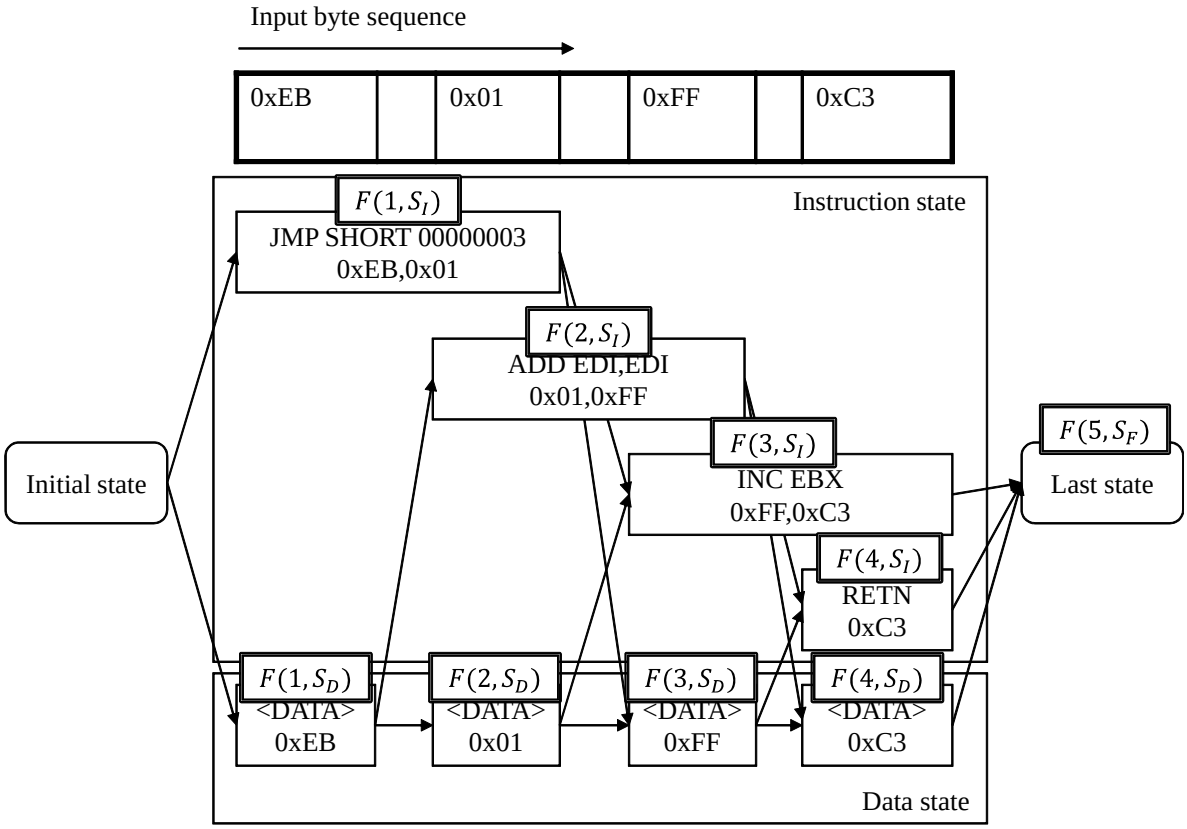

図 2.9　Forward アルゴリズムの処理

$F(1,S_D) = a_{S_B,S_D}$ とする．この $F(n,s)$ は以下の漸化式で表すことができる．

$$
\begin{aligned}
F(n,s) &= P(x_1^{n-1}, l_n = s|\theta) \\
&= \sum_{s',n':n'+|w_{s',n'}|=n-1} P(x_1^{n'-1}, w_{s',n'}, l_{n'} = s', l_n = s|\theta) \\
&= \sum_{s',n':n'+|w_{s',n'}|=n-1} P(x_1^{n'-1}, l_{n'} = s'|\theta) P(l_n = s, w_{s',n'}|x_1^{n'-1}, l_{n'} = s', \theta) \\
&\approx \sum_{s',n':n'+|w_{s',n'}|=n-1} P(x_1^{n'-1}, l_{n'} = s'|\theta) P(l_n = s|l_{n'} = s', \theta) P(w_{s',n'}|l_{n'} = s', \theta) \\
&\approx \sum_{s',n':n'+|w_{s',n'}|=n-1} F(n',s') P(l_n = s|l_{n'} = s', \theta) P(w_{s',n'}|l_{n'} = s', \theta) \\
&\approx \sum_{s',n':n'+|w_{s',n'}|=n-1} F(n',s') a_{s',s} b_{s',n'}
\end{aligned}
$$

この漸化式を利用し，$F(1,S_I), F(1,S_D), F(2,S_I), \ldots$ を順に算出していき，最終的に求まる $F(N+1,S_F)$ が $P(X|\theta)$ となる．例えばバイト列 $X = (0xEB, 0x01, 0xFF, 0xC3)$ が与えられたときは，図 2.9 に示すように $F(1,S_I), F(1,S_D), F(2,S_I), \ldots$ を算出してい

き，$F(5, S_F)$ が $P(X|\theta)$ となる．

次に，$l_n = s$ であるときに x_n^N を出力する確率を $B(n, s)$ とする．この $B(n, s)$ も漸化式で表すことができる．

$$
\begin{aligned}
B(n,s) &= P(x_n^N | l_n = s, \theta) \\
&= \sum_{s', n' : s' \in S, n' = n + |w_{s,n}|} P(w_{s,n}, x_{n'}^N, l_{n'} = s' | l_n = s, \theta) \\
&\approx \sum_{s', n' : s' \in S, n' = n + |w_{s,n}|} P(w_{s,n} | l_n = s, \theta) P(x_{n'}^N, l'_n = s' | l_n = s, \theta) \\
&\approx \sum_{s', n' : s' \in S, n' = n + |w_{s,n}|} P(w_{s,n} | l_n = s, \theta) P(l'_n = s' | l_n = s, \theta) P(x_{n'}^N | l_n = s, l_{n'} = s', \theta) \\
&\approx \sum_{s', n' : s' \in S, n' = n + |w_{s,n}|} P(w_{s,n} | l_n = s, \theta) P(l_{n'} = s' | l_n = s, \theta) P(x_{n'}^N | l_{n'} = s', \theta) \\
&\approx \sum_{s', n' : s' \in S, n' = n + |w_{s,n}|} P(w_{s,n} | l_n = s, \theta) P(l_{n'} = s' | l_n = s, \theta) B(n', s') \\
&\approx \sum_{s', n' : s' \in S, n' = n + |w_{s,n}|} b_{s,n} a_{s,s'} B(n', s')
\end{aligned}
$$

この漸化式を利用し，後方から $B(N, S_I), B(N, S_D), B(N-1, S_I), \ldots$ と順に算出することで，$P(X|\theta) = B(0, S_B)$ を求めることができる（図 2.10）．これが Backward アルゴリズムである．

ここで話を戻し，X と θ が与えられたときに $l_n = s$ となる確率 $P(l_n = s|X, \theta)$ を算出する．

$$
P(l_n = s | X, \theta) = \frac{P(X, l_n = s | \theta)}{P(X|\theta)}
$$

この分母 $P(X|\theta)$ は Forward/Backward アルゴリズムにより算出できる．一方，分子は以下のように変形できる．

$$
\begin{aligned}
P(X, l_n = s | \theta) &= P(x_1^{n-1}, x_n^N, l_n = s | \theta) \\
&= P(x_1^{n-1}, l_n = s | \theta) P(x_n^N | x_1^{n-1}, l_n = s, \theta) \\
&\approx P(x_1^{n-1}, l_n = s | \theta) P(x_n^N | l_n = s, \theta)
\end{aligned}
$$

ここに出現する $P(x_1^{n-1}, l_n = s|\theta)$ および $P(x_n^N | l_n = s, \theta)$ はそれぞれ前述の $F(n, s)$ と $B(n, s)$ である．よって $P(l_n = s|X, \theta)$ は以下の式で求めることができる．

$$
P(l_n = s | X, \theta) \approx \frac{F(n, s) B(n, s)}{F(N+1, S_F)}
$$

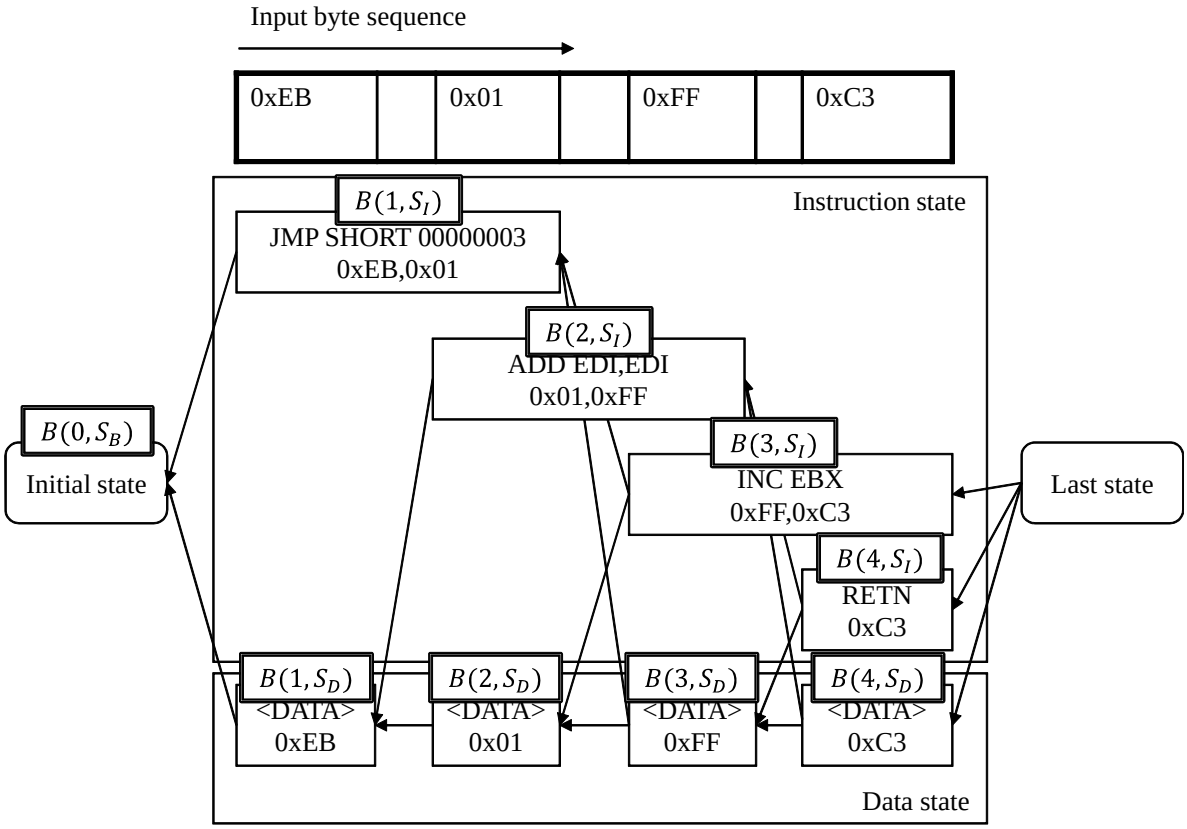

図 2.10　Backward アルゴリズムの処理

分岐区域数の期待値算出とコード領域の識別

対象となるプログラムコードのバイト列に関して，各バイトが機械語命令かデータかの確率を求められれば，同じく各バイトに関して，前述の分岐区域数の期待値を算出することが可能になる．ここで，X において相対分岐命令の先頭と解釈できる箇所を x_i，その宛先を x_j とした場合，$\mathbf{X}$ に存在する (i, j) の集合を J_X としておく．

通常，分岐命令の宛先は機械語命令の先頭である．よって $\mathbf{X}$ が与えられたとき，$(i, j) \in J_X$ に関して分岐が成立する確率は，$P(l_i = S_I, l_j = S_I | X, \theta)$ となる．

また，x_i が機械語命令の先頭である確率と x_j が機械語命令の先頭である確率が独立であるとすると，$P(l_i = S_I, l_j = S_I | X, \theta)$ は以下のように近似できる．

$$P(l_i = S_I, l_j = S_I | X, \theta) \approx P(l_i = S_I | X, \theta) P(l_j = S_I | X, \theta)$$

これにより x_n が属する分岐区域数の期待値 E_n は次のように算出できる．

$$E_n = \sum_{(i,j)\in J_X : i\leq n\leq j+|w_{S_I,j}|-1 \vee j\leq n\leq i+|w_{S_I,i}|-1} P(l_i = S_I|X,\theta)P(l_j = S_I|X,\theta)$$

OEP を中心として E_n が十分に大きい箇所は，同じプログラムコード領域内であると考えられる．しかしプログラムコード領域の始点・終点付近は，分岐区域内に収まらないこともある．この問題を解決するため，抽出対象の始点 x，終点 y を以下のように決定する．k は，連続するコミット済みメモリ領域の先頭からの OEP のオフセット，ϵ は 0 に近い値（例えば 0.5）である．

$$p^* = \begin{cases} 0 & if\ \{p|p<k, E_p<\epsilon\} = \phi, \\ \max\{p|p<k, E_p<\epsilon\} & otherwise. \end{cases}$$

$$q^* = \begin{cases} 0 & if\ \{q|q<k, E_q<\epsilon\leq E_{q-1}\} = \phi, \\ \max\{q|q<k, E_q<\epsilon\leq E_{q-1}\} & otherwise. \end{cases}$$

$$x = \frac{p^*+q^*}{2}$$

$$r^* = \begin{cases} N-1 & if\ \{r|k<r, E_r<\epsilon\} = \phi, \\ \min\{r|k<r, E_r<\epsilon\} & otherwise. \end{cases}$$

$$s^* = \begin{cases} N-1 & if\ \{s|k<s, E_s<\epsilon\leq E_{s+1}\} = \phi, \\ \min\{s|k<s, E_s<\epsilon\leq E_{s+1}\} & otherwise. \end{cases}$$

$$y = \frac{r^*+s^*}{2}$$

これらの点を図示すると，図 2.11 のようになる．

p^*, r^* は OEP に最も近い $E_n < \epsilon$ となる点であり，q^*, s^* は隣接するプログラム領域によって $\epsilon \leq E_n$ となる OEP に最も近い点である．x と y を，それぞれ p^*, q^* と r^*, s^* の中間点とすることで，OEP を含むプログラム領域が欠けることおよび，隣接するプログラムコード領域の混入を防ぐことができる．

2.5　多重パックへのアプローチ

従来のアンパック手法では，多重にパックされたバイナリを解析するときに，各層のアンパックが完了するたびにオリジナルコードの候補が抽出されてしまう．ここでは，複数のオリジナルコードの候補が得られたときに，その中から真のオリジナルコードを特定す

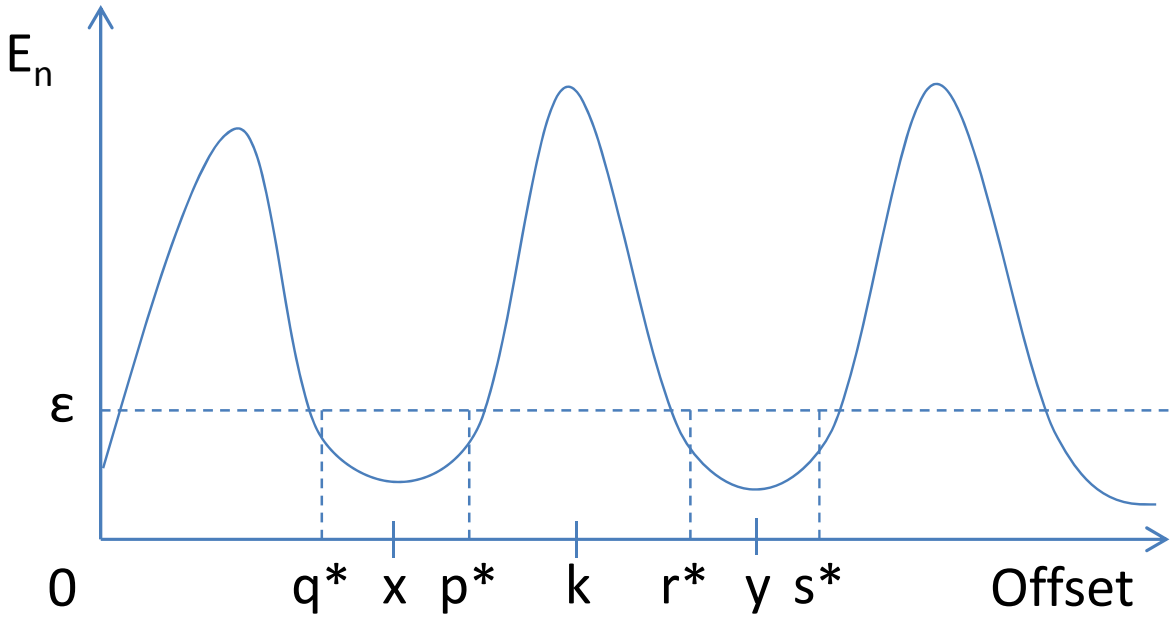

図 2.11　連続するメモリ領域における E_n と x,y の例

る手法を提案する．

まず，アンパックの処理ルーチンはコンパイラ出力コードとして尤もらしくなく，オリジナルコードはコンパイラ出力コードとして尤もらしいことを前提とする．そして，この前提に基づきコンパイラ出力コードを確率モデルにより表現することで，その尤度を算出しオリジナルコードを決定する．コンパイラ出力コードのモデルは，図 2.5 の実行ファイルの確率モデルを利用する．前述の通り Forward アルゴリズムを用いることで，$P(X|\theta)$ を求めることができる．これは θ を一般的なコンパイラで学習しておくことで，入力バイト列 X が与えられたときの θ の尤度，つまりコンパイラ出力コードとしての尤もらしさと捉えることができる．こうして得られた $P(X|\theta)$ と，事前の知見なしに求まる $P(X)$ [*4]の比 $\frac{P(X|M_\theta)}{P(X)}$（尤度比という）を算出し，その値が 1 以下であればコンパイラ出力コードらしくない，1 より大きければコンパイラ出力コードらしい，といった判断が可能となる．ただし，このモデルはあくまで命令やデータに関するバイト値の出現頻度と，各状態間の遷移確率のみをモデル化しているに過ぎないため，アンパックの処理ルーチンであっても尤度が高くなる状況が考えられる．

アンパックの処理ルーチンには，コンパイラ出力コードには見られない特徴を持つもの

[*4] ヌルモデルと呼ばれる．ここでは $P(X) = \frac{1}{256^N}$ とする．

がある．そのひとつとして分岐命令の宛先が命令の先頭からずれた場所を指すように見える，といった状況（ブランチバイオレーションと呼ぶ）が挙げられる．この意図は，先頭から順に逆アセンブルを行った際に本来の命令列を隠す（つまり解析を困難にする）ためであり，アンパックの処理ルーチンにはこういった分岐命令が頻繁に出現する．一方，通常のコンパイラ出力コードには，宛先が命令の先頭ではない分岐命令は存在しない．したがって，コンパイラ出力コードの尤もらしさを算出する際には，このブランチバイオレーションが起こる逆アセンブル結果を排除するべきである．

ここで分岐命令の先頭と解釈できる入力バイト値を x_i，その宛先を x_j とした場合に，入力系列 X に存在する (i,j) の集合を J_X，(i,j) がブランチバイオレーションを起こさない（$l_i \neq S_I$ もしくは $l_j = S_I$ を満たす）事象を $F_{i,j}$ とすると，入力バイト列 X を出力し，ブランチバイオレーションを全く起こさない確率は

$$
\begin{aligned}
&P(X, \bigcap_{(i,j)\in J_X} F_{i,j}|\theta) \\
&= P(X|\theta)P(\bigcap_{(i,j)\in J_X} F_{i,j}|X,\theta)
\end{aligned}
$$

と表現できる．しかし，これを直接算出するには全ての分岐を展開する必要があり，その展開数は分岐のオーバーラップ数に応じて指数関数的に増加するため，計算量の観点から困難である．そこで，「ある分岐命令がブランチバイオレーションを起こさない確率は，その他の分岐命令がブランチバイオレーションを起こさない確率と独立であり，分岐元の状態確率と分岐先の状態確率もまた独立である」と仮定する．これにより前式は以下のように近似することができる．

$$
\begin{aligned}
&P(X|\theta)P(\bigcap_{(i,j)\in J_X} F_{i,j}|X,\theta) \\
&\approx P(X|\theta) \prod_{(i,j)\in J_X} P(l_i \neq S_I \ or \ l_j = S_I|X,\theta) \\
&\approx P(X|\theta) \prod_{(i,j)\in J_X} 1 - P(l_i = S_I|X,\theta)P(l_j \neq S_I|X,\theta)
\end{aligned}
$$

ここで出てくる $P(l_i|X,\theta)$ は，前述のとおり Forward/Backward アルゴリズムにより容易に算出することができる．最終的には以下の尤度比をコンパイラ出力コードの判断基準

とする．

$$\frac{P(X|\theta)\prod_{(i,j)\in J_X} 1 - P(l_i = S_I|X,\theta)P(l_j \neq S_I|X,\theta)}{P(X)}$$

具体的には上式が 1 より大きい場合にはコンパイラが出力したコードとし，1 以下の場合はコンパイラが出力したコードではない，と判断する．

2.6　実装

本節では，パックされたマルウェアを入力とし，マルウェアのオリジナルコードを抽出する自動アンパックシステムについて説明する．本自動アンパックシステムは前述の二つの提案手法と，メモリ監視モジュールから構成される．ここでは，まずメモリ監視モジュールについて説明し，続いてメモリ監視モジュールと前述の二つの提案手法を組み合わせた，自動アンパックシステムの構成について説明する．

本システムでは，マルウェアを仮想 PC 上で動作させるが，仮想 PC 上で動作していることを検出し，その動作を停止するマルウェアも存在する．そこで本節の最後では，こうした仮想化ソフトウェアの検出方法についてと，その対処法について述べる．

2.6.1　メモリアクセス監視モジュール

本モジュールは Saffron と同様，カーネルドライバとして実装され，独自のページフォールトハンドラを用いることで，「書き込まれた後に実行される」メモリページを，オリジナルコードの候補として出力する．図 2.12 にメモリページの状態遷移を示す．まず対象プロセス空間内のすべてのメモリページに関して，書き込み監視状態とする．その後，書き込みが発生したメモリページに関して，実行監視状態に遷移させる．そして実行監視状態であるメモリページが実行された際に，そのメモリページが含まれる領域をファイルへ出力し，当該領域を再び書き込み監視状態に戻す．

メモリアクセス監視機構において提案手法と Saffron とが大きく異なる点は監視方法にある．Saffron では PTE に User accessible ビットをクリアしつつも，TLB では User accessible ビットが設定された状態を作り出し，メモリアクセスをフックした後の処理を続行させていた．一方，実際にマルウェアを動作させるアプローチでは，マルウェア感染による解析への影響をなくすため，一般的に仮想マシン上で解析し 1 検体毎に環境をロー

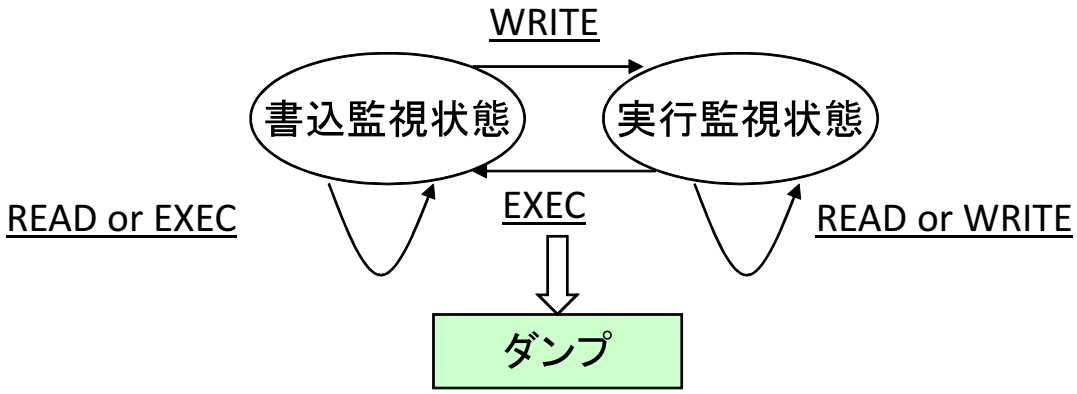

図 2.12 メモリアクセス監視モジュールの処理

ルバックする．しかし，仮想マシンにおける TLB は実機のそれを忠実に再現しているとは限らない．実際，著名な仮想マシンである VMware [49] においても，命令用 TLB が特権モードをまたぐ際にフラッシュされるとの報告 [50] もある．命令用 TLB が特権モードをまたぐ際にフラッシュされてしまうと，Saffron の実装では常に PTE に設定した User accessible ビットがクリアされた状態になるため，ユーザプロセスとページフォルトハンドラを行き来する無限ループに陥ってしまう．そのため，マルウェア解析では必須ともいえる仮想マシン上での解析が不可能となってしまう．こうした TLB への依存を無くすため，本研究で開発したメモリ監視モジュールは，まず書き込み監視するべきメモリページに対しては，対応する PTE の writable-bit を 0 に設定する．これにより書き込みが発生したときのみページフォルトを引き起こすことができる．また，実行を監視するべきメモリページに対しては，対応する PTE の XDbit [51] を 1 に設定することで実現する（図 2.13）．これには PAE(Physical Address Extensions) が有効になっており，かつ MSR（Model-Specific Registers）内の 11 ビット目を設定する必要があることに注意する [52]. 一方，PTE を操作し該当する監視状態に移行させた後に，当該メモリページがページアウトしてしまうと，PTE へ書き込んだ情報が失われてしまう．さらに動的に確保されたメモリページに対しても，それを検出し書き込み監視のために writable-bit を 0 に設定する必要がある．ただ，何れの状況においてもページフォルトが発生するため，ページフォルトの発生したメモリページが，現在どちらの状態（書き込み監視か実行監視）であるかに基づき，再度 writable-bit を 0，もしくは XDbit を 1 に設定することで，継続的に監視状態を保つことができる．

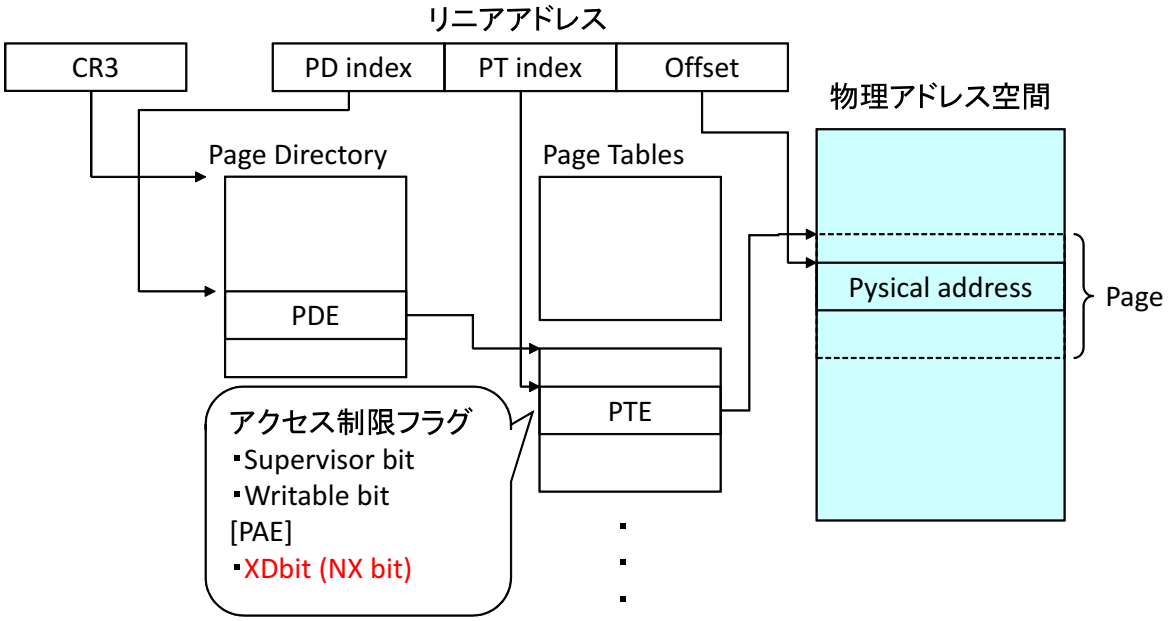

図 2.13　PTE 内の XDbit

こうした機能により TLB を用いることなく，監視する必要があるアクセスのみでページフォールトを発生させることができるようになったため，TLB の実装が実機のそれとは異なる VMware 等の仮想マシンであっても，問題なく動作可能となった．また，不要なページフォールトが発生しなくなったことにより速度面のパフォーマンス向上も見込むことができる．

2.6.2　自動アンパックシステムの構成

本システムの全体像を図 2.14 に示す．

メモリ監視モジュールは，Saffron と同様，マルウェアを実際に動作させ，当該プロセス空間において書き込みが発生したメモリページを，実行監視状態に遷移させる．そして実行監視状態のメモリページが実行された際に，そのときの命令ポインタ（OEP の候補）を含む連続するコミット済み領域を抽出する．ここで抽出された領域は，オリジナルコードの候補以外にも隣接する動的リンクライブラリが混在する可能性がある．この抽出処理は，事前に設定しておいた解析時間（特に断らない限り 3 分とする）が経過するまで繰り返される．マルウェアが多重にパックされている場合は，複数の領域が得られることになる．

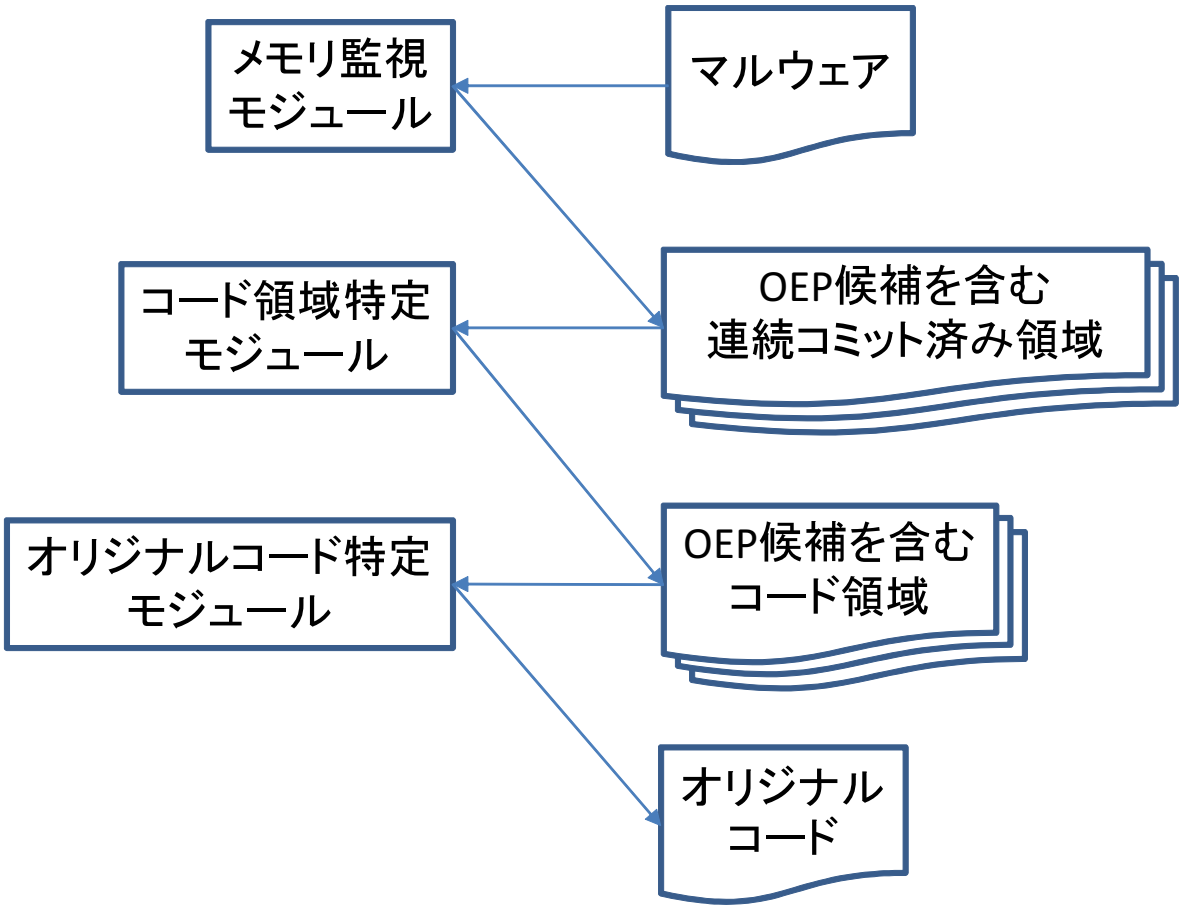

図 2.14 アンパックシステムの全体像

コード領域特定モジュールは，メモリ監視モジュールにより抽出された各メモリ領域に関して，OEP を含む一つのプログラムコード領域を抽出する．この処理によりオリジナルコードの候補に動的リンクライブラリ隣接する場合でも，オリジナルコードの候補のみが抽出される．

オリジナルコード特定モジュールは，コード領域特定モジュールにより抽出されたオリジナルコードの候補から，コンパイラ出力コードとして尤もらしい候補を真のオリジナルコードとする．

2.6.3 仮想化ソフトウェアの検出方法とその対処法

マルウェアを動作させその挙動を解析する場合，動作環境は当然そのマルウェアに感染することになる．このため，環境を元の状態に容易に戻すことができる仮想化ソフトウェ

アは，マルウェアを収集するハニーポットや，マルウェアの解析ツールを動作させるうえで，必須のツールとなっている．一方こうした状況に対し，自身が仮想化ソフトウェア上で動作していることを検出する機能を備えるマルウェアも出現している．

ここでは，自動アンパックシステムにて利用する VMware [49] に焦点を当て，仮想化ソフトウェアの検出方法とその対処法について述べる．

VMware バックドア I/O ポート

VMware には，ゲスト OS とホスト OS 間のシームレスなマウスポインタの移動や，ファイルのドラッグ＆ドロップ等を実現するために，バックドア I/O ポートと呼ばれる仕組みが備わっている．マルウェアはこの仕組みが存在するかどうかで，自身が VMware 上で動作しているかどうかを判断することができる．図 2.18 は Elias Bachaalany [*5] による VMware 検出プログラムである．

VMware バックドア I/O ポートを利用するには，EAX レジスタに'VMXh'，ECX レジスタにコマンド番号，EBX レジスタにコマンド引数，DX レジスタにポート番号'VX' を設定し，IN 命令（in eax, dx）を実行する．図 2.18 では，VMWare のバージョン番号を取得するコマンドの番号 10 を，ECX レジスタに設定している．当該プログラムが VMware 上で動作している場合，IN 命令の呼び出し後に，EBX レジスタに'VMXh' が設定され，EAX レジスタと ECX レジスタにバージョン番号が格納される．図 2.18 のプログラムは，EBX レジスタが'VMXh' であかどうかで，自身が VMware 上で動作しているかどうかを判断している．一方物理 PC 上の場合は，IN 命令で例外が発生し IsInsideVMWare 関数は false を返す．

この仮想化検出手法は，仮想マシンイメージが保存されているフォルダに存在する vmx ファイルを編集することで無効化することができる．具体的には，図 2.16 を追加すればよい．

この設定によりバージョン取得機能が無効化され，IN 命令を呼び出しても EBX レジスタ，EAX レジスタ，ECX レジスタは変化しなくなる．

しかしこの設定だけでは，IN 命令での例外発生までは再現されない．このため，例外が発生するかどうかを判断基準とする，図 2.17 のような検知手法には効果がない．

[*5] "Detect if your program is running inside a Virtual Machine", http://www.codeproject.com/KB/system/VmDetect.aspx

```
bool IsInsideVMWare()
{
  bool rc = true;
  __try
  {
    __asm
    {
      push   edx
      push   ecx
      push   ebx
      mov    eax, 'VMXh'
      mov    ebx, 0 // any value but not the MAGIC VALUE
      mov    ecx, 10 // get VMWare version
      mov    edx, 'VX' // port number
      in     eax, dx // read port
                     // on return EAX returns the VERSION
      cmp    ebx, 'VMXh' // is it a reply from VMWare?
      setz   [rc] // set return value
      pop    ebx
      pop    ecx
      pop    edx
    }
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    rc = false;
  }
  return rc;
}
```

図 2.15 バックドア I/O ポートの検出

```
isolation.tools.getVersion.disable = "TRUE"
```

図 2.16 getVersion の無効化

IN 命令による例外発生を再現するためには，VMware プロセスに修正を加える必要がある．VMware における仮想 PC は vmware-vmx.exe という名のプロセスとして，ホスト OS 上で動作する．このプロセス内には，以下の機械語命令列が存在する．これは IN 命令によりバックドア I/O ポートに対して通信があった際に，EAX レジスタに'VMXh' が設定しているかをチェックするコードである．この jz 命令を NOP 命令（0x90）に変更することで，IN 命令による例外発生を再現することが可能となる．

```
bool IsInsideVMWareEx()
{
  __try
  {
    __asm
    {
      push   edx
      push   ecx
      push   ebx
      mov    eax, 'VMXh'
      mov    ebx, 0 // any value but not the MAGIC VALUE
      mov    ecx, 10 // get VMWare version
      mov    edx, 'VX' // port number
      in     eax, dx // read port
                     // on return EAX returns the VERSION
      pop    ebx
      pop    ecx
      pop    edx
    }
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    rc = false;
  }
  return true;
}
```

図 2.17　例外発生の検出

```
cmp large dword ptr ds:0D018h, 'VMXh'
jz  loc_7CF2FF
```

図 2.18　vmware-vmx.exe における'VMXh' チェック

ITDR，LDTR

IA-32 では，IDT（Interrupt Descriptor Table）と呼ばれる，割り込みや例外発生時に呼び出される関数のテーブルが存在する．IDT は各論理 CPU 毎に存在し，そのアドレスは IDTR（IDT Register）に格納される．この IDTR の変更は，カーネルモードでのみ許可されているが，読み込みは SIDT 命令を使うことでユーザモードから行うことができる．具体的には，図 2.19 のプログラムで IDT のアドレスを取得できる．IDTR は 6 バイトで構成され，後半の 4 バイトに IDT のアドレスが格納されている．この IDT のアドレスは，

```
DWORD getIdtAddr()
{
    UCHAR   idtr[6];
    __asm sidt idtr;
    return *(DWORD*)&idtr[2];
}
```

図 2.19 IDTR の取得

```
__declspec(naked) USHORT getLdtSelector()
{
    __asm{
        sldt ax
        ret
    }
}
```

図 2.20 LDTR の取得

```
monitor_control.vt32 = "TRUE"
```

図 2.21 Intel VT の設定

実 PC の場合と VMware 上の場合とで異なる値を示す．

また，LDT（Local Descriptor Table）も同様に，VMware 上での動作を検出するために使われる．LDT のセグメントセレクタの値が，実 PC 上と仮想マシン上で異なってくる．LDT のセグメントセレクタの値は図 2.20 のプログラムにより取得することができる．getLdtSelector 関数は，実 PC 上では 0x0000 を返すが，VMware 上では 0 以外の値を返す．

IDTR や LDTR が物理 PC と仮想 PC とで変化する理由は，パフォーマンスを優先し，仮想化が完全に為されていないためである．VMware ではアクセラレーション機能を無効化するか，Intel VT 等のハードウェアによる仮想化支援機能を有効にすることで，対応できる．具体的には，vmx ファイルに図 2.21 を追加すればよい．

2.7　実験

本節では，コード領域特定モジュールとオリジナルコード特定モジュールの精度を実験により評価し，その結果を考察する．

2.7.1　コード領域特定モジュール

本節では，OEP を含む連続するコミット済みメモリ領域に，複数のプログラムコード領域が存在する場合に，OEP を含むひとつのプログラムコード領域全体を抽出可能かどうかについて実験を行う．この実験では，OEP を含む連続するコミット済みメモリ領域として，Windows 7 に付属する表 2.3 の 3 つの実行ファイルのメモリイメージを連結したバイト系列を利用した．このうち，notepad.exe のエントリポイントを 2.4.2 節における k

ファイル名	バージョン	ファイルサイズ（バイト）
kernel32.dll	6.1.7600.16481	875,088
notepad.exe	6.1.7600.16385	179,712
user32.dll	6.1.7600.16385	811,520

表 2.1　利用した実行ファイル

とし，この k を用いることで，notepad.exe のプログラムコード領域を抽出可能か調べた．確率的逆アセンブルに要する HMM のモデルパラメータに関しては，Firefox 3.0.1 [53] を Microsoft C++ Compiler(Ver. 15.00.21022.08) でコンパイルし，生成された xul.dll のアセンブリコードを学習データとした．また，分岐区域数の期待値 E_n に対する閾値 ϵ は 0.5 とした．

図 2.22 に，3 つの連続するメモリイメージにおける分岐区域数の期待値を示す．横軸はメモリイメージ上のオフセット n（16 進数表記）を表し，縦軸は各オフセットにおける分岐区域数の期待値 E_n である．先頭から kernel32.dll, notepad.exe, user32.dll の順でメモリイメージが並んでおり，注釈のついている箇所がそれぞれのコード領域である．2.4.2 節における k（OEP のオフセット）を，notepad.exe のエントリポイント 0x000D7689 とした場合，プログラムコード領域の始点 x・終点 y は，それぞれ図 2.22 に示す通りとなった．この x から y までの領域を抽出することで，notepad.exe のプログラムコード領域全

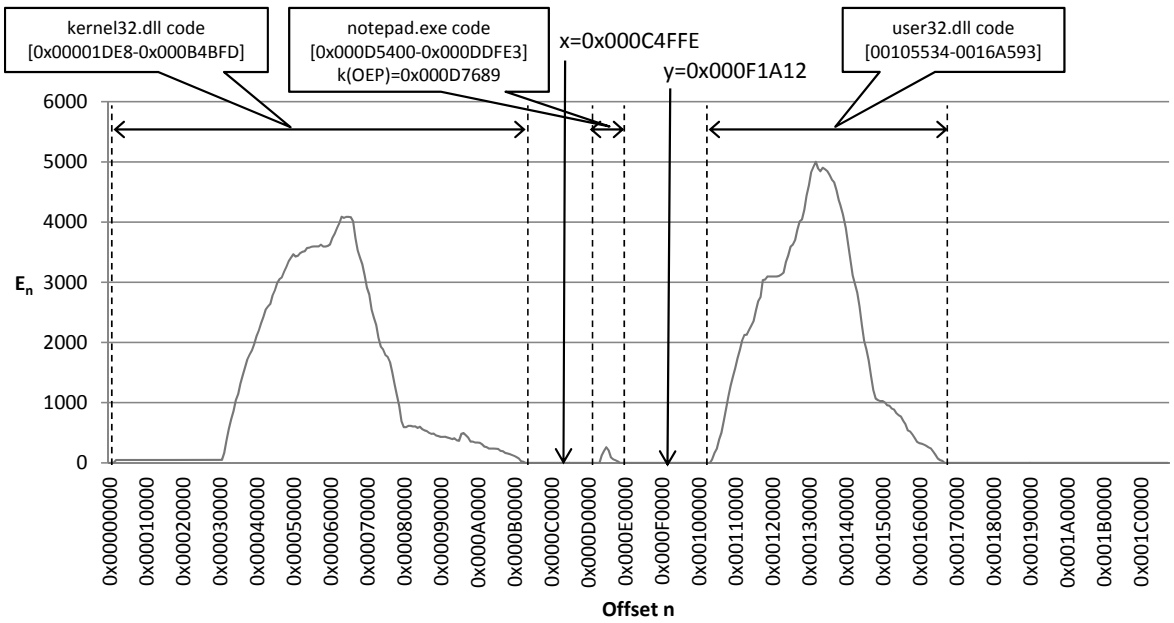

図 2.22　3 つの連続するメモリイメージにおける分岐区域数の期待値

体を，他の dll のプログラムコード領域が混入することなく，取得できることがわかる．

次に，p^* や q^* 付近の分岐区域数の期待値 E_n とアセンブリコードを示すことで，抽出対象の先頭を $x = \frac{p^*+q^*}{2}$ とすることの有効性について述べる．まず，notepad.exe の先頭付近に関する分岐区域の期待値 E_n を図 2.23 に示す．notepad.exe のプログラムコード領域は，0x000D5400 からはじまっている．一方で，$E_n > \epsilon$ となる境界 p^* は，プログラムコード領域の内部である 0x00D5404 となっている．以下は 0x00005400 付近のアセンブリである．

```
0x00005400                nop
0x00005401                nop
0x00005402                nop
0x00005403                nop
0x00005404                nop
0x00005405                mov     edi, edi
0x00005407                push    ebp
```

0x000D5405 は WinMain 関数 [54] の先頭であり，WinMainCRTStartup [55] と呼ばれる

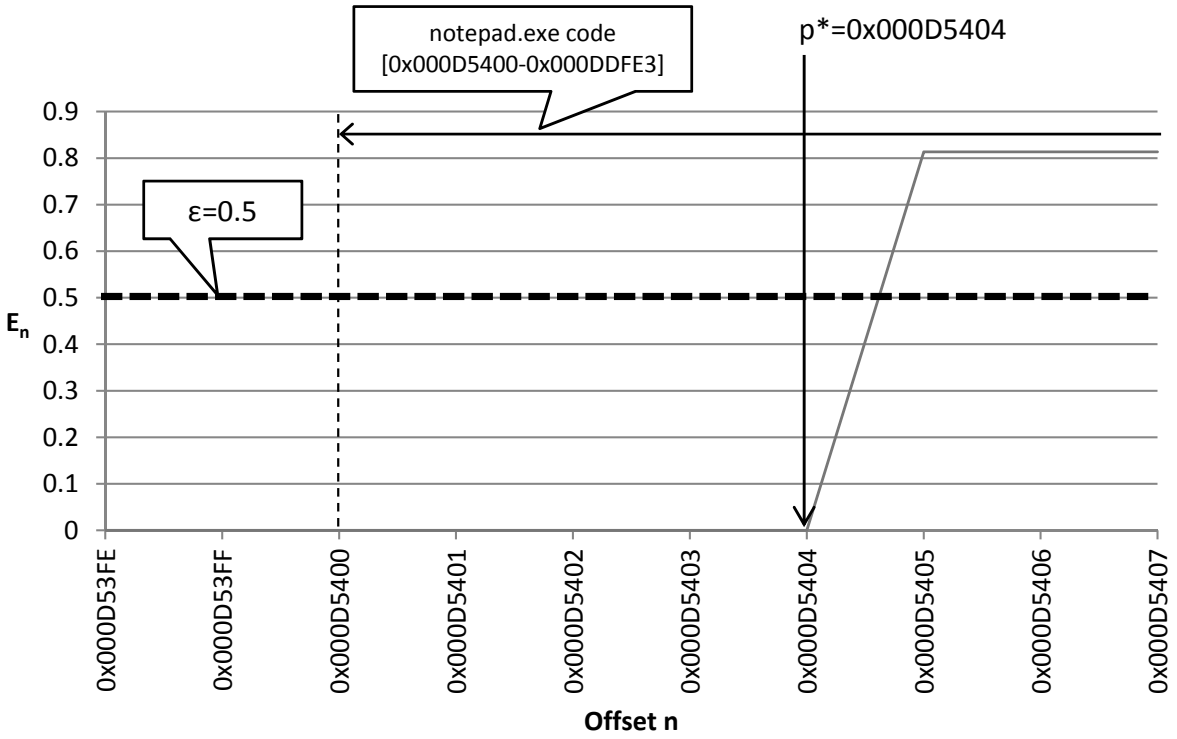

図 2.23　notepad.exe のコード領域の先頭付近における E_n

C ランタイムライブラリの初期化ルーチンから呼ばれている．このため 0x000D5405 における E_n は，1 に近い値となっている．しかし，0x00D5400 から 0x00D5404 までの 5 バイトは WinMain 関数に対するホットパッチ機構 [56] のための NOP 命令であり，ホットパッチが適用されていない場合は，0x00D5400 のアドレスは使われないことになる．このため 0x00D5400 からの 5 バイト分は，E_n の数値が 0 に近い値となった．つまり p^* を抽出対象の先頭としてしまうと，オリジナルのプログラムコード領域が欠けてしまうことになる．

さらに kernel32.dll の終点付近における E_n を図 2.24 に示す．kernel32.dll のプログラムコード領域は，0x000B4BFD までとなっている．しかしながら，$E_n > \epsilon$ となる境界 q^* は，プログラムコード領域の内部である 0x00B4BF9 となっている．実際の 0x000B4BF8 付近のアセンブリコードは次のとおりである．

```
0x000B4BF8                  pop     edi
0x000B4BF9                  pop     esi
0x000B4BFA                  leave
```

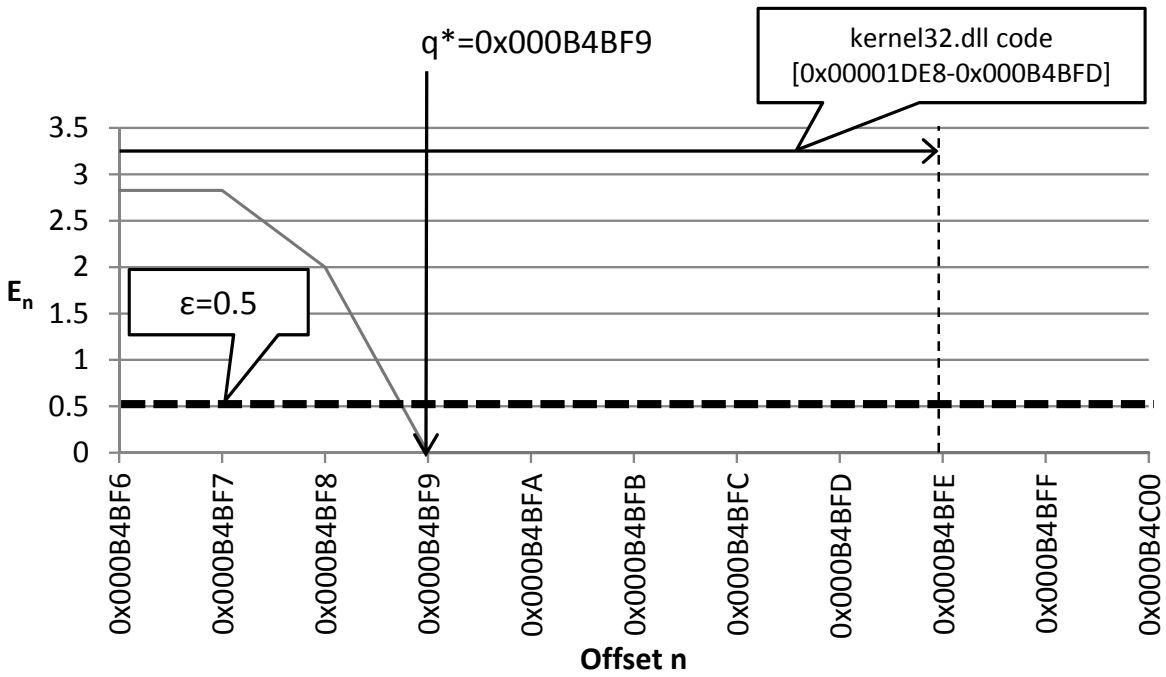

図 2.24 3 つの連続するメモリイメージにおける分岐区域数の期待値

```
0x000B4BFB                    retn    8
```

この関数は，kernel32.dll の一番最後の関数であり，0x000B4BF8 から retn までの命令は，レジスタの復元やスタックフレームの巻き戻しなど，関数のエピローグ処理となっている．こうした関数のエピローグ処理は，途中から実行されることがないため分岐先にもなりえず，E_n が 0 に近い値となった．これは，q^* を抽出対象の先頭としてしまうと，kernel32.dll のプログラムコードの一部が抽出対象に混入することを意味する．

提案手法ではこうした現象を鑑み，抽出対象の先頭を $x = \frac{p^* + q^*}{2}$ としており，実験ではプログラムコード領域の欠損や，他のプログラムコードの混入が生じていないことを確認できた．また抽出対象の終点についても同様であった．

ここではさらに，提案手法においてプログラムコード領域の識別に失敗する場合について考察する．典型的なものとして次の二つのケースが考えられる．

1. OEP のオフセットを k としたとき，E_k が ϵ 未満である．
2. パッカーが変更可能なメモリ領域（PE ヘッダ等）に相対分岐命令が書き込まれている．

2.7.1 章の実験結果として示したように，プログラムコード領域の境界付近では，たとえプログラムコード領域内であっても，E_n の値が ϵ を下回るケースが存在する．こうした箇所が OEP になった場合は，$E_k \geq \epsilon$ を前提とする本提案手法を適用することはできない．たとえば，まったく条件分岐やループが存在しない関数がプログラムコード領域の最終端に位置し，その関数の先頭が OEP となっている場合は，$E_k < \epsilon$ となりうる．こうした場合は，OEP から最も近い $E_n \geq \epsilon$ となる点を探し，そこから本提案手法を適用する等の対処法が考えられる．また OEP の特定手法の中には，OEP 以外にもオリジナルのプログラムコード内のアドレスを抽出できるものがある．こうしたアドレスの中で E_n が ϵ 以上となるアドレスを探し，それをもとに本提案手法を利用することもできる．

また図 2.22 中の $E_k < \epsilon$ となっている領域に，相対分岐命令を書き込まれた場合には，抽出対象に他のプログラムコード領域が混入する恐れがある．たとえば，PE ヘッダ等はロード済みのオリジナルのプログラムにとって不要であり，パッカーが自由にメモリ内容を変更することができる．もし，パッカーがこの領域に相対分岐命令を書き込んだ場合，提案手法における E_n は ϵ 以上になる状況も想定される．この対処としては，相対分岐命令の密度に制限を設けることや，閾値 ϵ の値を増やすことなどが考えられる．また，オリジナルのプログラムコード領域は改変されないため，オリジナルのプログラムコード領域における相対分岐命令が，パッカーによって変更可能な領域を指すことはないと考えられる．こうしたプログラムコード領域の特徴を用いることも，プログラムコード領域識別の一助となるであろう．

2.7.2　オリジナルコード特定モジュール

本章では，CCC DATAset 2008 におけるマルウェア検体を提案手法でアンパックした結果について述べる．

アプリケーション	Firefox 3.0.1
コンパイラ	Microsoft C++ Compiler Ver. 15.00.21022.08

表 2.2　学習用プログラムとコンパイル環境

提案手法におけるコンパイラ出力コードとしての尤度を算出するために必要な HMM のモデルパラメータに関しては，表 2.2 によって生成された xul.dll [*6]のアセンブリコードを学習データとし，命令状態における出力確率および状態遷移確率を算出した．またデータ状態における出力確率は，プログラムによって大きく異なると考え全シンボルに関して等確率 $\frac{1}{256}$ とした．

実験環境は表 2.3 の通りであり，マルウェアの検体をゲスト OS 上でアンパックした．

ホスト OS	Windows XP SP3
ホスト CPU	Intel Xeon 2.66GHz
ホストメモリ	3GB
仮想マシン	VMware Server 1.0.4
ゲスト OS	Windows XP SP1
ゲスト CPU 数	1
ゲストメモリ	512MB

表 2.3 実験環境

図 2.25 は，オリジナルコードの候補を抽出した後に，各候補に関してコンパイラ出力コードモデルとヌルモデルの尤度比を算出した結果である．各点はオリジナルコードの候補であり，全部で 230 程度の候補が抽出された．横軸はその候補が抽出された時間（秒），縦軸はその候補の尤度比の対数をとった値となっている．当該検体は起動した後に三回の再起動を繰り返すため，オリジナルコードの候補を表すマーカーをプロセス毎に変えている．図 2.25 において対数尤度比が 0 を超えた点は，コンパイラ出力コードとして尤もらしいことを表している．

当該検体において最初に対数尤度比が 0 を超えた候補は図 2.25 における (*1) であり，その領域を逆アセンブルした結果，ボットとしてのプログラムコードが確認された．またその他の対数尤度比が 0 を超える領域は，全て (*1) と同じくボットとしてのプログラムコードを持ち，対数尤度比が 0 以下の領域は，全てアンパック処理中のルーチンであることも確認された．

[*6] Firefox において最も大きな実行バイナリ (約 8.5MB) であり，学習データとして十分なサイズである．

図 2.25 からは他にもいくつかの興味深い結果を読み取れる．まず一番目のプロセス（PID=364H）には，全くオリジナルコードが出現していないことが分かる．このマルウェアに用いられたパッカーによりパッキングされた実行ファイルは，デバッガによる追跡等を困難にするため，アンパック前に一度再起動を行うプログラムに変更されると考えられる．また二番目のプロセスでオリジナルコードが出現した後に，再びプロセスが再起動しているが，これはマルウェア自身がシステムディレクトリへコピーされ，そこから再起動されているためである．つまり一度目の再起動はパッカーによりもたらされたものであり，二度目の再起動はアンパックされたマルウェアによりもたらされたものであることが分かる．その後，システムディレクトリ内で三番目のプロセスが起動し，さらにパッカーの機能により四番目のプロセスが起動している．実際，この四番目のプロセスではじめてボットとしての本来の活動が開始されていることも別途確認できた．

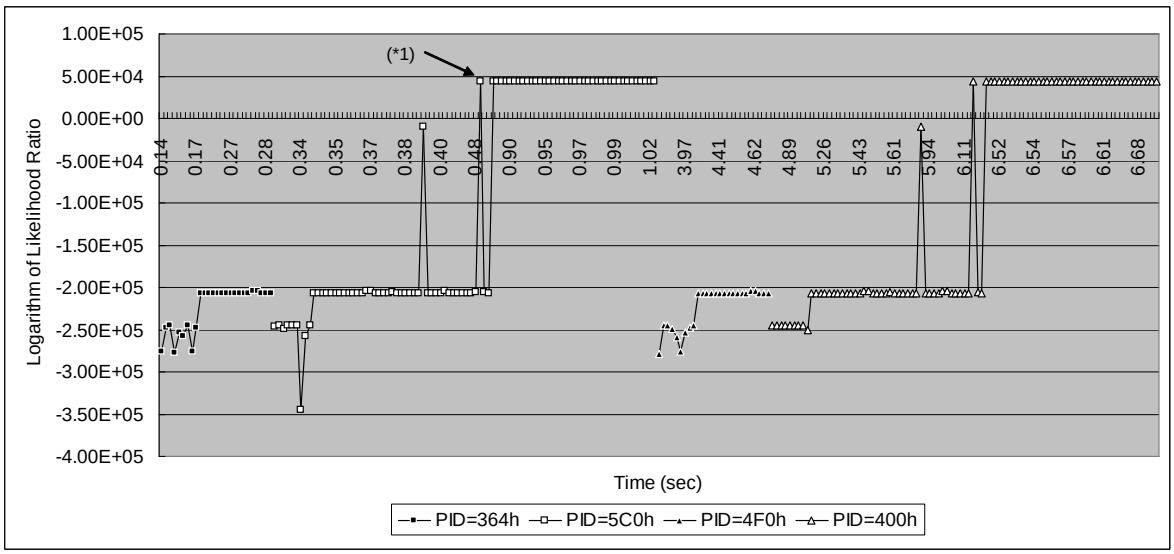

図 2.25　オリジナルコード候補の対数尤度比

2.8　むすび

ランタイムパッカーによる隠蔽処理は，リバースエンジニアリングや，プログラムコードに基づくマルウェアの分類を困難にする．このため，パッキングされたマルウェアからオリジナルのプログラムコードを抽出するアンパック手法の研究開発が進んでいる．しかしながら，汎用的なアンパックに関する従来研究では，二つの課題が存在していた．一つ

目の課題は，従来研究では主に OEP の特定方法に焦点が当てられており，抽出すべきプログラムコード領域がどこからどこまでの範囲なのかについて，特に言及されてこなかった点にある．本章では，まず OEP を含む連続するコミット済みメモリ領域を抽出し，このバイト列における相対アドレス指定の分岐命令に着目することで，OEP を含むひとつのプログラムコード領域全体を抽出する手法を提案した．また，本提案手法における実験では，対象となるプログラムコード領域の前後に，他のプログラムコード領域を含むメモリイメージが接している場合にも，分岐区域数の期待値をもとに，OEP を含むプログラムコード領域だけを識別できることを示した．また二つ目の課題は，マルウェアが多重にパックされている場合に，各層のアンパックが完了するたびに，オリジナルコードの候補が抽出されてしまっていた．これに対して本研究では，得られたオリジナルコードの候補に関して，確率モデルによってコンパイラ出力コードの尤もらしさを算出し，オリジナルコードを特定できる新たなアンパック手法を提案した．

同時に本研究では，TLB に依存しないメモリ監視モジュールを開発し，上記二つの提案手法を組み合わせることで，自動アンパックシステムを構築した．これにより，パックされたマルウェアに関しても，自動的にマルウェアのオリジナルコードを抽出することが可能になった．

第 3 章

逆アセンブル

3.1　はじめに

ソフトウェアの内部構造を理解するリバースエンジニアリングは，プログラムのデバッグや脆弱性の発見 [57–59]，マルウェア解析 [60–62] において根幹の技術となる．特にマルウェア解析においては，その対象となるプログラムはバイナリ形式でしか入手できない場合が多く，こうしたケースでは，バイナリ形式のプログラムを逆アセンブルする必要が生じる [63]．しかしながら，前章により得られるマルウェアのオリジナルコードには，逆アセンブルのヒントになりうる実行ファイルのヘッダ情報が含まれていない．またほとんどのマルウェアに関して，そのデバッグシンボル情報を入手することもまた困難である．さらに IA-32 の機械語命令は可変長であり，また Windows 向けコンパイラは，実行バイナリのサイズを削減するために，機械語命令とデータを混在させる傾向にある．このため，ヒントが無い状態での正確な逆アセンブルは難しい問題となる．

ここでは，まず従来の逆アセンブル手法とその限界を整理する．その後，確率的なアプローチを用いた新たな逆アセンブル手法を提案し，その有効性を示す．また本章では，インテル・アーキテクチャ (IA-32) の Windows で動作するプログラムに焦点を当て，その逆アセンブル手法について述べる．

3.2　従来の逆アセンブル手法

逆アセンブル手法は，動的なアプローチと静的なアプローチの二つに分けられる．

動的なアプローチは，逆アセンブル対象のプログラムを実際に動作させ，実行された機

械語命令を記録することで，機械語命令を特定する．このアプローチの短所は，解析中には偶然実行されなかった機械語命令が，機械語命令として抽出されないことにある．これは，攻撃者からの指令なしには動作しないボット [64] 等を逆アセンブル対象とした場合，深刻な問題となる．

一方静的なアプローチ [65,66] は，その精度は動的なアプローチと比べ劣るものの，プログラムコード全体を対象とする．ここでは，静的なアプローチに焦点を当て説明を続ける．静的なアプローチは主に二つの手法，もしくはこれらを組み合わせた手法 [67] に分類できる．Linear sweep と呼ばれる最初のアプローチは，プログラムを先頭から逆アセンブルし，命令として解釈できない部分はデータと解釈，次バイトから逆アセンブルを行い，この作業を終端まで繰り返す．GNU objdump [68] ではこの手法が使われている．このアプローチは，データ部も命令として解釈可能であれば，命令部として逆アセンブルしてしまうという欠点がある．また一度命令の先頭を見誤ると，連鎖的に後続のバイト列を別命令として逆アセンブルし，真の命令列とは異なる命令列が多数出力されることになる．

二つめの Recursive traversal [66] と呼ばれる手法は，プログラムの制御フローを辿ることで，データに関する解釈ミスを解決している．このアプローチは，プログラムのエントリポイントや頻出命令列とマッチする箇所を命令列の先頭とし，Linear sweep と同様に後続の命令を逆アセンブルしていき，無条件分岐命令（jmp 命令や ret 命令）が出現した時点で，逆アセンブルを止める．このとき，分岐命令の分岐先を新たな命令列の先頭として再帰的に逆アセンブルを進めていく．Recursive traversal の長所は，データと機械語命令が混在している場合にも，データを逆アセンブルすることを避けることができる点である．一方，分岐先を決定できない場合（間接分岐命令である場合）には，その分岐先が機械語命令として解釈できないという欠点もある．

こうした Recursive traversal の問題に対していくつか解決策が提案されている．Speculative disassembly [69] は，Recursive traversal で到達できなかった領域に対して，Linear sweep を用いることで，Linear sweep の精度を向上させている．BIRD [67] は，Recursive traversal と上述の動的なアプローチを組み合わせることで，間接分岐命令の分岐先の特定を試みる．また Exhaustive disassembly [65] は，Recursive traversal に基づき，考えられる全てのベーシックブロック境界を抽出し，その中からいくつかのヒューリスティックな想定をもとに，正確なベーシックブロックの境界の特定を試みる．

ただこうした従来技術やその組み合わせ技術には，依然として以下のいずれかのケースに対して課題が残っている．

- 機械語命令とデータが混在するケース
  Linear sweep や Speculative disassembly はデータ部分を誤って機械語命令として解釈してしまう．
- 分岐命令の分岐先を特定できないケース
  Recursive traversal は，分岐先を機械語命令として解釈できない．また BIRD も，ボットのようなプログラムの動作を網羅することが困難なケースでは，Recursive traversal と変わらない精度となってしまう．

次節以降ではこうした課題を解決するために，分岐命令に頼らずに機械語命令とデータを識別する確率的逆アセンブル手法を提案する．

3.3 提案手法

本章で提案する確率的逆アセンブル手法は，コンパイラが出力する機械語命令の傾向や，機械語命令からデータ，データから機械語命令への切り替わり頻度などをパラメータとし，そのパラメータに基づき確率的に最も尤もらしい逆アセンブル結果を出力する．これには，実行ファイルの確率モデル（図 2.5）を利用する．ここでは以下の記号を用いて説明する．

- N: プログラムコードのバイト数.
- $X = x_1 \cdots x_N$: プログラムコードのバイト列．ここでは $x_1 \cdots x_n$ を x_1^n とも表記する．
- $S = \{S_B, S_I, S_D, S_F, S_M\}$: HMM における状態の集合． S_B はプログラムコードの開始，S_I は機械語命令の先頭，S_D はデータ，S_F はプログラムコードの終端，S_M は機械語命令の 2 バイト目以降を表す．
- $L = l_1 \cdots l_N$: $x_i (1 \leq i \leq N)$ に割り当てられる状態系列． 便宜上，$l_0 = S_B, l_{N+1} = S_F$ とする．
- $w_{s,n}$: 状態 s において x_n を先頭とする単語． $w_{s,n}$ の長さ（バイト数）を $|w_{s,n}|$ と

すると，$w_{s,n}$ はバイト列 $x_n^{n+|w_{s,n}|-1}$ を意味する．

- $a_{s,s'}$: 状態 s から状態 s' へ遷移する確率．
- $b_{s,n}$: 状態 s で $w_{s,n}$ を出力する確率．

Forward アルゴリズムでは，モデルパラメータ θ（$a_{s,s'}$ と $b_{s,n}$）が与えられたときの，X の出力確率 $P(X|\theta)$ を算出していた．一方，逆アセンブル手法では Viterbi アルゴリズム [70] を用いることで，モデルパラメータ θ が与えられた際に，対象バイナリの出力と状態遷移（機械語命令・データのラベル付け）が最も高い確率となる状態系列 $\underset{L}{\operatorname{argmax}} P(X, L|\theta)$ を求める．

ここで，l_1^{n-1} の状態遷移系列で x_1^{n-1} を出力し，x_n に割り当てられる状態が s となる（つまり $l_n = s$ となる）確率のうち，最大となる値を $V(n, s)$ とする．この $V(n, s)$ は，以下の漸化式で表せる．

$$
\begin{aligned}
&V(n,s) = \max P(x_1^{n-1}, l_1^{n-1}, l_n = s|\theta) \\
&= \max_{s',n':n'+|w_{s',n'}|=n-1} P(x_1^{n'-1}, l_1^{n'-1}, w_{s',n'}, l_{n'} = s', l_n = s|\theta) \\
&= \max_{s',n':n'+|w_{s',n'}|=n-1} P(x_1^{n'-1}, l_1^{n'-1}, l_{n'} = s'|\theta) P(w_{s',n'}, l_n = s|x_1^{n'-1}, l_1^{n'-1}, l_{n'} = s', \theta) \\
&\approx \max_{s',n':n'+|w_{s',n'}|=n-1} V(n', s') P(w_{s',n'}|l_{n'} = s', \theta) P(l_n = s|l_{n'} = s', \theta) \\
&\approx \max_{s',n':n'+|w_{s',n'}|=n-1} V(n', s') b_{s',n'} a_{s',s}
\end{aligned}
$$

これは Forward アルゴリズムと同様，$V(1, S_I), V(1, S_D), V(2, S_I), \ldots$ を順に算出していき，最終的に求まる $V(N+1, S_F)$ が $\max P(X, L|\theta)$ となる．また，この最大値を探す際に，どの s', n' が最大となったかを記録しておくことで，$\underset{L}{\operatorname{argmax}} P(X, L|\theta)$ も算出することが可能となる．こうして求まった $\underset{L}{\operatorname{argmax}} P(X, L|\theta)$ が，モデルパラメータ θ が与えられたときの，最も尤もらしい逆アセンブル結果となる．例えばバイト列 $X = (0xEB, 0x01, 0xFF, 0xC3)$ が与えられたときは，図 3.1 に示すように $V(1, S_I), V(1, S_D), V(2, S_I), \ldots$ を順に算出していき，$V(5, S_F)$ が $\max P(X, L|\theta)$ となる．また，図 3.1 において各単語を結ぶ実線の矢印が，採用された最大値とすると，$V(5, S_F)$ から順に $V(4, S_I), V(3, S_D), V(1, S_I)$ と辿ることで，$\underset{L}{\operatorname{argmax}} P(X, L|\theta)$ を抽出することができる．

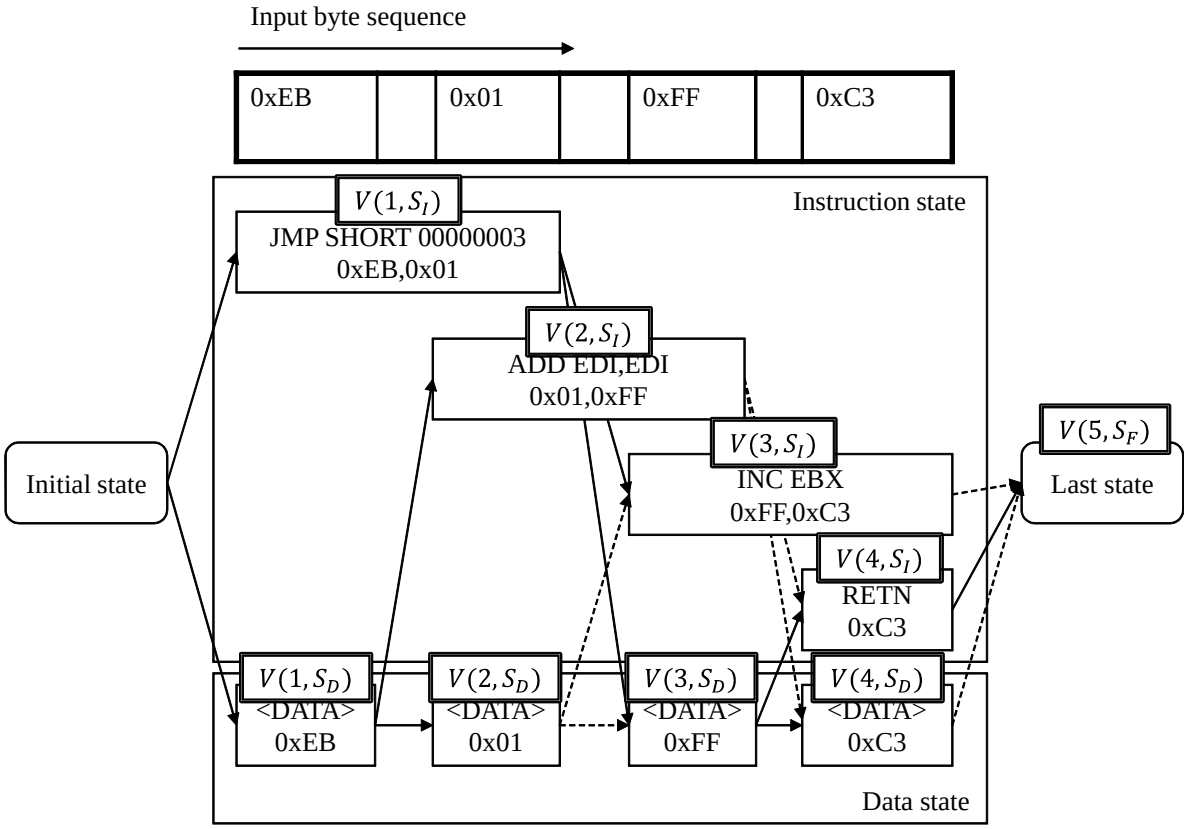

図 3.1 Viterbi アルゴリズムの処理

3.4 実験

提案手法を評価するため，ここでは 3 つのデータセット (A)(B)(C) を用いる．(A) と (B) は Visual C++ 9.0 Express Edition [71] でコンパイルされた 8 つのアプリケーション [72–79] から構成される．(A) と (B) の違いはコンパイル時のオプションであり，(A) は各アプリケーションパッケージに含まれるデフォルトオプション，(B) は/Ox（最大限の最適化）およびホットパッチ関連のコンパイラオプションを追加した状態でコンパイルした．(C) は Popov [80] らの難読化ツールにより難読化された SPECint-2000 ベンチマーク [81] に含まれる 10 のアプリケーションである．特に断らない限り提案手法の学習データセットとしては，(A)(B) の場合は評価対象以外の 7 アプリケーション，(C) の場合は評価対象以外の 9 アプリケーションを用いる．また，提案手法 (PD) の比較対象としてリニアスウィープアルゴリズムを実装した逆アセンブラ（LS）と，商用逆アセンブラである

IDA Pro4.9(IP) [82] を用意した．

3.4.1　精度

各逆アセンブル手法の精度評価には，以下の式により算出される MCC（Matthews correlation coefficient）[83] を用いた．

$$MCC = \frac{T_p T_n - F_p F_n}{\sqrt{(T_p + F_p)(T_p + F_n)(T_n + F_p)(T_n + F_n)}}.$$

この式の，T_p，T_n はそれぞれ True positive の数と True negative の数であり，F_p，F_n はそれぞれ False positive の数と False negative の数である．この MCC は，0% がランダムな予測，100% が完全に機械語命令とデータを判別できたことを意味する．また，ここで言う True positive はある機械語命令の先頭を正しく予測できたことを意味し，True negative はある機械語命令の先頭以外（機械語命令の 2 バイト目以降やデータ）を正しく予測できたことを意味する．同様に False positive は機械語命令の先頭ではない箇所を機械語命令の先頭と判断したことを意味し，False negative は機械語命令の先頭を予測できなかったことを意味する．

以下に各データセットの結果を示す．データセット (A) では表 3.1 に示す通り，IDA Pro

Program	LS	IP	PD
SpeakFreely7.2	96.47	99.87	99.83
analog 6.0	97.20	99.79	99.88
lame 3.96.1	95.26	99.84	99.95
make 3.81	93.75	99.73	99.85
ncftp 3.2.1	93.12	97.92	99.91
xpdf 3.00 (pdftops.exe)	93.54	99.14	99.90
putty for Win	98.94	99.99	99.92
TightVNC 1.3.9 (vncviewer.exe)	93.36	99.71	99.89
Mean	95.44	99.53	99.90

表 3.1　MCC (%) of dataset (A).

および提案手法で 99% を超える高い精度が得られた一方，リニアスウィープは 95% にとどまった．これはデフォルトのコンパイルオプションであっても，データ部分が命令中に混在しているためである．

Program	LS	IP	PD
SpeakFreely7.2	94.22	99.86	99.95
analog 6.0	90.31	89.16	99.91
lame 3.96.1	91.83	97.05	99.96
make 3.81	90.86	97.02	99.94
ncftp 3.2.1	89.85	92.69	99.94
xpdf 3.00 (pdftops.exe)	90.13	90.92	99.97
putty for Win	89.24	82.89	99.89
TightVNC 1.3.9 (vncviewer.exe)	90.31	94.11	99.95
Mean	90.58	91.34	99.93

表 3.2 MCC (%) of dataset (B).

また表 3.2 のデータセット B では，IDA Pro も大きく精度を落としている．IDA Pro は頻出命令をパターンとして事前に持っており，この頻出命令からリカーシブトラバーサル法で逆アセンブルしているため，最適化が進み頻出命令のパターンが保たれなくなったことにより，精度が低下したと考えられる．これに対して提案手法ではデータセット (A) とほぼ同じ精度を保てていることが分かる．

データセット (C) は Popov らにより難読化された実行バイナリである．この難読化手法は，対象プログラムのアセンブリコードを入力として，通常の分岐命令を使う代わりに，例外発生時のシグナルハンドリング機構を使うようにプログラムを変更している．最近のマルウェアに対する難読化は，ランタイムパッカーにより行われるのが一般的であり，このようにオリジナルコードを変異させたデータセットによる評価は，必ずしも本研究の趣旨に沿ったものではない．しかしながら，他の逆アセンブル手法と本提案手法を対比するうえで，各手法の特性を明らかにするには有用と考え，評価対象とした．表 3.3 が示す通り，リカーシブトラバーサルを用いる IDA Pro はデータセット (A)(B) と比べ，極

Program	LS	IP	PD
bzip2	34.78	48.08	91.72
crafty	38.66	50.64	91.15
gap	35.85	51.53	89.26
gcc	32.00	48.70	90.11
gzip	34.45	48.61	91.61
mcf	33.61	45.88	88.68
parser	33.93	46.45	91.45
perlbmk	32.92	49.34	91.87
twolf	37.64	48.50	92.12
vortex	40.97	46.61	93.68
vpr	37.03	49.63	91.24
Mean	35.22	48.69	91.13

表 3.3　MCC (%) of dataset (C).

端に精度が落ちている．これは Popov らによる難読化が，分岐命令を例外発生命令に変換している一方で，リカーシブトラバーサルが分岐命令に強く依存していることが原因である．これに対し，提案手法は分岐命令には全く依存していないため安定して 90% 前後の精度を達成できたと考えられる．表 3.4 は，データセット (C) に対する各逆アセンブル手法の True/False negative/positive の割合である．これからも分かるように，False

Program	Linear sweep	IDA Pro	Our tool
T_p	12.57	10.22	19.64
T_n	63.21	73.79	77.29
F_p	14.16	3.59	0.09
F_n	10.05	12.40	2.99

表 3.4　Ratios (%) of true and false positives/negatives to all obfuscated bytes.

positive/negative のいずれの割合においても，提案手法は従来技術より優れていることが分かる．

ここでデータセット (C) に関して，さらに詳しい実験結果について述べる．Linn と

Dabray は，難読化ツールの評価指標として Confusion factor(CF) を定義した [48]．ここで，正しい機械語命令の集合を V，逆アセンブラが出力した機械語命令の集合を O としたとき，CF は以下のように表現できる．

$$CF = \frac{|V - O|}{|V|}.$$

一方 Kruegel ら [65] は，逆アセンブラの評価指標 DA を $DA = 1 - CF$ と定義した．この DA によるデータセット (C) の評価結果は，表 3.5 となった．

Program	Linear sweep	IDA Pro	Our tool
bzip2	55.03	43.96	87.61
crafty	58.20	44.48	86.70
gap	56.25	48.29	84.00
gcc	53.29	44.28	85.49
gzip	54.94	43.69	87.47
mcf	54.21	44.18	83.02
parser	54.66	46.34	87.21
perlbmk	53.96	43.76	87.75
twolf	57.37	45.60	88.19
vortex	59.19	46.56	90.62
vpr	56.92	45.68	86.88
Geom.mean	55.79	45.14	86.79

表 3.5 Disassembly accuracy (%).

リニアスウィープと IDA Pro がそれぞれ約 55%，約 45% である一方，提案手法は約 86% と高い精度であることが分かる．また Popov ら [80] は，最先端の逆アセンブル手法である Kruegel らの Exhaustive disassembler [65] でさえも，データセット (C) に関しての DA は，50% を下回ると報告している．

最後に，極端に学習データが少ない状態での提案手法の実験結果について述べる．表 3.6 と表 3.7 は，ひとつのプログラムだけを学習データとした場合の評価結果である．左端の列は逆アセンブル対象のプログラムを意味し，上端の行は学習データとしたプログラムを意味する．

Program	bzip2	crafty	gap	gcc	gzip	mcf
bzip2	-	86.85	86.60	82.21	88.67	86.81
crafty	83.01	-	85.32	78.90	84.22	83.10
gap	80.01	84.85	-	79.30	79.63	78.19
gcc	79.21	84.25	85.60	-	84.31	74.34
gzip	88.13	87.14	86.21	82.48	-	86.98
mcf	72.03	83.25	79.15	70.72	73.21	-
parser	86.53	84.28	83.68	80.37	86.23	83.50
perlbmk	87.02	86.98	87.82	85.22	86.58	85.03
twolf	85.06	87.30	84.38	81.65	86.42	84.07
vortex	86.87	88.64	85.91	88.09	87.07	85.61
vpr	85.60	80.56	81.58	80.99	83.92	81.27
Geom.mean	83.21	85.38	84.59	80.87	83.91	82.80

表 3.6　Disassembly accuracy (%) for one-program training.

Program	parser	perlbmk	twolf	vortex	vpr
bzip2	86.99	84.14	85.94	79.33	86.75
crafty	81.43	80.30	83.69	74.71	81.01
gap	78.17	79.37	77.03	72.94	78.31
gcc	75.72	78.88	75.19	76.94	72.48
gzip	86.46	84.27	85.90	79.55	86.26
mcf	69.69	74.46	74.17	67.83	69.90
parser	-	82.01	85.13	76.02	85.78
perlbmk	85.76	-	83.64	78.56	85.62
twolf	86.93	82.60	-	77.61	85.84
vortex	85.87	87.37	87.77	-	86.95
vpr	85.88	82.81	85.80	77.72	-
Geom.mean	82.09	81.55	82.29	76.04	81.66

表 3.7　Disassembly accuracy (%) for one-program training.

この結果を注意深く見ると，crafty を学習データとしたときの精度は 85.38% と非常に高い一方で，vortex を学習データとしたときは 76.04% と精度が低くなっている．これまでの他の実験において，様々な種類の機械語命令が用いられているプログラムを学習データとすると，逆アセンブルの精度が向上することが観測されている．このため，なるべく多くのコンパイラにより学習用データセットを作成しておくことが望まれる．

3.4.2 処理時間

ここでは，提案手法において逆アセンブルに要する時間を計測した結果を示す．計測には，CPU が 3.2-GHz Pentium IV，メモリサイズが 2GB の環境を用いた．表 3.8 は，逆アセンブル対象のプログラムサイズ，逆アセンブルに要した時間，そしてスループットを示している．提案手法は，Viterbi アルゴリズムをベースとしているため，プログラムサイズ

Program	Size (bytes)	Time (s)	Throughput (KB/s)
mcf	624,008	5.04	120.84
bzip2	694,281	5.62	120.54
gzip	704,992	5.69	121.05
vpr	805,892	6.53	120.57
parser	838,332	6.76	121.04
crafty	908,242	7.35	120.73
twolf	936,112	7.58	120.56
gap	1,356,808	10.92	121.32
vortex	1,518,716	12.27	120.83
perlbmk	1,612,309	12.97	121.38
gcc	3,051,833	24.63	120.99
Mean	1,186,502	9.58	120.96

表 3.8 Disassembly times.

を N とした場合，その計算量は $O(N)$ である．このため，表 3.8 におけるスループットは，逆アセンブル対象のプログラムサイズに依らず，一定（約 120KB/s）に保たれている．

3.5　むすび

一般的にパックされたマルウェアのオリジナルコードには，逆アセンブルのヒントとなる PE ヘッダは存在せず，またデバッグシンボル情報の入手も困難である．そのため，機械語命令とデータが混在する傾向にある Windows 向けのプログラムコードに関して，完全な逆アセンブル結果を得ることは非常に難しい．そこで本章では，マルウェアによく使われるコンパイラが出力するプログラムコードを学習しておくことで，最も尤もらしい逆アセンブル結果を出力可能な手法を提案した．様々なデータセットに関する実験では，Linear sweep や Exhaustive disassembly，商用逆アセンブラと比較し，提案手法が安定して精度のよい逆アセンブル結果を出力できることを示した．

第 4 章

類似度算出

4.1　はじめに

マルウェアのプログラムコードに基づく分類手法は，二つのプログラムコードの類似度（もしくは距離）を，どう算出するかによって特徴づけられる．本章では，プログラムコードの類似度を算出する従来研究とその課題について述べ，自動マルウェア分類に適した手法を提案する．提案手法の有効性を評価した結果は，第 5 章にて自動マルウェア分類システムの説明とあわせて述べる．

4.2　従来研究とその課題

マルウェアのプログラムコードに基づく分類に関する従来研究は，プログラムコードの特徴として何に着目しているかで整理することができる．

N-gram に基づくアプローチ [16] では，まずマルウェアの逆アセンブル結果からオペレーションコード列を抽出し，連続する N 個のオペレーションコード列（N-gram）がそれぞれ何回出現したかを特徴ベクトルとする．この特徴ベクトルをマルウェアの特徴とし，マルウェアの非類似度をコサイン距離として定義する．N-gram は N 個のオペレーションコード列の順序が保存され，命令単位での並べ替えが生じると異なる N-gram として扱われてしまう．そのため，N-perm と呼ばれるオペレーションコード列の順序関係を考慮しない特徴を利用する方法も提案されている．N-gram/N-perm を利用した手法の特長は，ベクトル間のコサイン距離としてマルウェアの類似度が定義されるため非常に高速に処理可能なことである．

他にもプログラムのベーシック・ブロックに着目した手法 [17] も提案されている．ベーシック・ブロックとは，分岐命令・分岐先命令を含まない連続した命令列である．まずマルウェアの逆アセンブル結果から，プログラムコードをベーシック・ブロックに分割する．そしてベーシックブロック単位でレーベンシュタイン距離を算出し，それをマルウェアの非類似度とする．またベーシックブロック単位での並べ替えに対応するために，転置インデックスまたはブルームフィルタを用いる手法も提案されている．具体的には，まずサンプルとなるマルウェアの全ベーシックブロックをデータベースへ格納する．その後，対象となるマルウェアのベーシックブロックがデータベース内に存在したか否かのビットベクトルを求め，それをマルウェアの特徴とする．この手法もまた高速に類似度を算出することが可能である．

一方でプログラムのコールツリーを特徴とし，類似性を求める手法 [18] も存在する．コールツリーは，ソースコード上のプログラム構造が反映されるため，コンパイラの変更に強いと言われている．しかしながら，ツリー構造の厳密な比較は計算コストが非常に高い．このため，マルウェアの IAT(Import Address Table) を再構築した状態で，コールツリーの葉となる外部関数から近似的にコールツリーの類似度を算出する等の工夫が必要となる．

このようにプログラムコードに基づく分類技術に関して様々な研究がなされており，N-gram/N-perm やベーシックブロック，コールツリー等の各々の着眼点に関する類似度を算出することができる．しかし分類の全自動化や，マルウェアが備える脅威の全容を把握するという観点では，いくつかの課題が残る．

例えばベーシックブロックに依る手法では，マルウェアのプログラムコードをベーシックブロック単位に区切る必要がある．一方，C++ のようなポリモーフィズムを備えるプログラミング言語により開発されたプログラムでは，クラス継承を実現するために関数ポインタを利用した関数呼び出しが使われる．このように動的に呼び出し先が変化する命令を多用されると，分岐先命令を特定することができずベーシックブロックに区切ることも困難になる．コールツリーの構築もまた同様であり，現状では人手により正確な情報を構築する必要がある．

また別の課題として，N-gram/N-perm やコールツリーに依る手法では，マルウェア間の距離が近かったとしても，偶然 N-gram/N-perm やコールツリーが似ていただけで，全く異

なるマルウェアである可能性は否定できない．つまりマルウェアが備える脅威を完全に解明するには，結局全てのマルウェアの解析が必要となってしまう．他にも N-gram/N-perm による手法では，N-gram/N-perm の出現回数という一種の統計情報により類似度が定義されるため，実際に変化のあった場所を抽出することは難しいといった課題もある．

前述の課題を解決するため，我々は 2 つのマルウェアが与えられた際に，機械語命令を単位として共通する機械語命令列およびその数を高速に算出し，類似性を求める手法を提案する．本手法で必要となるのは，マルウェアのオリジナルコードの逆アセンブル結果のみであり，全自動化の足枷となるコールツリーやベーシックブロックの分割，IAT の再構築などは必要としない．また本手法では，機械語命令の追加や削除に応じた類似度を算出できるようになるとともに，実際に変更のあった箇所も機械語命令単位で提示することが可能になる．つまり 2 つの酷似したマルウェアを比較した場合，その共通部分を解析するだけで，2 つのマルウェアが備える機能の大半を把握したことになりうる．

ここでは，まず本提案手法のベースとなる LCS（Longest Common Subsequence）と，その高速化手法であるビットベクトル化について説明する．その後，機械語命令列においてビットベクトル化を適用する手法を提案する．

4.3 LCS(最長共通部分列)

提案手法では，機械語命令を 1 要素とし 2 つのマルウェアから得られた各機械語命令系列の LCS [84] およびその長さを算出することを目指す．LCS とは 2 つの系列の共通の部分列の中で最長の系列のことであり，例えば $dbca$ と $bdca$ の LCS は dca と bca となる．LCS 長の導出方法としては，2 つの系列の長さを M, N とした場合，計算量 $O(MN)$ で LCS を算出するアルゴリズム [84] が知られている．ここで 2 つの系列をそれぞれ $S = s_1^m, T = t_1^n$，S と T の LCS 長を $L(s_1^m, t_1^n)$ とすると，LCS の性質として次式が成り立つ．

$$L(s_1^i, t_1^j) = \begin{cases} i = 0 \; or \; j = 0 \rightarrow 0 \\ s_i = t_j \rightarrow L(s_1^{i-1}, t_1^{j-1}) + 1 \\ s_i \neq t_j \rightarrow max(L(s_1^i, t_1^{j-1}), L(s_1^{i-1}, t_1^j)) \end{cases} \tag{4.1}$$

この再帰式を利用することで，動的計画法により $O(MN)$ で LCS の長さを算出することができる．この作業は行と列にそれぞれ S, T を割り当てた LCS の行列（以下，DP 行列と呼ぶ）を算出することに他ならない．例えば $S =$ "$dbca$", $T =$ "$bdca$" としたときの

DP 行列は図 4.1 になる．

	b	d	c	a
d	0	1	1	1
b	1	1	1	1
c	1	1	2	2
a	1	1	2	3

図 4.1　DP 行列

一方，これまでの調査から 100,000 を超える命令数のマルウェアも多数確認されており，こうしたマルウェア同士の比較には相当な時間を要する．またマルウェア数も日々増加し，それらの組み合わせの数も膨大になっているため，高速に機械語命令系列から LCS およびその長さを算出する手法が望まれる．

4.4　LCS 算出のビットベクトル化

Crochemore ら [85] は LCS を算出する際に，DP 行列の各マスを 1 ビットとして扱い，and, or, not および add の 4 種類の演算で演算ワード長分のマス目をまとめて処理する手法を提案した．ここでもまた $S =$ "$dbca$", $T =$ "$bdca$" の 2 系列を例に説明する．まず T に出現するアルファベットが S のどの位置に出現するかを現すビット行列 M（図 4.2）を作成する．

	a	b	c	d
d	0	0	0	1
b	0	1	0	0
c	0	0	1	0
a	1	0	0	0

図 4.2　ビット行列 M

これに基づき下記演算を繰り返すことで，加算（下線部）で発生したキャリーの総和として LCS 長を算出することができる．

$$V_j = \begin{cases} j = 0 \rightarrow 1111 \\ 1 \leq j \leq n \rightarrow \underline{(V_{j-1} + (V_{j-1} \& M_{y_j}))} | (V_{j-1} \& M'_{y_j}) \end{cases} \tag{4.2}$$

本例では DP 行列における 4 マス分がまとめて処理されることになるが，IA-32 アーキテクチャ [41] における演算用レジスタのビット長は 32 ビットであるため，32 マス分ずつ処理できることになる．さらに SSE2 命令 [86] を用いることで andnot/or に関しては 128 ビット単位，add に関しては 64 ビット単位（このうち 1 ビットはキャリー用とする）で処理可能となる．

ただしこの際に事前に必要となる M は機械語命令の種類数分のメモリスペースを要する．IA-32 アーキテクチャにおける機械語命令は最長で 16 バイトにも及ぶため，そのまま機械語命令を一要素として扱うとメモリ領域が枯渇する恐れがある．

4.5 提案手法

4.5.1 縮約命令

ここでは機械語命令としての情報量を残しつつ，アルファベットサイズを削減することを目的として新たに縮約命令を提案する．IA-32 機械語命令は以下の 6 つの部位から構成される．

- Prefix（0〜4 バイト）

  機械語命令はオプションとなる Prefix（最大で 4 バイト）から始まることがある．この役割は，デフォルトの機械語命令の振る舞いを変更するために用いられる．例えば，命令セットが使うデフォルト・セグメントを強制的に切り替えたり，オペランドのデフォルト・サイズを強制的に 16 ビットや 32 ビットに切り替えるために用いられる．Prefix は 4 つのグループに分かれており，4 つのグループからひとつずつ利用することができる．また Prefix の順序に制限はない．
- Opcode（1〜3 バイト）

  Opcode は機械語命令の動作および機械語命令全体のフォーマットを決定づける．
- ModR/M（0〜1 バイト）

  ModR/M はオペランドのアドレッシング・モードを決定づける情報を含む．またオペランド中でレジスタが使われる場合は，そのレジスタ種別に関する情報も含む．Opcode によって存在しない場合もある．
- SIB（0〜1 バイト）

SIB（Scale Index Base）は，アドレッシング・モードに関する情報を含む．Opcode や Mod R/M によって存在しない場合もある．

- Displacement（0〜4 バイト）

  Displacement には機械語命令がアクセスするアドレス情報が格納される．Opcode によって存在しない場合もある．

- Immediate（0〜4 バイト）

  Immediate には機械語命令が使う即値が格納される．Opcode によって存在しない場合もある．

このうちディスプレースメントには分岐命令の分岐先情報が含まれ，分岐元と分岐先の間に新たな命令が追加されることで変化してしまう．また，メモリアクセスに必要となる絶対アドレスもディスプレースメントとして指定される．一方マルウェアが動的リンクライブラリとして実装されている場合，ロードされるアドレスは一定でない．つまり，環境によってこの絶対アドレスは変化する可能性がある．したがって，ディスプレースメントも類似性算出の情報としてしまうと，上記のような状況において必要以上に類似性が失われてしまう．そのため，縮約命令ではディスプレースメントの情報を含めないこととした．また即値に関しても同様にアドレス情報が含まれる場合があるため縮約命令には盛り込まない．以上を踏まえ，縮約命令は図 4.3 のエンコーディング規則に従い，1 つの機械語命令を 32 ビット値に変換する．Prefix1〜Prefix4 は命令プレフィックスを表しており，

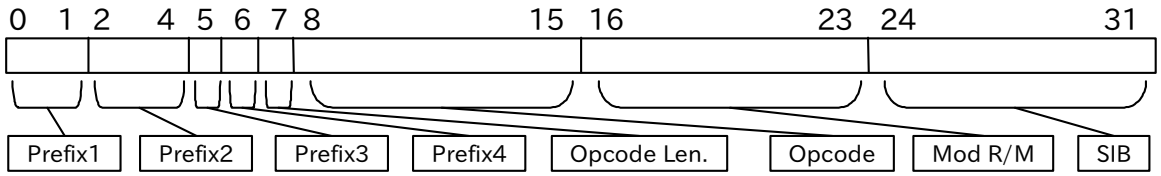

図 4.3　縮約命令の形式

命令プレフィックスの各グループにおいて指定された値を格納する．また Opcode Len はオペレーションコードの長さを表しており，オペレーションコードが 1 バイトの場合は 0，それ以外の場合は 1 を格納する．Opcode は実際のオペレーションコードを格納する変数であるが，オペレーションコードが 2 バイト以上のときは一番最後のオペレーションコードのバイト値を格納する．ModR/M および SIB は機械語命令に含まれる場合はその

値を，含まれない場合は 0 を格納する．こうして定義された縮約命令の種類数に関して事前に調査したところ，1 マルウェア当たりの機械語命令は多いもので約 50,000 種類あったのに対し，縮約命令の種類数は約 8,000 種類に抑えることができた．これはビットベクトル化の際に必要なメモリ量に換算すると約 100MB [*1] であり，複数の CPU コアで並列処理したとしても，現在の計算機性能で十分に処理可能となる．

4.5.2 類似度算出

本提案手法では，2 つのマルウェアが与えられた際にまず各機械語命令列を縮約命令列 A, B に変換し，それらの LCS 長（$|L|$ とする）を求める．こうして得られた $|L|$ を元にマルウェア間の類似度 S および非類似度（距離）D を以下のように定義する．

$$S = \frac{|L|}{|A| + |B| - |L|} \tag{4.3}$$

$$D = 1 - S \tag{4.4}$$

S は各縮約命令列 A,B を集合，LCS をそれらの積集合とみなしたときの Jaccard 係数に相当し，0 から 1 の値をとる．0 は 2 つのマルウェアに共通部分がないことを意味し，1 は 2 つのマルウェアが全く同一であることを意味する．

4.6 むすび

従来研究では，ベーシックブロックやコールツリー等，プログラム構造の再構築を要しており，これがマルウェア分類の全自動化の妨げとなっていた．また N-gram/N-perm による手法では，N-gram/N-perm の出現回数という一種の統計情報により，類似度を定義しているため，実際に変化のあった場所を抽出することは難しいといった問題もあった．

こうした問題に対し，本章では機械語命令単位でプログラムコードの LCS を抽出し，その LCS の長さに基づき類似度を決定する手法を提案した．マルウェアの中には，機械語命令数が 100,000 を超えるものも存在するため，単純に機械語命令列同士の LCS を抽出するには，多くの計算時間を要する．そのため提案手法では，機械語命令を独自の縮約命令で表現することで，LCS 抽出アルゴリズムのビットベクトル化を可能にした．これにより，SSE2 命令を用いた実装では，単純な LCS 抽出アルゴリズムと比較し 100 倍程度

[*1] 比較対象の縮約命令数を 100,000 とした場合．

の高速化を見込むことができる．また，本手法が必要とするのは，第 3 章の確率的逆アセンブル手法により生成できる逆アセンブル結果のみであるため，容易にマルウェア分類を自動化することができる．さらに，提案手法により算出された類似度（LCS の長さ）は機械語命令単位であるため，解析に要する作業量を正確に見積もることも可能となる．

第 5 章

自動マルウェア分類システム

5.1　はじめに

本章では，第 2 章・第 3 章・第 4 章で述べた提案手法を組み合わせた自動マルウェア分類システムについて述べ，いくつかのデータセットによる分類結果を示しながら，その有効性を明らかにする．

5.2　システム構成

本システムは前述の 3 つの手法に対応する，アンパックモジュール，逆アセンブルモジュール，類似度算出モジュールより構成される（図 5.1）．

まず本システムに対してマルウェアが与えられると，アンパックモジュールにおいて，パッキングされているマルウェアからオリジナルのプログラムコードを含むメモリダンプが出力される．アンパックモジュールにより出力されたメモリダンプは逆アセンブルモジュールにおいて逆アセンブルされ，機械語命令列が得られる．類似度算出モジュールは，各マルウェアの機械語命令列を元に類似度を算出し，マルウェアの全組み合わせ関する類似度行列を作成する．これにより入力されたマルウェアを系統別に分類することが可能となる．

アンパックモジュールは，仮想マシン VMware を用いて構築されている．仮想マシンとしては Windows XP と 2.6.1 節で開発したカーネルドライバをインストールした状態である．まずホスト側でマルウェアを受け取ると，仮想マシンである Windows XP を起動し受け取ったマルウェアを起動する．この際，カーネルドライバは動的に生成されたコード

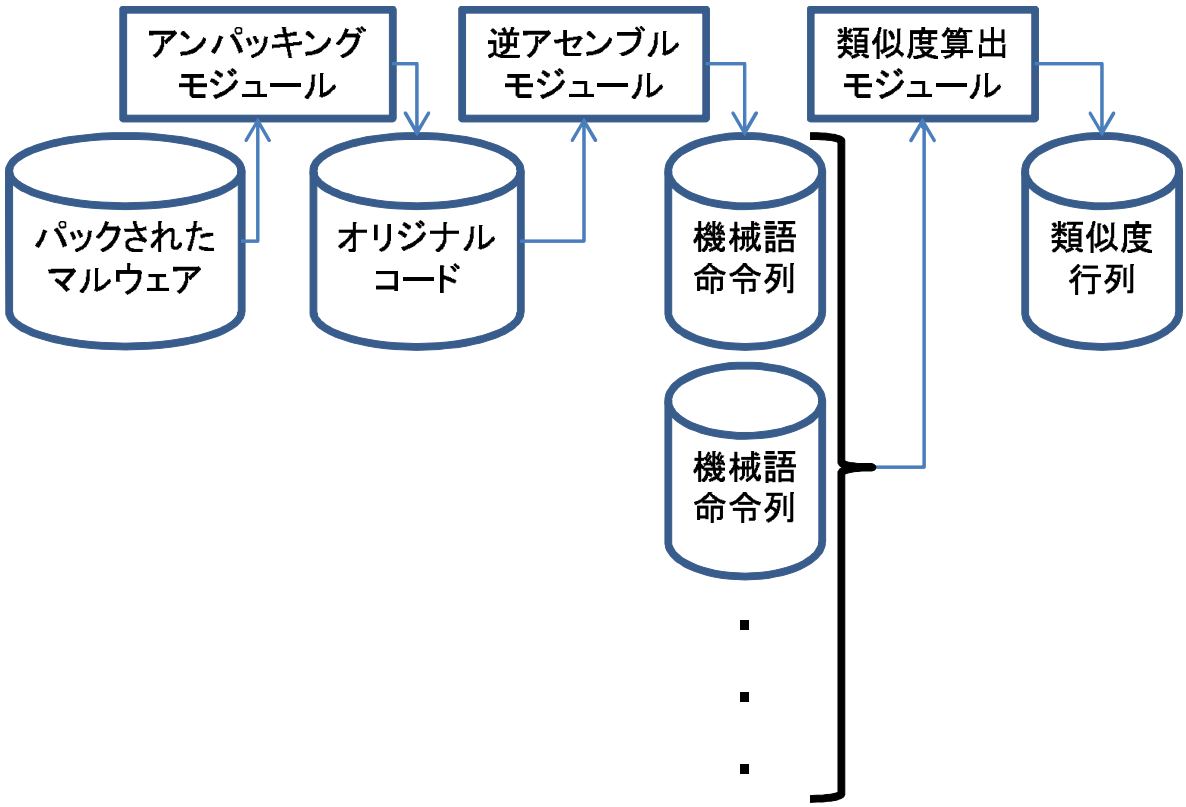

図 5.1　自動マルウェア分類システム

を含むメモリ領域をダンプし続ける．一定時間が経過すると，得られたメモリダンプを逆アセンブルモジュールに渡す．その後，ホスト側から仮想マシンである Windows XP をロールバックし，次のマルウェアを待つ．ここでは，マルウェアが多重にパッキングされている場合等において，マルウェアのオリジナルコードを含む複数のメモリダンプが得られることがある．

次に逆アセンブルモジュールでは，アンパックモジュールで得られた複数のメモリダンプを全て逆アセンブルする．本論文では特に明示しない限り，その中で最も命令数の多い逆アセンブル結果を当該マルウェアの逆アセンブル結果とする．

こうして得られたマルウェアの逆アセンブル結果を用いて，類似度算出モジュールはマルウェアの全組み合わせの類似度を算出し，類似度行列を作成する．そして最終的には，この類似度行列を使い階層的クラスタ分析を行うことで，マルウェアの系統樹を描くことが可能になる．

5.3 実験

本章ではまず 3 つのデータセット A,B,C に関して，我々が開発した自動マルウェア分類システムにより類似度を算出し，群平均法を利用して階層的クラスタ分析を行った結果を示す．ここでは，各データセットにおけるマルウェアの類似度とそれらの解析結果を挙げながら，解析作業の効率化に関する本システムの有効性について述べる．３つのデータセット A,B,C は，実行ファイル形式のマルウェアで構成される．一番目のデータセット A は CCC DATAset 2009 [87] のマルウェア検体 10 種である．二番目のデータセット B は 2009 年 7 月に京都大学において収集した 702 種類の SHA1 ハッシュ値が重複しないマルウェア検体である．収集には独自に開発したハニーポットを利用している．このハニーポットは Windows XP SP0 をベースとしたハイインタラクション型の受動的ハニーポットであり，内部ではマルウェアの起動を禁止しおり，いわゆる一次検体のみを収集している．このため，ダウンローダ等を介したマルウェアは収集対象となっていない．三番目のデータセット C は 2 つ目のデータセットとは異なる場所で，前記ハニーポットを用いて収集した 3,233 種類の SHA1 ハッシュ値が重複しないマルウェア検体である．

また本章では，本システムの分類精度について考察する．具体的にはソースコードが存在する複数のマルウェアと複数のコンパイラを用いることで，ソースコードやコンパイラ等の変化が，本システムによる類似度に与える影響について述べる．

5.4 データセット A の分類結果

CCC DATAset 2009 のマルウェア検体の分類結果をデンドログラムとして図 5.2 に示す．各葉はマルウェア検体におけるオリジナルコードの縮約命令列を表す．横軸は非類似度となっており，左方で繋がっていれば類似した検体（クラスタ）同士であることを意味している．縮約命令列の名前は”HASH_ハッシュ値の先頭 4 文字”とした．またこのデータセットに関しては，複数のオリジナルコードが存在したものには名前の末尾に識別番号（00 など）を付けて，各オリジナルコードを区別している．

当該データセットにおける 393F,84E9,1D23（グループＡと呼ぶ）は何らかの関連性を持つとされており，7190,CD91（グループＢと呼ぶ）もまた何らかの関連性を持つとされ

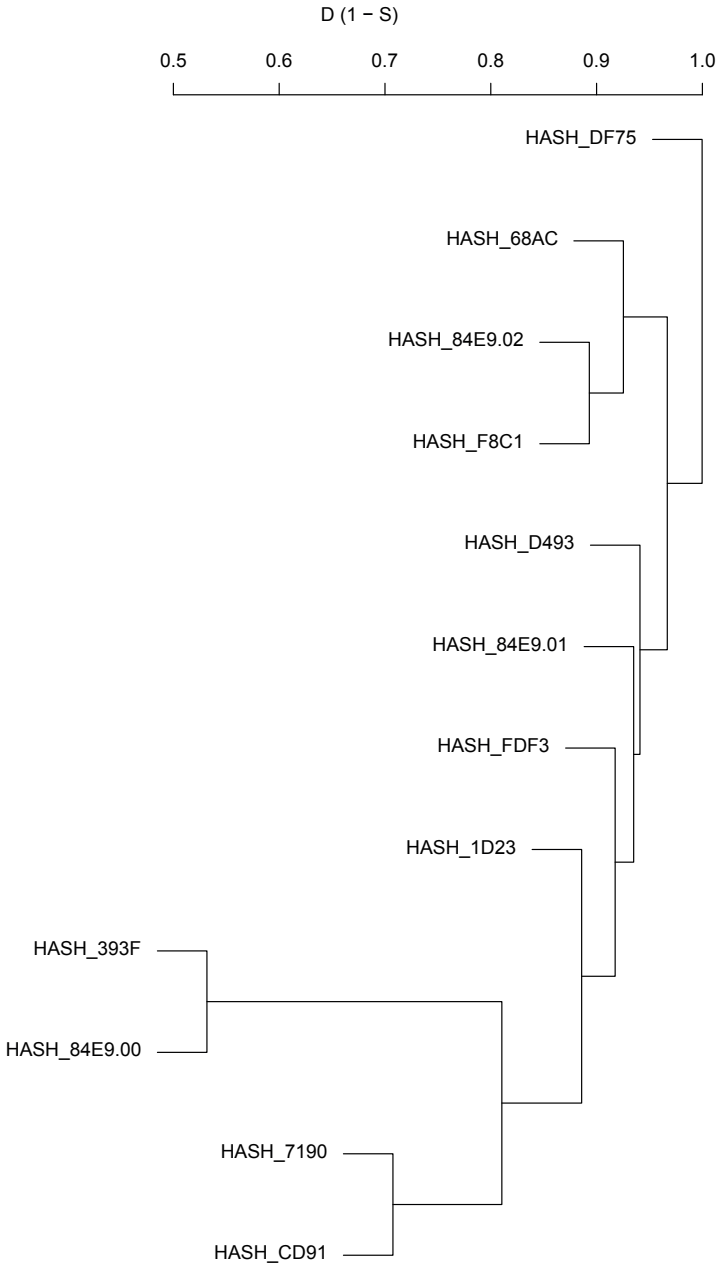

図 5.2　データセット A のデンドログラム

ている．図 5.2 では，実際にグループＡの 393F および 84E9 に関して類似度 $S = 0.47$ 程度の一致がみられた．

またグループＢの 7190 および CD91 に関しても類似度 $S = 0.29$ 程度の一致がみられ，その一致箇所には TCP 139/445 番を用いた通信ロジックが含まれていた．さらに

7190 の検体には，他ホストへのスキャン活動後に IRC サーバへのスキャン結果通知用のメッセージを作成するロジックが含まれていた．このスキャン結果に関するロジックは，EnterCriticalSection API [88] と LeaveCriticalSection API [89] に挟まれており，実際に IRC サーバとの通信を行うスレッドとの排他制御が行われている．一方で，CD91 の検体にも 7190 の検体と全く同じ他ホストへのスキャン活動を行うロジックが含まれていたが，IRC サーバへの通知に関わるロジックは存在しなかった．しかし，排他制御すべき処理が存在しないにもかかわらず，EnterCriticalSection API と LeaveCriticalSection API を連続して呼び出す処理は存在していた 5.3.

図 5.3　7190 検体と CD91 検体の比較

これはあくまで推測であるが，7190 の検体から IRC 関連の機能を取り除き，感染活動に特化させたマルウェアが CD91 検体であり，クリティカルセクション関連 API の呼び出しは，その際に削除し損じた処理であると考えられる．実際，CD91 検体にだけは autorun.inf を悪用した比較的新しい感染活動に関するロジックが含まれており，こうした状況からも，7190 検体の後に CD91 検体が開発されたと考えられる．

このように複数のマルウェアを解析する際に，本システムによりマルウェアを事前に分類しておくことで，類似度が高いマルウェアを優先的に解析する事が可能になる．さら

に，機械語命令単位での共通部分や差分の明確化は，マルウェアの効率的な解析や，マルウェア間の関連性の推定において，その一助となる．

5.5　データセット B の分類結果

本システムを利用し京都大学で収集されたマルウェア 702 検体を分類した．ここで類似度 S が 0.8 以上のマルウェアをひとつのクラスタとしてまとめた場合，クラスタ数はわずか 7 つであることが確認された．この中で最も大きいクラスタに含まれるマルウェアは 679 種類にものぼり，リバースエンジニアリングの結果，これは Conficker.B および Conficker.B++ であった [90]．また類似度 S について 0.91 を閾値とすると，当該クラスタは Conficker.B（446 検体）と Conficker.B++（233 検体）の 2 つのクラスタに分割されることも確認できた．さらに Conficker.B と Conficker.B++ の LCS を求め当該検体同士の差分を解析したところ，Conficker.B++ には新たに CreateNamedPipeA [91] 等の名前付きパイプに関連する処理等（図 5.4）が追加されていることを確認できた．

SRI International は Conficker に関して，B++ には遠隔から実行ファイルをアップデートするために，新たにバックドアが追加されたと報告しているが，これはまさしく名前付きパイプを利用したものである．

5.6　データセット C の分類結果

データセット C は 3233 種類のマルウェア検体である．分類結果のデンドログラムを極座標変換したものを図 5.5 に示す．具体的には図 5.2 における縦軸が円の半径，横軸が角度（0〜2 π）を表しており，図 5.2 における上端が円の中心，下端が円周上を表している．

分類の結果，類似度 90% を閾値としてクラスタに分解すると，全体の約 50%(1593 検体) を占めるクラスタ（図 5.5 において，円の上方やや右から，左回りに下方やや左あたりまでを占める）と，約 25% を占めるクラスタ（図 5.5 において，円の下方やや左から，左回りに右方やや下あたりまでを占める）と，が存在することが分かった．さらに，アンチウィルスソフトによる検出名や静的解析の結果，各ファミリは Conficker [92]（約 50%）と Rahack [93]（約 25%）であることが分かった．また Conficker のクラスタを，類似度 91% を閾値としてさらに分解してみると，3 つのクラスタとなった．これらは，アンチ

```
seg000:009577F6                 lea     eax, [esp+1A8h+var_198]
seg000:009577FA                 push    eax
seg000:009577FB                 push    esi
seg000:009577FC                 push    esi
seg000:009577FD                 push    offset sub_959102
seg000:00957802                 push    esi
seg000:00957803                 push    esi
seg000:00957804                 call    ds:CreateThread
seg000:0095780A                 push    eax
seg000:0095780B                 call    ds:CloseHandle
.
.
seg000:00959102                 push    ebp
seg000:00959103                 mov     ebp, esp
seg000:00959105                 sub     esp, 510h
seg000:0095910B                 push    ebx
seg000:0095910C                 push    esi
seg000:0095910D                 push    edi
seg000:0095910E                 lea     eax, [ebp+var_110]
seg000:00959114                 push    104h
seg000:00959119                 push    eax
seg000:0095911A                 call    sub_958D4E
seg000:0095911F                 mov     edi, ds:CreateNamedPipeA
seg000:00959125                 pop     ecx
seg000:00959126                 pop     ecx
seg000:00959127                 mov     ebx, 3E8h
seg000:0095912C                 mov     esi, 400h
seg000:00959131                 jmp     short loc_959192
seg000:00959133 loc_959133:
seg000:00959133                 push    0
seg000:00959135                 push    [ebp+var_4]
seg000:00959138                 call    ds:ConnectNamedPipe
seg000:0095913E                 mov     [ebp+var_C], eax
seg000:00959141                 call    ds:RtlGetLastWin32Error
seg000:00959147                 cmp     [ebp+var_C], 0
seg000:0095914B                 jnz     short loc_959154
seg000:0095914D                 cmp     eax, 217h
```

図 5.4 Conficker.B++ における名前付きパイプ作成処理

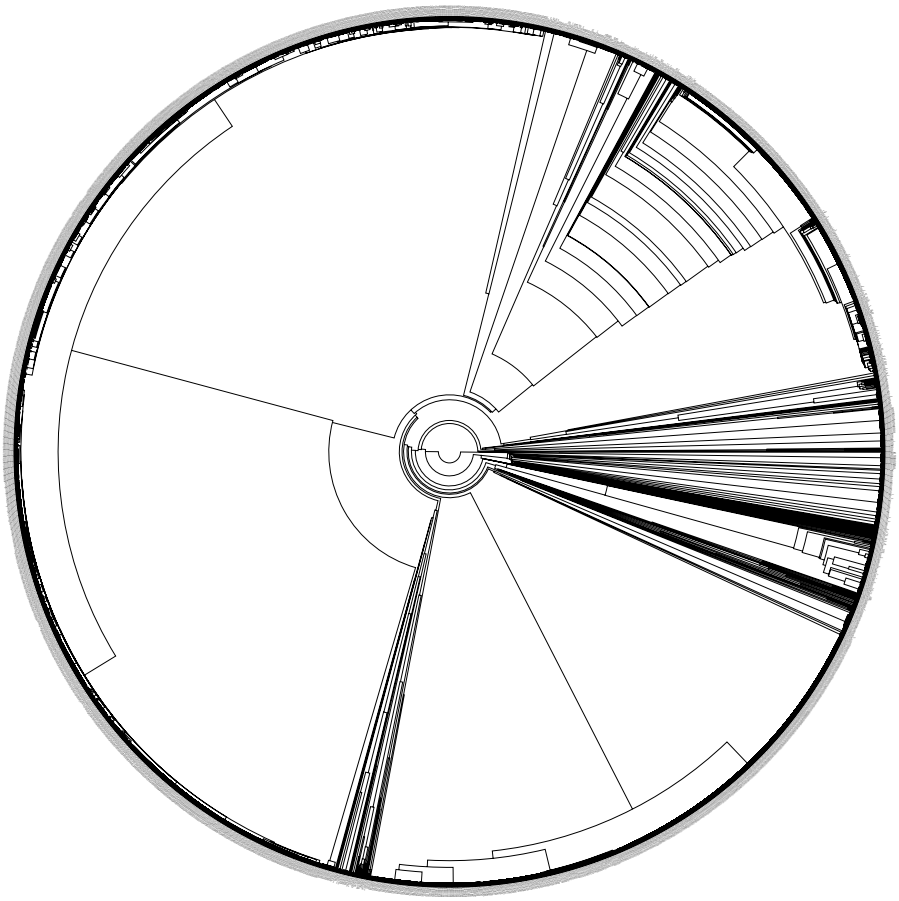

図 5.5　データセット C のデンドログラム

ウィルスソフトの検出名や静的解析結果を参考にすると，Conficker.A/B/B++ の三種類であることが分かった．また Rahack に関しては，類似度 94% を閾値とすると Rahack.W と Rahack.H の 2 種類のクラスタに分けられた．Rahack.W の多様性はなく一種類のプログラムコードである一方，Rahack.H に関しては，類似度 98% を閾値として分類してみると，4 種類に分けられた．特にそのうち 1 種類に関して，アンチウィルスソフトでは半数以上が Backdoor.Trojan [94] として検出されていることが分かった．

図 5.5 において右側上方のクラスタは階段状になっている．これは，他の 2 つのファミリと比べてプログラムコードの多様性に富んでいることを示している．このクラスタに含まれるマルウェアをいくつか静的解析した結果，IRC 関連の処理が含まれるボット [95] であることが分かった．ただし，アンチウィルスソフトの検出名をみてみると，IRCBot の他にも Virut [96] と識別されているものが多く含まれていた．Virut のプログラムコー

ドは IRC 関連の処理と比べると小さい．今回の実験では，複数のメモリダンプが得られた際には，最も命令数が多い逆アセンブル結果を用いて分類した．このため，アンチウィルスの検出名とは異なる分類結果が得られたと考えられる．

次にデータセット C に関して，できるだけ多くのマルウェアを少ない手間で解析することを念頭に置き，類似度を 95% としたときにできるクラスタをみてみよう（図 5.6）．図 5.6 において，マルウェア数が多い上位 5 つのクラスタはそれぞれ，Conficker.B,

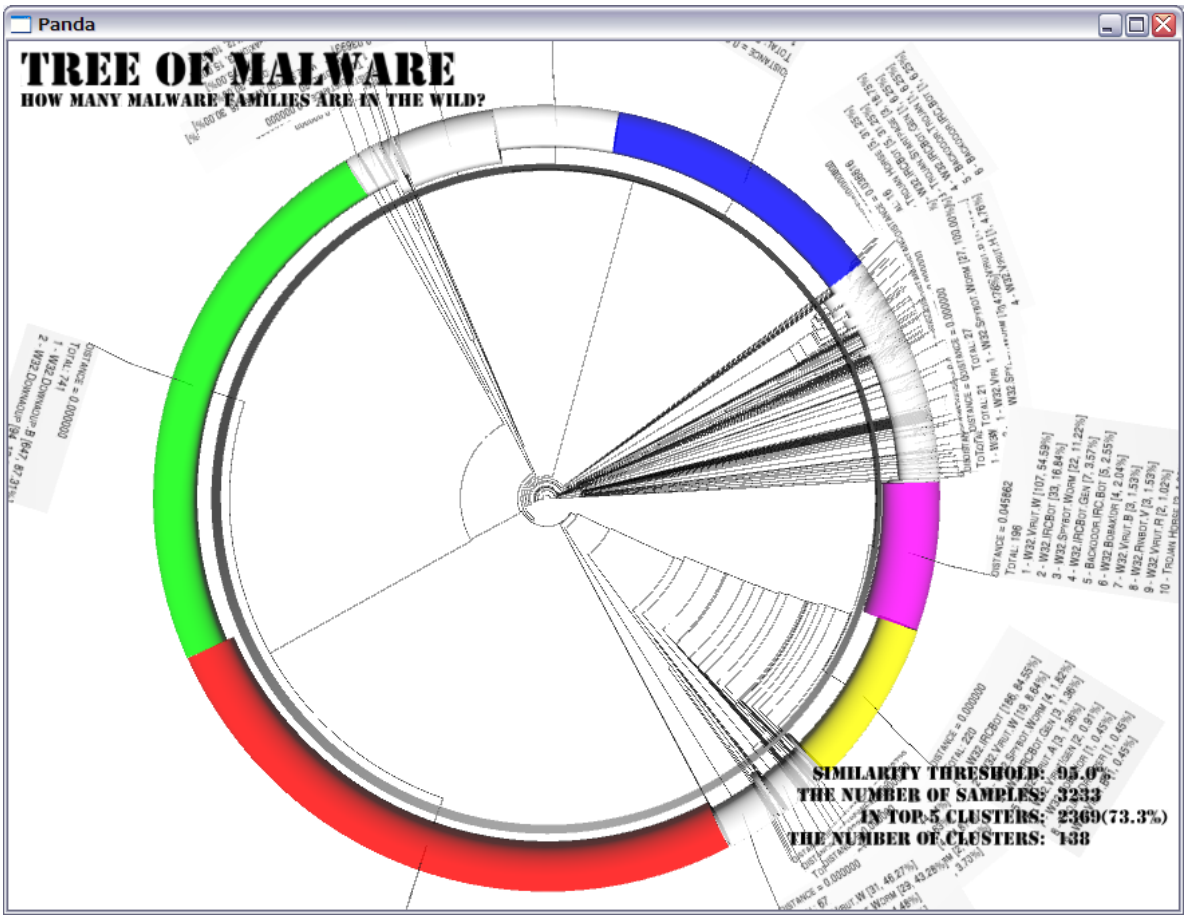

図 5.6　データセット C のデンドログラム（類似度 95%）

Conficker.B++，Rahack.W，IRCBot（2 種類）となった．これらのクラスタに含まれるマルウェア数は，全体の 73.3% に及ぶ．つまり，上位 5 つのクラスタからそれぞれ 1 検体を選び，それら 5 検体を解析するだけで，データセット C における全てのマルウェアのプログラムコードのうち，約 66%（95% × 73.3%）をカバーしたことになる．

さらに，類似度 90% を閾値としたときにできるクラスタを表したのが図 5.7 である．こ

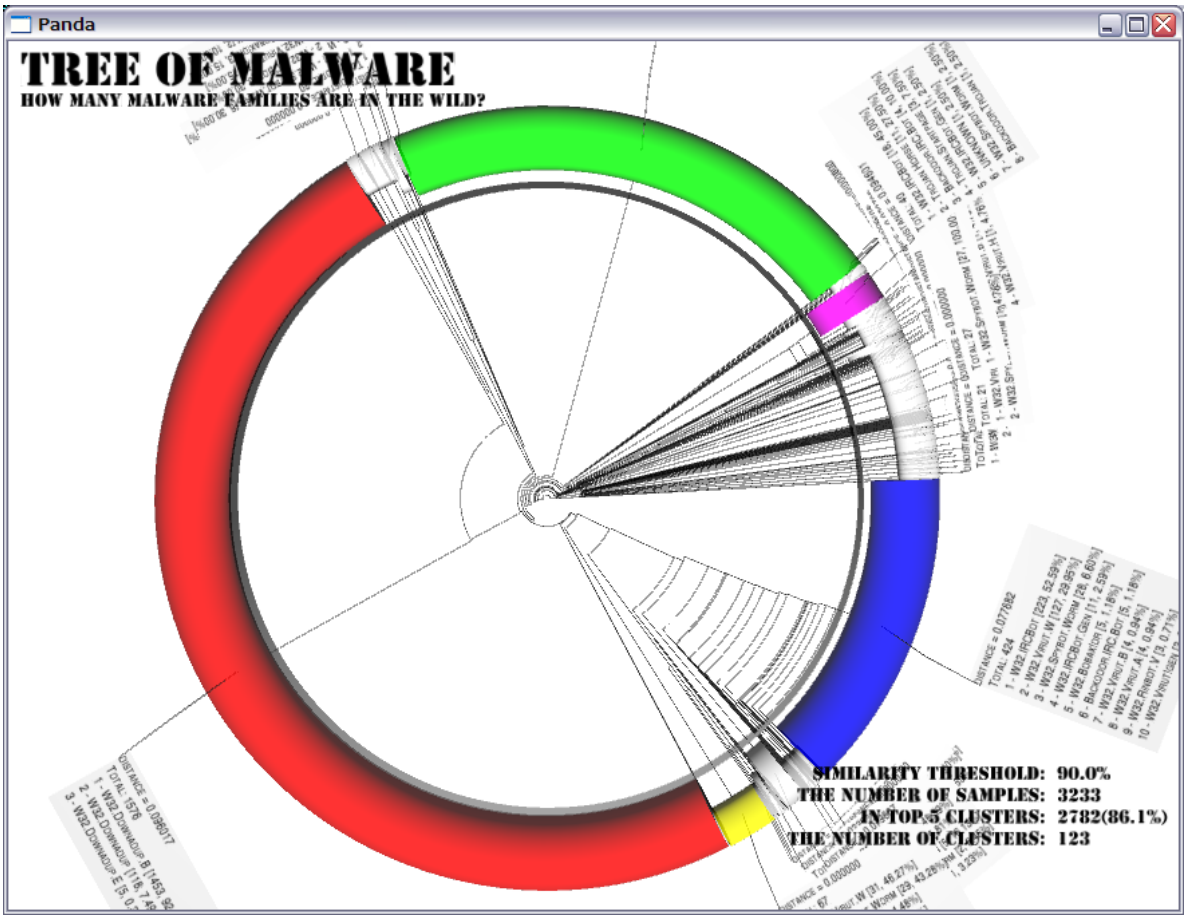

図 5.7　データセット C のデンドログラム（類似度 90%）

こでは，Conficker.B と Conficker.B++ は同じクラスタとなり，また Rahack.W と Rahack.H も同じクラスタとなる．他にも IRCBot の多くの亜種も 1 つのクラスタとして分類されている．このとき，マルウェア数が多い上位 5 クラスタに含まれるマルウェア数は，実に全体の 86.1% にのぼる．つまり類似度 90% を閾値とした場合は，5 検体を解析するだけで，全マルウェアの約 77.5%（90% × 86.1%）をカバーしたことになる．言い換えると，約 2,500 検体（3,233 検体の 77.5%）分のプログラムコードを把握するために要する労力が，本システムによる分類結果により，わずか 5 検体分の労力にまで削減できているとも解釈できる．

図 5.8 は，類似度 50% を閾値としたときの分類を表している．このときの上位 5 クラスタに含まれるマルウェア数は全体の約 87.8% であり，類似度 90% 程度のときの 86.1% と比較し，あまり増加していない．

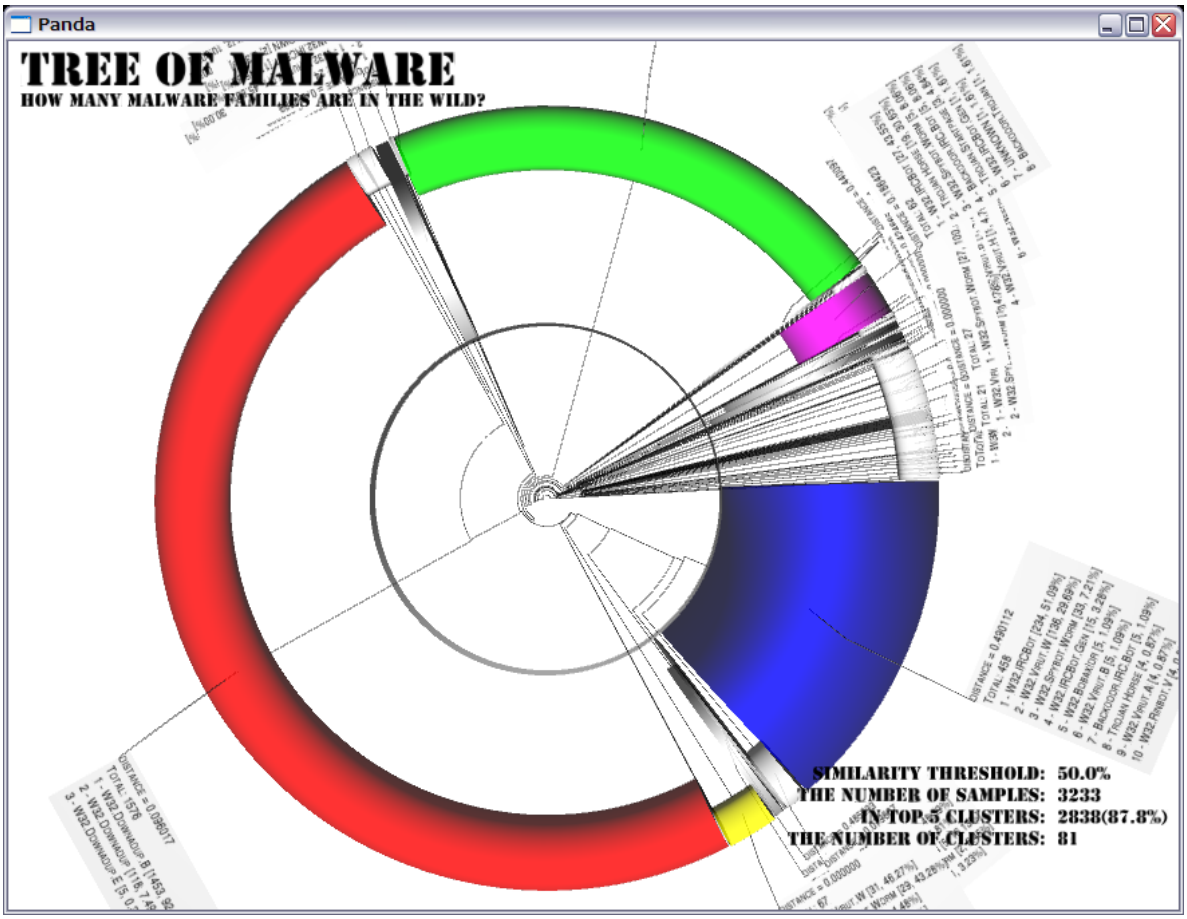

図 5.8　データセット C のデンドログラム（類似度 50%）

さらに類似度 20% を閾値としたとき（図 5.9）でさえも，上位 5 クラスタに含まれるマルウェア数は全体の約 89.9% とマルウェア数は増えず，これ以上のマルウェアのプログラムコードを把握するためには，5 検体より多くのマルウェアを解析する必要があることを示している．

5.7　本システムの分類精度に関する考察

ここでは，ソースコードが存在する 4 種類のマルウェア（表 5.1）[97,98] を，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（表 5.2）によりコンパイルすることで，ソースコードやコンパイラ・最適化オプションの変化が類似度に与える影響について考察する．

具体的には以下の 2 点に着目し，その考察を与える．

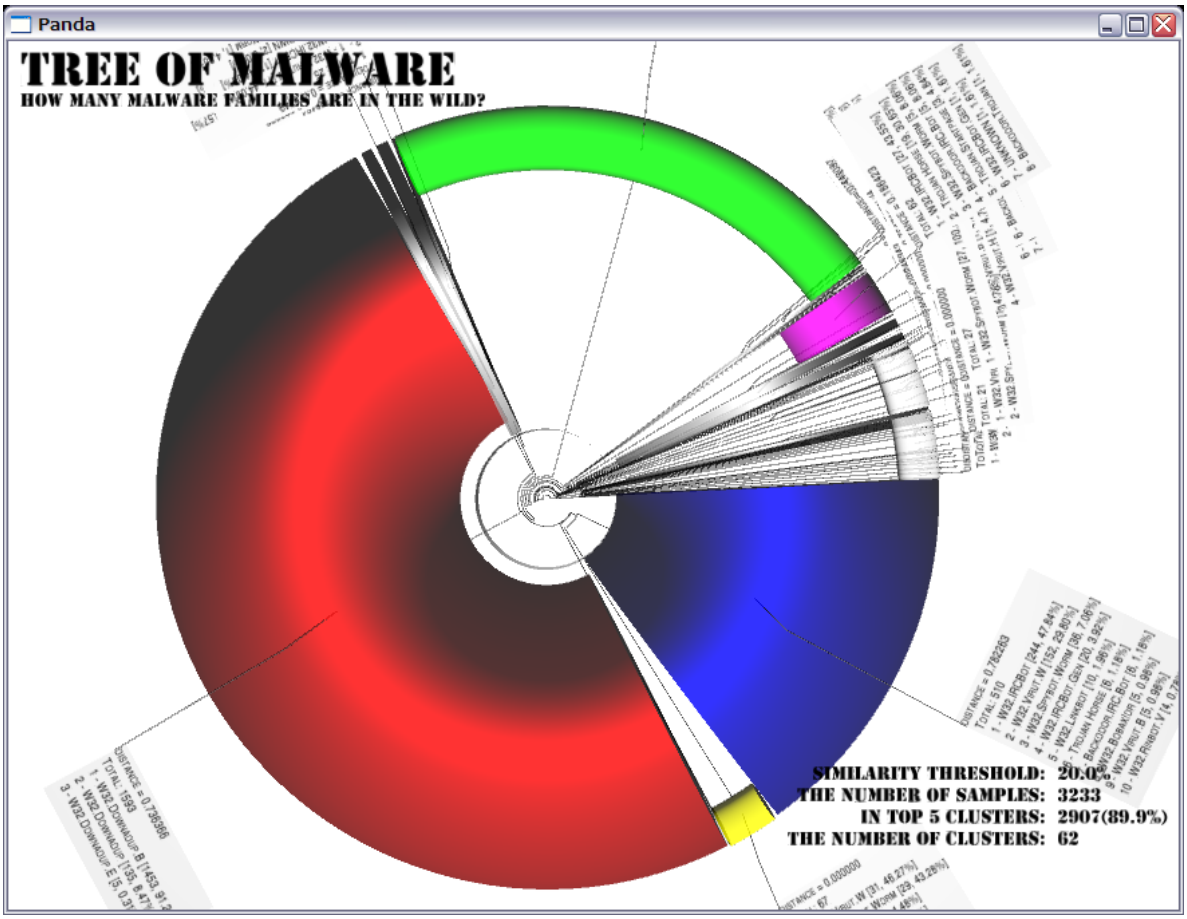

図 5.9　データセット C のデンドログラム（類似度 20%）

名称 (本稿での略称)	対応コンパイラ
sdbot v0.4b (sdbot04b)	Visual C++, lcc-win32
sdbot v0.5a (sdbot05a)	Visual C++, lcc-win32
sdbot v0.5b (sdbot05b)	Visual C++, lcc-win32, MinGW
rxBot v0.7.7 Sass (rxBot)	Visual C++

表 5.1　評価用マルウェア

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

コンパイラ	バージョン情報	最適化オプション
Visual C++	32-bit C/C++ Optimizing Compiler Version 15.00.21022.08	/Od（最適化を無効化） /Ox（最大限の最適化）
MinGW	gcc version 3.4.5 (mingw-vista special r3)	-O0（最適化を無効化） -O3（最大限の最適化）

表 5.2　評価用コンパイラ

まずは，同じコンパイラ・最適化オプションで作成した 4 種類のマルウェアに関する類似度について説明する．ここでは，表 5.1 の全てのマルウェアをコンパイル可能な Visual C++ を用いた．表 5.3，表 5.4 は，各マルウェアをそれぞれ 2 種類の最適化オプション Od（最適化を無効化）, Ox（最大限の最適化）でコンパイルし，それらの類似度行列を本システムにより算出した結果である． どちらの最適化オプションであっても，3 種類の sdbot

	rxBot	sdbot04b	sdbot05a
sdbot04b	19.16	-	-
sdbot05a	20.22	52.80	-
sdbot05b	19.48	45.65	77.54

表 5.3　類似度行列 (%), 最適化オプション Od

	rxBot	sdbot04b	sdbot05a
sdbot04b	19.97	-	-
sdbot05a	21.17	66.51	-
sdbot05b	18.94	55.17	75.09

表 5.4　類似度行列 (%), 最適化オプション Ox

同士の類似度は約 45%〜77% となっている．一方，rxBot と sdbot との類似度は 18%〜21% 程度に留まっており，亜種と別種との関係が，類似度に反映されていることが見てとれる．

ここではさらに，本システムの類似度とソースコードの類似度との比較を試みる．sdbot は main 関数を含む唯一のソースコードファイルから成る．表 5.5，5.6 は，3 種類の sdbot

のソースコードの行数，および各ソースコードに関して diff コマンドを用いて算出した共通する行数をあらわしている． ここで，ソースコードの行数を集合の要素数，共通行の数

名称	行数
sdbot04b	1599
sdbot05a	1902
sdbot05b	2173

表 5.5　sdbot のソースコード行数

	sdbot04b	sdbot05a
sdbot05a	1312	-
sdbot05b	1191	1660

表 5.6　共通行の数

を積集合の要素数とみなすことで，本システムと同様，Jaccard 係数に基づき類似度を算出することができる．表 5.7 にその算出結果を示す．sdbot05a と sdbot05b の類似度が最

	sdbot04b	sdbot05a
sdbot05a	59.94	-
sdbot05b	46.14	68.74

表 5.7　diff に基づく類似度 (%)

も高く，次に sdbot04b と sdbot05a の類似度が高く，sdbot04b と sdbot05b の類似度が最も低くなっている．表 5.3・表 5.4 の結果とは，類似度の違いはあるものの，それらの大小関係は維持されていることが分かる．

最後に，MinGW と Visual C++ の両コンパイラに対応している sdbot05b に関して，コンパイラ・最適化オプションを変化させながら類似度を算出した結果を，表 5.8 に示す．同じマルウェアであっても，コンパイラが異なるとそれらの類似度は約 5%〜10% になっていることが分かる．また MinGW に関して，最適化オプションが O0（最適化を無効化）と O3（最大限の最適化）の場合で，類似度が 18.46% となっており，表 5.3 や表 5.4 で示した，RxBot と sdbot の類似度より低くなっている．つまりコンパイラや最適化オプショ

	MinGW (O0)	MinGW (O3)	Visual C++ (Od)
MinGW (O3)	18.46	-	-
Visual C++ (Od)	10.20	5.74	-
Visual C++ (Ox)	6.15	5.36	55.30

表 5.8 sdbot05b に関する類似度 (%)

ンの違いが，種の違い以上に強く類似度に反映されてしまっていることが分かる．一方で Visual C++ に関しては，最適化オプションが Od と Ox の場合で，類似度が 55.3% となっており，MinGW の場合に比べて高い数値となっている．この結果からコンパイラによって，最適化オプションの類似度に対する影響度合いが異なるとも言える．

5.8 むずび

本論文では，機械語命令列の類似度算出手法を提案した．また，これまで我々が開発してきたマルウェアのアンパック手法および逆アセンブル手法を組み合わせ，マルウェアの分類作業を全自動化するシステムを構築した．ハニーポットで収集した 3,233 種類のマルウェアを分類した結果では，代表的な 5 つのクラスタから，それぞれ 1 検体ずつ選択し解析するだけで，全体の約 77.5% 程度のマルウェアの機能を把握できることを示した．また，本システムの分類結果とアンチウィルスソフトによる検出名の比較では，本システムが同一と判断したマルウェアに関して，アンチウィルスソフトでは異なる複数の検出名が確認される状況もあり，マルウェアに別のマルウェアが感染している状況等において，マルウェアに対する命名の難しさが明らかになった．他にも，ソースコードが存在するマルウェアを用いた実験では，コンパイラや最適化オプションが同じであれば，ソースコードの類似度と同じ大小関係を維持できていることが分かった．一方，同じソースコードのマルウェアであっても，コンパイラや最適化オプションが異なる状況では，種の違いよりも大きく，その類似度が低下することも確認された．ただ，メタモーフィック・エンジンやポリモーフィック・エンジンとは異なり，マルウェア開発によく使われるコンパイラやそのオプションの種類には限りがある．このため，マルウェア作者がコンパイラやそのオプションを変化させマルウェアを作成したとしても，依然として，本提案システムは解析コストの削減に効果を発揮するであろう．

第 6 章

インポートアドレステーブル格納場所の特定方法

6.1 はじめに

マルウェアの機能を把握するためには，マルウェアが利用する外部関数（Win32 API 等）の特定が重要となる．Windows 用の実行ファイルにおいて，暗黙的なリンクにより外部関数を利用する際には，外部関数のアドレスが IAT（Import Address Table）に格納される．ここで，Windows における実行ファイルの形式について説明する．Windows における実行ファイルは図 6.1 に示す IMAGE_DOS_HEADER からはじまる．

e_magic には，必ず 0x5A4D（文字列で’MZ’）が格納されており，e_lfanew に PE ヘッダが格納されているアドレスが保存されている．PE ヘッダは図 6.2 の構造になっている．

Signature には，0x00004550（文字列で’PE\0\0’）が格納される．さらに OptionalHeader は図 6.3 の構造になっている．

IMAGE_OPTIONAL_HEADER32 構造体における DataDirectory は，全部で IMAGE_NUMBEROF_DIRECTORY_ENTRIES（16）個のエントリ（図 6.4）からなり，各エントリは図 6.5 に示す用途に使われている．

ここで DataDirectory[1] は，インポートディレクトリと呼ばれ，図 6.6 に示す IMAGE_IMPORT_DESCRIPTOR 構造体の形をとる．

通常，この IMAGE_IMPORT_DESCRIPTOR 構造体を参照することで，IAT の場所を特定することができる．しかしながら，パックされたマルウェアには，パック以前のプログラムコード（以下，オリジナルコード）用の PE ヘッダが含まれない場合がある．これは，

```
// DOS .EXE header
typedef struct _IMAGE_DOS_HEADER {
    WORD   e_magic;      // Magic number
    WORD   e_cblp;       // Bytes on last page of file
    WORD   e_cp;         // Pages in file
    WORD   e_crlc;       // Relocations
    WORD   e_cparhdr;    // Size of header in paragraphs
    WORD   e_minalloc;   // Minimum extra paragraphs needed
    WORD   e_maxalloc;   // Maximum extra paragraphs needed
    WORD   e_ss;         // Initial (relative) SS value
    WORD   e_sp;         // Initial SP value
    WORD   e_csum;       // Checksum
    WORD   e_ip;         // Initial IP value
    WORD   e_cs;         // Initial (relative) CS value
    WORD   e_lfarlc;     // File address of relocation table
    WORD   e_ovno;       // Overlay number
    WORD   e_res[4];     // Reserved words
    WORD   e_oemid;      // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;    // OEM information; e_oemid specific
    WORD   e_res2[10];   // Reserved words
    LONG   e_lfanew;     // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

図 6.1　IMAGE_DOS_HEADER 構造体

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

図 6.2　IMAGE_NT_HEADERS32 構造体

PE ヘッダが OS の備えるローダに対する情報であり，パックされたマルウェアのロードを担うのは，独自のローダであることに起因する．つまり，マルウェアのオリジナルコードのロードに必要な情報は，必ずしも PE ヘッダのフォーマットに従う必要はなく，パッカーが用意する独自の構造で管理しておけばよい．実際，オリジナルコード用の PE ヘッダを削除する難読化手法 [99] も存在する．こうした状況では，オリジナルコードが利用する IAT エントリの格納場所を特定することが困難になる．

本稿ではこうした課題を解決するために，オリジナルコードの PE ヘッダが存在しない状態であっても，IAT エントリの格納場所を精度よく特定する手法を提案する．

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    // Standard fields.
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    // NT additional fields.
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
        DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

図 6.3　IMAGE_OPTIONAL_HEADER32 構造体

6.2　従来技術

ここでは，オリジナルコードが利用する IAT エントリ格納場所を特定する従来手法について説明する．

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

図 6.4　IMAGE_DATA_DIRECTORY 構造体

```
// Directory Entries

#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7
// Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10
// Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11
// Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13
// Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

図 6.5　IMAGE_OPTIONAL_HEADER32 構造体

6.2.1　0xFF, 0x15/0x25 の探索

通常，IAT エントリは間接分岐命令により利用される．この手法は，間接分岐命令と解釈できるバイト列を網羅的に探し出し，当該命令の分岐先が格納されているアドレスを，IAT エントリ格納場所として抽出する [38, 100]．コンパイラが出力するプログラムコードにおいて，IAT エントリを参照する機械語命令は，主に以下の間接分岐命令である．

- JMP [IAT エントリのアドレス]
- CALL [IAT エントリのアドレス]

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
                // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;
                // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;
            // 0 if not bound,
            // -1 if bound, and real date\time stamp
            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;
            // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;
            // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

図 6.6 IMAGE_IMPORT_DESCRIPTOR 構造体

これらの間接分岐命令は，0xFF, 0x25（JMP の場合）もしくは 0xFF, 0x15（CALL の場合）から始まり，その直後に 4 バイトの IAT エントリのアドレスが続く．そこでこの手法では，オリジナルコードから 0xFF, 0x25 もしくは 0xFF, 0x15 を探し出し，後続の 4 バイトを IAT エントリ格納場所として出力する．当然のことながら，0xFF, 0x25（もしくは 0xFF, 0x15）が間接分岐命令ではなくオペランド部やデータとして存在する場合は，IAT エントリ格納場所ではないアドレスが誤って出力されてしまう（この事象を False positive とする）．通常のプログラムの場合，この False positive の割合は単純計算で $\frac{1}{2^{16}} \times 2$ 程度（32KB に 1 つの割合）と非常に低いが，ランタイムパッカーの実装方法によっては，False positive を意図的に誘発することが可能となる．

その説明の前に，まずランタイムパッカーの特徴を整理する．ランタイムパッカーは，オリジナルの実行ファイルを入力として受け取り，実行ファイル全体（もしくはコードセクション等）に対し何らかのエンコードを行う．その後，エンコードされたバイナリとこれを復元する特殊なローダをつなぎ合わせ，新たな実行ファイルを生成する．このようにランタイムパッカーは実行ファイルを入力として受け付けるため，マルウェア作者は自身の開発環境を変更することなく，マルウェアのオリジナルコードを隠ぺいすることができ

る．一方，この実行ファイル生成（コンパイル）とパッキングの分業は，ランタイムパッカーによるオリジナルコードの変異（命令の置換や順序入れ替え等）を困難にする側面もある．これは実行ファイルの正確な逆アセンブルが困難であることに依る．この理由として，Windows 向けのコンパイラは実行ファイルの削減やキャッシュヒット率の向上を見込み，機械語命令とデータを混在させる傾向にあることが挙げられる．また IA-32 [41] の機械語命令が可変長であることも，逆アセンブルを難しくしている一因である．このためオリジナルコードの変異は，基本的にオリジナルエントリポイントから辿れる範囲等，非常に限定的であると考えられる．

一方で，Windows の実行ファイルにはリロケーション・テーブル（6.7，6.8）と呼ばれる情報を持たせることができる． リロケーションテーブルは，想定されるベースアドレ

```
//
// Relocation format.
//

typedef struct _IMAGE_RELOCATION {
    union {
        DWORD   VirtualAddress;
        DWORD   RelocCount;
        // Set to the real count when IMAGE_SCN_LNK_NRELOC_OVFL is set
    };
    DWORD   SymbolTableIndex;
    WORD    Type;
} IMAGE_RELOCATION;
typedef IMAGE_RELOCATION UNALIGNED *PIMAGE_RELOCATION;
```

図 6.7　IMAGE_RELOCATION 構造体

スではない場所に実行ファイルがロードされる場合に，実行ファイル中の変更すべき箇所を記録しておくテーブルである．具体的には，このテーブルの内容は絶対アドレスを利用する機械語命令のオペランドを指している．ローダは，このリロケーション・テーブルを参照するだけで，絶対アドレスの参照先を実際にロードされるベースアドレスに合わせた値に補正することができる．これは，正確な逆アセンブル結果を必要とせずに，機械語命令のオペランド部分を変更することができることを意味する．

ここで，False positive 誘発の説明に戻る．まず，実行ファイルのベースアドレスが 0x00400000 のときの，以下の機械語命令について考えてみる．

```
//
// I386 relocation types.
//
#define IMAGE_REL_I386_ABSOLUTE         0x0000
    // Reference is absolute, no relocation is necessary
#define IMAGE_REL_I386_DIR16            0x0001
    // Direct 16-bit reference to the symbols virtual address
#define IMAGE_REL_I386_REL16            0x0002
    // PC-relative 16-bit reference to the symbols virtual address
#define IMAGE_REL_I386_DIR32            0x0006
    // Direct 32-bit reference to the symbols virtual address
#define IMAGE_REL_I386_DIR32NB          0x0007
    // Direct 32-bit reference to the symbols virtual address,
    // base not included
#define IMAGE_REL_I386_SEG12            0x0009
    // Direct 16-bit reference to the segment-selector bits
    // of a 32-bit virtual address
#define IMAGE_REL_I386_SECTION          0x000A
#define IMAGE_REL_I386_SECREL           0x000B
#define IMAGE_REL_I386_TOKEN            0x000C
    // clr token
#define IMAGE_REL_I386_SECREL7          0x000D
    // 7 bit offset from base of section containing target
#define IMAGE_REL_I386_REL32            0x0014
    // PC-relative 32-bit reference to the symbols virtual address
```

図 6.8　IMAGE_RELOCATION 構造体における Type

```
MOV EAX, DWORD PTR DS:[0x0040BEEF]
```

この機械語命令のバイト列表現は，0xA1, 0xEF, 0xBE, 0x40, 0x00 となる．次に，実行ファイルのベースアドレスが 0x15FF0000 に変更されたときの，上記の機械語命令を示す．

```
MOV EAX, DWORD PTR DS:[0x15FFBEEF]
```

この機械語命令のバイト列表現は，0xA1, 0xEF, 0xBE, 0xFF, 0x15 となり，機械語命令中に 0xFF, 0x15 が出現していることが分かる．つまり，実行ファイルのベースアドレスを 0x15FF0000（もしくは 0x25FF0000）とすることで，実行ファイルの先頭 64KB の領域（たとえばグローバル変数）にアクセスする機械語命令のオペランド内に，0xFF, 0x15（もしくは 0xFF, 0x25）を出現させることが可能になる．これは正確な逆アセンブル結果を把握していないランタイムパッカーであっても，IAT エントリの格納場所を錯乱させること

が可能になることを意味する．また，リロケーション・テーブルも PE ヘッダの一部であるため，パッカーは独自の構造でこれを管理することで，解析者からリロケーション情報を隠ぺいすることができる．

6.2.2　逆アセンブラによる間接分岐命令の探索

この手法は事前に逆アセンブラを利用することで，間接分岐命令を探索し，そこで利用される IAT エントリを抽出する．逆アセンブラとしては，IDA Pro [82] 等が存在する．ただ，前述したように Windows の実行ファイルを正確に逆アセンブルすることは難しい．特に，パックされたマルウェアの場合，逆アセンブルのヒントとなる PE ヘッダが消失している場合もあり，正確な逆アセンブル結果は望めない．このため IAT エントリ格納場所の抽出精度にも課題が残る．

6.3　提案手法

オリジナルコードが利用する IAT エントリには，以下の特徴がある．

- IAT エントリの参照元は複数存在する場合がある
- IAT エントリの参照元は Thunk とよばれる間接 JMP 命令の場合があり，Thunk は直接 CALL 命令により呼び出される．

また，隠れマルコフモデル（HMM）に基づく確率的逆アセンブル手法では，Forward/Backward アルゴリズムにより，対象バイナリ中の各バイト値が命令の先頭になる確率を算出することができる．本稿ではこうした IAT エントリの特徴と，確率的逆アセンブル手法により求められる各分岐命令が機械語命令と解釈される確率を利用することで，IAT エントリの抽出精度を向上させる手法を提案する．

図 6.9 に本手法の処理概要を示す．本手法は，まずプログラムコード中に含まれる分岐命令と解釈できるバイト列を探し出す．そして，確率的逆アセンブル手法により当該バイト列が機械語命令である確率を算出し，暫定分岐命令情報を作成する．例えば，表 6.1 の分岐命令と解釈できるバイト列を含むプログラムコードがあったとする．CALL/JMP X は X へ分岐する直接分岐命令で，CALL/JMP [X] は X に格納されたアドレスへ分岐する間接分岐命令を表す．このプログラムコードに対する暫定分岐命令情報は，例えば表 6.2

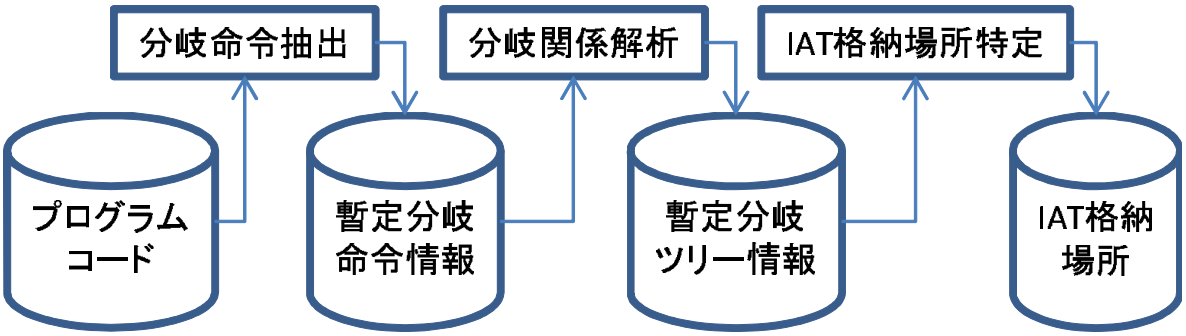

図 6.9　提案手法の処理概要

表 6.1　プログラムコード

アドレス	ニーモニック
0x0005	CALL 0x000F
0x000F	CALL 0x1004
0x001F	JMP 0x1004
0x002F	CALL [0x2004]
0x1004	JMP [0x2004]
0x1100	CALL [0x20AA]

表 6.2　暫定分岐命令情報

アドレス	種別	命令確率	アドレス
0x0005	直接 CALL	0.2	0x000F
0x000F	直接 CALL	0.4	0x1004
0x001F	直接 JMP	0.6	0x1004
0x002F	間接 CALL	0.8	0x2004
0x1004	間接 JMP	0.1	0x2004
0x1100	間接 CALL	0.4	0x20AA

のようになる．次に暫定分岐命令情報を参照し，間接分岐命令からその呼び出し元となる分岐命令を探索することで，例えば図 6.10 のような暫定分岐ツリー情報を生成する．この暫定分岐ツリーは，間接分岐命令の分岐先格納アドレス（つまり IAT エントリの候補）を根とする．

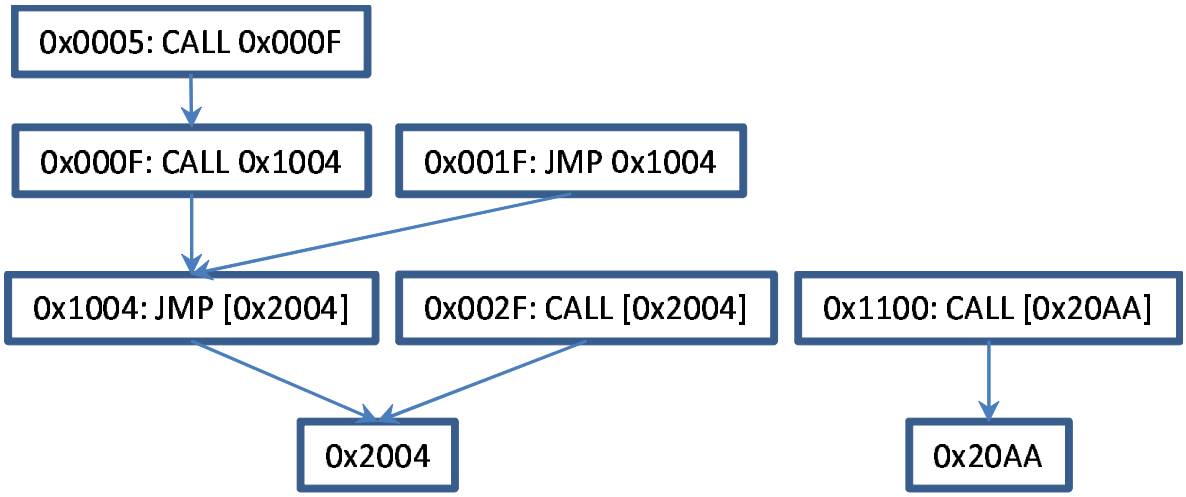

図 6.10　暫定分岐ツリー情報

こうして得られた暫定分岐ツリーから，分岐先格納アドレスの候補（例えば図 6.10 では 0x2004 と 0x20AA）に着目し，それらが真の分岐先格納アドレスである確率を算出する．ここで，図 6.10 において「0x2004 が真の分岐先格納アドレスではない」ときを考えてみる．0x2004 が分岐先格納アドレスでないためには，その参照元となる JMP [0x2004] および CALL [0x2004] が機械語命令として解釈されてはいけない．また，JMP [0x2004] が機械語命令として解釈されない状況では，JMP 0x1004 と CALL 0x1004 も機械語命令として解釈されてはいけない．これは，機械語命令として解釈されない箇所を分岐先とする分岐命令が存在しないためである．したがって，「0x2004 が真の分岐先格納アドレスではない」ということは，0x2004 と同じ暫定分岐ツリー内に含まれるすべての分岐命令は，機械語命令として解釈されないことを意味する．

ここで，入力となるオリジナルコードのバイト列を X，確率的逆アセンブル手法で用いる HMM のモデルパラメータを θ，分岐先格納アドレスの候補 r が真の分岐先格納アドレスである事象を E_r，その確率を $P(E_r)$ とする．また r と同じ分岐命令ツリーに含まれる分岐命令を $b_{r,i}(1 \leq i \leq N_r, N_r$ は r と同じ暫定分岐ツリーに含まれる分岐命令数$)$ とし，$b_{r,i}$ が機械語命令として解釈されない事象を $F_{b_{r,i}}$，その確率を $P(F_{b_{r,i}})$ とする．前述の暫定分岐ツリーの特徴から，X, θ のもとで r が真の分岐先格納アドレスである確率は，以

下の通りである．

$$P(E_r|X,\theta) = 1 - P(\bigcap_{i=1}^{N_r} F_{b_{r,i}}|X,\theta)$$
$$= 1 - \frac{P(X, \bigcap_{i=1}^{N_r} F_{b_{r,i}}|\theta)}{P(X|\theta)} \quad (6.1)$$

ここに出てくる $P(X|\theta)$ は，X が与えられた際に Forward アルゴリズムで算出し，全ての r について再利用できる．一方，$P(X, \bigcap_{i=1}^{N_r} F_{b_{r,i}}|\theta)$ の算出は，全ての $b_{r,i}(1 \leq i \leq N_r)$ に割り当てる状態を，データもしくは機械語命令の 2 バイト目以降とした上で，Forward アルゴリズムにより X の出力確率を計算することに他ならない．つまり $\{b_{r,i}|1 \leq i \leq N_r\}$ が変化するたびに（分岐先格納アドレスの候補 r ごとに），対象となる X の出力確率を算出する必要がある．確率的逆アセンブル手法の計算量は，対象となるプログラムコードサイズを $|X|$ とすると $O(|X|)$ となる．そのため，全ての分岐先格納アドレスの候補数を $|R|$，暫定分岐ツリーの平均ノード数を $\bar{N}$ とすると，全ての $P(E_r|X,\theta)$ を求めるには $O(|X|\bar{N}|R|)$ の計算量が必要となってしまう．ここで，計算量を削減するために $F_{b_{r,i}}(1 \leq i \leq N_r)$ が互いに独立であると仮定する．すると $P(E_r|X,\theta)$ は以下のように算出できる．

$$P(E_r|X,\theta) \approx 1 - \prod_{i=1}^{N_r} \frac{P(X, F_{b_{r,i}}|\theta)}{P(X|\theta)} \quad (6.2)$$

式 6.2 中の $P(X, F_{b_{r,i}}|\theta)$ は，X に関する Forward/Backward アルゴリズムの実行結果を保持しておくことで，O(1) で算出できる．よってこの近似により，全ての $P(E_r|X,\theta)$ を求める計算量を $O(|X| + \bar{N}|R|)$ に抑えることができる．

本提案手法では，こうして r ごとに算出された $P(E_r|X,\theta)$ が 0.5 以上のときに，当該 r を IAT エントリ格納場所として出力する．

6.4 実験

ここでは以下の三つの手法について，IAT エントリ格納場所の特定精度を評価する．

- 0xFF, 0x15/0x25 の網羅的な抽出による手法（Exhaustive）
- IDA Pro の逆アセンブル結果から間接分岐命令を特定する手法（IDA Pro）

表 6.3　ベースアドレス 0x00400000

手法	TP	TN	FP	FN	PPV	NPV	MCC
Exhaustive	343	0	18	0	0.9501	NaN	0.0000
IDA Pro	293	18	0	50	1.0000	0.2647	0.4755
Probabilistic	343	18	0	0	1.0000	1.0000	1.0000

表 6.4　ベースアドレス 0x15FF0000

手法	TP	TN	FP	FN	PPV	NPV	MCC
Exhaustive	343	0	2,839	0	0.1078	NaN	0.0000
IDA Pro	293	2,839	0	50	1.0000	0.9827	0.9162
Probabilistic	343	2,829	10	0	0.9717	1.0000	0.9840

表 6.5　ベースアドレス 0x25FF0000

手法	TP	TN	FP	FN	PPV	NPV	MCC
Exhaustive	343	0	2,839	0	0.1078	NaN	0.0000
IDA Pro	293	2,839	0	50	1.0000	0.9827	0.9162
Probabilistic	342	2,837	2	1	0.9942	0.9996	0.9951

- 提案手法（Probabilistic）

評価対象とするプログラムコードは，アンパック後のメモリイメージを想定し，メモリ上に展開され PE ヘッダが存在しない状態のオリジナルコードとする．正解データは，パックされていない状態の実行ファイル（PE ヘッダを含んだ状態）を IDA Pro 6.0 で逆アセンブルした結果から生成しておく．この正解データを生成するため，MWS 2011 Datasets [101] のマルウェア検体のうち，パックされていない 5 つの検体（ハッシュ値先頭 2 バイトが 5e2d, 15a4, 542a, 7586, ae58 の検体）を利用した．また，提案手法で用いる HMM のモデルパラメータは，Firefox 3.0.1 に含まれる xul.dll を Microsoft C++ Compiler(Ver. 15.00.21022.08) でコンパイルした結果から学習した．

まず表 6.3 に，対象プログラムコードのベースアドレスが 0x00400000 であった場合の評価結果を示す．表の列は左から順に True positive 数 (TP)，True negative 数 (TN)，False positive 数 (FP)，False negative 数 (FN)，Positive predictive value(PPV)，Negative predictive

value(NPV)，Matthews correlation coefficient(MCC) となっている．ここで True negative とは，0xFF, 0x15/0x25 の網羅的な抽出により得られた IAT エントリ格納場所の候補のうち，IAT エントリ格納場所ではない箇所を特定できたことを示す．また TP, TN, FP, FN は，5 検体に関する総計である．PPV/NPV/MCC は，それぞれ以下の式で表すことができる．

$$PPV = \frac{TP}{TP + FP} \tag{6.3}$$

$$NPV = \frac{TN}{TN + FN} \tag{6.4}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6.5}$$

MCC は二値分類器の評価指標として用いられ，TP, TN, FP, FN の相関係数である．具体的には 1 が完全に正しい予測，0 がランダムな予測，-1 が全く逆の予測を意味する．Exhaustive アプローチは，全ての IAT エントリ格納場所の候補を出力する．このため False negative は存在せず，正解となる 343 箇所の IAT エントリ格納場所を全て網羅できている．一方で 0xFF, 0x15/0x25 というバイト列が，プログラムコード中のデータ部分等に 18 箇所存在するため，その数だけ False positive が発生している．ただ，正解数と比較すると十分少ない数であるため，PPV は 95.01% と高い値を示している．対照的に IDA Pro を用いた手法では，False negative が 50 箇所で発生している．IDA Pro は Recursive traversal と呼ばれるアルゴリズムに基づいている．この Recursive traversal はプログラムのエントリポイントや，頻出命令列を起点としてプログラムの制御フローを辿りながら逆アセンブルを進める．このため動的に分岐先が決まる状況では，その分岐先が機械語命令として解釈されない場合がある．こうした理由から，IDA Pro は False negative を生じさせやすい傾向にあると考えられる．一方提案手法では，18 箇所の 0xFF, 0x15/0x25 というバイト列に惑わされずに，完全に正確な予測を行えていることが分かる．

次に，対象となるプログラムコードのベースアドレスを 0x15FF0000 としたときの評価結果を表 6.4 に示す．このデータセットでは，0xFF, 0x15 というバイト列が機械語命令のオペランド部分に頻出しており，その数は 2,839 箇所に及ぶ．このため Exhaustive アプ

ローチの PPV は 10.78% まで急落している．これに対し IDA Pro はベースアドレスの影響を全く受けず，0x00400000 のときと同じ False positive/negative 数となっている．また提案手法では 10 箇所の False positive が発生しているが，IDA Pro との MCC の比較では十分に優れた抽出精度となっている．

さらに表 6.5 はベースアドレスを 0x25FF0000 とした場合である．Exhaustive アプローチと IDA Pro はベースアドレスが 0x15FF0000 のときと全く変わらない抽出精度である．一方，提案手法では 0x15FF0000 のときと比較し，MCC が 98.40% から 99.96% へと向上していることが分かる．

6.5　むすび

パックされたマルウェアに関して，オリジナルコードの IAT エントリ格納場所を特定することは，その機能を把握するために重要な作業となる．従来の手法では，不正確な逆アセンブル結果に基づき IAT エントリを特定していたため，その特定精度に課題があった．本稿では，IAT エントリ格納場所の参照元が複数ある場合等に着目し，確率的逆アセンブル手法により算出される機械語命令の確率を利用することで，精度よく IAT エントリ格納場所を特定する方法を提案した．また実験により，提案手法が従来手法を上回る特定精度であることを示した．今後は IAT エントリが指す外部関数を特定することで，IAT 再構築を自動化する手法を検討していく．

第 7 章

結論

個人情報の売買やスパムメールの配信代行といったブラックマーケットの成熟に伴い，マルウェアには，迅速なバグ改修や機能追加等の効率的な開発が求めらるようになった．また，アンチウィルスソフトによる検出を回避するためには，マルウェア自身の変異も求められる．こうしたマルウェアの効率的な開発と多様化を両立するため，現在の多くのマルウェアは，高級言語で開発された後にランタイムパッカーで多様化されている．その結果，マルウェアの種類数は日々増加の一途をたどり，マルウェアの全貌を明らかにすることはおろか，そのトレンドを把握することも困難になっている．

こうした状況を鑑み，本研究ではマルウェアを自動的に分類し，解析対象を代表的なマルウェアに絞り込むことで，マルウェアの解析コストを削減することを目指した．プログラムコードに基づくマルウェアの分類は，大きく以下の 3 つのプロセスに分割することができる．

1. アンパック
2. 逆アセンブル
3. プログラムコードの類似度算出

本論文では，まずこれら各プロセスを自動化する際の課題を抽出し，それらを解決する手法を提案するとともに，その有効性を示した．また，アンパック・逆アセンブル・類似度算出に関する提案手法を組み合わせることで，自動マルウェア分類システムを構築し，実際にインターネットで収集された 3,233 検体を分類した．その結果，代表的な 5 つのクラスタから 1 検体ずつを選択し解析するだけで，全マルウェアの約 77.5% のプログラム

コードを把握できることが明らかとなった．加えて，マルウェアの機能把握の要となるインポートアドレステーブル格納場所を，特定する手法を提案し，従来研究より高い精度で特定可能なことを示した．以下では各章で述べた内容の要旨を示す．

第 1 章では，社会問題化するマルウェアの背景について述べ，マルウェアが日々増加している要因と，その解決となるマルウェア分類に関する従来研究について触れた．またマルウェア分類の全自動化を目指す上で，３つのプロセス（アンパック・逆アセンブル・類似度算出）が課題となることを明らかにし，こうした課題を解決する方針を示し本論文の導入とした．

第 2 章では，ランタイムパッカーによりパックされたマルウェアから，オリジナルコードを抽出することを目的とし，まず従来のアンパック手法における以下の二つの課題を指摘した．

1. マルウェアのプログラムコード領域を特定できない
2. 多重にパックされたマルウェアに関してオリジナルコードの特定が困難

一点目のプログラムコード領域特定の課題に対しては，相対分岐命令に着目したプログラムコード領域の特定方法を提案した．一般的に，パックされたマルウェアのオリジナルコードには，逆アセンブルのヒントとなる PE ヘッダが存在しない．このため，まず提案手法では，実行モジュールを隠れマルコフモデルで表現することにより，実行モジュール内の各バイト値が機械語命令である確率を算出する手法を提案した．そして，これに基づき分岐区域数の期待値を求めることで，実行モジュールの領域を特定することを可能にした．二点目の多重にパックされたマルウェアについては，前述の隠れマルコフモデルを利用することで，コンパイラ出力コードとしての尤もらしさを算出し，オリジナルコードを特定する手法を新たに提案した．パックされたマルウェアに対する実験では，約 230 のオリジナルコードの候補から，提案手法が正確に真のオリジナルコードを抽出できることを示した．

第 3 章では，PE（Portable Executable）ヘッダが存在しない状況においても，Viterbi アルゴリズムを用いることで，最も尤もらしい逆アセンブル結果を得る手法を提案した．様々なデータセットを用いた実験では，従来研究や商用逆アセンブラと比較し，提案手法が安定して高い精度の逆アセンブル結果を出力できることを示した．

第 4 章では，機械語命令列単位で最長一致部分列（LCS）を算出することで，プログラムコード間の類似度を算出する手法を提案した．マルウェアによっては，その機械語命令数が 100,000 を超える場合もあり，こうしたマルウェアの比較には，LCS 算出アルゴリズムのビットベクトル化による高速化が必要となる．提案手法では，機械語命令からオペランド情報を取り除いた縮約命令を定義し，機械語命令の種類数増加によるメモリ枯渇の問題を解決することで，プログラムコード間の類似度算出の高速化を達成した．

第 5 章では，前述のアンパック・逆アセンブル・類似度算出に関する提案手法を組み合わせることで，自動マルウェア分類システムを構築した．実際のインターネットで収集されたマルウェアに対する実験では，代表的な 5 つのクラスタから 1 検体ずつを選択し解析するだけで，全マルウェアの約 77.5% のプログラムコードを把握できることが明らかとなった．

第 6 章では，マルウェアの機能を把握するために必要となる IAT エントリ格納場所の特定方法について述べた．ある従来技術では，実行モジュールに再配置による錯乱手法に対して弱く，また逆アセンブル手法に基づく従来技術では，逆アセンブル結果の不正確さが IAT エントリ格納場所の特定にも強く影響していた．そこで本研究では，実行モジュール内の各バイト値が機械語命令である確率と，IAT エントリを根とするコールツリーから，IAT エントリ格納場所を精度よく抽出する手法を提案した．実験では，提案手法が各種従来技術よりも高い精度で IAT エントリ格納場所を特定できることを示した．

参考文献

[1] 井上大介, 中尾康二. 1. マルウェアって? 情報処理, Vol. 51, No. 3, pp. 237–243, 2010.

[2] McAfee. Computer security research - mcafee avert labs blog. http://www.avertlabs.com/research/blog/index.php/2009/07/22/.

[3] McAfee. Mcafee 脅威レポート: 2011 年第 1 四半期. http://www.mcafee.com/japan/media/mcafeeb2b/international/japan/pdf/threatreport/threatreport11q1.pdf.

[4] Symantec. Symantec global internet security threat report trends for 2009. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf.

[5] Ero Carrera and Halvar Flake. Automated structural classification of malware. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.

[6] Chet Hosmer. Polymorphic and metamorphic malware. http://www.blackhat.com/presentations/bh-usa-08/Hosmer/BH_US_08_Hosmer_Polymorphic_Malware.pdf.

[7] Tom Brosch and Maik Morgenstern. Runtime packers: The hidden problem? https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.

[8] Michael Bailey, Jon Andersen, Z. Morleymao, and Farnam Jahanian. Automated classification and analysis of internet malware. Technical report, In Proceedings of Recent

Advances in Intrusion Detection (RAID’ 07), 2007.

[9] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW ’10, pp. 45:1–45:4, New York, NY, USA, 2010. ACM.

[10] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW ’10, pp. 45:1–45:4, New York, NY, USA, 2010. ACM.

[11] Ulrich Bayer, Paolo Milani, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[12] Tony Lee and Jigar J. Mody. Behavioral classification. In *Proceedings of EICAR Conference 2006*, 2006.

[13] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA ’08, pp. 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM’09, pp. 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[15] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD ’04, pp. 470–478, New York, NY, USA, 2004. ACM.

[16] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *JOURNAL IN COMPUTER VIROLOGY*, Vol. 1, pp. 13–23, 2005.

[17] M. Gheorghescu. An automated virus classification system. In *VIRUS BULLETIN CONFERENCE OCTOBER 2005*, pp. 294–300, 2005.

[18] E. Carrera and G. Erdelyi. Digital genome mapping - advanced binary malware analysis. In *In Proc. Virus Bulletin Conf.*, pp. 187–197, 2004.

[19] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, ICICS '08, pp. 238–255, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] R. Tian, L.M. Batten, and S.C. Versteeg. Function length as a tool for malware classification. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pp. 69 –76, oct. 2008.

[21] Ronghua Tian, L. Batten, R. Islam, and S. Versteeg. An automated classification system based on the strings of trojan and virus families. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pp. 23–30, 2009.

[22] R. Islam, Ronghua Tian, L. Batten, and S. Versteeg. Classification of malware based on string and function feature selection. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, pp. 9 –17, july 2010.

[23] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pp. 611–620, New York, NY, USA, 2009. ACM.

[24] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *In RAID*, pp. 207–226. Springer-Verlag, 2005.

[25] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156–, Washington, DC, USA, 2001. IEEE Computer Society.

[26] Qinghua Zhang and D.S. Reeves. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 411 –420, dec. 2007.

[27] Thomas Dullien, Rolf Rolles, and Ruhr universitaet Bochum. Graph-based comparison of executable objects. In *University of Technology in Florida*, 2005.

[28] I. Briones and A. Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.

[29] Superman. Polymorphism in c. `http://www.codeproject.com/KB/cpp/PolyC.aspx`.

[30] Microsoft Corporation. Microsoft pe and coff specification. `http://msdn.microsoft.com/en-us/windows/hardware/gg463119`.

[31] snaker, Qwerton, Jibz, and xineohP. Peid. `http://www.peid.info/`.

[32] Aaron. Aaron's homepage. `http://www.exetools.com/unpackers.htm`.

[33] Joe Stewart. Ollybone. `http://www.joestewart.org/ollybone/`.

[34] DataRescue. Using the universal pe unpacker plug-in included in ida pro 4.9 to unpack compressed executables. `http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf`.

[35] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pp. 46–53, New York, NY, USA, 2007. ACM.

[36] Lutz Bohne. Pandora's bochs: Automatic unpacking of malware. `http://www.0x0badc0.de/PandorasBochs.pdf`.

[37] V. Smith D. Quist. Covert debugging: Circumventing software armoring. In *Blackhat USA 2007 / Defcon 15*, 2007.

[38] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 289 –300, dec. 2006.

[39] Cesare Silvio and Xiang Yang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*, AusPDC '10, pp. 61–70, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[40] BitBlaze project. Temu: The bitblaze dynamic analysis component. `http://`

bitblaze.cs.berkeley.edu/temu.html.

[41] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. http://www.intel.com/products/processor/manuals/.

[42] Microsoft Corporation. Memory_basic_information structure. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366775(v=vs.85).aspx.

[43] Microsoft Corporation. Virtualalloc function (windows). http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx.

[44] Microsoft Corporation. Virtualprotect function (windows). http://msdn.microsoft.com/en-us/library/aa366898(v=VS.85).aspx.

[45] Microsoft Corporation. Memory protection constants. http://msdn.microsoft.com/en-us/library/aa366786(VS.85).aspx.

[46] Microsoft Corporation. Using __declspec(dllimport) and __declspec(dllexport). http://msdn.microsoft.com/en-us/library/aa271769(v=vs.60).aspx.

[47] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, Vol. 77, No. 2, pp. 257–286, Feb 1989.

[48] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pp. 290–299, New York, NY, USA, 2003. ACM.

[49] Vmware: Virtualization via hypervisor, virtual machine & server consolidation - vmware. http://www.vmware.com/.

[50] Alan Bradley. Tron: He fights for the user. http://www.openrce.org/repositories/users/AlanBradley/Tron-TC8.pdf.

[51] Intel Corporation. Execute disable bit and enterprise security. http://www.intel.com/technology/xdbit/index.htm.

[52] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1. http://download.intel.com/

design/processor/manuals/253668.pdf.

[53] Mozilla Developer Network. Downloading source archives. https://developer.mozilla.org/en/Mozilla_Source_Code_(HTTP\%2F\%2FFTP).

[54] Microsoft Corporation. Winmain entry point. http://msdn.microsoft.com/library/ms633559(VS.85).aspx.

[55] Microsoft Corporation. /entry (entry-point symbol). http://www.spec.org/cpu2000/CINT2000/.

[56] Microsoft Corporation. /hotpatch (create hotpatchable image). http://msdn.microsoft.com/en-us/library/ms173507.aspx.

[57] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Vol. 19, No. 1, pp. 42–51, 2002.

[58] Jay-Evan J. Tevis and John A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pp. 197–202, New York, NY, USA, 2004. ACM.

[59] Jay-Evan J. Tevis and Jr. John A. Hamilton. Static analysis of anomalies and security vulnerabilities in executable files. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pp. 560–565, New York, NY, USA, 2006. ACM.

[60] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pp. 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[61] Jr Daniel J. Sanok. An analysis of how antivirus methodologies are utilized in protecting computers from malicious code. In *InfoSecCD '05: Proceedings of the 2nd annual conference on Information security curriculum development*, pp. 142–144, New York, NY, USA, 2005. ACM.

[62] Arun Lakhotia, Eric Uday Kumar, and Michael Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, Vol. 31, No. 11, pp. 955–968, 2005.

[63] Wilson C. Hsieh, Dawson R. Engler, and Godmar Back. Reverse-engineering instruction encodings. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pp. 133–145, Berkeley, CA, USA, 2001. USENIX Association.

[64] Barford Paul and Yegneswaran Vinod. *An Inside Look at Botnets*. Advances in Information Security. Springer Verlag, 2006.

[65] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pp. 255–270, Berkeley, CA, USA, 2004. USENIX Association.

[66] B. Schwarz, , S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 45–54. IEEE Computer Society, 2002.

[67] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pp. 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[68] Free Software Foundation. *GNU Binary Utilities*, May 2002. `http://www.gnu.org/software/binutils/manual/`.

[69] C. Cifuentes, M. Van Emmerik, D. Ling, D. Simon, and T. Waddington. Preliminary experiences with the use of the uqbt binary translation framework. In *Proceedings of the Workshop on Binary Translation*, Oct 1999.

[70] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, Vol. 13, No. 2, pp. 260 – 269, apr 1967.

[71] Microsoft Corporation. Microsoft visual studio express. `http://www.microsoft.com/japan/msdn/vstudio/express/`.

[72] Brian C. Wiles and John Walker. Speak freely. `http://www.speakfreely.org/`.

[73] Stephen Turner. Analog: Www logfile analysis. `http://www.analog.cx/`.

[74] The LAME Project. Lame mp3 encoder. http://lame.sourceforge.net/.

[75] Free Software Foundation. Make for windows. http://gnuwin32.sourceforge.net/packages/make.htm.

[76] NcFTP Software. Ncftp client home page. http://www.ncftp.com/ncftp/.

[77] LLC Glyph & Cog. Xpdf: Home. http://foolabs.com/xpdf/home.html.

[78] PuTTY team. Putty: A free telnet/ssh client. http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[79] TightVNC project. Tightvnc: Vnc-compatible free remote control / remote desktop software. http://www.tightvnc.com/.

[80] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *Proceedings of the 16th conference on USENIX Security Symposium*, pp. 275–290, Berkeley, CA, USA, 2007. USENIX Association.

[81] Standard Performance Evaluation Corporation. Spec cint2000 benchmarks. http://msdn.microsoft.com/en-us/library/f9t8842e(VS.71).aspx.

[82] Hex-Rays. Ida pro disassembler - multi-processor, windows hosted disassembler and debugger. http://www.hex-rays.com/idapro/.

[83] P. Baldi, S. Brunak, Y. Chauvin, C. A. Andersen, and H. Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, Vol. 16, No. 5, pp. 412–424, May 2000.

[84] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, Vol. 21, pp. 168–173, January 1974.

[85] Maxime Crochemore, Costas S. Ilipoulos, and Yoan J. Pinzon. Speeding-up hirschberg and hunt-szymanski lcs algorithms. *Fundam. Inf.*, Vol. 56, pp. 89–103, October 2002.

[86] Intel Corporation. Using streaming simd extensions 2 (sse2). http://software.intel.com/en-us/articles/using-streaming-simd-extensions-2-sse2/.

[87] 畑田充弘, 中津留勇, 寺田真敏, 篠田陽一. マルウェア対策のための研究用データセットとワークショップを通じた研究成果の共有. マルウェア対策人材育成ワークショップ, 2009.

[88] Microsoft Corporation. Entercriticalsection function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms682608(v=vs.85).aspx.

[89] Microsoft Corporation. Leavecriticalsection function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms684169(v=vs.85).aspx.

[90] SRI International. An analysis of conficker. http://mtc.sri.com/Conficker/.

[91] Microsoft Corporation. Createnamedpipe function. http://msdn.microsoft.com/en-us/library/aa365150(VS.85).aspx.

[92] Symantec Corporation. W32.downadup. http://www.symantec.com/security_response/writeup.jsp?docid=2008-112203-2408-99.

[93] Symantec Corporation. W32.rahack. http://www.symantec.com/security_response/writeup.jsp?docid=2005-010614-1404-99.

[94] Symantec Corporation. Backdoor.trojan. http://www.symantec.com/security_response/writeup.jsp?docid=2001-062614-1754-99.

[95] Symantec Corporation. W32.ircbot. http://www.symantec.com/security_response/writeup.jsp?docid=2002-070818-0630-99.

[96] Symantec Corporation. W32.virut. http://www.symantec.com/security_response/writeup.jsp?docid=2007-041117-2623-99.

[97] Symantec Corporation. Backdoor.sdbot. http://www.symantec.com/security_response/writeup.jsp?docid=2002-051312-3628-99.

[98] JJC. Security - the global perspective. http://global-security.blogspot.com/2007/07/rxbot.html.

[99] Masaki Suenaga. A museum of api obfuscation on win32. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/a_museum_of_api_obfuscation_on_win32.pdf.

[100] Daniel Quist, Lorie Liebrock, and Joshua Neil. Improving antivirus accuracy with

hypervisor assisted analysis. *J. Comput. Virol.*, Vol. 7, pp. 121–131, May 2011.

[101] 畑田充弘, 中津留勇, 秋山満昭. マルウェア対策のための研究用データセット～mws 2011 datasets～. マルウェア対策人材育成ワークショップ, 2011.

謝辞

本論文は, 私が早稲田大学大学院基幹理工学研究科情報理工学専攻博士後期課程に在籍していた期間に取り組んだ研究をまとめたものです. 同専攻教授村岡洋一先生，同専攻教授筧捷彦先生，同専攻教授松山泰男先生，同専攻教授中島達夫先生には，本論文の審査を行っていただきました.

村岡洋一先生には指導教官として，私が学部 4 年生のときから修士で卒業するまでの期間に加え，社会人博士としての期間と，長きにわたりご指導を賜りました．先生からは研究への取り組みの意義や，ときには研究分野そのものの意義を問う多くのご指摘をいただき，研究への主体的な取り組み等，研究者としての心構えを学ばせていただきました.

筧捷彦先生には，確率的逆アセンブルにおける課題に関するご指摘等，今後の研究の展望を見据えた貴重なご意見・ご助言をいただきました．また学部・修士の学生の頃，先生の授業によりプログラミングの楽しさを学ぶことができ，それが今の私の原動力になっていることは疑う余地もありません.

松山泰男先生には，他分野での機械学習の適用例や評価結果の捉え方に関する数々の有益なご助言をいただきました．こうしたご指導を通して，コンピュータセキュリティ分野に留まらない視野の広さを，そして新たな発想へのきっかけを得ることができました.

中島達夫先生には，分類したマルウェアの網羅性に関するご指摘等を通して，本研究分野の本質的な課題を考えるきっかけを与えてくださいました．また本論文の執筆に関しても詳細かつ具体的なご指導をいただきました.

職場である日本電信電話株式会社では, 博士後期課程在学期間中，たくさんの方々からのご協力・ご助言をいただきました.

ＮＴＴアドバンステクノロジ株式会社の伊藤光恭氏には，日本電信電話株式会社にご所属のときより，長きに渡り本研究をサポートしていただきました．新しいことを追い求め

続ける姿勢や，人脈を次々に構築していくそのお姿は，私の研究への取り組み方の礎となりました．

日本電信電話株式会社の針生剛男氏，谷本直人氏，八木毅氏，川古谷裕平氏，青木一史氏，秋山満昭氏，中山心太氏，塩治 榮太朗氏には，会社の上司や同僚として，学業と仕事を両立するうえで様々なご協力をいただくとともに，本研究に関する活発な議論を行わせていただきました．

日本電信電話株式会社の石橋圭介氏，佐藤一道氏，インターネットマルチフィード株式会社の豊野剛氏とは，DNS やネットワークトラフィックの観点での議論を行わせていただき，端末とネットワークが連携するマルウェア対策の重要性を示唆していただきました．

日本電信電話株式会社の冨士仁氏，ＮＴＴコミュニケーションズの柏大氏には，入社当時よりお世話になり，社会人，そして研究者として一からご指導いただきました．

また，ここに挙げることができなかった職場の上司・同僚の方々にも多くのご支援をいただきました．深く感謝いたします．皆様のサポートなくして本研究を続けることはできませんでした．

ラックホールディングス株式会社の新井悠氏，株式会社セキュアブレインの星澤裕二氏は，アナライジング・マルウェアの執筆等，多くの刺激的なお仕事でご一緒させていただきました．良き理解者として，またときには良きライバルとして，お付き合いいただけていることに大変感謝しております．

村岡研究室の諸氏には研究室での生活を通して，本研究に対する多くのアドバイスをいただきました．

秋岡明香氏には，研究室での一つ上の先輩として温かく本研究を見守っていただきました．また機械学習やデータマイニングの観点でのご指摘は，分野横断的な視点を与えてくださいました．

池田潤一氏とは，ネットワーク視点でのマルウェア検出方法に関して，多くの議論を交わさせていただきました．

学部・修士の学生時代にお世話になった皆様，さらには社会人として村岡研究室に戻ってきたときにも，温かく迎え入れてくださった皆様には深く感謝しております．

その他にも，多くの方からご助力を頂きました. 私の研究を支えてくださった全ての方

に心より感謝いたします.

最後に，些細なことにも喜び元気づけてくれる両親・祖母・叔母，また，村岡研究室の仲間として，研究のアドバイザとして，様々な支えとなってくれた妻に深く感謝いたします.

著者研究業績

論文

種別	題名	発行年月	掲載箇所	著者
論文誌 1 ○ 査読付 国際会議 2 ○	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 Towards Efficient Analysis for Malware in the Wild	2010 年 9 月 2011 年 6 月	情報処理学会論文誌 Vol.51, No.9, pp.1622–1632 Proceedings of IEEE International Conference on Communications 2011	岩村誠 伊藤光恭 村岡洋一 Makoto Iwamura Mitsutaka Itoh Yoichi Muraoka

総説

種別	題名	発行年月	掲載箇所	著者
3	Anti-Malware Technologies	2010 年 7 月	NTT Technical Review, Vol.8, No.7	Mitsutaka Itoh Takeo Hariu Naoto Tanimoto Makoto Iwamura 他 5 名
4	マルウェア対策技術	2010 年 3 月	NTT 技術ジャーナル,2010 年 3 月号	伊藤光恭 針生剛男 谷本直人 岩村誠 他 5 名
5 ○	研究用データセット：マルウェア検体編　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	2010 年 3 月	情報処理学会会誌 Vol.51, No.3, pp.292–295	岩村誠 伊藤光恭

講演

種別	題名	発行年月	掲載箇所	著者
6 ○	IAT エントリ格納場所の特定方法	2011 年 10 月	マルウェア対策研究人材育成ワークショップ 2011	岩村誠 伊藤光恭 村岡洋一
7 ○	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	2010 年 6 月	電子情報通信学会, 情報通信システムセキュリティ研究会	岩村誠 伊藤光恭 村岡洋一
8	次世代ネットワークを変容させるネットワークセキュリティ技術：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（招待講演）	2010 年 3 月	情報処理学会創立 50 周年記念（第 72 回）全国大会	岩村誠
9 ○	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	2009 年 10 月	マルウェア対策研究人材育成ワークショップ 2009	岩村誠 伊藤光恭 村岡洋一
10 ○	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	2008 年 10 月	マルウェア対策研究人材育成ワークショップ 2008	岩村誠 伊藤光恭 村岡洋一
11 ○	隠れマルコフモデルに基づく新規逆アセンブル手法	2008 年 3 月	電子情報通信学会総合大会	岩村誠 伊藤光恭 村岡洋一

著書

種別	題名	発行年月	掲載箇所	著者
12	アナライジング・マルウェア——フリーツールを使った感染事案対処	2010 年 12 月	オライリー・ジャパン	新井悠 岩村誠 川古谷裕平 青木一史 星澤裕二

その他

種別	題名	発行年月	掲載箇所	著者
論文誌				
13	Design and Implementation of High Interaction Client Honeypot for Drive-by-download Attacks	2010 年 5 月	IEICE Transactions on Communication, Vol.E93-B No.5 pp.1131–1139	Mitsuaki Akiyama Kazufumi Aoki Yuhei Kawakoya Makoto Iwamura Mitsuataka Itoh
14	能動的攻撃と受動的攻撃に関する調査及び考察	2009 年 9 月	情報処理学会論文誌 Vol.50, No.9, pp. 2147?2162	青木一史 川古谷裕平 秋山満昭 岩村誠 針生剛男 伊藤光恭
査読付国際会議				
15	Controlling Malware HTTP Comunications in Dynamic Analysis System using Search Engine	2011 年 9 月	The 3rd IEEE International Workshop on Cyberspace Safety and Security	Kazufumi Aoki Takeshi Yagi Makoto Iwamura Mitsutaka Itoh
16	Memory behavior-based automatic malware unpacking in stealth debugging environment	2010 年 10 月	Proceedings of 5th IEEE International Conference on Malicious and Unwanted Software	Yuhei Kawakoya Makoto Iwamura Mitsutaka Itoh
17	MARIONETTE: Client Honeypot for Investigating and Understanding Web-based Malware Infection on Implicated Websites	2009 年 8 月	Joint Workshop on Information Security 2009	Mitsuaki Akiyama Yuhei Kawakoya Makoto Iwamura Kazufumi Aoki Mitsutaka Itoh
講演				
18	実行命令トレースに基づく動的パッカー特定手法	2011 年 10 月	マルウェア対策研究人材育成ワークショップ 2011	川古谷裕平 岩村誠 針生剛男
19	Dense Ship:サーバ型ハニーポット用仮想マシンモニタ	2011 年 6 月	電子情報通信学会, 情報通信システムセキュリティ研究会	川古谷裕平 岩村誠 伊藤光恭

種別	題名	発行年月	掲載箇所	著者
20	メモリ拡張によるアドレスに依存しないブレークポイント技術の提案	2010 年 10 月	マルウェア対策研究人材育成ワークショップ 2010	中山心太 青木一史 川古谷裕平 岩村誠 伊藤光恭
21	動的解析における検体動作時間に関する検討	2010 年 10 月	マルウェア対策研究人材育成ワークショップ 2010	青木一史 川古谷裕平 岩村誠 伊藤光恭
22	検索エンジンによるマルウェア接続先評価手法の提案	2010 年 6 月	電子情報通信学会, 情報通信システムセキュリティ研究会	青木一史 秋山満昭 岩村誠 伊藤光恭
23	Gumblar の長期観測による分析	2010 年 6 月	電子情報通信学会, インターネットアーキテクチャ研究会	秋山満昭 佐藤一道 岩村誠 伊藤光恭
24	OEP 自動検出によるマルウェアアンパック手法	2010 年 6 月	電子情報通信学会, 情報通信システムセキュリティ研究会	川古谷裕平 岩村誠 伊藤光恭
25	通信トラフィックの時系列分析によるボット活動の可視化と特徴検出	2009 年 10 月	マルウェア対策研究人材育成ワークショップ 2009	池田潤一 岩村誠 秋岡明香 村岡洋一
26	クライアントハニーポットにおける攻撃検知手法の実装と評価	2009 年 10 月	コンピュータセキュリティシンポジウム 2009	秋山満昭 川古谷裕平 岩村誠 青木一史 伊藤光恭
27	半透性仮想インターネットによるマルウェアの動的解析	2009 年 10 月	マルウェア対策研究人材育成ワークショップ 2009	青木一史 川古谷裕平 岩村誠 伊藤光恭

種別	題名	発行年月	掲載箇所	著者
28	DNS クエリグラフを用いた悪性ドメイン名リスト評価	2009 年 3 月	電子情報通信学会, インターネットアーキテクチャ研究会	石橋圭介 豊野剛 佐藤一道 岩村誠
29	Improving Accuracy of Black Domain List by Using DNS Query Graph	2008 年 12 月	The 2008 International Workshop on Information Network Design (WIND 2008)	Keisuke Ishibashi Tsuyoshi Toyono Makoto Iwamura
30	クライアントハニーポットを用いた Web 感染型マルウェアの実態調査	2008 年 10 月	コンピュータセキュリティシンポジウム 2008	秋山満昭 川古谷裕平 岩村誠 伊藤光恭
31	ステルスデバッガを利用したマルウェア解析手法の提案	2008 年 10 月	マルウェア対策研究人材育成ワークショップ 2008	川古谷裕平 岩村誠 伊藤光恭
32	Detecting Botnet by Matching DNS and Honeypot Data	2008 年 9 月	2008 OARC Workshop	Keisuke Ishibashi Tsuyoshi Toyono and Makoto Iwamura
33	半透性仮想ネットワークを用いたボットの動的解析手法の提案	2008 年 3 月	電子情報通信学会総合大会	青木一史 岩村誠 伊藤光恭
34	クライアント型ハニーポットによる悪意ある Web サイトの検出について	2008 年 3 月	電子情報通信学会総合大会	北村真一 岩村 誠 伊藤 光恭
35	Honey Patch : Honeypot における攻撃検知手法の提案	2006 年 3 月	電子情報通信学会総合大会	川古谷裕平 柳原忠明 岩村誠 伊藤光恭
36	バッファオーバーフロー発生要因の特定方法	2004 年 8 月	情報科学技術フォーラム	岩村誠 冨士仁
37	バイナリプログラムにおけるバッファオーバーフロー攻撃検知法と攻撃痕跡抽出法	2004 年 3 月	情報処理学会, コンピュータセキュリティ研究会	岩村誠 柏大

種別	題名	発行年月	掲載箇所	著者
特許 (登録済) 38	解析システム、解析方法および解析プログラム	2011 年 6 月	特許第 4755658 号	青木一史 岩村誠
39	攻撃検知装置、攻撃検知方法および攻撃検知プログラム	2011 年 5 月	特許第 4739962 号	川古谷裕平 岩村誠 柳原忠明
40	サービス不能攻撃検知システムおよびサービス不能攻撃検知方法	2011 年 3 月	特許第 4709160 号	濱田雅樹 冨士仁 岩村誠
41	バッファオーバーフロー脆弱性分析方法、データ処理装置、分析情報提供装置、分析情報抽出処理用プログラムおよび分析情報提供処理用プログラム	2010 年 12 月	特許第 4643201 号	岩村誠 冨士仁
42	中継装置および中継装置用プログラム	2010 年 7 月	特許第 4551316 号	岩村誠 副島裕司
43	スタックスマッシング攻撃防御方法、スタックスマッシング攻撃防御装置及びスタックスマッシング攻撃防御プログラム	2009 年 8 月	特許第 4358648 号	岩村誠 柏大
44	不正アクセス対処ルール生成方法及びその装置と、不正アクセス対処方法及びその装置と、不正アクセス対処ルール生成プログラム及びそのプログラムを記録した記録媒体と、不正アクセス対処プログラム及びそのプログラムを記録した記録媒体	2009 年 1 月	特許第 4253215 号	岩村誠 柏大

種別	題名	発行年月	掲載箇所	著者
45 特許 (出願中)	不正アクセス対処ルール生成方法, 不正アクセス対処方法, 不正アクセス対処ルール生成装置, 不正アクセス対処装置およびスタックスマッシング攻撃対策システム	2008 年 3 月	特許第 4091528 号	岩村誠 柏大
46	解析システム、解析方法および解析プログラム	2011 年 8 月	特開 2011-154727	青木一史 岩村誠
47 ○	類似性算出装置、類似性算出方法および類似性算出プログラム	2011 年 4 月	特開 2011-86147	岩村誠 伊藤光恭
48 ○	オリジナルコードの抽出装置、抽出方法、および抽出プログラム	2010 年 4 月	特開 2010-92179	岩村誠 伊藤光恭
49 ○	逆アセンブル方法および逆アセンブル装置	2009 年 8 月	特開 2009-193161	岩村誠 伊藤光恭

特記事項

- 論文 7 によりインターネットアーキテクチャ研究会学生研究奨励賞を受賞.
- 論文 9 が情報処理学会コンピュータセキュリティ研究会推薦論文に選出.
- 論文 28 によりインターネットアーキテクチャ研究賞を受賞.
- 論文 36 により FIT2004 ヤングリサーチャー賞を受賞.