課題研究報告書

マルウェアによるパッカー利用の解析

1310707　　冨沢 篤史

主指導教員　　小川 瑞史
審査委員主査　　小川 瑞史
審査委員　　緒方 和博
　　廣川 直

北陸先端科学技術大学院大学
情報科学研究科

平成 29 年 8 月

目次

1.背景

ICT の利用が今後もこれまで以上の速さで拡大していくことが明らかな中で、マルウェアもその活動領域を拡大し、これまで以上に社会に影響を及ぼすことが避けられない状況にある。そのような背景もありマルウェアに対する研究が様々な観点で行われているが、その中の領域の一つにマルウェアが自身の隠蔽化のために利用するパッカーについての研究がある。パッカー自身はプログラムのフットプリント低減やリバースエンジニアリング防止を目的とした汎用的なものであるが、その特性ゆえにマルウェアの性能向上に寄与するため、かなりの割合でマルウェアにおいてパッカーが利用されている[1,2]。

マルウェアによるパッカー利用の研究は大まかに以下の二つに分けられる。

・使用されるパッカー自身の特性や同定手法についての研究

・パッカーにより圧縮（packed）されたマルウェアの解凍(unpacking)に関する研究

どちらの研究においても、パッカーによる圧縮ロジックの解析を行い、マルウェア自身のプログラム開始点（Original Entry Point：OEP）の同定を行うことが主要なテーマとなる。

パッカーの同定においてはプログラムのバイナリーパターンを解析する手法が最も一般的な方式である[3,4,5]。exe ファイルの PE（Portable Executable）ヘッダーを CFF Explorer のようなバイナリーデータ解析ツール等を使用して直接観察することが可能であり広く利用されている手法であるが、マルウェアバイナリーのバイト列を一部改変することで検出が難しくなるという問題がある。そのためバイナリーのパターンマッチング方法にはさまざまな工夫が考案されている。また Sandbox の中で実際にマルウェアプログラムを実行し、ランタイムの挙動から実行するアプローチもあるが、パッカーにはデバッグされていることを検知した場合に実行を止めるなどの振る舞いを変える機能を持っているために必ずしも有効な手段とは言えない。

本学では小川研究室を中心にプログラムの制御フローグラフを生成する BE-PUM（ Binary Emulator for PUshdown Model generation ）をオンライン利用可能にしており（bepum.jaist.ac.jp）、これらを用いたマルウェアプログラムの逆アセンブルによるパッカー同定の研究[6,7]が行われている。BE-PUM による解析によってパッカー並びに隠蔽化テクニックの同定に成果を上げており、マルウェアによる情報流出を広く知らしめるきっかけにもなった EMDIVI のパッカーと OEP の同定にも成功している。

本課題研究ではこれまでに上述の BE-PUM によるパッカー同定が行われたマルウェア分析結果の逆アセンブルデータを直接観察し、その結果から OEP の検出を試みた。

パッカー自身がリバースエンジニアリング防止のために実施している各種の隠蔽化テクニックにはパッカー毎に利用傾向があることが既存研究から明らかになっている。そこでそれぞれの隠蔽化テクニックにより実行される CPU 命令には

規則性があるとの仮定を行い、特徴的な実行命令を逆アセンブルデータから抽出したものを用意し命令パターンの検出とマッチングを行うことでパッカーによる解凍コードとマルウェアのプログラムによる命令の分離を行った。

使用するサンプルは VXHeaven(http://vxheaven.org)で公開されているマルウェアデータを使用した。
VXHeaven はコンピューターウィルスに関する情報データベースであり、ウィルスのサンプル、ソースコード、生成を行うためのプログラムやチュートリアルなどを提供するサイトである。このサイトから入手可能なウィルスは 2000 年代に流通した比較的旧式のもので、最新の Windows OS やセキュリティ対策ソフトウェアを適用していれば悪用されることはないものであるが、マルウェアサイトのメインページに"Viruses don't harm, ignorance does!"との記載があるように、マルウェアの感染や攻撃手法のアイデアは昔のウィルスを参考にしているものも多く、ウィルス対策の研究素材の入手先として利用できるサイトである。
本課題研究は VXHeaven より入手可能なマルウェアサンプルのうち、約 5000 種類を BE-PUM によりパッカー同定を行った際の分析結果ファイル（逆アセンブルファイル）を使用し、利用パッカー別に分類して観察を行った。

以下に本課題研究の章立てを示す。
2 章では本課題研究の前提となるマルウェアの典型的な手法とそこで用いられるパッカーの典型的な隠蔽化テクニックについて説明する。3 章では BE-PUM の概要をまず述べて上でパッカーの隠蔽化テクニック同定のために行われた機能拡張について説明している。
4 章は本課題研究におけるパッカーならびに OEP 同定のアプローチについて解説を行い、5 章から 8 章にかけてパッカー別の観察結果と考察を述べる。今回観察したサンプルは ASPack, PEComnact, UPX, Upack の 4 種類である。9 章は本課題研究のまとめと今後の課題を記す。

関連研究

BE-PUM を使ったパッカーの隠蔽化テクニック同定の研究[7]においては、パッカーで行われる典型的な隠蔽化テクニックを既存のサーベイ結果[8]を基に行った分類とマルウェアサンプル観察を通して得た検出方法を定義し、それらを検出するための BE-PUM 拡張を行っている。これら分類に沿った隠蔽化テクニックの出現頻度をマルウェアサンプルにて計数、その計数結果をパッカーのメタデータとして利用することでパッカー同定を行っている。当手法によるパッカー同定判断には、[9]のアイデアを参考にカイ二乗検定を適用している。パッカー毎に定義された隠蔽化テクニックそれぞれの出現頻度を BE-PUM の逆アセンブル結果が超えることでパッカーを同定するこの方法を使用すると、パッカーが同定された命令の近傍にマルウェア自身のコードの始まり、つまり OEP の近傍の検出も併せて可能になる。 これにより、例えば BE-PUM による EMDIVI の解析では、まず UPX をパッカーとして使用していることを同定し、UPX による元のファイルの解凍コードが popa 命令で終わるという既知の情報と併せて OEP のアドレスを探し出している。

上述のパッカー同定方法は VXheaven データベースの 5374 種類のマルウェアサンプルに対して実験で適用することで評価を行い、市販のパッカー（CEiD、CFF Explorer および VirusTotal）よる解析では 327 サンプルで同定ができなかったのに対して、BE-PUM では機能拡張による同定対応ができない 1 サンプルを除きすべてのデータで同定ができた。

本課題研究で OEP 検出に用いたサンプルデータは当該マルウェアサンプルの逆アセンブル結果である。

2.マルウェアの典型的手法とパッカーの隠蔽化テクニック

○マルウェア

マルウェア（malware）とは malicious と software を組み合わせた造語であり、コンピューターの利用ユーザーが意図、もしくは期待していない動作を実行することで迷惑をかけることを目的とするソフトウェアを意味する。自己増殖を行うもの（ワーム、ウィルス）、コンピューターの制御の乗っ取りやデータの消失や外部流出を行うもの（トロイの木馬、スパイウェア）がマルウェアに該当する。またそれらの攻撃を予告、もしくはすでに行った攻撃による損失の回避を語り金銭を要求するランサムウェアもマルウェアの一種である。

○パッカー

パッカー（packer）はソフトウェアの実行ファイル（.exe ファイル）を実行可能な状態を保持したままでそのサイズを圧縮するツールである。パッカー自体は格納ストレージの容量を削減したり、ファイル転送時にネットワークの負担を低減したりと、ごく一般的な用途で利用されているものであるが、その特性が以下のような形でマルウェアにも悪用されている：

- 元のファイルとは異なる性質（名前、サイズ）を持たせることができるため、標的とする環境（ユーザー、マシン）での実行を起こしやすくする。
- 圧縮の過程で行われるプログラムコードの難読化（Obfuscation）により、セキュリティ対策ソフトウェアによる検出を難しくする。

パッキングされたファイルはコードセクションにあるパッキングを元に戻すプログラム（unpacking コード）と、データセクションにある圧縮された元の実行プログラムファイル（packed ファイル）から構成されている。

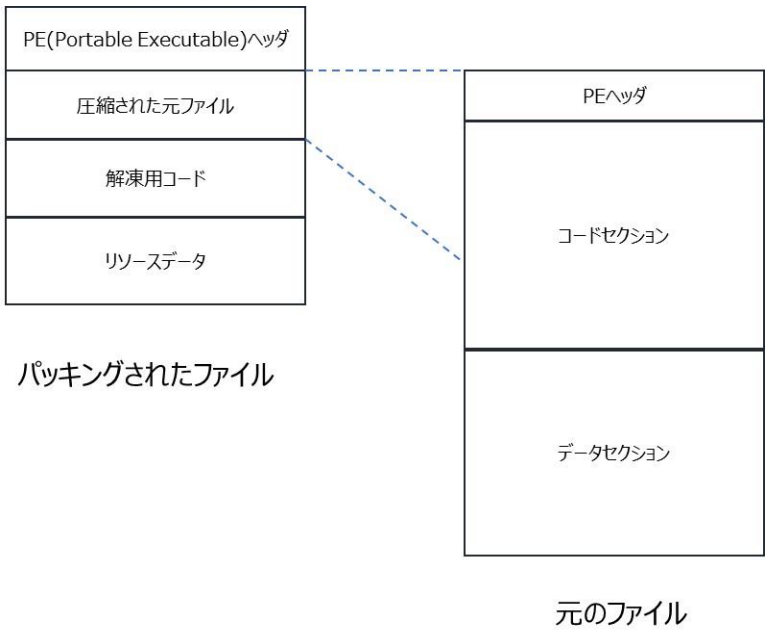


図 2-1：元のファイルとパッキングされたファイル

パッキングされたファイルが実行されると、まず unpacking コード実行されることで packed ファイルがコードセクション上に解凍・展開され、引き続きこの解凍された元ファイルが実行される。メモリ上で unpacking コードの実行終了後に元ファイルの実行が始まる点を OEP（Original Entry Point）と呼び、パッキングされたファイルをアンパックして元のファイルを解析するためには OEP の特定が必要となる。

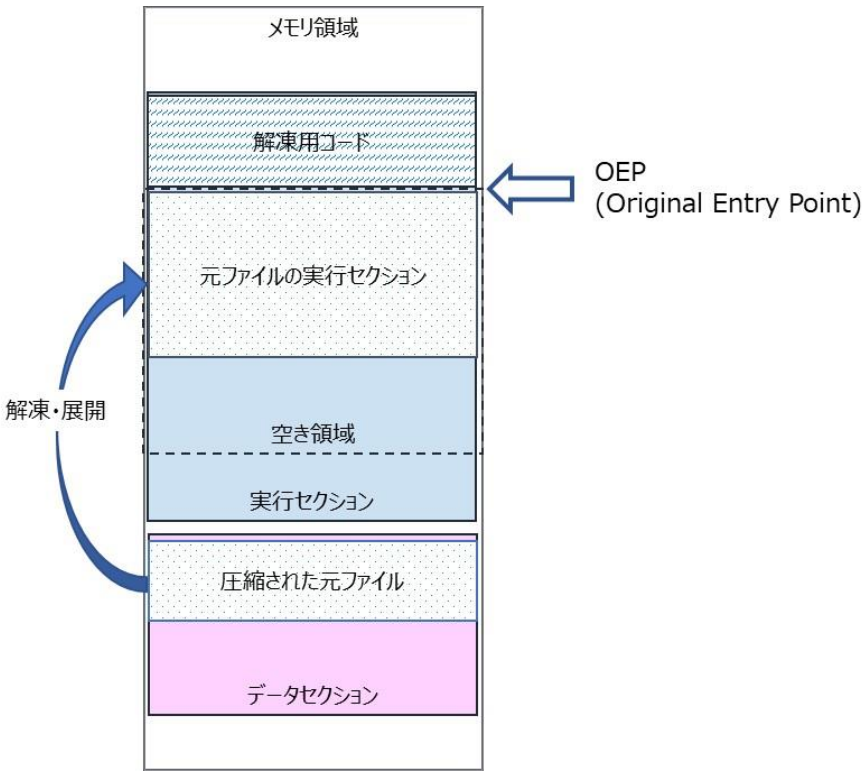


図 2-2：パッキングされたファイルの実行イメージ

○マルウェアの隠蔽化テクニック

本課題研究では既存研究にならい以下の 6 グループに分類した難読化テクニックに着目し、それらが実行される際に呼ばれる命令の出現傾向を観察する。ここではそれぞれのグループに分類されるテクニックと発生する命令について概説する。

< 1. Entry/code placing obfuscation: Code layout >

可変長命令セットを持つ x86 アーキテクチャーの特性を利用してひとつの命令列を複数ブロックにまたがって保持し、プログラムの実行過程で結合して目的の命令を呼び出す。

< 2. Self-modification code（Dynamic code: overwriting and packing/unpacking） >

コードの一部または全部の上書きや暗号化・複合化処理を行う。

< 3. Instruction obfuscation: Indirect jump >

call,jump,return と共に利用するオペランドの値を書き換えることでそれぞれの実行後に飛ぶアドレスを変更する。

< 4. Anti-tracing: SEH（structural exception handler）, 2API（kernel32.dll 内の LoadLibrary と GetProcAddress の利用） >

構造化例外処理で提供している例外ハンドラーの悪用や隠蔽化を行う際に利用する Windows API（LoadLibrary@kernel32.dll,GetProcAddress@kernel32）の呼び出しが見られる。

< 5. Arithmetic operation: Obfuscated constants, check summing >

オペコードに演算をかけることで直接値を保持しないようにする、セキュリティチェックが行うデバッグによりで発生する命令の置き換えを確認するためにチェックサムを導入する。

< 6. Anti-tampering; Timing check, anti-debugging, anti-rewriting, and hardware breakpoints >

セキュリティチェックによる改竄などを検知するために、IsDebuggerPresent の値や実行時間（デバッグによる解析が間に挟まれることで想定以上の実行時間がかかっていないか）のチェックが行われる。またデバッガによるブレークポイントの設置の確認や、もしくはブレークポイントを迂回するため関数の先頭を自身が確保したメモリ領域にコピーして利用する（stolen bytes）なども行われる。

3. BE-PUM の概要とパッカー同定

BE-PUM （ Binary Emulator for PUshdown Model generation ）はプログラムの制御フローグラフを生成するツールである。インテル x86 アーキテクチャーのバイナリーコードの逆アセンブルに対応し、典型的なマルウェアの隠蔽化テクニックの解析に対するこのツールの有効性は既存研究にて示されている。

BE-PUM ではプログラムのフローグラフ生成のためにコンコリックテストを活用している。コンコリック（concolic） は concrete と symbolic から作られた混成語であり、コンコリックテストはプログラムの各ステップを実現するためのシンボルの具体的（concrete）なインスタンスを計算しながらシンボリック実行を行う手法である。プログラムの次のステップに進む際に各シンボルが満たすべき値をソルバーによって算出し、それらを次ステップへの入力として使用していくことで、単純なシンボリック実行よりも効率よくプログラムのパスを網羅することができる。

BE-PUM はシンボリック実行、バイナリーエミュレーション、および生成した制御フローグラフのストレージの 3 要素で構成されている。バイナリーコードの逆アセンブルには JakStab 0.8.3[10,11] を使用し、コンコリックテストにおける変数の具体的なインスタンスの生成には SMT ソルバー Z3 4.4 を用いている。

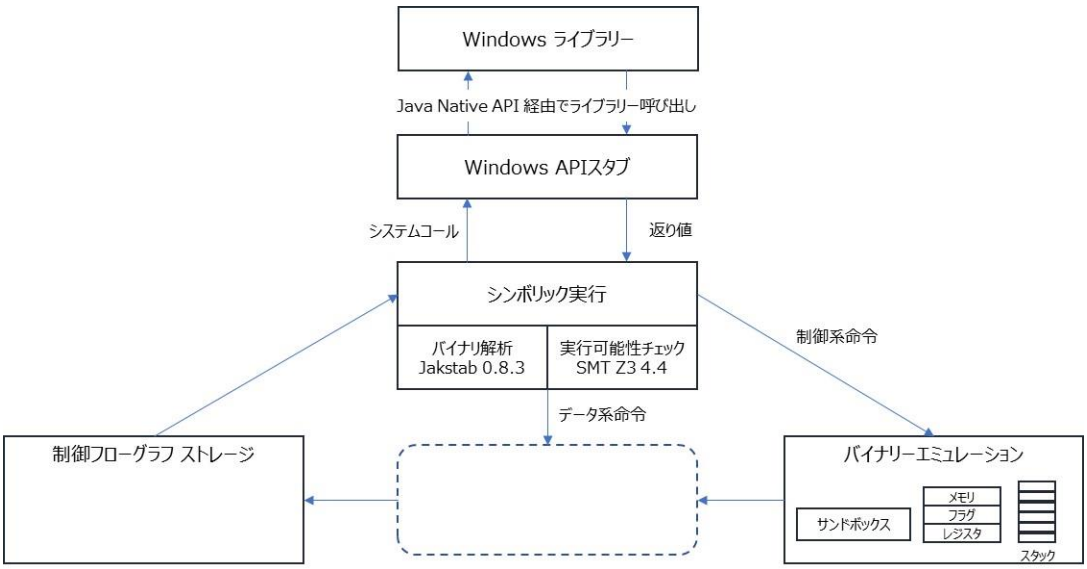


図 3-1: BE-PUM 概要

バイナリーエミュレーションは x86 命令の解釈および Windows API スタブの呼び出しを実行する。Windows API スタブを呼び出した場合は Java Native API（JNA）を介して実際の Windows API を実行し、返り値と

プロセスの更新情報のフィードバックを受ける。バイナリーエミュレーションの実行結果を基に次の実行パスの状態を算出するためのコンコリックテストが行われ、プロセスのメモリ状態などを表すシンボルのインスタンス情報が次のステップに向けて更新される。

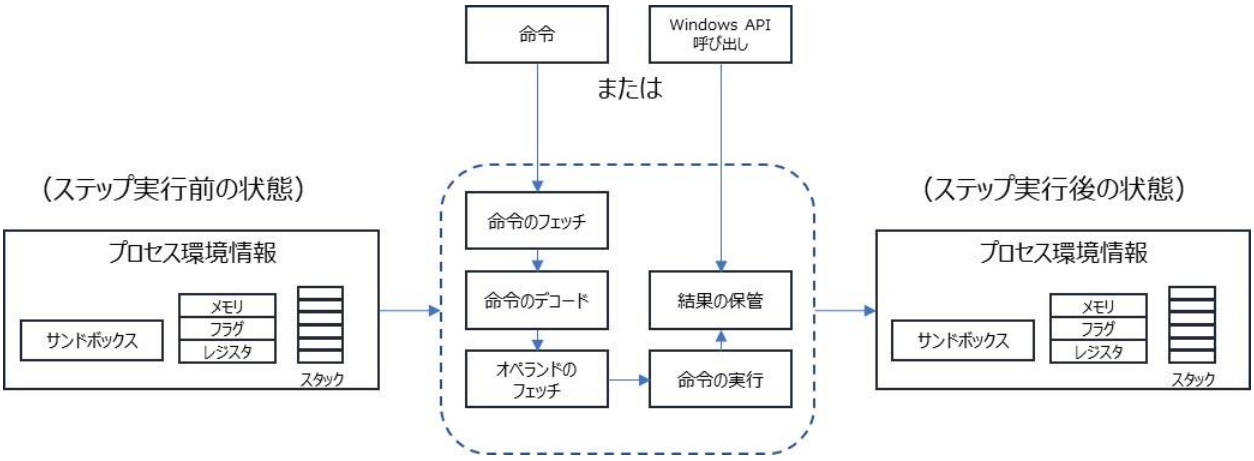


図 3-2: BE-PUM によるバイナリーエミュレーション

BE-PUM がサポートしていない命令や Windows API の実行や例外処理に移った場合は BE-PUM の処理が停止する。

パックされたコードが実行されるときに呼び出される x86 命令や Windows API について<既存の研究>から以下の特徴があることが明らかになっている。

・解凍用コードによる元のファイルの解凍

元のファイルをメモリ上にロードするために、VirtualAlloc@kernel32.dll で新たに仮想メモリ空間の割り当てを行うか、または空きセクションを再利用する。圧縮されている元のファイルはその領域に解凍が行わる。併せて解凍用コードは元のファイルが利用する Windows API のインポートテーブルも展開し、GetProcAddress@kernel32.dll と LoadLibrary@kernel32.dll を使用してインポートするライブラリを動的にロードする。

・完全性チェック対応

アンチウィルス対策などによる分析の検知を行うため、自身の実行時間を計測することで分析等によって想定以上の時間がかかっていないかを確認する仕組みを持つものがある。具体的には x86 アーキテクチャーで提供されている

RDTSC 命令を使用することで CPU クロックごとに加算されるタイムスタンプを適当な区切りで取得することで実行時間を計測する。

・デバッグモード検出

パッカーの中には自分自身が実行されている環境を確認し、マルウェア解析でよく用いられるデバッグモードでの実行やエミュレータでの実行を検出した場合は挙動を変え、目的の動作を行わないようにするものも存在する。デバッグモードでの稼働有無については、 IsDebuggerPrsent@kernel32.dll の使用が最も基本的な確認方法である。この命令を直接使用する他にも、Yoda のように実行中のスレッド情報を保持するデータ構造体である TIB（Thread Information Block）から実行プロセスの各種情報を保持した構造体である PEB（Process Environment Block）のアドレスを取得（32bit モードでは TIB の構造体のオフセット 0x30 に PEB のアドレスが入っている）、BeingDegguged 変数の値を読み取って判別を行っているものも存在する。

上記の特徴を踏まえ、現在の BE-PUM ではパッカー同定のために、パッカーによる使用頻度が高い 200 の x86 命令と 140 の Windows API の追加サポートやバイナリーエミュレーション実行時の実メモリのシミュレーション、トラップフラグとデバッグレジスターの追加などの機能拡張が行われている。

BE-PUM ではパッキングされたファイルを直接逆アセンブルして実行命令列を生成することを利用して元のファイルの OEP の近傍を特定することができる。これと利用されているパッカーの補足情報と併せると元のファイルの OEP を同定することが可能になる。例として EMDIVI の解析では、まず利用しているパッカーが UPX3.0 であることを同定し、UPX3.0 の解凍用コードが POPA 命令で終わるという事実から OEP 近傍の POPA 命令を検索することで EMDIVI 本体の OEP 同定に成功している。

4. パッカーおよび OEP の同定アプローチ

本章ではパッカーおよび OEP 同定を行う手法を説明する。最初に OEP の定義を明らかにした上で既存研究および本課題研究それぞれで採用した同定手法について説明する。

OEP(Original Entry Point)とはパッカーを通して実行されたプログラムが開始するメモリ上の場所であり、パッカーによる解凍用コードの最後の命令によって本来のプログラムがメモリ上に復元されたタイミングで CPU が参照しているメモリアドレスを指す。OEP の近傍を同定することでプログラム、すなわちマルウェアの状態を復元して解析することが可能になる。また、OEP を正確に同定することができれば、パッカーからマルウェアのプログラムを分離することができるため、バイナリーレベルでのマッチングが可能になり、異なる種類のパッカーを利用することで隠蔽化が行われるような場合においてもマルウェアの同定が可能になることが考えられる。

BE-PUM を用いたパッカー同定のアプローチとして、隠蔽化テクニックの形式的な定義とその出現頻度に基づいて統計的に同定を行うアプローチが確立されている。以降でその具体的な手法を説明したのち、本課題研究で採用した制御フローグラフから特定の命令を抽出することで抽象化列を生成・比較する手法を説明する。

マルウェアの逆アセンブル結果の観察により、解凍用コードが実行される際には様々な隠蔽化テクニックが組み合わされて実行される。各テクニックの実行回数、実施順序にはパッカー毎に固有の傾向がみられることがわかっている。そこで 2 章で紹介した 6 種類あるパッカーの典型的な隠蔽化手法から 14 種類の具体的な隠蔽化手法に対して形式的な定義を行い、それらを BE-PUM で検出するための機能拡張が行われた。BE-PUM を使用することで逆アセンブル結果から各テクニックの出現を逐次カウントできるので、その計数結果とパッカー毎にサンプルプログラムを通して算出した各隠蔽化テクニックの出現頻度を都度比較していくことでパッカー同定を行っている。その際に各隠蔽化アプローチの出現頻度の確からしさはカイ二乗検定により確認している。

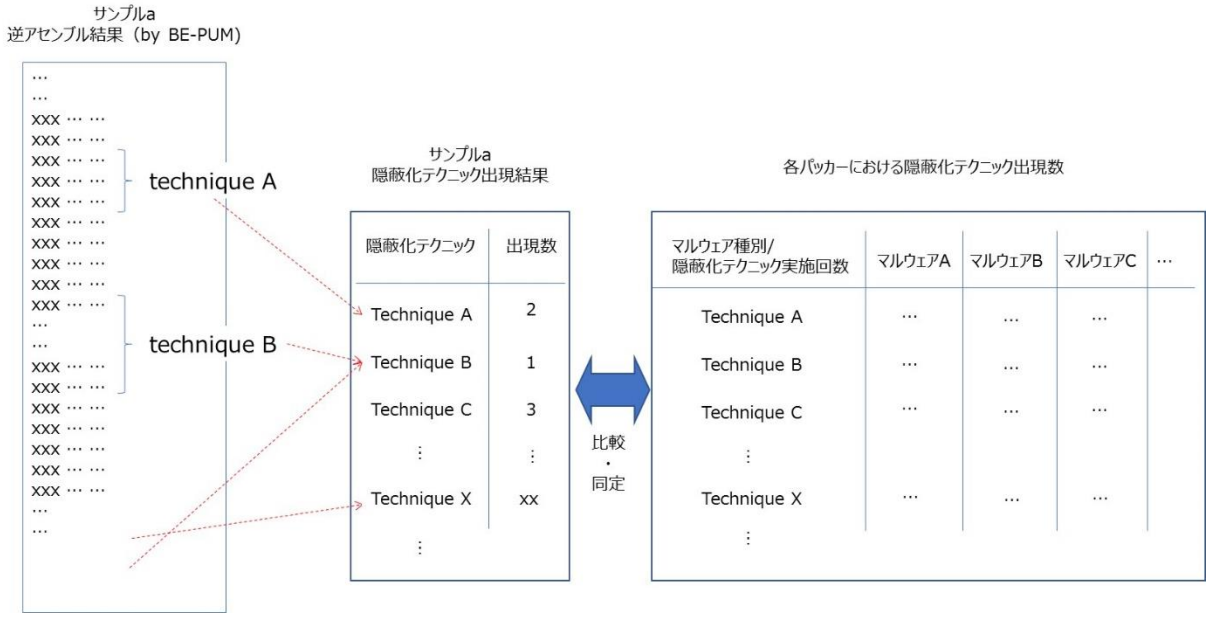


図 4-1: 既存研究におけるパッカー同定アプローチ

一方、本課題研究では BE-PUM の解析結果を直接観察し、パッカーならびに OEP の同定の試行を以下に説明するアプローチで実施した。

まず観察を始めるにあたり、解凍した元ファイルの実行領域を格納するためにメモリ割り当てを行うパッカープログラムのメモリ構造の特性と、[7]によりパッカーの隠蔽化テクニックの出現順序に規則性があることが判明していることを踏まえ、以下の仮説を立てる。

（仮説 1）：隠蔽化テクニックと x86 命令群について
各テクニックの実行結果としての命令列には規則性があり、同じパッカーを使用していれば BE-PUM の制御フローグラフの直接観察にて命令の出現に共通した傾向が見て取れる。

（仮説 2）：Windows API やライブラリ・関数呼び出しの出現傾向について
パッキングされたファイルが実行される際には、解凍用コードおよび解凍された元のファイルが実行される際にそれぞれ Windows API が呼ばれる。つまり解凍用コード実行後に元のファイルによる Windows API 呼び出しや関数・プロシージャーの実行のための Call 命令が観察される。

仮説 1 に従えば同じパッカーを使用しているマルウェアサンプルの命令実行列を横に並べていけば、解凍用コードの実行を行う部分には規則性が見られるはずであり、OEP の前では共通の命令が実行され、かつその後各マルウェアの実行目的に合った処理が行われるために命令の出現には規則性が見られなくなることが推測される。
また仮説 2 と併せるとまず解凍用コードの実行時に呼び出される Windows API や関数・プロシージャーの呼び出しが各サンプルで共通してあらわれ、その後処理が進んで元のファイル（マルウェア）が実行されるタイミングで再度 Windows OS への接触が Windows API の利用という形で現れるはずである。

上記仮説のもとにパッカー毎に逆アセンブル結果から命令出現に規則性が観察できれば、それはパッカーの特性にほかならず、出現パターンを観察することでパッカーの同定が可能になる。

今回は観察結果を俯瞰、相互比較しやすくするため、下図のように制御フローグラフから特定の命令列のみを抽出したデータのサブセットを作成することで抽象化を行い、その抽象化した命令列を比較をすることにした。サブセットデータもまた特定の出現パターンを保持するため、観察対象の情報量を削減でき、多数のサンプルデータを同時に観察しやすくした。
また、パッキングされたファイルは[解凍用コード] -> [元のファイルの実行コード]の順番に実行されるため、同じパッカーを使用したサンプルの逆アセンブル結果では、[パッカー毎に固有の命令実行パターン] -> [マルウェアごとの異なる命令実行結果] の形で命令が並ぶので、この境界が OEP 近傍を表すことになる。

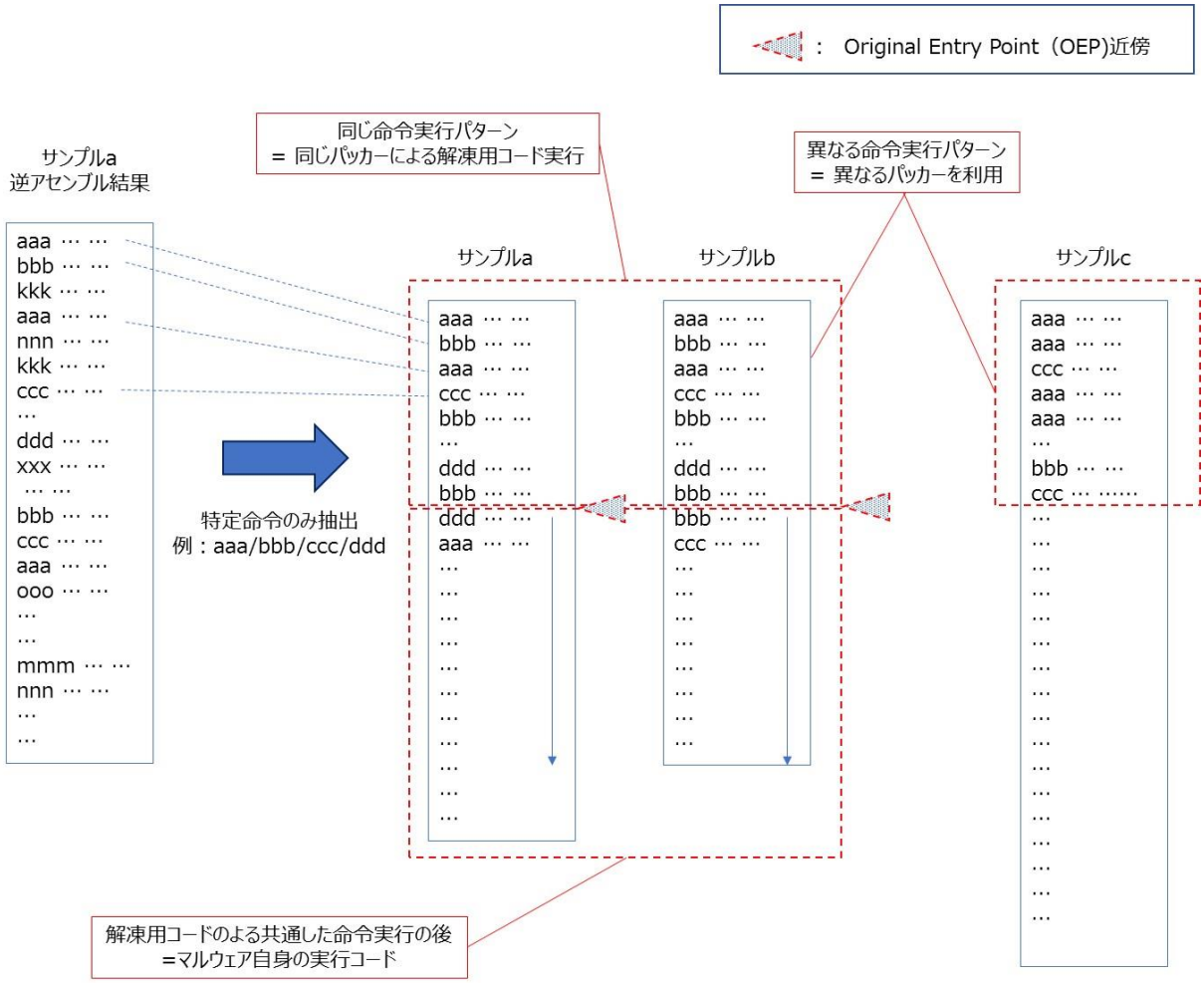


図 4-2: 観察およびパッカー同定のアプローチ

もとの逆アセンブル結果に戻りサブセットデータより得た OEP 近傍を示す命令列より下を詳細に観察し、パッカー毎のサンプルデータに共通する命令の実行パターンを探し出すことで最終的な OEP を求めることができる。

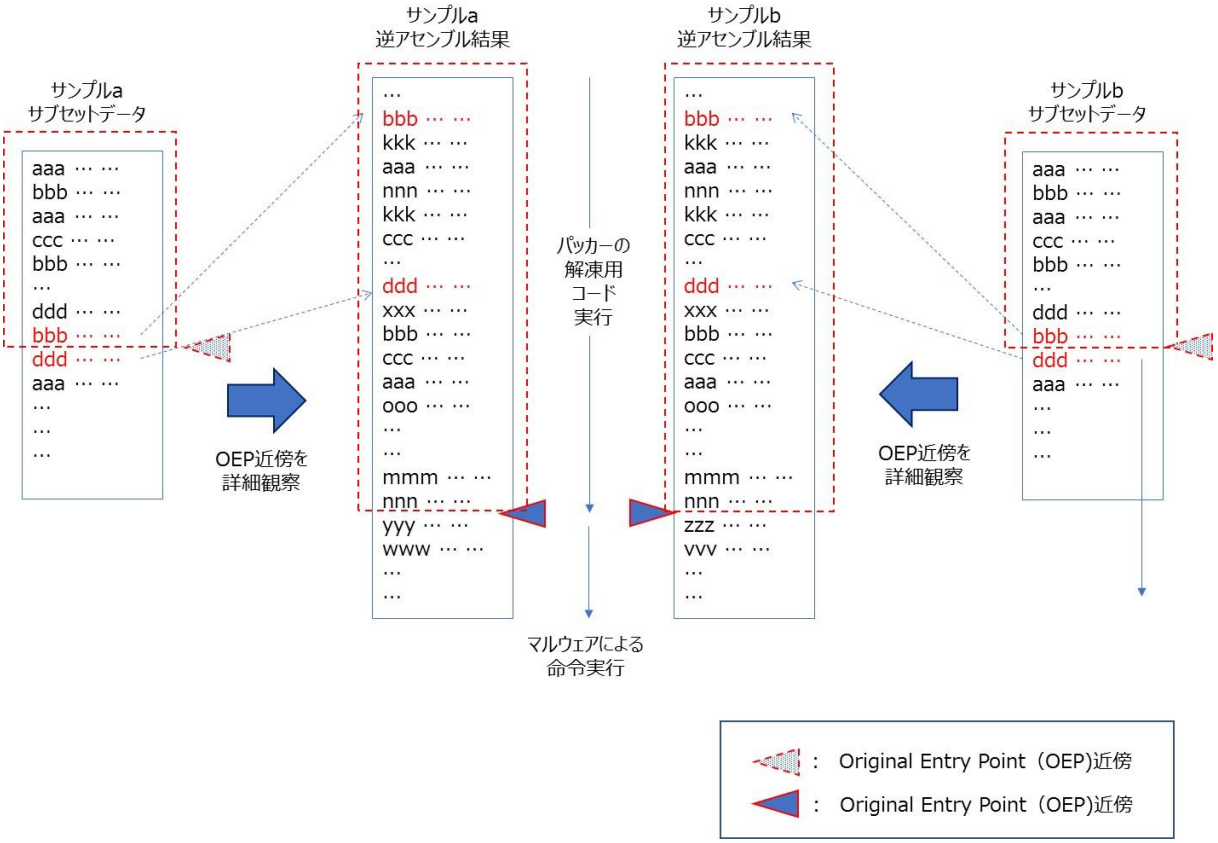


図 4-3: OEP 同定アプローチ

上記手法にて BE-PUM 実行結果を観察するにあたり以下の準備を行った。

（準備 1）利用パッカー別 BE-PUM 実行結果リスト（以下、フルセットデータと呼ぶ）

Excel に各 BE-PUM 実行結果データを 1 列ずつ並べたものである。
各列ともに上から、データ管理用 ID、マルウェアサンプル名、マルウェア同定結果（ヘッダー解析）、マルウェア同定結果（BE-PUM による解析結果）を記載し、空白を挟んで BE-PUM 解析による命令列のダンプを張り付ける。観察を容易にする上でメモリアドレス情報が付与されている生データと実行命令のみを選択的に張り付けたものを 2 つ用意し、観察時には後者を、メモリアドレス検索の際には前者という形で使い分けを行う。

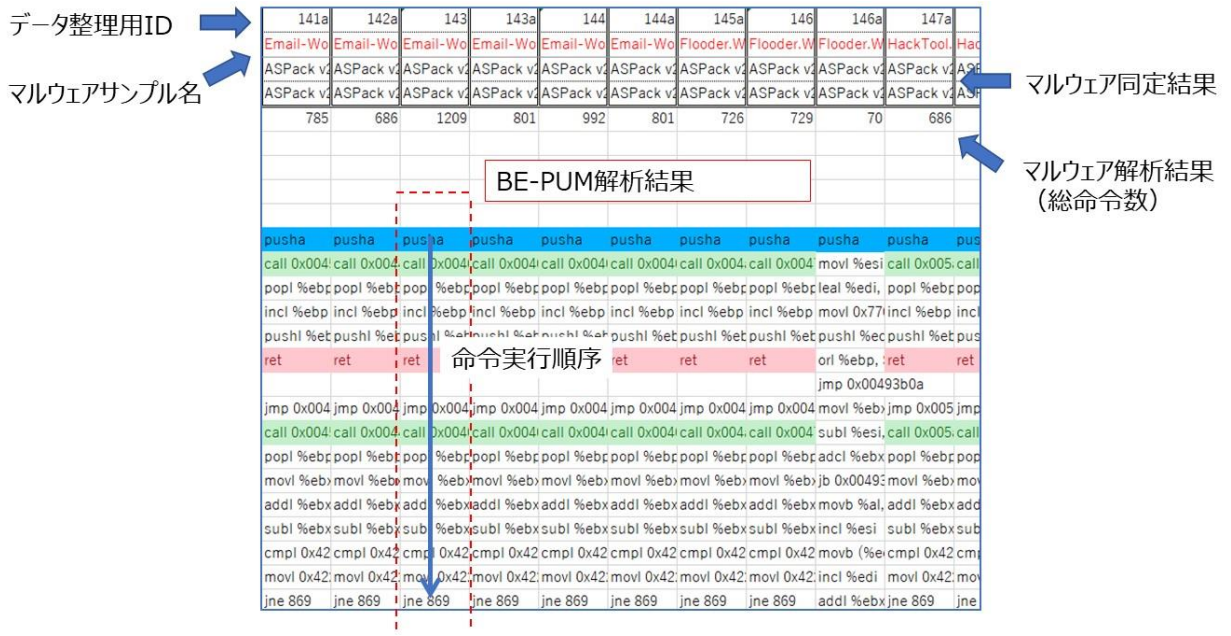


図 4-4: フルセットデータ

（準備 2）フルセットデータから特定の命令列のみを抽出した結果リスト（以下、傾向分析サブデータ）

複数サンプルの比較観察を容易にするためにフルセットデータを抽象化したものである。上述の仮定のもとで命令の出現傾向の観察を容易にするため、フルセットデータから Windows API 呼び出しならびに関数・プロシージャーの実行を表す以下の命令群のみを抽出したリストを用意する。その際に各命令・呼び出しの実行結果に対してカッコ内で表す着色を施し、スプレッドシート上にマルウェアサンプル毎に転記することで、視覚的に出現パターンの有無を確認することができるようにする。

・Windows API（ XXXX@kernel32.dll ）の呼び出しが出現するすべての命令 (黄色)
・call 命令（緑色）
・ret 命令（薄赤色）
※実際にはこれらに加えて既存研究で UPX の解凍用コードの最後に現れることがわかっている popa と pusha 命令（青色）も併せて残している。

今回上記 Windows API 呼びだしを抽出対象にしたのは、解凍用コードの中でマルウェア本体の実行時に必要な API ライブラリのロードや最終的に実行される悪意のあるプログラムによる API の実行が必ず行われるため、BE-PUM によって解凍用コードの実行完了まで解析が成功していることを確認でき、かつ他の命令列よりも出現頻度が

相対的に低いためより抽象度の高い命令列を生成できると考えたからである。同じ理由でプロシージャー的なプログラムの実行及び終了時に呼び出される call および ret も対象に加えた。

以下に傾向分析サブデータを用いた具体的な観察例を示す。ここではデータ ID146a 以外のサンプルにおいて青->緑->薄赤->緑->黄色…というパターンが出現していることがわかる。今回の仮説に基づけば、それぞれのパッカー毎に解凍用コードの命令列から抽出した傾向分析サブデータではこのようなパターンが出現することが予想される。以降ではこの特定の命令を抽出して色付けしたものを命令出現パターンと表現する。

図 4-5: 傾向分析サブデータ

フルセットデータならびに傾向分析サブデータはマルウェアが利用するパッカー毎に用意する。観察対象としては、既存研究[7]でパッカー同定実験を行った際に入手した 2000 サンプルを含む VXHeaven データベースから入手可能なマルウェアの BE-PUM による逆アセンブル結果（具体的には JakStab が出力した asm ファイル）を利用す

る。

今回観察したサンプルデータの諸元を以下に示す。

表 4-1: 観察対象のマルウェア

パッカー	バージョン	サンプル数
ASPack	2.12	194
PECompact	v2.0	494
UPack	v0.37-v0.39	382
UPX	V3.0	2535

各サンプルの傾向分析サブデータから命令出現パターンの有無を観察し、その観察結果を基に解凍用コードの終了点近傍を突き止めたのち、後続の命令群の出現傾向や特徴からパッカー毎に OEP の同定を行う。

5. ASPack のパッカーと OEP の同定

5.1. 分析結果の観察

今回観察した ASPack サンプルデータの諸元を以下に示す。

表 5-1: マルウェアサンプル（ASPack）

諸元	値	備考
サンプル数	194	
ヘッダー解析による同定	153	
BE-PUM による同定	194	
最小命令数	4	マルウェア名：欄外に記載（※）
最大命令数	5145	マルウェア名：Backdoor.Win32.Hupigon.y

※サンプル名：

069d840b55c4ec59ae66e1a780677e96d0032288b5aa3da4bb313d542636b110（原文まま）

次頁に示す図はフルセットデータから全サンプルの最初から 245 番目までの命令を抜き出したものである。

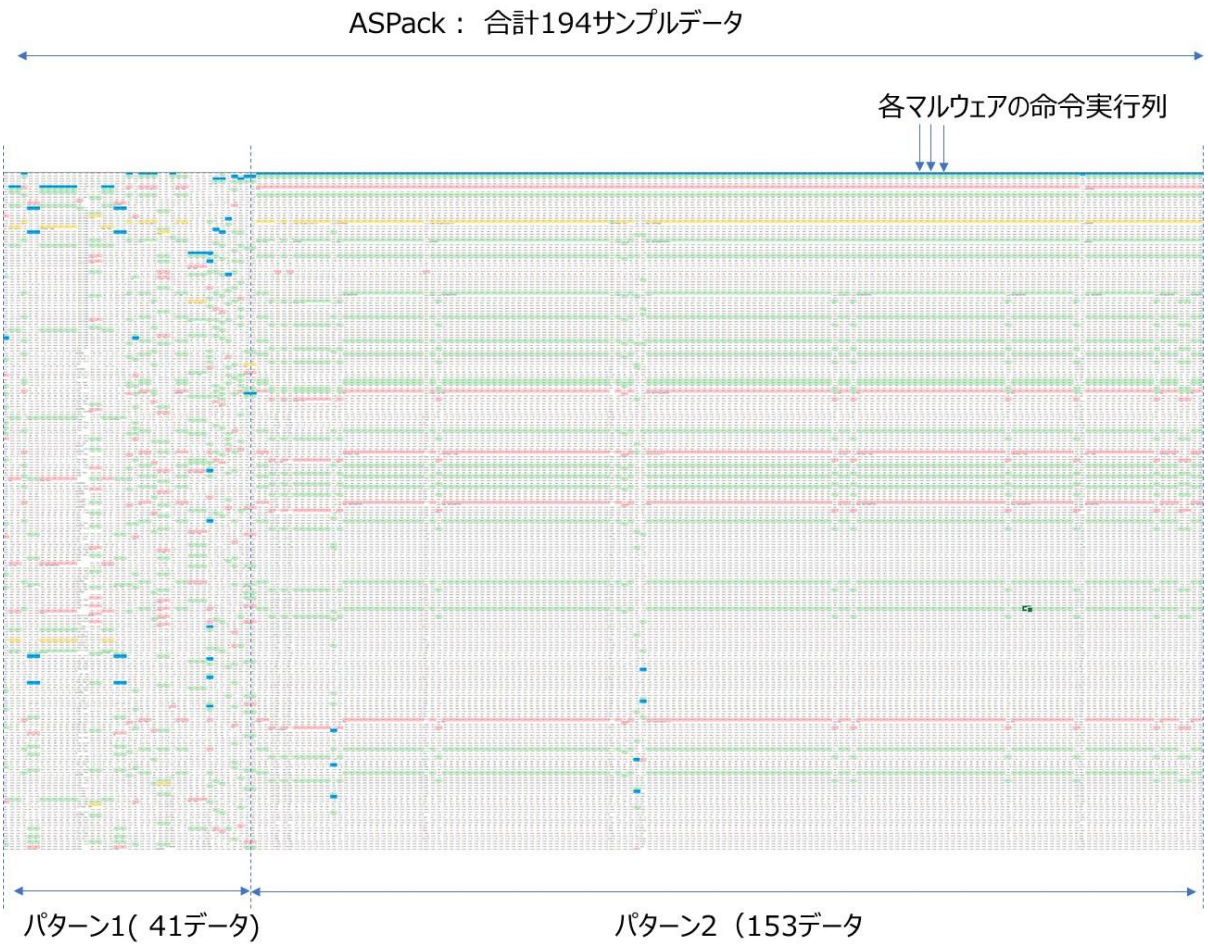


図 5-1: フルセットデータ（ASPack）

ASPack のサンプルデータにはヘッダー解析によりパッカー同定ができなかったが、BE-PUM で同定結果がでているものがある。そこで前者をパターン A-1、後者をパターン A-2 としてまとめ直すと以下のようになる。

表 5-2: BE-PUM のみパッカー同定ができたサンプル（パターン A- 1 ）

諸元	値	備考
サンプル数	41	
最小命令数	4	マルウェア名：上述の記載を参照
最大命令数	3908	マルウェア名：欄外に記載（※）

※サンプル名：

070ca820eda066a995a9b94dd536f369e6375969105e1e663b7254e6eac6fdbe

表 5-3: パターン A-1 に属するマルウェア

以下のマルウェアは同じパターン（A-1）	
マルウェア種別	サンプルデータ数
その他	30
Backdoor.Win32	3
Net-Worm.Win32	1
Trojan.Win32	3
Trojan-Downloader.Win32	3
Trojan-PSW.Win32	1
Virus.Win32	2

表 5-4: ヘッダー解析/BE-PUM 両方でパッカー同定ができたサンプル（パターン２）

諸元	値	備考
サンプル数	153	
最小命令数	70	マルウェア名：Flooder.Win32.Delf.v
最大命令数	5145	マルウェア名：Backdoor.Win32.Hupigon.y

表 5-5: パターン A-2 に属するマルウェア

以下のマルウェアは同じパターン（A-2）	
マルウェア種別	サンプルデータ数
Backdoor.Win32	119
Constructor.Win32	3
その他	1
Email-Flooder.Win32	2
Email-Worm.Win32	6
Flooder.Win32	3
HackTool.Win32	4
Net-Worm.Win32	5
not-virus_Hoax.Win32	1
Packed.Win32	1

Trojan-AOL.Win32	1
Trojan-Clicker.Win32	7

また、ASPack の傾向分析サブデータを以下に示す（それぞれ最初から 150 個分の抽出した命令を図示している）

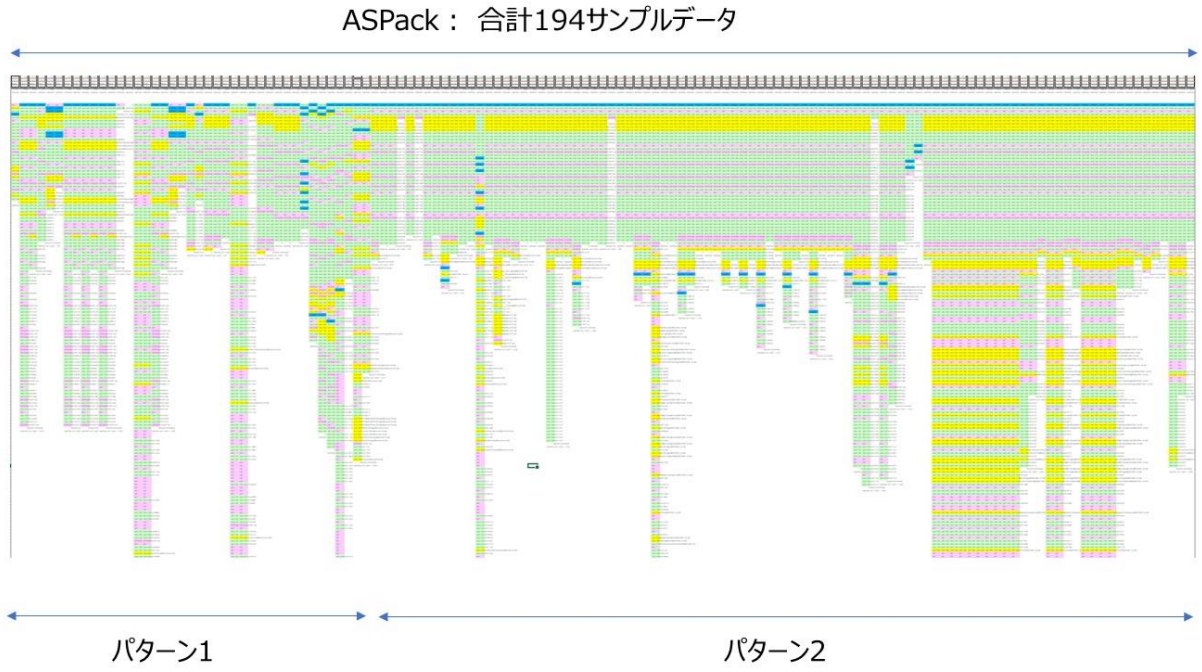


図 5-2:　傾向分析サブデータ(ASPack)

パターン A-2 に属する 153 のサンプルデータについては、合計命令数が 300 以下の 6 つとサンプル中最大の命令数（5145）をカウントした Backdoor.Win32.Hupigon.y を除きすべてにおいて命令の出現傾向が同じであった。

傾向分析データにおける命令順序を上げていくと、最初は pusha 命令で始まり（フルセットデータも同じ：つまりすべてのパッキングされたファイルは pusha から始まっている）、以下の順序で命令が実行されている。
pusha（開始） -> call -> ret -> call -> call GetModuleHandleA@kernel32.dll -> call GetProcAddress@kernel32.dll -> call GetProcAddress@kernel32.dll (2 回目) -> call VirtualAlloc@kernel32.dll -> call VirtualAlloc@kernel32.dll (2 回目) -> call -> ...

途中 VirtualAlloc@kernel32.dll が呼ばれており、ここでは解凍用コードが元のファイルを差し込む領域を確保するための空き領域の割り当てを行っていることが推測される。

その後、ret 命令と call 命令が続き、総命令数が約 700 を超えるサンプルにおいては再度 Windows API の呼び出しが見られる。ただし名前から明らかに亜種もしくは同種であるものを除くと、ここからの命令は各サンプルで規則性は見られないため、ここではすでに元ファイルの実行が行われているものと推測された。なお、総命令数が 700 を超えないサンプルについては、他のサンプルと同じ命令の出現傾向が見られるが Windows API の呼び出しは行われずに終了している。これについては以下の 2 点が考えられるが、フルセットデータに立ち戻って他の命令も含めた実行結果を見ると、すべて同じ傾向が見られたため、後者であると推測する。

1. Windows API 呼び出しが発生しない形でマルウェアの実行（潜伏・発症など）が完了した。
2. BE-PUM による解析が元ファイルの実行前のところで途中終了した（例：サポートしていない命令・API 呼び出しの発生など）。

命令数が比較的少ないパターン A-2 の下記サンプルについても、途中で命令出現傾向から外れる（命令数 70,71 のマルウェアは最初の命令「pusha」のみ一致、他の 4 つは 13 命令目まで一致）。

表 5-6: 命令数の少ないサンプルデータ(ASPack)

命令数	マルウェア名
70	Flooder.Win32.Delf.v
70	Backdoor.Win32.Rbot.rv
91	Backdoor.Win32.Hackdoor.q
91	Backdoor.Win32.Hupigon.aoa
91	Backdoor.Win32.PcClient.ox
228	Backdoor.Win32.SdBot.acs

これらが他のマルウェアサンプルと比較して、命令の絶対数が少ない理由としては以下のことが考えられる。

・パッキング（圧縮）されなかった：圧縮対象となるファイルと使用するパッカーの組み合わせにより圧縮されないことがあり得るため。

・BE-PUM がサポートしていない命令/Windows API 呼び出しが行われた。

マルウェアデータの中で命令数が同じものが複数見られるケースがあった。例えば以下に示すマルウェアは命令数がすべて 686 である。

表 5-7: 同じ命令数をもつマルウェア(ASPack)

以下の 20 マルウェアで同じ命令数	
命令数	マルウェア名
686	Backdoor.Win32.JustJoke.20
686	Backdoor.Win32.Lemerul.20.g
686	Backdoor.Win32.Likun.60
686	Backdoor.Win32.Polodor.a
686	Backdoor.Win32.Rbot.aia
686	Backdoor.Win32.Rbot.ajy
686	Backdoor.Win32.Rbot.anc
686	Backdoor.Win32.RCServ.e
686	Backdoor.Win32.SdBot.sb
686	Backdoor.Win32.Shadow.a
686	Backdoor.Win32.Shadow.b
686	Backdoor.Win32.Stark.a
686	Constructor.Win32.HkDown.b
686	Email-Flooder.Win32.Delf.k
686	Email-Flooder.Win32.Delf.l
686	Email-Worm.Win32.Mydoom.bc
686	HackTool.Win32.Delf.bi
686	HackTool.Win32.SqlTool.b
686	HackTool.Win32.WinArpAttacker
686	Trojan-Clicker.Win32.Chimoz.f

これらのうち、名前の接尾語の以外は同じ名前を持っているもの（例： Backdoor.Win32.Rbot.aia の aia の部分だけ異なる）、名称が異なるが種別が同じもの（上記例で Backdoor.Win32. まで同じ場合）については同種のものが名前違いで流通したウィルスであると推測される。

また、種別が異なるサンプル間においても、個々の実行されている命令を見ていくと、個々の命令のオペランドとして出現するメモリアドレスはそれぞれ異なるが、実行される命令の順序の視点で見ると 400 行目まではどれも同じ命令順序で実行されており、400 行目以降も終わりまでほぼ同じ命令列・順序で実行されている。そのため、これらはマルウェア名が異なるものの、亜種、もしくは名前違いの同種であることが推測される。

また、これらとほぼ同じ実行命令数を持つ以下の 3 つマルウェアサンプルにも同じ傾向が見られるため、これらもマルウェア名が異なるものの、亜種、もしくは名前違いの同種として流通したものであることが推測される。

Backdoor.Win32.Nethief.39 ： 命令数 687
Backdoor.Win32.Prorat.19.p ： 命令数 689
Backdoor.Win32.Sealer.a ： 命令数 689

・今回の観察で着目した call 命令のオペランドに同じメモリアドレスが複数回出現するケースが見られた。アドレスは観察したマルウェアデータで異なる場合も、同一の場合もある。また複数のアドレスが繰り返し呼び出し対象となっているケースもあった。call 命令の後に続く命令群についても同一の命令群のパターンが出現する場合としない場合があり、前者はプログラム中のループ処理に起因するものと推測される。

最後にパターン A-1 について述べる。パターン A-1 では下記 5 マルウェアが上述のヘッダー解析/BE-PUM 双方で同定できたものと同じ命令出現傾向が見られた。

表 5-8: パターン A-1 のマルウェアサンプル(1)

命令数	マルウェア名
725	00d527e1cb4447ca77e1bd7add266a5a095715d52317e82f1dcc13b3feb0c5b7
802	0a843675721cf47daa4ad522787ae5f66164acd4108ca019053c13800adf3d48
759	0c4981a755552caae82e7a61af62108a767a74ced8fde3b143cf5099b8be115b
752	0c4981a755552caae82e7a61af62108a767a74ced8fde3b143cf5099b8be115b
690	0c6806aaa5a1bdc24aee2cbdb843702af0dcc28db72c085079f9512c8e65913f

また、以下のマルウェアにはパターン A-2 とは別の出現傾向が観察された。命令数が等しいデータについては同じ命令の出現傾向が見られるため同一のウィルスであると推測される。

表 5-9: パターン A-1 のマルウェアサンプル(2)

以下の 10 マルウェアで同じ命令出現傾向	
命令数	マルウェア名
1582	00d2ff09bb2b686ffcc24f698b20886a635c37987e0326d9f362d5611ab47800
806	00d2ff09bb2b686ffcc24f698b20886a635c37987e0326d9f362d5611ab47800
1582	03764fc6d3bfe98e9a8dbc71c00ec9c970612e36432b9f97858d90b037c800c5
806	03764fc6d3bfe98e9a8dbc71c00ec9c970612e36432b9f97858d90b037c800c5
1582	045e108c28ae968d91045d86773e5f7aa7e0d9886bfef2e3464ce9cd8b649d29
806	045e108c28ae968d91045d86773e5f7aa7e0d9886bfef2e3464ce9cd8b649d29
1582	047f909dcc6f5e23e1b185cd631f8d934737304b9e771c4b6e6e1d2432e6d2c8
806	047f909dcc6f5e23e1b185cd631f8d934737304b9e771c4b6e6e1d2432e6d2c8
1582	087c99fb20cf1fd987f78300d80ba06bfe52e4597333414e9b86ffc93d1320af
806	087c99fb20cf1fd987f78300d80ba06bfe52e4597333414e9b86ffc93d1320af
以下の 3 マルウェアで同じ命令出現傾向	
命令数	マルウェア名
716	Backdoor.Win32.Rbot.fl
716	Backdoor.Win32.Rbot.jl
716	Backdoor.Win32.Rbot.jv

今回のサンプルでは命令列が極端に少ないがヘッダー解析/BE-PUM どちらの結果でもパッカーを同定しているものがあるが、これについてはデータ作成時にプログラムによる自動判別を行っているため、例外とみるべきデータに対しても同定が行われている可能性がある。そのため以降の OEP 同定ではパターン A-1 のサンプルと、途中で分析が終了してしまったと推測されるパターン A-2 の合計命令数 700 以下のデータは場外した。

以上から ASPack では、全 194 サンプルの傾向分析サブデータを観察することで 153 サンプル（77%）において、各サンプルのプログラム開始時から解凍用コードによる共通の命令出現傾向を観察し、パッカーの同定ができた。

5.2.OEP の同定

サンプルデータの観察結果から、解凍用コードによる仮想メモリ確保フェーズにおける GetModuleHandleA@kernel32.dll ， GetProcAddress@kernel32.dll, および VirtualAlloc@kernel32.dll の呼び出し以降、Windows API 呼び出しが発生していないものが多く確認されている。一般にマルウェアが目的とする活動を行う上では Windows API の何らかの呼び出しが必要と考えられることから、これらサンプルデータについては、BE-PUM による制御フローグラフの作成が何らかの理由でできていない可能性がある。

以上の仮説を踏まえ、傾向分析サブデータにおいて、後続の命令で Windows API 呼び出しが発生している部分抜き出したものを一部のマルウェアサンプルについて抜粋したものが下図である。

共通実行フェーズの終盤にてロジックの連続呼び出し（ メモリアドレスのオペランドが等しい Call 命令の連続実行）の後に ret $0x10<UINT16> （A）が呼び出され、その後 call -> ret を経て VirtualAlloc@kernel32.dll （B）の呼び出しが行われている。そのため（A）と（B）の間に Original Entry Point があると考えられる。

図 5-3: OEP 着眼点(ASPack)

そこでフルセットデータを用いていくつかのサンプルデータを抽出し、(A)から（B）にかけての実行された命令を観察していくと、ret の前まではどのサンプルにおいても同じ命令が実行されており、その以降はサンプル毎に実行される命令にばらつきが見られるようになった。

以下に示す図はサンプルデータのうち

Backdoor.Win32.SdBot.bs（実行命令数:2639）

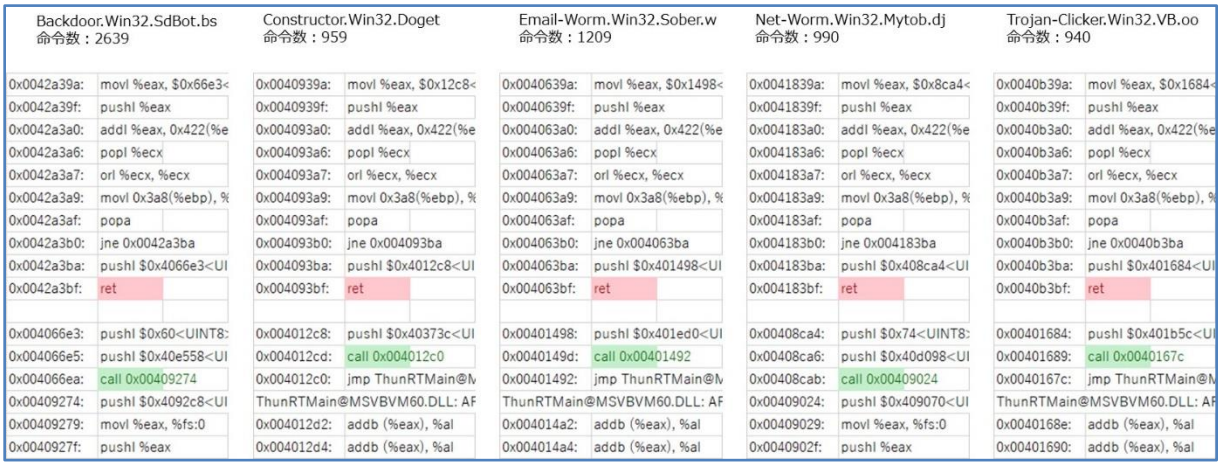
Constructor.Win32.Doget(959)

Email-Worm.Win32.Sober.w(1209)

Net-Worm.Win32.Mytob.dj(990)

Trojan-Clicker.Win32.VB.oo(940)

のそれぞれのフルデータにおける ret 前後の実行命令を抜粋したものである。



Backdoor.Win32.SdBot.bs 命令数：2639		Constructor.Win32.Doget 命令数：959		Email-Worm.Win32.Sober.w 命令数：1209		Net-Worm.Win32.Mytob.dj 命令数：990		Trojan-Clicker.Win32.VB.oo 命令数：940	
0x0042a39a:	movl %eax, $0x66e3<	0x0040939a:	movl %eax, $0x12c8<	0x0040639a:	movl %eax, $0x1498<	0x0041839a:	movl %eax, $0x8ca4<	0x0040b39a:	movl %eax, $0x1684<
0x0042a39f:	pushl %eax	0x0040939f:	pushl %eax	0x0040639f:	pushl %eax	0x0041839f:	pushl %eax	0x0040b39f:	pushl %eax
0x0042a3a0:	addl %eax, 0x422(%e	0x004093a0:	addl %eax, 0x422(%e	0x004063a0:	addl %eax, 0x422(%e	0x004183a0:	addl %eax, 0x422(%e	0x0040b3a0:	addl %eax, 0x422(%e
0x0042a3a6:	popl %ecx	0x004093a6:	popl %ecx	0x004063a6:	popl %ecx	0x004183a6:	popl %ecx	0x0040b3a6:	popl %ecx
0x0042a3a7:	orl %ecx, %ecx	0x004093a7:	orl %ecx, %ecx	0x004063a7:	orl %ecx, %ecx	0x004183a7:	orl %ecx, %ecx	0x0040b3a7:	orl %ecx, %ecx
0x0042a3a9:	movl 0x3a8(%ebp), %	0x004093a9:	movl 0x3a8(%ebp), %	0x004063a9:	movl 0x3a8(%ebp), %	0x004183a9:	movl 0x3a8(%ebp), %	0x0040b3a9:	movl 0x3a8(%ebp), %
0x0042a3af:	popa	0x004093af:	popa	0x004063af:	popa	0x004183af:	popa	0x0040b3af:	popa
0x0042a3b0:	jne 0x0042a3ba	0x004093b0:	jne 0x004093ba	0x004063b0:	jne 0x004063ba	0x004183b0:	jne 0x004183ba	0x0040b3b0:	jne 0x0040b3ba
0x0042a3ba:	pushl $0x4066e3<UI	0x004093ba:	pushl $0x4012c8<UI	0x004063ba:	pushl $0x401498<UI	0x004183ba:	pushl $0x408ca4<UI	0x0040b3ba:	pushl $0x401684<UI
0x0042a3bf:	ret	0x004093bf:	ret	0x004063bf:	ret	0x004183bf:	ret	0x0040b3bf:	ret
0x004066e3:	pushl $0x60<UINT8:	0x004012c8:	pushl $0x40373c<UI	0x00401498:	pushl $0x401ed0<UI	0x00408ca4:	pushl $0x74<UINT8:	0x00401684:	pushl $0x401b5c<UI
0x004066e5:	pushl $0x40e558<UI	0x004012cd:	call 0x004012c0	0x0040149d:	call 0x00401492	0x00408ca6:	pushl $0x40d098<UI	0x00401689:	call 0x0040167c
0x004066ea:	call 0x00409274	0x004012c0:	jmp ThunRTMain@M	0x00401492:	jmp ThunRTMain@M	0x00408cab:	call 0x00409024	0x0040167c:	jmp ThunRTMain@M
0x00409274:	pushl $0x4092c8<UI	ThunRTMain@MSVBVM60.DLL: AF		ThunRTMain@MSVBVM60.DLL: AF		0x00409024:	pushl $0x409070<UI	ThunRTMain@MSVBVM60.DLL: AF	
0x00409279:	movl %eax, %fs:0	0x004012d2:	addb (%eax), %al	0x004014a2:	addb (%eax), %al	0x00409029:	movl %eax, %fs:0	0x0040168e:	addb (%eax), %al
0x0040927f:	pushl %eax	0x004012d4:	addb (%eax), %al	0x004014a4:	addb (%eax), %al	0x0040902f:	pushl %eax	0x00401690:	addb (%eax), %al

図 5-4: フルセットデータにおける OEP 近傍（ASPack）

以上から ASPack の OEP はこの ret の直下のアドレスであると推測した。

なお、ret の直前に実行される命令を 5 つ抽出すると、

orl -> movl -> popa -> jne -> pushl -> ret

となる（オペランドの記述は省略）

このように今回の手法によって、パッカーの同定ができた ASPack のサンプルについて、傾向分析サブデータによるパッカーの解凍用コード以降の命令列を詳細に観察することで OEP の同定を行うことができた。OEP は

orl -> movl -> popa -> jne -> pushl -> ret

の次に実行される命令のアドレスである。

6. PECompact のパッカーと OEP の同定

6.1.分析結果の観察

今回観察した PECompact サンプルデータの諸元を以下に示す。

表 6-1： マルウェアサンプル（PECompact）

諸元	値	備考
サンプル数	494	
ヘッダー解析による同定	494	
BE-PUM による同定	494	
最小命令数	189	マルウェア名：Email-Worm.Win32.Colevo.b 他 2
最大命令数	2206	マルウェア名：Trojan-Downloader.Win32.Delf.es

ASPack のマルウェアサンプルにはヘッダー解析によるパッカー同定ができなかったものがあったが、PECompact のサンプルデータはすべてヘッダー解析/BE-PUM の両方で同定ができた。

また、ASPack と同様にマルウェアデータの中で命令数が同じものが複数見られるケースがあった。名前の接尾語の以外は同じ名前を持っている場合（例： P2P-Worm.Win32.SpyBot.dk の dk の部分だけ異なる）、名称が異なるが種別が同じ場合（上記例で P2P-Worm.Win32. まで同じ場合）が観察できたため、同種のものが名前違いで流通したウィルスであると推測される。

以下に示す図はフルセットデータから全サンプルの最初から 245 番目までの命令を抜き出したものである。

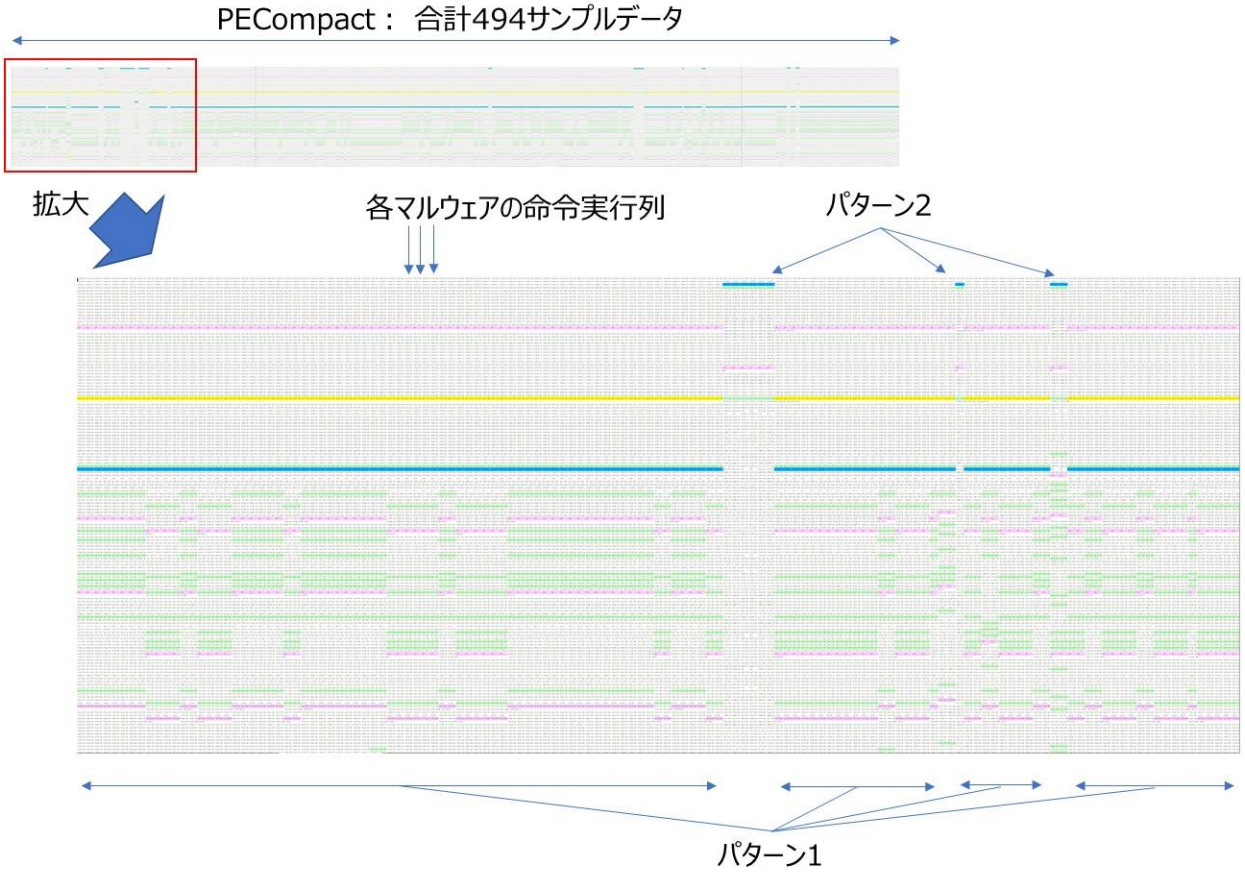


図 6-1: フルセットデータ（PECompact）

また、PECompact の傾向分析サブデータを以下に示す。なお、サンプル数が膨大なため以後の観察結果で出てくるパターン C-1 および C-2 の部分を抜粋し、かつ各サンプルデータにおいて最初から 150 個分の抽出した命令に図示の範囲を絞る。

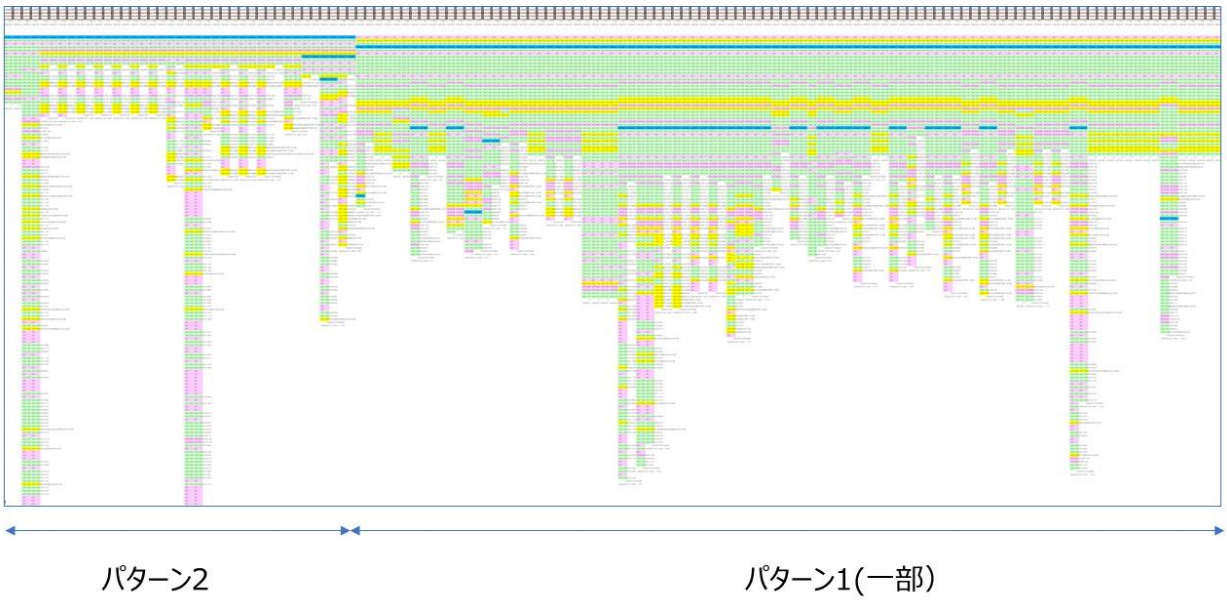


図 6-2: 傾向分析サブデータ（PECompact）

今回の観察で注目している 4 種類の命令の出現傾向については、どのサンプルデータにおいても以下に示す 2 パターンのどちらかに属している。

（パターン C-1）プログラム開始時に

ret -> call VirtualAlloc@kernel32.dll -> call -> pusha

の順で命令の実行結果が表れる。なおフルセットデータを見ると ret は 17 番目に実行される命令である。

表 6-2: マルウェアサンプル（PECompact）：パターン C-1 のデータ

諸元	値	備考
サンプル数	454	
最小命令数	190	マルウェア名：Trojan-Spy.Win32.Banker.bht
最大命令数	1791	マルウェア名：Trojan-Downloader.Win32.Delf.la

（パターン C-2）プログラム開始時に

pusha -> call -> ret -> call の順で命令が実行される。フルセットデータを見ると、pusha は 3 番目に実行されている。

表 6-3: マルウェアサンプル（PECompact）：パターン 2 のデータ

諸元	値	備考
サンプル数	40	
最小命令数	189	マルウェア名：Email-Worm.Win32.Colevo.b 他 2
最大命令数	2206	マルウェア名：Trojan-Downloader.Win32.Delf.es

命令数の大きさと所属パターンに関連性はなく、また（サンプルデータの名前から判断した際の）マルウェアの種別との関連性も見当たらなかった。

パターン C-1 に属するマルウェアにおいては、以下の表に挙げるようにマルウェアの種別と命令数に関係なく解凍用コードと考えられる最初の命令列の出現傾向がほぼ同じだった。

表 6-4: 同じ命令数をもつマルウェア

パターン C-1 に属するマルウェア	
マルウェア種別	サンプルデータ数
Trojan-Spy.Win32	53
Trojan-PSW.Win32	11
Trojan-Proxy.Win32	17
Trojan-Dropper.Win32	30
Trojan-Downloader.Win32	278
Trojan-Clicker.Win32	8
P2P-Worm.Win32	5
Net-Worm.Win32	4
IRC-Worm.Win32	2
Email-Worm.Win32	19
Backdoor.Win32	25

また、call 命令のオペランドに同じメモリアドレスが複数回出現した。アドレスは観察したマルウェアデータで異なる場合も、同一の場合もある。また複数のアドレスが繰り返し呼び出し対象となっているケースもあった。同一のアドレスを call した後に続く命令の種類に規則性は見つからなかった。

一方でパターン 2 に属するマルウェアにおいては上述のような同じアドレスの Call 命令が複数回出現したサンプルデータは以下に挙げる 5 つあった。そのうち命令数 189 の 3 データについてはサンプルデータ名からそれぞれ同種のものであることが推測され、一方で残りの 2 サンプルについてはサンプルデータの名前から同じマルウェアの本体とそのテスト用アプリケーションであった。

表 6-5: パターン 2 で同じアドレスへの Call 命令を行っているマルウェア

以下の 20 マルウェアで同じ命令数	
命令数	マルウェア名
189	Email-Worm.Win32.Colevo.b
189	Email-Worm.Win32.Colevo.d
189	Email-Worm.Win32.Colevo.e
1663	Trojan-Dropper.Win32.Delf.ao
1611	Trojan-Dropper.Win32.Delf.ao（test）

なお、この 5 つについては他のサンプルデータと特性が異なるため、以後の OEP 同定はおこなわないこととした。

以上から PECompact では、すべてのサンプルデータにてプログラム開始時から解凍用コードによる共通の命令出現傾向を観察し、パッカーを同定することができた。

6.2.OEP の同定

ASPack と同様に傾向分析サブデータの観察結果を基に OEP の同定を行ってみる。

パターン C-1（パッキングされたファイルが ret を最初に呼び出す）の傾向分析サブデータの抜粋を以下に示す。

パターン C-1 では、GetProcAddress@kernel32.dll の呼び出しの後、ret $0x4<UINT16> （A）が実行され、その後 VirtualAlloc@kernel32.dll （B）の呼び出しが行われている。 ここで（A）と（B）の間に Original Entry Point があると考えられる。

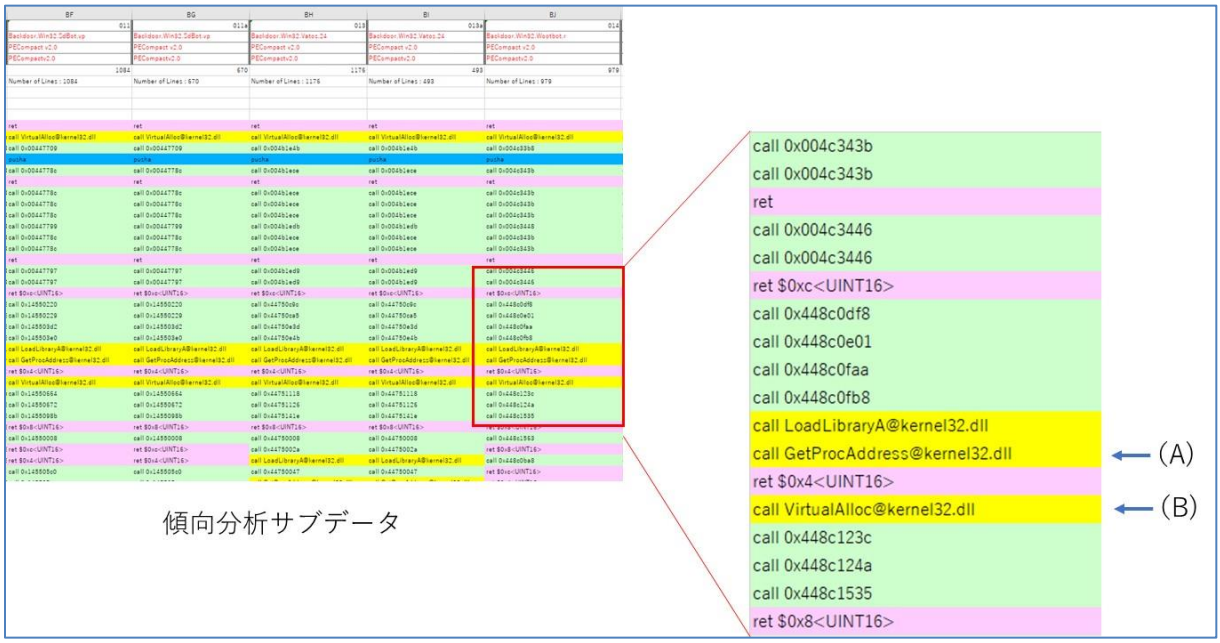


図 6-3: OEP 着眼点(PECompact:パターン C-1)

次にフルセットデータからパターン C-1 のいくつかのサンプルデータを抽出する。

ここでは以下の 5 つのサンプルを例示する。

Backdoor.Win32.Delf.air（実行命令数 1148）

Email-Worm.Win32.Anker.qline（1167）

Net-Worm.Win32.Mofeir.n（785）

P2P-Worm.Win32.SpyBot.dw（683）

Trojan-Downloader.Win32.Banload.af（1166）

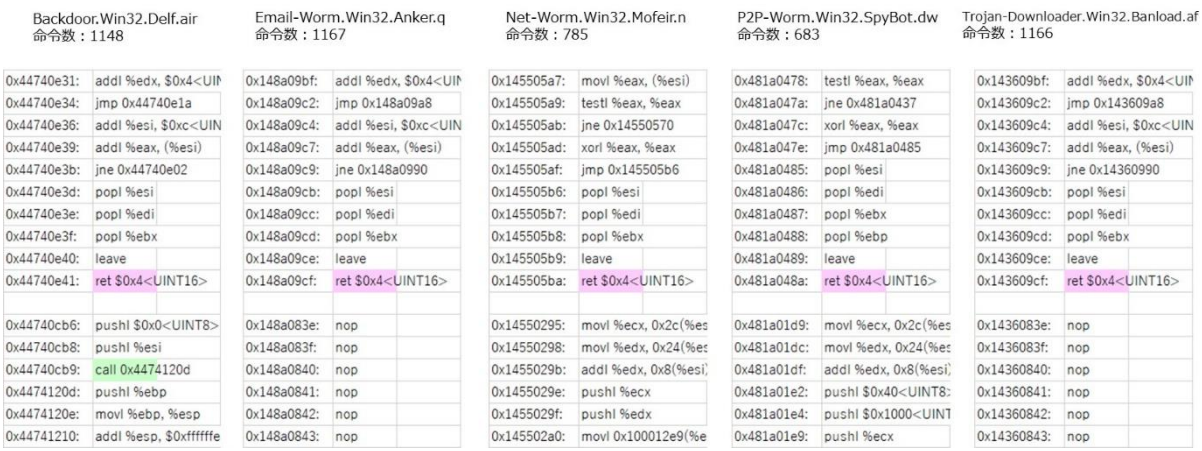


図 6-4: フルセットデータにおける OEP 近傍（PECompact:パターン C-1）

ret 実行以降の命令はマルウェアサンプルによりそれぞれ異なっていることから、パターン C-1 の PECompact はこの ret 命令の後のアドレスが OEP であると推測した。

なお、ret の直前に実行される命令は 5 つ抽出すると以下のようになっている（オペランドの記述は省略）

(※) -> popl -> popl -> popl -> leave -> ret

なお※の部分はマルウェアサンプル毎に異なっているが、どの場合も ret 直前の命令は条件つきジャンプ（je）または jmp 命令の後に popl が何回か続き、その後 leave -> ret という順番で実行されていた。

このように今回の手法によって、PECompact の傾向分析サブデータによるパッカーの解凍用コード以降の命令列を詳細に観察することで OEP の同定を行うことができた。OEP は

popl -> popl -> popl -> leave -> ret

の次に出現する命令のアドレスである。

7. UPX のパッカーと OEP の同定

7.1.分析結果の観察

今回観察したサンプルデータは全部で 2535 であり、マルウェア種別で見た内訳は以下のとおりである。

表 7-1： マルウェアサンプル（UPX3.0）マルウェア種別

マルウェア種別	サンプルデータ数
Backdoor.IRC	1
Backdoor.Win32	437
Constructor.Win32	30
DoS.Win32	7
Email-Flooder.Win32	9
Email-Worm.Win32	230
Exploit.Win32	20
Flooder.Win32	12
HackTool.Win32	28
IM-Flooder.Win32	3
IM-Worm.Win32	12
IRC-Worm.Win32	13
Net-Worm.Win32	60
not-virusBadJoke.Win32	3
not-virusHoax.Win32	6
Nuker.Win32	2
P2P-Worm.Win32	115
Packed.Win32	1
Rootkit.Win32	1
SMS-Flooder.Win32	2
Sniffer.Win32	1
SpamTool.Win32	2
Trojan-AOL.Win32	1
Trojan-Clicker.Win32	87
Trojan-Downloader.Win32	895
Trojan-Dropper.Win32	254

Trojan-Notifier.Win32	6
Trojan-Proxy.Win32	75
Trojan-PSW.Win32	142
Trojan-Spy.Win32	62

これらのサンプルデータのうち BE-PUM の分析が最後まで完了できていないと推測されるデータを除外しフルセットデータが 300 命令以上ある命令を抽出すると 42 データが残った。なお、UPX のサンプルデータは今回のサンプルデータ全体の中で過半数の割合を占めているが、全体として ASPack や PECompact よりも各サンプルデータの逆アセンブル結果が小さいものが多かった。

以下に示す図は上述した 42 サンプルのフルセットデータの最初から 135 番目までの命令を抜き出したものである。

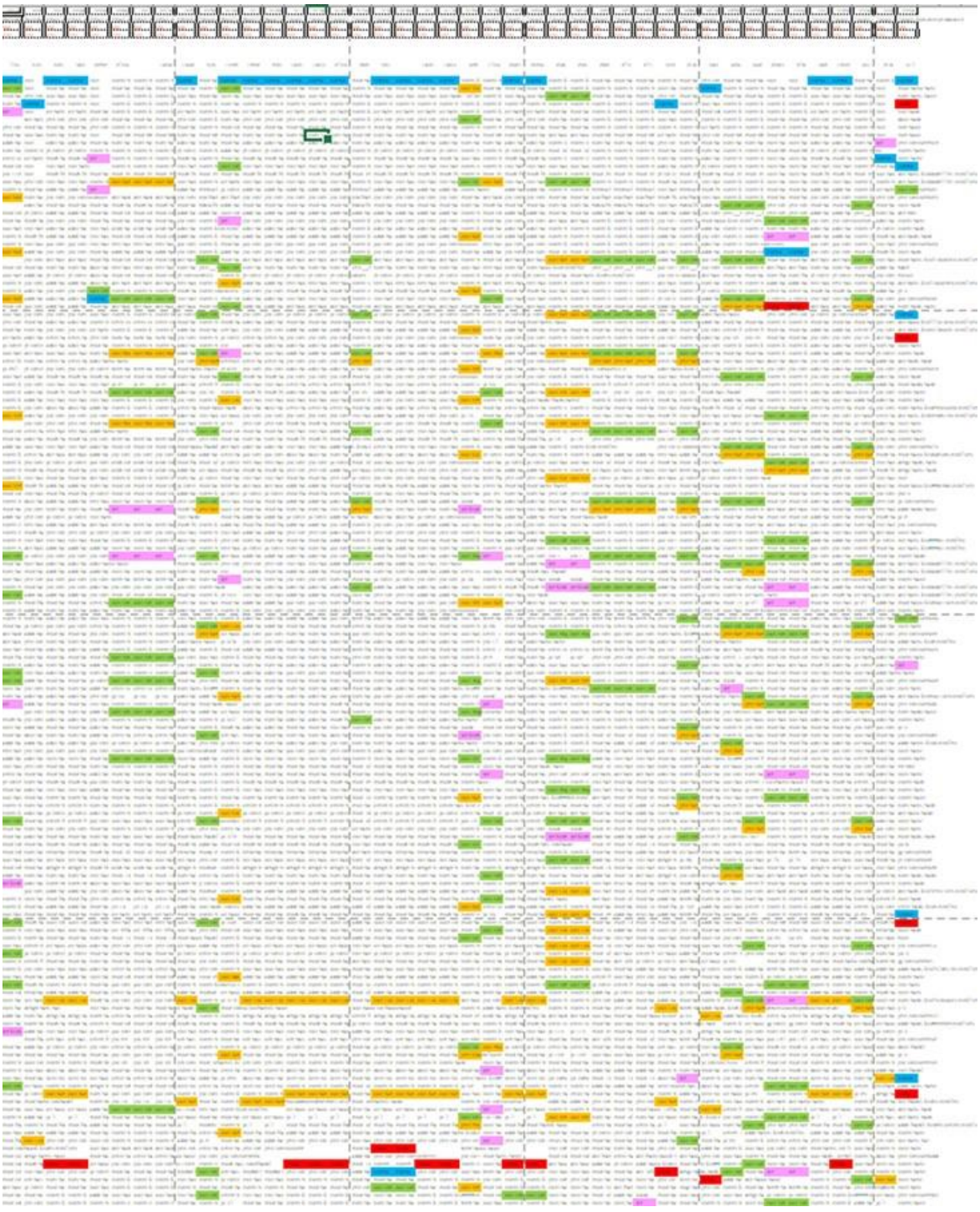


図 7-1: フルセットデータ（UPX3.0）

これらのうち、さらに VirtualAlloc@kernel32.dll や HeapAlloc@kernel32.dll などのメモリ呼び出し行う命令列をもつ 25 サンプルを絞り込み作成した傾向分析サブデータを以下に示す（最初の 88 命令を抽出している）。

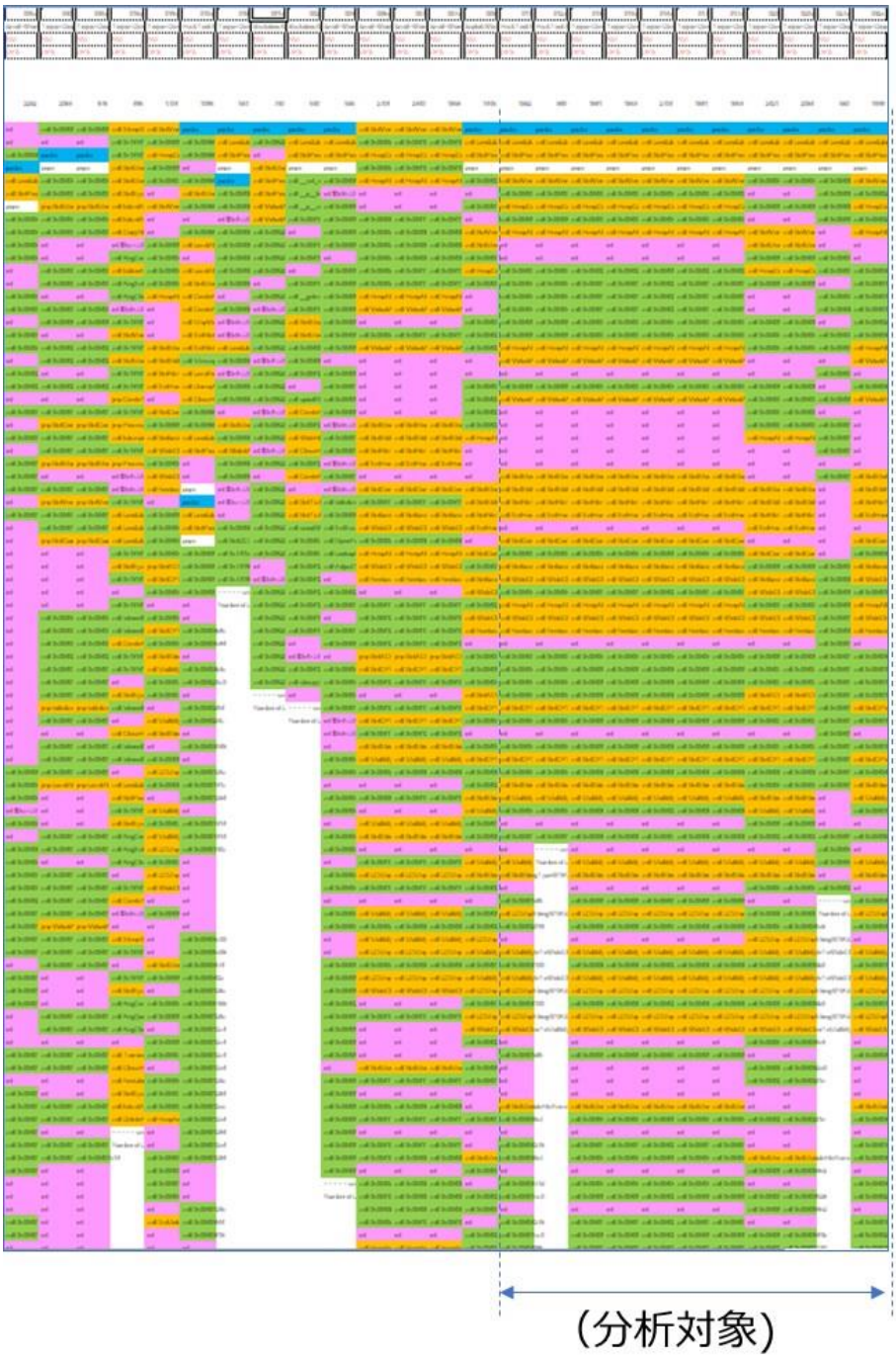


図 7-2: 傾向分析サブデータ（UPX3.0）

図 7-2 右側のサンプルデータ群に解凍用コードとみられる共通項が見られるため、これらに対する OEP の同定を次節にて行う。なお、分析対象に含まれる 25 のマルウェアサンプル種別は HackTool.Win32、Trojan-Downloader、Trojan-Dropper の 3 つである。

傾向分析サブデータにおいて、下記 2 点を観察することができる。
・popa 以降も同じ命令列が続いている

図 7-2 にて分析対象としているサンプルでは見やすくするために popa 命令を白色で抜き出しているが、パターンを見るとわかるように popa（サブデータの比較的上部に出現）より下の命令にも共通したパターンのようなものが見られる。
・popa が 1 度も出現しないサンプルが多数存在する
観察開始時には UPX の OEP 近傍に popa があるという情報を得ていたが、実際に個々のサンプルデータを観察すると popa が出現しないうちに BE-PUM の解析が終了しているサンプルがあった。全般に逆アセンブル結果ファイルの大きさが小さいことから、UPX による解凍用コードで行われている命令に BE-PUM 未サポートのものがあり、それが分析完了まで到達できたサンプルが母数に対して少ない結果となったのではと推測している。

以上をまとめると、UPX のサンプルでは取得できた各サンプルデータのフルセットデータの大きさおよび傾向分析サブデータの観察結果から、少なくとも解凍用コードの実行完了まで BE-PUM が解析を行ったデータは少なかったものの、一部のデータにおいて共通した命令列の出現を観察することができた。全般において UPX の OEP 同定において重要な命令である popa を検出できていないデータが多く存在し、解凍用コードの実行完了まで BE-PUM が解析を実行できていないことが示唆された。次節においてそれらのデータを用いた OEP の同定を試みる。

7.2.OEP の同定

以下に示すのは図 7-2 で示した分析対象の一部抜粋である。これを見ると popa 命令（白色で表現）以降でサンプル毎に出現する命令が異なっていることがわかる。

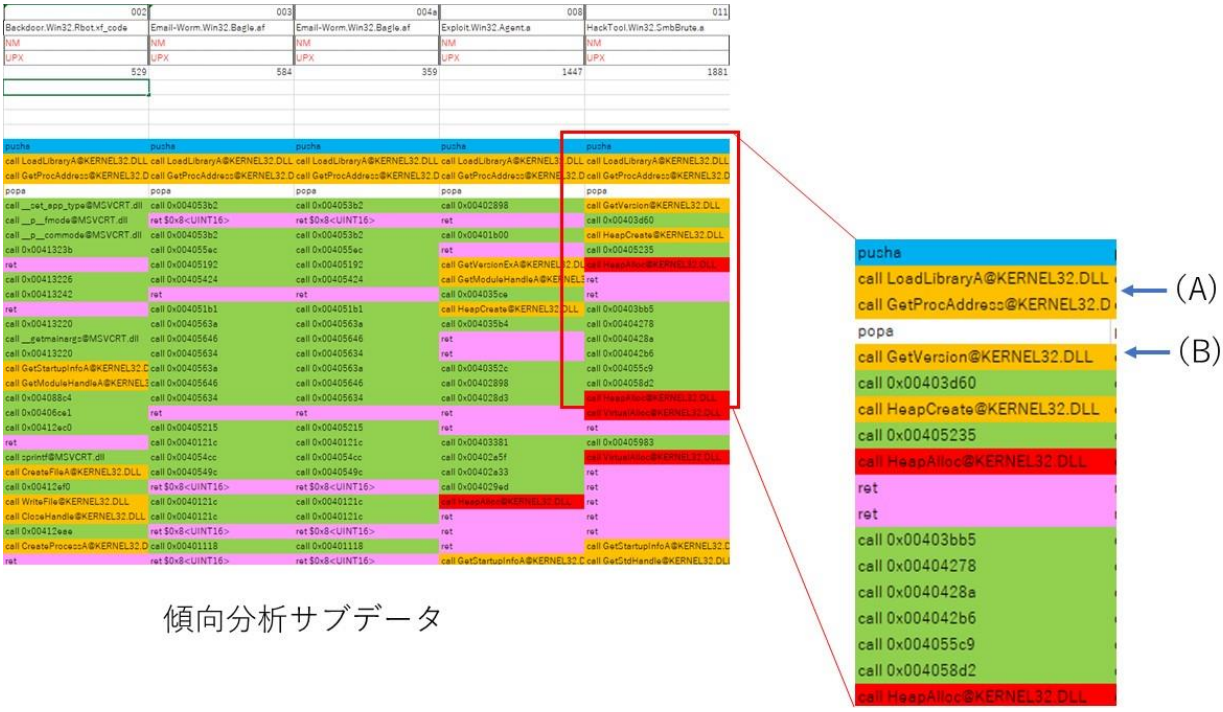


図 7-2: OEP 着眼点(UPX3.0)

そこで popa の前後（A）と（B）の間のフルセットデータを見てみることにする。以下の図は 4 つサンプルデータについての実行命令列を示したものである。

図 7-3: フルセットデータにおける OEP 近傍（UPX）

一番左のサンプルである HackTool.Win32.SmbBrute.a では popa の直前に 5 つ他の命令が実行されているが、どのサンプルにおいても popa の直前に je -> movl -> addl -> jmp -> GetProcAddress@Kernel32.dll -> LoadLibraryA@Kernel32.dll の呼び出しが行われている。また既存研究にある通り popa の後に jmp 命令が実行されており、この命令のオペランドで記されているアドレスが OEP ということになる。

以上から、UPX の傾向分析サブデータによるパッカーの解凍用コード以降の命令列を詳細に観察することで、UPX の解凍用コードが popa で終わるという既存研究での報告を今回のサンプルでも一部のデータにおいて確認することができた。

8. Upack のパッカー同定と考察

今回観察した Upack サンプルデータの諸元を以下に示す。

表 8-1: マルウェアサンプル（Upack）

諸元	値	備考
サンプル数	382	
ヘッダー解析による同定	378	
BE-PUM による同定	382	
最小命令数	10	マルウェア名：別表にて記載
最大命令数	487	マルウェア名：Email-Worm.Win32.Hybris.h

表 8-2: マルウェアサンプル（Upack）マルウェア種別

マルウェア種別	サンプルデータ数
Backdoor.Win32	14
Email-Worm.Win32	28
IM-Worm.Win32	26
IRC-Worm.Win32	2
Net-Worm.Win32	4
Trojan-Downloader.Win32	128
Trojan-Dropper.Win32	38
Trojan-Proxy.Win32	14
Trojan-PSW.Win32	64
Trojan-Spy.Win32	64

以下に示す図はフルセットデータから全サンプルの最初から 245 番目までの命令を抜き出したものである。

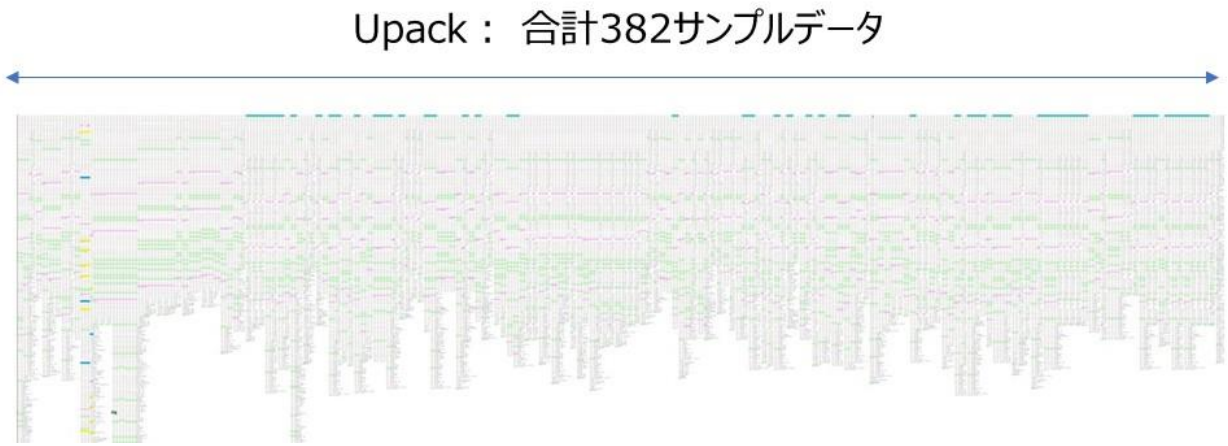


図 8-1: フルセットデータ（Upack）

ヘッダー解析によりパッカー同定ができず、BE-PUM で同定結果が出たサンプルについて、Upack では Email-Worm.Win32 に属する 4 サンプルが該当した。

次に Upack の傾向分析サブデータの抜粋を以下に示す（それぞれ最初から 150 個分の抽出した命令を図示している）

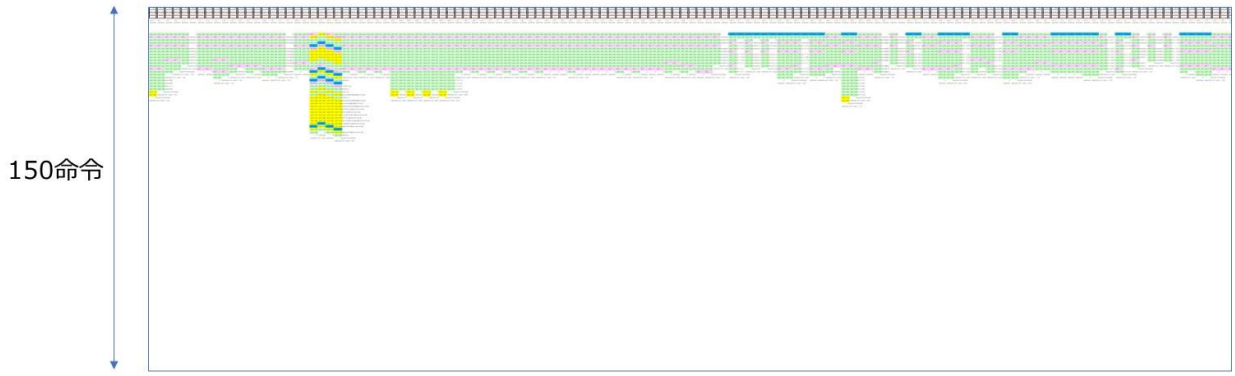


図 傾向分析サブデータ(Upack)

フルセットデータ、および傾向分析サブデータの全般的な傾向を見ると、ASPack, PECompact と比較して以下の点が特徴として挙げられる。

・Upack のマルウェアサンプルは命令数全体的に少ない。総命令数 30 以下のサンプルが全 384 データのうち 71 あり、200 を超えるサンプルは 30 しかない。

・プログラムデータ格納のために使用する VirtualAlloc@kernel32.dll の呼び出しを行うサンプルが 4 つしかなく、かつそれらも 2 回以上呼び出しを行っているものがない。

・命令の出現順序について、異なるマルウェア種別にまたがって形で共通で出現するものがない。

解凍コードの実行を示唆すると考えている VirtualAlloc@kernel32.dll が出現せず、かつ異なるマルウェアにまたがって共通の命令実行パターンが見られないことから、BE-PUM がサポートしてない命令もしくは Windows API の呼び出しなどの理由により Upack の解析が正しく行われていない可能性がある。

以上から Upack サンプルへ今回のアプローチを適用してパッカーおよび OEP の同定は不可能と判断した。

9.本課題研究のまとめと今後の課題

本課題研究では VXHeaven データベース上のマルウェアを BE-PUM を用いて逆アセンブルした結果を直接観察し、パッカーの解凍用コードを示す命令列パターンを検出することでパッカーの同定とマルウェアの Original Entry Point（OEP)の検出を試みた。ASPack と PECompact をパッカーとして用いられているサンプルデータについてパターンの検出を行うとともに OEP と考えられる地点の特定を行うことができた。

UPX については観察による OEP 同定結果と既存研究との比較を行い、解凍用コードが popa 命令で終了するという結果の確認を行った。

一方で BE-PUM による逆アセンブルが完了していない、もしくはできていないと考えられるサンプルも多く、Upack においては命令パターンの検出と OEP 同定の考察まで至らなかった。
また、UPX の解析結果についても popa 以降もパターンが見られる部分があり、それがマルウェア側に共通して出てきているものであるのか、もしくは解凍用コードが popa 出現以降も続いていることを示唆しているものなのかは不明であり、今後の課題となった。課題解決のためには以下の方策が有効ではないかと推測した。

- BE-PUM が対応する命令/Windows API を拡張することで実行可能対象データのカバー率を向上させる。

- 今回着目した命令/Windows API コールとは異なるセットを用いたパターン生成

後者について、今回採用した観察手法では観察する分析用サブデータの作成を一定のルールに基づいて実施しているため、ロボットによる自動実行や機械学習の利用を併用することにより、効率的かつ多角的な観察が可能になると考えられる。

本課題研究で採用したアプローチを用いるとパッカーの解凍用コードの制御フローグラフを抽出できるため、一度パッカーの解析と同定が行えると、例えばサンドボックスの中でマルウェアを実行することで、解析データを基にパッカーとマルウェアのプログラムを分離することができるのではないかと考えている。

また、別の課題として今回推測した OEP の検証方法の確立がある。今回のサンプルデータは比較的古い種類のマルウェアで構成されており、そこで使用されているバージョンのパッカーは入手が難しい。そのため現在使用できるバージョンのパッカーで疑似マルウェアプログラムをパッキングしたアプリケーションを用意し、BE-PUM による逆アセンブルを行った結果と疑似マルウェアのみを逆アセンブルした結果を比較することが有効ではないかと考える。さらにこの結果を用いることで比較的最近出現したマルウェアのパッカーおよび OEP の同定を試行することも今後の展望として考えられる。

参考文献

1. Anti-virus technology whitepaper. Technical report, BitDefender, 2007.

2. M. Morgenstern, A. Marx. Runtime packer testing experiences. In *CARO*, pp.288-305, 2008. LNCS6174.

3. T. Ban, R. Isawa, S. Guo, D. Inoue, K. Nakao. Efficient malware packer identification using support vector machines with spectrum kernel. In *AsiaJCIS*, pp.69-76, 2013.

4. T. Ban, R. Isawa, S. Guo, D. Inoue, K. Nakao. Application of string kernel based support vector machine for malware packer identification. In *IJCNN*, 2013.

5. L. Sun, S. Versteeg, S. Boztas, T. Yann. Pattern recognition techniques for the classification of malware packers. In ACISP, Berlin, pp.370-390, 2010. LNCS 6168.

6. M. H. Nguyen, T. B. Nguyen, T. T. Quan, M. Ogawa. A hybrid approach for control flow graph construction from binary code. In *IEEE APSEC*, pp.159-164, 2013.

7. M. H. Nguyen, M. Ogawa, T. T. Quan. Obfuscation code localization based on CFG generation of malware. In *FPS*, pp.229-247, 2015. LNCS 9482.

8. K. A. Roundy, B. P. Miller. Binary-code obfuscations in prevalent packer tools. In *ACM Comput. Surv*, 46, pp.4:1-4:32, 2013.

9. V.S. Sathyanarayan, P. Kohli, B. Bruhadadeshwar. Signature Generation Detection of Malware Families. In *ACISP*, pp.336-349, 2008. LNCS 5107.

10. J. Kinder. *Static Analysis of x86 Executables.* PhD thesis, Technische Universitaet Darmstadt, 2010.

11. J. Kinder, F. Zuleger, H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, pp.214-228, 2009. LNCS 5403.

謝辞

本課題研究をまとめるにあたり、また社会人学生として休学期間を含め 4 年半の間、終始熱心なご指導をいただいた小川瑞史教授に感謝の意を表します。また機会を作っては石川キャンパスに滞在をしておりましたが、そこで小川・浅野研究室を始めとする情報科学科の皆様にはさまざまな刺激と示唆をいただきました。ありがとうございました。

最後に小川研究室所属の東京サテライト学生である鎌田さん、村田さん、田中さんにお礼を申し上げます。皆さんとはセミナーや自主勉強会の時などでしかお会いできず、わずかな会話しかできませんでしたが、一緒に頑張っている皆様の存在があったからこそ、ここまで来ることができました。本当にありがとうございました。