情報処理学会論文誌 Vol. 51 No. 9 1622–1632 (Sep. 2010)

推薦論文

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

岩村 誠[†1,†2] 伊藤光恭[†1] 村岡洋一[†2]

本論文では，機械語命令列の類似度算出手法および自動マルウェア分類システムを提案する．機械語命令列の類似度算出に関する提案手法では，新たなマルウェアが出現した際に，過去に収集されたマルウェアとの近さを算出するとともに，過去のマルウェアと共通の命令列および実際に変更のあった箇所を推定可能にする．一方，昨今の多くのマルウェアはパッカによりそのプログラムコードが隠蔽されている．こうした課題に対しこれまで我々は，マルウェアのアンパッキング手法および逆アセンブル手法を開発してきた．本論文では，これらの手法に機械語命令列の類似度算出に関する提案手法を組み合わせ，マルウェア分類システムを構築した．実験では本システムを利用し，3 グループのマルウェア検体を分類した．その結果，ハニーポットで収集した約 3,000 種類のマルウェアであっても，わずか数種類のマルウェアを解析することで，全体の 75%程度のマルウェアの機能を把握できることが分かった．さらに，類似度の高いマルウェアに関しては，それらの差分を推定でき，変更箇所にのみ着目した解析が可能なことも分かった．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，マルウェアに対する命名の難しさが明らかになった．

Automatic Malware Classification System Based on Similarity of Machine Code Instructions

Makoto Iwamura,[†1,†2] Mitsutaka Itoh[†1] and Yoichi Muraoka[†2]

We propose the method for calculating the similarity of machine code instructions and an automatic malware classification system based on the method. Our method enables malware analysts to calculate the distance of known malware to new one and estimate the part that are different. By the way, the machine code instructions of many recent malware are hidden by a packer. For solving the problem, we developed an unpacker and a disassembler. In this paper, we build the automatic malware classification system with a combination of the unpacker, the disassembler and the proposal method for the similarity of machine code instructions. In the experiment, we classified 3 groups of malware. The experiment results show that we can understand 75% of the whole malware functions by analyzing several different types of malware. For the similar pair of malware, the system could estimate the difference, i.e., we could analyze the malware focused on the different part. In the comparison with our system results and the names by anti-virus software, the problems of naming malware emerged, e.g., anti-virus software indicated different names about the malware that our system identified as the same malware.

1. はじめに

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いる[1)]．こうして，効率的なマルウェアの亜種の開発が行われている．加えて，マルウェアはアンチウィルスソフトによる検出から逃れるために，パッカと呼ばれる一種の圧縮ツールにより，同じ機能を持ちながらも外観が異なる形式で大量に生産されている[2)]．このような一連の開発サイクルにより，近年のマルウェア数は増加の一途をたどり[3)]，それらすべてに対策を打つことはおろか，対策の優先度を決めることさえ困難になっている．

本研究の目的はこうした背景をふまえ，マルウェアが備える脅威を解明する作業を削減することにある．本論文では，まず 2 つの機械語命令列の類似度算出手法を提案する．これは 2 つの機械語命令列の最長一致部分列を算出することで，それらの長さから類似度を算出する手法である．これにより新たなマルウェアが出現した際に，過去に収集されたマルウェアとの近さを算出できるとともに，過去のマルウェアと共通の命令列および実際に変更のあった箇所を推定することも可能になる．つまり変更があったと推定された箇所を，実際に解析する必要のある命令列と見なせば，マルウェアが備える脅威を解明するために必要となるリ

†1 NTT 情報流通プラットフォーム研究所
NTT Information Sharing Platform Laboratories
†2 早稲田大学
Waseda University
本論文の内容は 2009 年 10 月のマルウェア対策研究人材育成ワークショップ 2009 にて報告され，コンピュータセキュリティ研究会主査により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

バースエンジニアリングの作業を削減できることになる．

また，これまで我々はマルウェアの自動分類に向けて，他にも 2 つのマルウェア解析手法を提案してきた．1 つ目は動的生成コードを自動的に抽出するアンパッキング手法である．これにより，パッキングされたマルウェアからオリジナルコードを含むバイナリ列を取得することができる．もう 1 つは，命令とデータが混在するバイナリ列において，両者を識別する確率的逆アセンブル手法である．この逆アセンブル結果からマルウェアの機能を表す機械語命令列が得られる．

我々は，これらのアンパッキング手法と逆アセンブル手法に，機械語命令列の類似度算出に関する提案手法を組み合わせ，自動マルウェア分類システムを構築した．また実験として，3 グループのマルウェア検体を本システムにより分類した．その結果から，約 3,000 検体の SHA1 ハッシュ値が異なるマルウェアに関して，約 50%が Conficker，約 25%が W32.Rahack.H および W32.Rahack.W であることが分かった．つまりこのデータセットに関しては，わずか数種類のマルウェアを解析することで，全体の 75%程度のマルウェアの機能を把握できることになる．

また同データセットに関するアンチウィルスソフトのスキャン結果と本システムによる分類結果との比較では，W32.Rahack.H が Backdoor.Trojan として識別される等，ほぼ同一の機械語命令列であるにもかかわらず異なる検出名となる状況が確認された．他にも，PE 感染型マルウェアに感染しているマルウェアに関しては，どちらか一方の名前で識別されてしまう等，マルウェアの命名に関する課題も明らかになった．

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といった課題も明確になった．

本論文は次のように構成される．まず 2 章で，従来のマルウェアの分類技術とその課題について述べる．次に 3 章で，我々の提案する機械語命令列の類似度に基づくマルウェア分類手法について述べる．そして 4 章では，これまで我々が提案してきたアンパッキング手法および逆アセンブル手法について概説したうえで，これらの手法と 3 章の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説明する．5 章では，こうして構築した自動マルウェア分類システムを用い，マルウェア検体を分類，解析した結果とその考察について述べる．最後に 6 章でまとめる．

2. 従来研究とその課題

マルウェアの分類技術には，大きく分けて 2 つのアプローチが存在する．1 つは，マルウェアの挙動により分類する方法[4] である．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もと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．一方，ボットのような攻撃者からの指令な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に関して，その挙動を網羅的に抽出することは非常に難しい．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，マルウェアを分類する手法も存在する．これは挙動による分類とは異なり，マルウェアが潜在的に備える機能をふまえつつ分類できる．ここでは，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研究に焦点を当てる．

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[5] 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[6] 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[7]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いわれている．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．

たとえばベーシックブロックによる手法では，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 やコールツリーによる手法では，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，まった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，結局すべ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．

3. 提 案 手 法

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 の再構築等は必要としない．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．

	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 1　DP 行列
Fig. 1　DP matrix.

3.1 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 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，たとえば *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[8]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．

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

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．たとえば $S =$ "*dbca*"，$T =$ "*bdca*" としたときの DP 行列は**図 1** になる．

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どうし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，それらの組合せの数も膨大になっているため，高速に機械語命令列から LCS およびその長さを算出する手法が望まれる．

	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 2　ビット行列 M
Fig. 2　Bit vector M.

3.2　ビットベクトル化

Crochemore ら[9] 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 2**）を作成する．

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．

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

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 1 要素として扱うとメモリ領域が枯渇する恐れがある．

3.3　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 機械語命令は命令プレフィックス，オペレーションコード，ModR/M，SIB，ディスプレースメント，即値の 6 つの部位から構成される[10]．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．以上をふまえ，縮約命令は**図 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，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 マルウェアあ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 100 MB [⋆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．

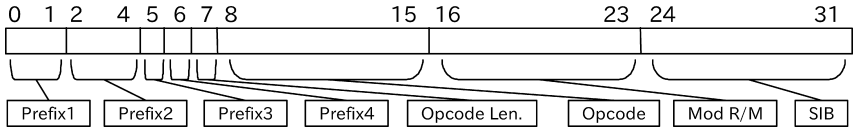

図 3　縮約命令の形式
Fig. 3　Reduced instruction format.

3.4　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 $|L|$ をもと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．

⋆1 比較対象の縮約命令数を 100,000 とした場合.

$$S = \frac{|L|}{|A| + |B| - |L|}$$
$$D = 1 - S$$

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 つのマルウェアがまったく同一であることを意味する．

4. 自動マルウェア分類システム

我々はこれまでにマルウェアの分類を全自動化するために，汎用アンパッキング手法[11]および確率的逆アセンブル手法[12]の研究開発を行ってきた．本章では，まずこれらの手法について関連研究とともに概説し，その後 3 章の提案手法を組み合わせた，自動マルウェア分類システムの構成について述べる．

4.1 アンパッキング手法

4.1.1 関 連 研 究

実行ファイルがどのパッカ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．たとえば，PEiD[15]は単純なパターンマッチングによりパッカを識別することができる．こうしてパッカ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キングが可能であるが，パッカを識別でき，なおかつ対応するアンパッカ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が利用された場合には適用できない．

一方，パッカごとのアルゴリズムに依存しない動的なアンパッキング手法も提案されている．ランタイムパッカ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，リロケーション等）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キング手法はこの事実を前提として開発されている．OllyBonE[16]や Universal PE Unpacker[17]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 Saffron[18]や Renovo[19]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．

ただ，Renovo は TEMU 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，そのエミュレーション精度の問題からアンチデバッグ機構により解析できない状況も報告されている．またエミュレーションの性質上，解析速度の低下も免れない．

これに対して，Saffron は OS のページフォ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）内のスーパバイザビットを設定することで，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 における TLB と PTE の同期処理は，通常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あえて省略することで PTE にはスーパバイザビットを設定しつつも，TLB ではスーパバイザビットが設定されていない状態を作り出し，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．ただしこれにも大きな問題が存在する．一般的に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，最も著名な仮想マシンである VMware[20]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もある[21]．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 の実装ではつねに PTE に設定したスーパバイザビットが有効になってしまうため，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，マルウェア解析に非常に有効である仮想マシン上での解析が実質不可能となってしまう．

4.1.2 TLB に依存しないアンパッキング手法

我々はこうした課題を解決するために，TLB の実装に依存せずにメモリへの書き込みおよび実行を監視する手法を提案した[11]．これは Saffron と同様にページフォルトハンドラをフックすることで，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．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．一方，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 を 1 に設定することで実現す

る．これには PAE（Physical Address Extensions）が有効になっており，かつ MSR レジスタ内の 11 ビット目を設定する必要があることに注意する．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，いず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．

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，Saffron では TLB に存在しないページにアクセスするたびにページフォルトが発生していたのに対し，我々の手法では，監視すべきアクセスのときのみにページフォルトが生じるようになったため，速度面のパフォーマンス向上も見込むことができる．

4.2　逆アセンブル手法

4.2.1　関 連 研 究

従来の逆アセンブル手法は主に以下の 2 つの手法，もしくはこれらを組み合わせた手法で構成される．

(1) Linear sweep　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，次バイト値から逆アセンブルを行い，この作業を終端まで繰り返す．

(2) Recursive traversal　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，命令として解釈できない部分までを逆アセンブルする．また，逆アセンブルの過程で分岐命令が現れたときは，その分岐先を新たな命令列の先頭とし，再帰的に逆アセンブルする．

Linear sweep 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．また可変命令長の場合は，1 度命令の先頭を見誤ると連鎖的に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．一方 Recursive traversal は，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していくが，その分岐先が動的に決まる場合は，分岐先が命令列として解釈されなくなる．こうした分岐先が動的に決まるケースは，C++における仮想関数テーブルをはじめとして，その他のコンパイラによる出力でも散見される．

4.2.2　確率的逆アセンブル手法

我々はこうした従来技術の欠点をふまえ，分岐命令等に頼らず確率的に命令かデータかを判断する逆アセンブル手法を提案した[12]．この手法では隠れマルコフモデルを用いることで，コンパイラ出力コードの特徴をモデル化する．簡単なモデルとしては，機械語命令を 1 命令出力する状態とデータ 1 バイトを出力する状態の 2 状態の Ergodic HMM が考えられる．また学習データとして，バイト列と命令かデータかを判断できる逆アセンブル結果を事前に用意しておく．この学習データは，マルウェアの開発によく利用される開発環境（Visual C++や Delphi，Borland C++等）により，事前に様々なソフトウェアをコンパイルし作成してもよいし，また実際のマルウェアをいくつか解析し作成してもよい．そして各状態における出力確率および状態遷移確率を，この学習データにより学習させてモデルパラメータを決定し，Viterbi アルゴリズムを用いることで，未知のバイト列に対する最ももっともらしい逆アセンブル結果を算出することが可能になる．

4.3　システム構成

ここでは，自動マルウェア分類システムについて述べる．本システムは前述の 3 つの手法に対応する，アンパッキングモジュール，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．まず本システムに対してマルウェアが与えられると，アンパッキング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．アンパッキング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，マルウェアの全組合せに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．

アンパッキングモジュールは，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 4.1 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，アンパッキングモジュールで得られた複数のメモリダンプをすべ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合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. 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あげながら，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．1 番目のデータセット A は CCC DATAset 2009 [13] のマルウェア検体 10 種である．2 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．収集には我々が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．3 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，我々のハニーポットを用いて収集した 3,232 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．

5.1 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 4** 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．縦軸は非類似度となっており，下方でつながっていれば類似した検体（クラスタ）どうし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 等）を付けて，各オリジナルコードを区別している．

図 4　データセット A のデンドログラム
Fig. 4　Dendrogram for dataset A.

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ている．図 4 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．また 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 と LeaveCriticalSection API 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 の検体とまった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．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ことが可能になる．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．

5.2　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 1 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．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 つのクラスタに正確に分割されることも確認できた．さらに Conficker.B と Conficker.B++の LCS を求め当該検体どうしの差分を解析したところ，Conficker.B++には新たに CreateNamedPipeA 等の名前付きパイプに関連する処理等が追加されていることを確認できた．SRI International[14] 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．

5.3　データセット C の分類結果

データセット C は 3,232 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** に示す．具体的には図 4 における縦軸が円の半径，横軸が角度（0〜2π）を表しており，図 4 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%（1,593 検体）を占めるクラスタ（図 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 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（約 50%）と Rahack（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 3 種類であることが分かった．

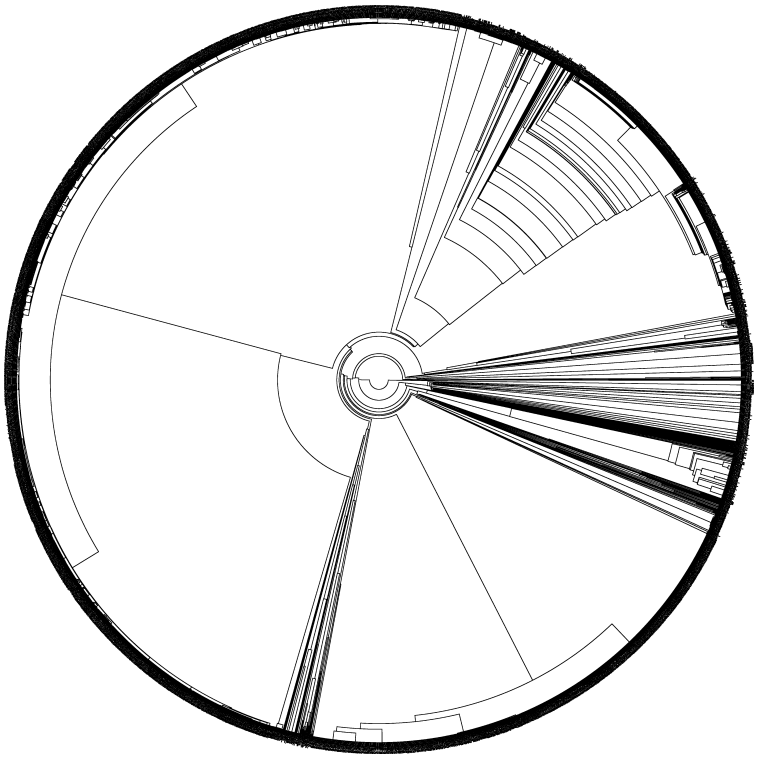

図 5　データセット C のデンドログラム
Fig. 5　Dendrogram for dataset C.

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 1 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 として検出されていることが分かった．一方，図 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

表 1　評価用マルウェア
Table 1　Malware for evaluation.

名称（本論文での略称）	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 2　評価用コンパイラ
Table 2　Compilers for evaluation.

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）

みてみると，IRCBot のほかにも Virut 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．一方，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．

5.4　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 1**）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 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．

(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

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 1 のすべてのマルウェアをコンパイル可能な Visual C++を用いた．**表 3**，**表 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 どうしの類似度は約 45%〜77%となっている．一方，rxBot と sdbot との類似度は 18%〜21%程度にとどまっており，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．

表 3　類似度行列 (%)，最適化オプション Od
Table 3　Similarity matrix (%), optimazation option Od.

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

表 4　類似度行列 (%)，最適化オプション Ox
Table 4　Similarity matrix (%), optimazation option Ox.

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

表 5　sdbot のソースコード行数
Table 5　The number of lines for sdbot.

名称	行数
sdbot04b	1,599
sdbot05a	1,902
sdbot05b	2,173

表 6　共通行の数
Table 6　The number of common lines.

	sdbot04b	sdbot05a
sdbot05a	1,312	—
sdbot05b	1,191	1,660

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なる．**表 5**，**表 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表している．ここで，ソースコードの行数を集合の要素数，共通行の数を積集合の要素数と見なすことで，本システムと同様，Jaccard 係数に基づき類似度を算出するこ

表 7　diff に基づく類似度 (%)
Table 7　Similarity matrix based on diff (%).

	sdbot04b	sdbot05a
sdbot05a	59.94	—
sdbot05b	46.14	68.74

表 8　sdbot05b に関する類似度 (%)
Table 8　Similarity matrix for sdbot05b (%).

	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

とができる．**表 7** にその算出結果を示す．sdbot05a と sdbot05b の類似度が最も高く，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 3 ・表 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 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 3 や表 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いえる．

6. ま と め

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キング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．これにより，ハニーポットで収集した約 3,000 種類のマルウェアを分類でき，さらにその分類結果より，わずか数種類のマルウェアを解析することで，全体の 75%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．今後は，コンパイラや最適化オプションの変更がプログラムコードに対して及ぼす影響をふまえつつ，マルウェアの分類技術を洗練化していく．

謝辞　本研究を進めるにあたり，ハニーポット設置環境およびマルウェア検体をご提供いただいた京都大学学術情報メディアセンター高倉弘喜准教授に深く感謝いたします．

参 考 文 献

1) Flake, H.: マルウェアの分類とアンパッキングの自動化，Black Hat Japan (2007).
2) サイバークリーンセンター．https://www.ccc.go.jp/report/h20ccc_report.pdf
3) Computer Security Research : McAfee Avert Labs Blog. http://www.avertlabs.com/research/blog/index.php/2009/07/22/
4) Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F. and Nazario, J.: Automated Classification and Analysis of Internet Malware, *RAID* (2007).
5) Md. Karim, E., Walenstein, A., Lakhotia, A. and Parida, L.: Malware phylogeny generation using permutations of code, *European Research Journal of Computer Virology*, Vol.1, No.1-2, pp.13–23 (Nov. 2005).
6) Gheorghescu, M.: An automated virus classification system, *Virus Bulletin Conference* (Oct. 2005).
7) Carrera, E. and Erdelyi, G.: Digital Genome Mapping – Advanced Binary Malware Analysis, *Proc. Virus Bulletin Conf.*, pp.187–197 (2004).
8) Wagner, R.A. and M.J. Fischer, The String-to-String Correction Problem, *J. ACM*, Vol.21, No.1, pp.168–173 (1974).
9) Crochemore, M. Iliopoulos, C.S. and Pinzon, Y.J.: Speeding-up Hirschberg and Hunt-Szymanski LCS algorithms, String Processing and Information Retrieval, pp.59–67 (2001).
10) Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/products/processor/manuals/
11) 岩村　誠，伊藤光恭，村岡洋一：コンパイラ出力コードの尤度に基づくアンパッキン

グ手法，*MWS2008*, pp.103–108 (2008).
12) 岩村　誠，伊藤光恭，村岡洋一：隠れマルコフモデルに基づく新規逆アセンブル手法，*Proc. 2008 IEICE General Conference* (2008).
13) 畑田充弘ほか：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，MWS2009 (Oct. 2009).
14) SRI International: An Analysis of Conficker. http://mtc.sri.com/Conficker/
15) PEiD. http://peid.has.it/
16) OllyBonE. http://www.joestewart.org/ollybone/
17) Using IDA to deal with packed executables. http://www.hex-rays.com/idapro/unpack_pe/
18) Quist, D., Smith, V. and Debugging, C.: Circumventing Software Armoring, *Black-hat USA 2007 / Defcon 15*, Las Vegas, NV (2007).
19) Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables, *Proc. 2007 ACM Workshop on Recurring Malcode*, pp.46–53 (2007).
20) VMware: Virtualization via Hypervisor, Virtual Machine & Server Consolidation. http://www.vmware.com/
21) Tron: He Fights for the User Alan Bradley. http://www.openrce.org/repositories/users/AlanBradley/Tron-TC8.pdf

(平成 21 年 11 月 30 日受付)
(平成 22 年 6 月 3 日採録)

推　薦　文

本研究では，ページフォールトハンドラを利用した汎用アンパッキング手法，アンパッキングされたマルウェアのメモリイメージから機械語命令列を特定する確率的逆アセンブル手法，そして機械語命令列の類似性に基づくマルウェアの分類手法を組み合わせて，アンパッキングから分類までの一連の作業を全自動化するツールとして組み上げている．さらに、本システムを用いた実証実験から，マルウェア本体の解析作業支援に適用して評価している．取り組み内容についても新規性が高く，多くの知見が得られている．開発したツールの効果も高く，有効性も高い．以上より，十分な新規性と有用性を認め推薦する．

（コンピュータセキュリティ研究会主査　菊池浩明）

岩村　　誠（正会員）

2000 年早稲田大学理工学部情報学科卒業．2002 年早稲田大学大学院理工学研究科修士課程修了．同年日本電信電話株式会社に入社，情報流通プラットフォーム研究所勤務（現職）．早稲田大学大学院基幹理工学研究科博士後期課程在籍．マルウェア対策の研究・開発に従事．

伊藤　光恭（正会員）

1982 年早稲田大学理工学部数学科卒業．1984 年早稲田大学大学院理工学研究科修士課程修了．同年日本電信電話公社入社．現在，NTT 情報流通プラットフォーム研究所勤務．

村岡　洋一（正会員）

1965 年早稲田大学理工学部電気通信学科卒業．1971 年イリノイ大学電子計算機学科博士課程修了．同学科助手の後，日本電信電話公社電気通信研究所に入所．1985 年より早稲田大学理工学部教授．