実行命令系列の比較によるアンパッキング手法の提案

伊沢 亮一[1]　森井 昌克[2]　井上 大介[1]

概要：我々はパッキングされたマルウェアのオリジナルエントリポイント（OEP）を特定する手法の研究を進めている．マルウェアを実行後，OEP にブレークポイントを設定することで，解析者はマルウェア本来のコードに焦点を充て解析することができる．昨年の CSS ではパッキングされたマルウェア 2 つの実行命令系列を比較すると，使用されているパッカーに依らず，共通するオリジナルコードの部分が類似度が高くなることを報告した．これは異なるマルウェアであったとしても，コードの使い回しにより機能が実装されることが多く，共通のコードを有しているためである．本稿では類似度が高い部分を基に OEP の候補アドレスを提示する手法を提案する．提案手法により提示される候補アドレスは尤もらしい順にソートされている．評価実験では 25 個のパッカーを用いて 747 個のサンプルを作成した．実験結果から，87%のサンプルに対して，候補アドレスの先頭 16 個に OEP が含まれていることを確認した．

キーワード：マルウェア，ソフトウェアパッカー，難読化，コード解析，サイバーセキュリティ

An Unpacking Method based on Instruction-trace Similarity

Ryoichi Isawa[1]　Masakatu Morii[2]　Daisuke Inoue[1]

Abstract: Detecting the original entry point (OEP) of malware is one of major challenges for malware analysis. This is becuase if an analyst recognizes the OEP, she can smoothly begin analyzing the original binary of malware starting at the OEP with debuggers. Considering how to invent an OEP detection method, we compared the instruction traces of malware samples, and reported that two malware samples shared some similar chunks of instructions in CSS (Computer Security Symposium) 2015. We also reported that the chunks constituted original binary. The reason is because malware authors often generate their malware with widespread collections of malicious code on the Internet. In this paper, we propose an OEP detection method based those shared chunks. With experiments using 747 malware samples packed with 25 packers, we confirmed that our method could provide a candidate set containing 16 memory addresses that contained in fact the OEP for each of 87% malware samples.

Keywords: Malware, Software packer, Obfuscation, Code analysis, Cybersecurity

1. はじめに

マルウェアの多くはパッキングされており，マルウェア本来のコード（オリジナルコード）を静的に解析することは難しい．そのため，解析者はデバッガーなどでマルウェアを実行し，メモリ上から自己解凍されるオリジナルコードを探してから解析する．このとき，オリジナルコード以外にも，自己解凍用ルーチン（アンパッキングルーチン）がメモリ上に展開，実行されるため，オリジナルコードを見つける作業は煩雑になりやすい．解析者のこのような労力に対し，マルウェア作成者は単にパッカーと呼ばれるツールを使用するだけで，マルウェアをパッキングすることができる．特に，パッカーは多種多様に存在し，使用されるパッカーにより，オリジナルコードの展開されるメモリアドレスやタイミングが異なり，オリジナルコードを探す作業をより煩雑にしている．パッカーの例として，UPX[1]

[1] 国立研究開発法人情報通信研究機構
National Institute of Information and Communications Technology
[2] 神戸大学大学院工学研究科
Graduate School of Engineering, Kobe University

や ASProtect[2]，Molebox[3]，Themida[4] があげられる．

本研究ではオリジナルエントリポイント（OEP）の候補アドレスを提示する手法を提案する．提案手法はパッカーの種類に依存しない汎用的な手法を目指す．従来においても汎用的にオリジナルコードもしくは OEP を特定を目指した手法 [5], [6], [7], [8] は提案されているが，昨年の IEEE S&P の Ugarte-Pedrero らの報告 [9] にもあるように，従来手法は必ずしもパッカーの共通の特徴を捉えておらず，対応できないパッカーも存在する．本研究の課題はパッカーの種類を問わない汎用的な手法を実現するため，どのような情報を用いればよいかを発見することである．

我々はコードの使い回しによりマルウェアが作成されることが多いことに着目した．パッキングされたマルウェアはメモリ上で自己解凍（自動でアンパッキング）されるため，メモリ上にオリジナルコードが展開されることになる．問題はオリジナルコードがどのタイミングでアンパッキングが終わるか分からないことである．そこで，解析対象のマルウェアの実行命令系列（実行された命令とアドレスの集合）と別のマルウェアの実行命令系列を比較すれば，使い回されているコードの部分の類似度が高くなり，それを基にすれば，解析対象の OEP を求められるのではないかと考えた．昨年の CSS では，ある 2 つのパッキングされたマルウェアを比較したとき，類似度が高い部分はアンパッキングルーチンかオリジナルコードであることを報告した [10]．このとき，2 つのマルウェアが異なるパッカーでパッキングされていることが分かれば，類似度が高い部分はオリジナルコードとなる．異なるパッカーはアンパッキングルーチンが異なるため類似度は高くならないためである．

提案手法では 2 つの実行命令系列の序盤のほうの系列を比較して，パッカーの同一性を判定する．これはアンパッキングルーチンが実行直後に実行されることを利用している．もし同じパッカーが使用されていれば，解析対象ではないほうのマルウェアを別のものに入れ替えて再度比較する．次に，解析対象の実行命令系列のうち，類似度が高い部分にメモリアドレス距離が近い部分は同じくオリジナルコードとして扱う．これはアンパッキングルーチンとオリジナルコードはメモリ上である程度の距離離れた位置に展開される点に着目している．もし，それらを近い距離に展開しようとすると，互いを上書きしてしまう可能性がでてくるためである．その後，オリジナルコードと判定された部分のうち，実行順などを考慮して，オリジナルエントリポイントの候補アドレスを抽出する．この処理を解析対象ではないほうのマルウェアを任意の回数入れ換えながら候補アドレスを抽出する．最終的に，抽出された回数順に候補アドレスをソートして提示する．提案手法は単に解析対象のマルウェアともう一方のマルウェアを比較するだけでよく，解析者にマルウェアの事前知識を求めない．

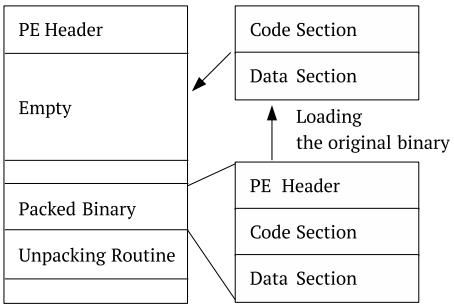

図 1 UPX でパッキングされたプログラムの構造と動作

Address	Instruction
7c95be1b	push esi
7c95be1c	push dword ptr [ebp+0x8]
7c95be1f	call 0x7c95bdab
7c95bdab	mov edi, edi
...	...
0041081c	push edi
0041081d	or ebp, 0xffffffff
00410820	jmp 0x410832
00410832	mov ebx, dword ptr [esi]
...	...
00407f21	call 0x4017f0
004017f0	push ebp
004017f1	mov ebp, esp
004017f3	mov eax, 0x1f94
...	...

図 2 実行命令系列の一例

評価実験では 25 個のパッカーで 30 個のマルウェアをパッキングして作成した，747 個のマルウェアを用いた．このとき動作しないものは除いている．これらのうち，78%のサンプルに対し，候補アドレスの先頭に OEP が位置し，87%のサンプルに対し，候補アドレスの先頭から 16 個に OEP が含まれていることを確認した．

2. 基礎知識

2.1 パッカー

パッカーの動作を説明するために，UPX でパッキングされたプログラムの構造を図 1 に示す [8]．パッキングされたプログラムは，PE ヘッダ，空のセクション，パッキングされたバイナリ，UPX のアンパッキングルーチンで構成される．このプログラムが実行されると，パッキングされたバイナリがアンパッキングルーチンによりメモリ上で復号される．そして，復号されたオリジナルコードが空のセクションに書き込まれる．その後，オリジナルコードが実行される．UPX はシンプルなパッカーであるが，Themida や telock など，複雑なパッカーガ多く存在する．

2.2 実行命令系列

PIN[11] や Valgrind[12] などの Dynamic Binary Instru-

mentation Tool（DBI ツール）や，QEMU[13] や Xen[14] の仮想環境上でマルウェアを実行することにより，実行された命令郡（実行命令系列）が取得できる．実行命令系列の一例を図 2 に示す．“push” や “call” のような実行された命令が取得できる．パッキングされたマルウェアであれば，この実行命令系列にアンパッキングルーチンとオリジナルコードが含まれる．

2.3 N-gram 類似度

ある 2 つの文字列の類似度を N-gram により比較する方法として Dice coefficient が提案されている [15]．N-gram とは，与えられた文字列のうち，連続した N 個の文字列のことを指す．本稿では実行命令系列を文字列に対応付け，オペコードを文字に対応付ける．例えば，“push push call mov” であれば，2-grams は “push push”，“push call”，“call mov” となり，3-grams は “push push call”，“push call mov” となる．そして，以下の Dice coefficient の計算方法 [15] により与えられた 2 つの実行命令系列の類似度を計算する．

$$S_{(A,B)} = \frac{2|\text{N-grams}(A) \cap \text{N-grams}(B)|}{|\text{N-grams}(A)| + |\text{N-grams}(B)|} \quad (1)$$

ここで，A と B は与えられた実行命令系列，N-grams(·) は N-gram の集合，$|\cdot|$ は N-gram の数をそれぞれ意味し，$0 \leq S_{(A,B)} \leq 1$ である．本稿では $S_{(A,B)}$ を N-gram 類似度と呼ぶ．

3. 提案手法

3.1 基本アイデア

実行命令系列の類似度の高い部分と，アンパッキングルーチンとオリジナルコーオのメモリアドレス距離に着目することが提案手法の基本アイデアである．

もし異なるパッカーでパッキングされた 2 つのマルウェアがあったとして，それらの実行命令系列を比較したとき，類似度が高い部分がオリジナルコードとなる [10]．同じパッカーでパッキングされているかどうかは事前には分からないため，提案手法では実行命令系列の序盤の命令の類似度を見ることで判定する．序盤の命令というのは，具体的には実行前から存在する命令のことを指す．オリジナルコードがパッキング（暗号化もしくは圧縮）されているのに対し，アンパッキングルーチンは実行前から存在する．そこで，実行前から存在する命令を比較することで，アンパッキングルーチンが似ているかどうかを調べ，似ていれば同じパッカーでパッキングされたものと判定する．

類似度が高い部分からメモリアドレス距離の近い命令を同じオリジナルコードとして扱う．これはアンパッキングルーチンとオリジナルコードが離れた位置に展開されることに着目した．もし互いの距離が近いとそれぞれが上書きされることを防ぐためである．オリジナルコードと判定された部分からアドレスを抽出して OEP の候補アドレスとする．詳細な手順は次節で述べる．

3.2 候補アドレス抽出の手順

2 つの実行命令系列を比較するときに，全体を比較するのではなく，部分部分を比較するためにある基準で実行命令系列を分割する．このとき，アンパッキングルーチンとオリジナルコードがある程度事前に分かれていることが望ましい．この分割方法として Ugarte-Pedrero らが提案したレイヤー [9] を用いる．図 3 に示すように，プログラムが実行されているメモリにおいて，レイヤー L_i $(i = 0, 1, 2, \cdots)$ の命令がある領域にデータを書き込み，そのデータが実行されたときに新しいレイヤーが定義される．このとき，新しいレイヤーの番号は書き込み元のレイヤー番号に 1 を加えた値となる．例えば，パッカーのアンパッキングルーチンが，次のアンパッキングルーチンを書き込み，実行するのであれば，次のレイヤーには動的に生成されたアンパッキングルーチンが格納される．

解析対象のマルウェア（以下，*Target*）ともう一方のマルウェア（以下，*Opposite*）の実行命令系列をレイヤー単位で N-gram 類似度を求め，表 1 のような類似度行列を取得する．レイヤー L_0 は実行前から存在する命令列であり，パッカーのアンパッキングルーチンに対応する．そこで，$S_{(L_0^{(T)}, L_0^{(O)})}$ の類似度が低いと異なるパッカーでパッキングされているものと判定する．異なるパッカーでは類似度はほぼ 0 となるため [10]，本稿では $S_{(L_0^{(T)}, L^{(O)})}$ が 0.3 以上のときは同じパッカーでパッキングされているものとし，*Opposite* を捨て，別のサンプルを使用する．

$S_{(L_i^{(T)}, L_j^{(O)})}$ のいずれかの類似度が高い $L_i^{(T)}$ を選択する．このとき予め与えられる閾値 T_{sim} より $S_{(L_i^{(T)}, L_j^{(T)})}$ が高ければ，その類似度は高いとする．類似度が高いレイヤーにはオリジナルコードが多く含まれているということに起因する．次に，$L_x^{(T)}$ $(x = 1, 2, \cdots)$ から $L_j^{(O)}$ と共通する N-gram を抜き出し，それらの中で最初に実行された N-gram の先頭アドレス *Addr* を選ぶ．類似度が高い $L_x^{(T)}$ が複数あれば各レイヤーから *Addr* を抽出し，$(Addr_1, Addr_2, \cdots)$ $(y = 1, 2, \cdots)$ を得る．

アンパッキングルーチンとオリジナルコードのアドレス距離が離れることに着目し，*Addr* から閾値 T_{dist} 以内の命令を抽出する．さらに抽出された命令のアドレスを起点として同じく T_{dist} 以内の命令を抽出し，これを繰り返す．抽出された命令のうち，最初に実行された命令のアドレスを候補アドレス c_1 として抽出する．これを $(Addr_1, Addr_2, \cdots)$ それぞれに対して行い，$(c_1, c_2, \cdots, c_k)$（k は c の個数）を得る．これを実行順にソートして，$(sc_1, sc_2, \cdots, sc_k)$ とし，各アドレスに以下の式により重みを与える．

$$w_{sc_l} = \frac{k - l + 1}{\sum_{x=1}^{k} x} \quad (2)$$

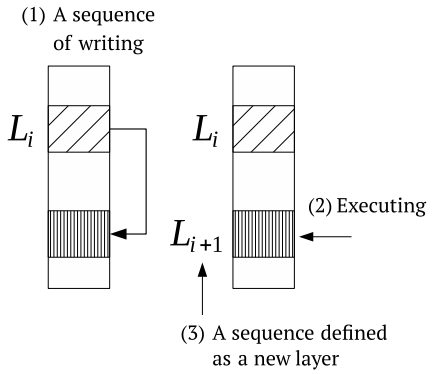

図 3 Layer の定義

表 1 レイヤーの類似度行列（(T): *Target*, (O): *Opposite*）

		(O)			
		$L_0^{(O)}$	$L_1^{(O)}$	$L_2^{(O)}$	$\cdots$
(T)	$L_0^{(T)}$	$S_{(L_0^{(T)},L_0^{(O)})}$	$S_{(L_0^{(T)},L_1^{(O)})}$	$S_{(L_0^{(T)},L_2^{(O)})}$	$\cdots$
	$L_1^{(T)}$	$S_{(L_1^{(T)},L_0^{(O)})}$	$S_{(L_1^{(T)},L_1^{(O)})}$	$S_{(L_1^{(T)},L_2^{(O)})}$	$\cdots$
	$L_2^{(T)}$	$S_{(L_2^{(T)},L_0^{(O)})}$	$S_{(L_2^{(T)},L_1^{(O)})}$	$S_{(L_2^{(T)},L_2^{(O)})}$	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

例えば (sc_0, sc_1) であれば，$(w_{sc_0}, w_{sc_1}) = (0.66, 0.33)$ となる．任意の個数の *Opposite* と比較し，$(sc_1, sc_2, \cdots)$ を取得し，sc_x $(x = 1, 2, \cdots)$ が示すアドレスに重みを加算していく．最後に，アドレスを重み順に並べ，最も重みが大きいアドレスが最尤の OEP 候補アドレスとする．

4. 評価実験

4.1 データセットと実験環境

パッキングされていない 30 個のマルウェアを次のように選択した．過去 7 年間に渡りハニーポットや Web クローラーで収集したマルウェアや協力組織から提供されたマルウェアがある．これらのマルウェアに対し，シマンテックのアンチウイルスソフトでマルウェア名を付加して分類したところ，135 のカテゴリができた．例えば，‘Backdoor.IRC.Bot’ や ‘Unknown’ である．このうち，Unknown を除く，マルウェア数の多い上位 30 カテゴリに対し，PEiD（シグネチャによるパッカー判定ツール）および Lyda らのエントロピ解析手法 [20] を用いて各マルウェアのパッキングの有無を調べた．そして，各カテゴリのパッキングされていないマルウェアから無作為に 1 検体ずつ抽出した．これは正解となる OEP を確実に得るためである．これによりマルウェア名の異なる 30 個のパッキングされたマルウェアを選んだ．また，これら 30 個の SHA256[21] のハッシュ値は重複しないことを確認した．その後，25 種類のパッカーにより 30 個のマルウェアをパッキングし，動作しないものを捨て，最終的に 747 個のパッキングされたサンプルを得た．

実験環境として，Windows 7 を QEMU にインストールした．Windows 7 のカーネルドライバと QEMU に提案手法を実装し，実行命令系列やレイヤー分けなどができるようにした．この Windows 7 上で各検体を実行し，OEP の候補アドレスを抽出し，OEP が何番目までに含まれるかを評価した．なお，本稿では Windows 7 に提案手法を実装しているが基本アイデアは他のプラットフォームにも適用可能であると考えている．

4.2 実験結果

提案手法は *Target* と複数の *Opposite* を比較した結果，投票のようにして候補アドレスを尤もらしい順に並べる．しかし，各単一の比較で OEP をある程度特定することができないのであれば，投票をしても OEP を選出することはできない．そこで，*Target* と 1 つ 1 つの *Opposite* の比較結果を検証した．具体的には，*Target* と 1 つの *Opposite* を比較し，最も高い類似度のレイヤー $L(T)_i$ からのみ候補アドレス *Addr* を抽出した．そのアドレスが OEP かどうかを調べた．表 2 はその結果を示している．*Target* として Backdoor.IRC.Bot を ACProtect，ASPack，exe32pack でパッキングした 3 検体を用いた．*Opposite* には *Target* 以外の 747 検体とし，それぞれ独立に比較した．この比較により，抽出したアドレスが OEP だったとき，Success に 1 を加え，OEP でなかったとき Failure に 1 を加える．*Target* と *Oppoiste* が同一のパッカーでパッキングされていると判定された，もしくは類似度の高い部分がなかったときは，その *Opposite* は破棄されて，Discarded に 1 が加えられる．ACProtect でパッキングされた Backdoor.IRC.Bot は 67 個の *Opposite* の比較により OEP の特定に成功している．ASPack でパッキングされた Backdoor.IRC.Bot は Failure が 32 あるが，Success が 64 あるため，投票により OEP が尤もらしいアドレスとして選ばれる．exe32pack は全ての *Opposite* が破棄されたが，解析者には破棄されたことが分かるため，OEP のアドレスが誤って伝わることはないという点で最悪の結果ではない．投票をしなくてもある程度 OEP が特定できることが確認できた．

次にあるサンプルを *Target* とし，残りの 747 個の *Opposite* と比較することで候補アドレスを取得した．このとき，先頭 $m(= 1, 2, \cdots)$ 番目以内に OEP が含まれていれば成功とし，パッカー毎に成功した *Target* の割合を求めた．この割合を評価指標 Recall （OEP を逃さず取得できた割合を意味する）と呼ぶ．パラメータチューニングでは N-gram$N$ を $\{4, 8, 16, 32, 48, 64, 72, 80\}$，$T_{sim}$ を $\{0.002, 0.004, 0.008, 0.016, 0.032, 0.064, 0.128, 0.256\}$，$T_{dist}$ を $\{32, 64, 128, 256, 512, 1024, 2048, 4096\}$ の総当たりで調べ，$m = 1$ の結果が最も良かった組み合わせが $N = 16$，T_{sim}=0.256，T_{dist}=64 だった．この結果を表 3 に載せる．表中の ACProtect を見ると，ACProtect でパッキングされた *Target* は 29 個あり，1 番目の候補アドレスが OEP だった割合が 93%（27 個）だったことを示している．$m = 1$ の

表 2 各比較から候補アドレスを 1 つだけ抽出した結果（*Target* は Backdoor.IRC.Bot，Packer: Backdoor.IRC.Bot に使用されているパッカー，Success: 抽出したアドレスが OEP だった回数，Failure: OEP ではなかった回数，Discarded: 捨てられた *Opposite*，# of *Opposite*: 比較したマルウェア数）

Packer	Success	Failure	Discarded	# of *Opposite*
ACProtect	67	0	680	747
ASPack	64	32	651	747
exe32pack	0	0	747	747

Average を見ると 78%となっており，*Target* の 78%に対して一意に OEP が特定できたことを示す．$m = 16$ を見ると 87%の *Target* に対して，候補アドレスの 16 番目までに OEP が格納されていた．$m = 64$ と $m = 128$ の Average はともに 0.89 であり，候補数（m の値）を増やしても問題ない場合は，別のパラメータの組み合わせの方が有効であると考えられる．

5. 関連研究

これまでにもアンパッキングの手法は提案されている．OS のバージョンが更新されていく中，手法の実装が古いものもあるが，それぞれのアンパッキングの基本的な考え方は現在においても有効であると考えられる．

PolyUnpack[5] は動的に生成されたコードを取得することを目的としている．アンパッキングの対象となる実行ファイルを 1 命令ずつ実行し逆アセンブルする．命令列と初期状態のパッキングされたファイルと比較して，その命令列がファイル中に存在しなければ動的に生成されたコードとする．

Renovo[7] は JMP 命令に着目して OEP を検出する．JMP 命令により書き込まれたコードに処理が繊維したとき，EIP のアドレスを OEP とする．対象のファイルを 1 命令ずつ実行し逆アセンブルすることで W&X (Written&eXecuted) ページ検出（書込/実行された箇所の抽出）を実装している．Ether[16] は仮想環境の Xen[14] を基にパッカーが持つアンチデバッグ機能を回避する環境を構築し，その上で Renovo と同じアンパッキング手法を実装している．

Kim らの手法 [18] はメモリへの書き込みと実行の遷移に着目して OEP を検出する．パッキングされたファイルにより書き込まれたメモリ上の各コードが連続して実行されたとき，その実行の開始地点を OEP の候補とする．

Kawakoya らの手法 [17] はメモリアクセスの傾向に着目して OEP を検出する．メモリアクセスとは書き込みと読み込み，実行のことであり，これらを式によりこれらの傾向を数値化して最も変化した地点を OEP とする．

OmniUnpack[6] は書込/実行されたコードを判定し，そのコードが危険なシステムコールを呼び出したときに，アンチウイルスソフトを呼び出す．実行の例外の度にアンチウイルスソフトを呼び出すと処理が遅くなるため，その呼び出し回数を削減することが目的である．パッカーで内容が変更されたマルウェアをアンチウイルスソフトで検出することを目的としており OEP は検出しない．なお，OmniUnpack は書込/実行されたコードの判定に OllyBonE[19] を用いている．OllyBonE は処理速度を向上させるために，メモリに書込/実行の禁止属性を設定することで W&X ページ検出が可能である．

Guo ら [8] は OmniUnpack と同様にアンチウイルスソフトでパッキングされたマルウェアを検査することを目的にしている．OmniUnpack との主な差異は OEP を起点にアンチウイルスソフトで検査することにあり，Guo らは OEP を判定するヒューリスティックを 3 つ提案している．1 つ目は読み込んでいるメモリ（メモリページ）にアンパッキングルーチンらしきものが含まれていないことのチェック，2 つ目はスタックポインターが起動時の状態と同じであること，3 つ目は実行時のコマンドライン引数へのアクセスがあること，である．それぞれの基準を組み合わせるのではなく，W&X ページ検出といずれか 1 つを組み合わせる．例外が発生し，基準を見たしたとき，例外が発生した地点を起点にアンチウイルスソフトに検査させる．

6. まとめ

本稿の実験では，1 組のパラメータの結果のみ示した．今後の予定として Precision（m の中に OEP の含まれる割合）と Recall（OEP を捕捉できる割合）のトレードオフの関係を調べる予定である．また，T_{sim} に対して，0.256 が最も良い結果だったが，それ以上の値に対しては調べていない．T_{sim} を 0.256 より大きい値にしたときの結果についても調べる予定である．

参考文献

[1] Oberhumer, M. F., Molnar, L. and Reiser, J. F.: UPX: Ultimate Packer for eXecutables, http://upx.sourceforge.net/.

[2] StarForce Technologies Ltd.: ASPack Software, http://www.aspack.com/.

[3] DesaNova Ltda.: Virtualization and Protection Tools at Molebox.com, http://www.molebox.com/.

[4] Oreans Technologies: Themida, http://www.oreans.com/themida.php.

[5] Royal, P., Halpin, M., Dagon, D., Edmonds, R. and Lee, W.: PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware, Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06), pp. 289300 (2006).

[6] Martignoni, L., Christodorescu, M. and Jha, S.: OmniUnpack: Fast, Generic, and Safe Unpacking of Malware, Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07), pp. 431441 (2007).

[7] Kang, M. G., Poosankam, P. and Yin, H.: Renovo: A Hidden Code Extractor for Packed Executables, Proceedings of the 5th ACM workshop on Recurring Mal-

表 3 Recall (N=16, T_{sim}=0.256, T_{dist}=64)

Packer	#	$m=1$	4	8	16	32	64	128
ACProtect	29	0.93	1.00	1.00	1.00	1.00	1.00	1.00
ASPack	30	0.93	1.00	1.00	1.00	1.00	1.00	1.00
ASProtect	30	0.67	0.67	0.70	0.87	0.90	0.90	0.90
exe32pack	9	0.00	0.00	0.00	0.00	0.00	0.00	0.00
exeStealth	30	0.77	0.77	0.77	0.77	0.80	0.80	0.80
Ezip	30	0.93	1.00	1.00	1.00	1.00	1.00	1.00
FSG	29	0.93	1.00	1.00	1.00	1.00	1.00	1.00
Mew	28	0.96	1.00	1.00	1.00	1.00	1.00	1.00
Molebox	30	0.53	0.60	0.60	0.60	0.60	0.63	0.63
Mpress	30	0.87	0.97	1.00	1.00	1.00	1.00	1.00
Npack	27	0.93	1.00	1.00	1.00	1.00	1.00	1.00
NSpack	29	0.93	1.00	1.00	1.00	1.00	1.00	1.00
Packman	30	0.90	1.00	1.00	1.00	1.00	1.00	1.00
PEcompact	29	0.93	1.00	1.00	1.00	1.00	1.00	1.00
PEpack	27	0.74	1.00	1.00	1.00	1.00	1.00	1.00
Pespin	29	0.34	0.34	0.34	0.38	0.38	0.45	0.45
Petite	17	0.41	0.53	0.65	0.94	0.94	0.94	0.94
Pklite32	11	1.00	1.00	1.00	1.00	1.00	1.00	1.00
RLpack	29	0.90	1.00	1.00	1.00	1.00	1.00	1.00
Scrambler	26	0.92	1.00	1.00	1.00	1.00	1.00	1.00
Simplepack	28	0.93	1.00	1.00	1.00	1.00	1.00	1.00
telock	19	0.47	0.47	0.47	0.53	0.58	0.74	0.74
Themida	30	0.57	0.57	0.60	0.67	0.73	0.77	0.77
Upack	28	0.93	1.00	1.00	1.00	1.00	1.00	1.00
UPX	30	0.93	1.00	1.00	1.00	1.00	1.00	1.00
Winupack	28	0.93	1.00	1.00	1.00	1.00	1.00	1.00
Wwpack32	19	0.89	0.89	0.89	0.89	0.89	0.89	0.89
Yodaprotector	30	0.60	0.63	0.67	0.80	0.83	0.87	0.87
Average		0.78	0.84	0.85	0.87	0.88	0.89	0.89

code (WORM'07), New York, NY, USA, ACM, pp. 4653 (2007).

[8] Guo, F., Ferrie, P. and Chiueh, T.-C.: A Study of the Packer Problem and Its Solutions, Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08), Berlin, Heidelberg, Springer-Verlag, pp. 98115 (2008).

[9] Ugarte-Pedrero, X., Balzarotti, D., Santos, I. and Bringas, P. G.: SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers, Proceedings of 2015 IEEE Symposium on Security and Privacy, pp. 659673 (2015).

[10] 伊沢亮一，森井昌克，井上大介，“汎用的なアンパッキング手法の検討: 実行命令系列の類似度比較,” コンピュータセキュリティシンポジウム 2015（CSS2015）予稿集，pp.473-479, 2015.

[11] Intel Corporation: Pin - A Dynamic Binary Instrumentation Tool, available at http://software.intel.com/en-us/articles/
pin-a-dynamic-binary-instrumentation-tool.

[12] Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, SIGPLAN Not., Vol. 42, No. 6, pp. 89100 (2007).

[13] Bellard, F.: QEMU, http://www.qemu.org/.

[14] Xen Project: The Xen Project, http://www.xenproject.org/.

[15] Brew, C. and McKelvie, D.: Word-pair extraction for lexicography, Proceedings of the 2nd Intl Conf. on New Methods in Language Processing, pp. 4555 (1996).

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," Proceedings of the 15th ACM conference on Computer and communications security, pp.51–62, CCS'08, ACM, New York, NY, USA, 2008.

[17] Kawakoya, Y., Iwamura, M. and Itoh, M.: Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment, Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE'10), pp. 3946 (2010).

[18] Kim, H. C., Orii, T., Yoshioka, K., Inoue, D., Song, J., Eto, M., Shikata, J., Matsumoto, T. and Nakao, K.: An Empirical Evaluation of an Unpacking Method Implemented with Dynamic Binary Instrumentation, IEICE Transactions, Vol. 94-D, No. 9, pp. 17781791 (2011).

[19] J. Stewart, "Ollybone v0.1, break-on-execute for ollydbg". http://www.joestewart.org/ollybone/.

[20] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," IEEE Security and Privacy, vol. 5, no. 2, pp. 4045, Mar. 2007.

[21] P. G. John Bryson, "Secure hash standard (shs) (federal in- formation processing standards publication 180-4)," available at http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, 2012.