5　サイバーセキュリティ技術：進化する脅威への対策技術と成果展開
5-1　難読化されたマルウェアに対する解析支援技術

伊沢亮一　班　涛

マルウェア*1 の脅威に対処するために、マルウェアの捕獲、解析、対策の導出を迅速に行うことが求められる。これら 3 つの段階のうち、我々は解析の効率化に焦点をあて、パッキング（暗号化／圧縮）*2 されたマルウェアに対する汎用アンパッキングシステムを研究してきた。本システムにより、パッキングの種類を問わず、マルウェア本来のコードを自動抽出でき、解析の効率化につながる。本稿では我々の汎用アンパッキングシステムを概説する。

1　まえがき

日々発生するサイバー攻撃の件数は増加の一途をたどっており、それらサイバー攻撃にはマルウェアが使用されることが多い。例えば、バンキングマルウェアといえばここ数年で一般に知られる脅威となったのではないだろうか。このマルウェアは PC からオンラインバンクの情報を不正に取得し、口座から金銭を奪うといった挙動を行う。他にも、PC 内の文書を暗号化して閲覧不可能にするランサムウェアも猛威をふるっている。ランサムウェアに PC が感染すると、攻撃者に送金する方法が画面に表示されることがあり、実際に送金すると、暗号化されたファイルを復号（以下、アンパッキング）するための鍵が送られてくることがある。もし、正しい鍵が取得できればファイルの閲覧が可能となるため、どうしてもファイルを閲覧したいという被害者は送金する他ない状況に追い込まれる。ただし、送金したからといって鍵が送られてくる保証は一切ない。このようなマルウェアの感染を未然に防ぐ、もしくは、いち早く感染後の被害から回復するためには、マルウェアの捕獲、解析、対策の導出を効率的に行うことが求められる。これは、マルウェア解析により感染経路などの挙動を把握した後、その情報をもとに対策を講じるためである。本研究では解析を効率化することに焦点をあて、研究を進めている。

マルウェアの多くは、コード解析*3 を妨害するために、パッキングが施されている。そのため、解析をする際、解析者はパッキングの種類を特定してから、アンパッキングするための手順を考え、マルウェア本来のコード（以下、オリジナルコード）を抽出しなければならず非常に手間がかかる。さらに、マルウェアの作成者は UPX[1] や Themida[2] などのパッキングツール（以下、パッカー）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、パッキングの労力はほぼない。加えて、多くの種類のパッカーが流通しているため、あるマルウェアに対してどのパッカーが使用されたかはマルウェア捕獲時点では解析者側には分からず、捕獲の都度、使用されているパッカーの特定やアンパッキングをしなければならない。そこで、オリジナルコードを自動で抽出することができれば、解析を開始するまでの手間が削減でき、結果的に解析の大幅な効率化につながる。我々は解析の効率化のため、汎用アンパッキングシステムの研究開発に取り組んできた。

従来から、汎用アンパッキングシステムは盛んに研究されており [3]–[9]、パッキングされたプログラムに共通する次のような動作に着目して汎用アンパッキングを行う〈パッキングされたプログラムが実行された後、アンパッキングルーチン（復号用のコード）が実行される。アンパッキングルーチンはパッキングされているオリジナルコードをアンパッキングしながらメモリ上に書き込み、その後、オリジナルコードが実行される。ここで、オリジナルコードの最初に実行された地点をオリジナルエントリポイント（以下、OEP）と呼ぶ。〉このように、いずれのパッカーにおいても、オリジナルコードはメモリに書き込まれてから実行されるため、書込／実行されたコードのいずれかにオリジナルコードは存在することになる。このとき、OEP を検出することができれば、それ以降にオリジナル

＊1　悪意のあるソフトウェア
＊2　マルウェアに使用される難読化技術のうち、本稿ではパッキング（暗号化／圧縮）を取り扱う。
＊3　プログラミングコードから情報を得る解析

コードが実行されることが分かるため、OEP の検出がひとつの課題となっている。

書込／実行されたコードを検出する方法として、逐次実行による方法とデータ実行防止による方法が挙げられる。逐次実行による方法では、プログラムを実行した後、1 命令*4 ごとに実行を停止しながら、書き込まれた命令とそのアドレスを記録していく。書き込まれた命令が実行されたとき、その実行されたアドレスを OEP の候補として検出する。この方法では 1 命令ごとに実行を停止させるため、実行時間が非常に長くなることが欠点となる。データ実行防止による方法では、書き込みがあったメモリ領域を実行禁止に設定していく。もし、実行禁止に設定されたメモリ領域の命令が実行されようとすると、実行禁止の例外が発生するため、書込／実行を検出できる。この方法では逐次実行による方法に比べ高速に処理が可能であるが、例外が発生したときのみにしか CPU レジスタなどの情報を得ることができない。そのため、書込／実行された命令の中から OEP の命令を特定するための情報量が少ないといった欠点がある。

本研究ではマルウェアが日々大量に発生していることにかんがみ、高速に処理が可能な後者のデータ実行防止による方法を採用する。その上で、この方法をもとに、OEP の検出精度が高い汎用アンパッキングシステムを提案する。提案システムは 2 つの機能を有している。1 つめの機能では、書込／実行された命令を生み出した親命令をアンパッキングルーチンとして検出する。これは、アンパッキングルーチンが、メモリ上に命令を書き込むという点に着目している。アンパッキングルーチンの直後に実行された命令のアドレスを OEP の最有力候補とする。2 つめの機能では、OEP の最有力候補をもとに他の候補をもっともらしい順に並べ替える。解析者は最有力候補を確認した後、もし OEP ではなかった場合、次の候補を確認していくことで、より早く OEP にたどりつける。なお、開発した汎用アンパッキングでは提案システムの前段にパッキングされたプログラムのパッカー特定 [10] を行う。パッカー特定により、既知のパッカーで、かつ、アンパッキングのアルゴリズムが実装されていた場合には、その実装でアンパッキングをする。もし、システムに登録されていない未知パッカーであれば、汎用アンパッキングシステムにより OEP の特定を試みる。

マルウェア解析の効率化はマルウェアが多発している現状において非常に重要な課題であり、NICT が取り組むべき課題あると考えている。より OEP の検出精度の高いシステムを研究開発するとともに、実用的には従来システムと組み合わせながら使用することで、解析者の負担の軽減につなげる。

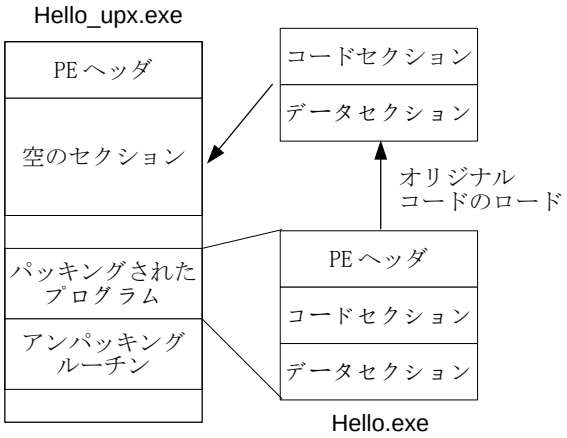

図 1　UPX でパッキングされたプログラムのファイル構造と挙動

本稿の構成は次の通りである。**2** では基礎知識として、パッカーの基本的な動作やデータ実行防止による方法について述べる。**3** では主に汎用アンパッキングシステムについて説明する。**4** でその評価実験の結果を示し、**5** でまとめる。なお、本稿の内容は文献 [10]、[11] で詳細な成果を報告している。

2　基礎知識

2.1　パッカー

パッカーとはプログラムをパッキング（暗号化／圧縮）するソフトウェアツールである。パッカーは対象のプログラムをパッキングし、それにアンパッキングルーチンを付加する。このアンパッキングルーチンにより、自己解凍及び実行が可能となっている。例として、UPX でパッキングされたプログラム Hello_upx.exe のファイル構造を図 1 に示す。Hello_upx.exe は PE ヘッダと空のセクション、パッキングされたプログラム（Hello.exe）、アンパッキングルーチンで構成されている。Hello_upx.exe が実行されると、アンパッキングルーチンが先に実行され、パッキングされたプログラムをアンパッキングし、メモリ上に展開する。アンパッキングされた Hello.exe は空のセクションに書き込まれた後、実行される。

UPX は非常に単純なパッカーだが、Themida や ASProtect[12] などは複雑なパッキングアルゴリズムを有しており、パッキングに使用されたパッカーによりプログラムの動きは異なってくる。ただし、いずれのパッカーでパッキングされたとしても、元のプログラムは必ずメモリ上でアンパッキングされた後、書き込みされてから実行される。書込／実行されたコード

*4　命令とは機械語命令のことを意味し、mov や jmp などがある [18]。

の中からいかにして OEP を検出するか、もしくはオリジナルコードを抽出するかが、汎用アンパッキングの要点となる。

2.2　解析基盤

逐次実行による方法とはプログラムを 1 命令ずつ逆アセンブル*5 しながら実行、停止を繰り返すことを指す。このようにすることで、実行された命令、命令の実行順、メモリの変化、CPU レジスタなどの情報を命令実行ごとに取得できる。この逐次実行による方法は Xen[13] や KVM[14]、QEMU[15] などの仮想環境か、PIN[16] や Valgrind[17] に代表される DBI ツール（Dynamic Binary Instrumentation）により実現できる。

データ実行防止の方法とは指定した範囲のメモリ領域上のコードを実行不可にできる機能である。Intel 64 and IA-32 アーキテクチャ [18] などの CPU ではメモリ空間を 4096 バイトのメモリページで仮想的に区切り、それぞれのページに対して書き込み許可／禁止や実行許可／禁止の属性を管理できる。図 2 に、書込／実行された命令を検出するための方法を示している。はじめにメモリ全体を R/X（Read-only/eXecutable。つまり、書き込み禁止）に設定しておく。その後、図中ではメモリページ Q に対して書き込みが行われている。このとき、書き込み禁止の例外が発生し、書き込まれたメモリページが Q であることが通知されるため、Q を W/NX（Writable/None-eXecutable。つまり、書き込み許可、実行禁止）に設定する。次に、Q 上の命令が実行されたとき、実行禁止の例外が発生するため、Q 上の命令を書込／実行された命令として判定する。実行禁止の例外の後、再度 Q を R/X に設定しておく。このようにすることで、OEP 候補となる命令のアドレスが取得できる。なお、メモリページ単位で管理しているため、ある 1 命令だけを実行禁止にするといったことはできない。また、CPU レジスタなどの情報取得のタイミングが、例外が発生したときに限定されるため、逐次実行による方法に比べると、得られる情報量は少なくなる。ただし、実行が停止する回数が逐次実行による方法に比べ少ないため、高速に処理できる点がデータ実行防止による方法の利点である。

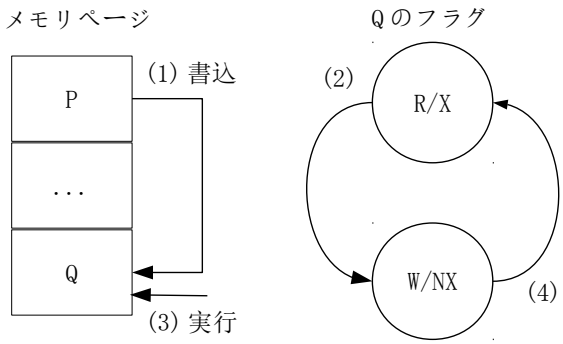

図 2　フラグの状態遷移（R/X: 読込のみ可（書込不可）／実行可、W/NX: 書込可／実行不可）

3　汎用アンパッキングシステム

3.1　基本アイデア

パッキングされたプログラム全てを汎用アンパッキングシステムの対象とはせず、前段でパッカー特定手法 [10] により、パッカーを特定する。既知のパッカーで、かつ、そのパッカーに対応するアンパッカーがあれば、それでアンパッキングをする。一般的に、あるパッカー専用に作られたアンパッカーの方が、汎用アンパッキングシステムよりも精度よく OEP を特定できるためである。

汎用アンパッキンシステムでは、より早く OEP 特定をするため、データ実行防止の方法を利用する。同方法で単に書込／実行された命令のアドレスを出力するだけでなく、次に述べる 2 つの機能により、OEP 候補に優先順位をつけて出力する。1 つめの機能ではパッキングされたプログラムを実行した後、書込／実行された命令のうち、アンパッキングルーチンを構成する命令を識別する。**2.1** で説明したことから分かるように、アンパッキングルーチンは書込／実行される命令を生み出すルーチンである。そこで、書込／実行される命令を生み出した親命令を、データ実行防止により検出して、アンパッキングルーチンの命令とする。そして、アンパッキングルーチンの直後に実行された命令のアドレスを OEP の最有力候補とする。2 つめの機能では、最有力候補のアドレスに実行順が近いアドレスほど、より OEP らしいアドレスとして扱う。この考えのもと、書込／実行された全ての命令のアドレスをソートして出力する。

図 3 に OEP の最有力候補を特定するためのケーススタディを示す。ケース 1 ではメモリページ E の命令が F にデータを書き込んだ後、G にも書き込みをしている。次に、F 上の命令が実行されていることから、E 上の全ての命令はアンパッキングルーチンと判定する。その後、F 上の命令が E に書き込んでいる。このとき、アンパッキングルーチンと判定されたメモリページに対して書き込みを行ったページもアンパッキングルーチンとして判定する。これはアンパッキングルーチン同士でデータを共有しているものと考えられるためである。最後に、G 上の命令が実行される。

*5　バイナリデータを命令に変換する処理

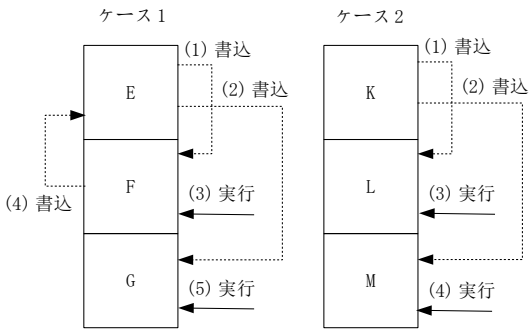

図 3　OEP 特定のケーススタディ (E-G、K-M: メモリページ)

ここで、E と F はアンパッキングルーチンとして判定されているため、ステップ (1) ～ (4) はアンパッキングルーチンの命令と判定される。そのため、その次に実行されているステップ (5) の命令のアドレスを OEP の最有力候補とする。

ケース 2 では K 上の命令が L と M に書き込み、L 上の命令が実行されるため、K をアンパッキングルーチンとする。ケース 2 では K のみがアンパッキングルーチンと判定され、K 上の最後の命令の直後に実行された命令 (ステップ (3)) のアドレスが OEP の最有力候補になる。

3.2　汎用アンパッキングのアルゴリズム

メモリページの種類を 2 つ定義する。ある命令 A が任意の場所にデータ*6 を書き込み、書き込まれたページ上のいずれかの命令が実行されたとする。このとき、命令 A が属するメモリページを“コード生成ページ”として定義する。次に、ある命令 B がコード生成ページにデータを書き込んだとする。このとき、命令 B が属するメモリページを“データ共有ページ”とする。図 3 ケース 1 では、E がコード生成ページで F がデータ共有ページとなる。図 3 ケース 2 では、K がコード生成ページとなる。コード生成ページとデータ共有ページは、**3.1** で述べた理由からアンパッキングルーチンとして判定する。

メモリページの種類を判定するため、パッキングされたプログラムを実行してから次の処理を行う。書き込み禁止の例外が発生するたびに書き込みを行った命令のアドレス (以下、src) と書き込み先のアドレス (以下、dest) の組を配列 **W** に保存していく。並行して、実行禁止の例外が発生したときに、実行された命令のアドレスを配列 **X** に保存していく。これらの操作をパッキングされたプログラムが終了するまで、もしくは、あらかじめ設定しておいたタイムアウト時間が経過するまで続ける。次に、**W** の各組に対して、dest が属するメモリページが、**X** のいずれかのアドレスが属するメモリページと一致するかどうかを確認する。もし含まれていたらその dest の組となっている src が属するメモリページをコード生成ページとする。その後、**W** の各組に対して、dest がコード生成ページにデータを書き込んでいるか確認し、書き込んでいれば src が属するメモリページをデータ共有ページとする。次に、**X** の各アドレスに対して、コード生成ページかデータ共有ページに属しているかを確認し、属していればアンパッキングルーチンのアドレスとする。最後に、**X** のアドレスと **W** の全ての src を併せて実行順にソートし、アンパッキングルーチンの最後のアドレスを探し、その直後にある **X** のアドレスを OEP の最有力候補とする。

OEP 候補の優先順位を付けるため、**X** の全てのアドレスを実行順にソートし、これを $\mathbf{X}=(x_0,x_1,\cdots,x_i,\cdots,x_p,\cdots,x_{l-1})$ とする。ここで、x_p は OEP の最有力候補のアドレス、p はそのインデックス、x_{l-1} は最後に実行された命令のアドレス、l は **X** のアドレス数、$i=0,1,2,\cdots,l-1$ である。次に、以下の式に従い **X** をソートする。

$$j=\begin{cases}0 & (i=p)\\ 2|p-i|-(1+sign(p-i))/2 & (\text{otherwise})\end{cases} \quad (1)$$

ここで、j は x_i のソート後のインデックス、$|\cdot|$ は絶対値、$sign(\cdot)$ は入力が負の値なら −1 を返し、それ以外なら 1 を返す関数である。つまり、**X** は $(x_p,x_{p-1},x_{p+1},x_{p-2},x_{p-2},\cdots)$ にソートされる。例えば、**X** が $(x_0,x_1,x_2,x_3,x_4,x_5,x_6)$ で、$p=2$ とすると、ソート後は $(x_2,x_1,x_3,x_0,x_4,x_5,x_6)$ となる。

図 3 ケース 1 では、ステップ (5) の命令のアドレスが OEP の最有力候補になる。これはステップ (4) がアンパッキングルーチンの最後のアドレスであり、その直後に実行された命令がステップ (5) のためである。図 3 ケース 2 ではステップ (3) の命令のアドレスが OEP の最有力候補になる。

4　評価実験

4.1　データセットと実験環境

実験に用いるマルウェア 35 検体を以下の方法で選択した。我々が過去数年間で収集したマルウェア数十万検体をシマンテックの『アンチウイルスソフト』で分類すると 135 カテゴリができた。例えば、“Backdoor.IRC.Bot”や“Unknown”などである。ここから各カテゴリのマルウェア数が多い順に 35 カテゴリを選択する。このとき、Unknown は除いた。その

*6　データの書き込み時点では、命令なのか、プログラミングでいう変数の値なのかは区別が難しいため、総称してデータと呼ぶ。

次に、各カテゴリの検体がパッキングされているか否かを検査した。この検査にはPEiDとLydaらの手法[19]を用いた。PEiDはシグネチャによるパッカー判定ツールである。Lydaらの手法はエントロピを用いてパッキングの有無を判定する手法である。その後、パッキングされていないと判定された検体を各カテゴリから無作為に1つ選択した。これで計35検体になる。これらの検体に対しSHA256[20]のハッシュ値を求め、ハッシュ値の重複がないことを確認した。次に、25個のパッカーで35検体をパッキングしてから、動作しないものを破棄した。パッキングされた検体は753個となり、これらを実験の対象とした。表1に、パッキングに使用したパッカー名を載せている。なお、パッキングされていない検体を用いる理由はOEPを確実に得るためである。

実験環境として、Windows XPをVirtualBox[21]のゲストOSとしてインストールした。このゲストOSの上で、各検体をアンパッキングし、汎用アンパッキングシステムの精度を検証する。なお、**3.1**の基本アイデアで述べたパッカー特定は使用せず、**3.2**のアルゴリズムのみでアンパッキングを行った。

4.2　実験結果

データ実行防止により、書込／実行された命令のアドレス数の平均と標準偏差を表1の4列めと5列めにそれぞれ記載している。例えば、ASPack 2.33でパッキングされた検体は35個あり、ASPack 2.33は平均で9.23個の書込／実行された命令を生み出したことが分かる。このときの標準偏差は5.14である。

我々の汎用アンパッキングシステムにより、書込／実行された命令のアドレスをOEPの有力である順にソートする。このとき、OEP候補のn番め以内にOEPが含まれていたとき、アンパッキングは成功したものとする。各パッカーの検体に対して、nを変化

表1　実験結果（'#'は検体数を示し、'—'は100%を示す）

	パッカー		OEP候補数		Recall（%）						
No.	名称	#	平均	標準偏差	n = 1	2	4	6	8	16	32
1	ASPack 2.33	35	9.23	5.14	94	97	100	–	–	–	–
2	ASProtect 1.70	35	67.20	12.66	37	40	43	43	43	43	86
3	exe32 pack 1.42 trial	10	7.00	2.86	90	100	–	–	–	–	–
4	Exe Stealth 2.73 trial	35	9.43	6.23	97	97	97	100	–	–	–
5	Ezip 1.0	35	10.00	6.03	94	97	97	100	–	–	–
6	FSG 2.0	34	8.76	5.37	94	97	100	–	–	–	–
7	Mew11 SE 1.2	33	11.42	7.50	94	97	97	100	–	–	–
8	MoleBoxPro 2.6.4 trial	35	23.29	5.16	0	3	6	37	97	100	–
9	mpress 2.19	31	9.87	4.46	94	97	100	–	–	–	–
10	nPack 1.1.300	33	9.39	5.27	97	100	–	–	–	–	–
11	NsPack 3.7 trial	34	10.00	6.15	94	97	97	100	–	–	–
12	Packman 1.0	35	9.29	5.08	97	100	–	–	–	–	–
13	PECompact 2.79 trial	34	10.94	6.05	94	97	97	100	–	–	–
14	PESpin 1.33	34	13.12	5.51	94	97	97	97	97	97	100
15	Petite 1.4	18	11.56	7.10	56	61	94	100	–	–	–
16	PKLITE32 1.1	14	9.57	5.43	14	100	–	–	–	–	–
17	RLPack 1.20	34	10.00	5.20	94	97	100	–	–	–	–
18	SimplePack 1.0	33	10.00	6.14	94	97	97	100	–	–	–
19	tElock 0.99	19	8.42	5.32	95	100	–	–	–	–	–
20	Themida 2.2.7.0	32	300.47	16.57	0	0	3	44	75	97	97
21	Upack 0.399	26	11.73	6.18	88	92	96	100	–	–	–
22	UPX 3.08	34	10.09	6.05	94	97	97	100	–	–	–
23	WinUpack 0.31	33	10.21	6.17	94	97	97	100	–	–	–
24	WWPack32 1.20 trial	22	9.18	4.41	95	100	–	–	–	–	–
25	yoda's protector 1.02	35	14.71	5.58	97	97	97	97	97	100	–
合計		753		平均	81	85	87	92	96	97	99

させながら、アンパッキングに成功した検体数の割合を調べた。この評価指標は Recall[22][23] と呼ばれ、各パッカーに対する Recall の値を表 1 に載せる。例えば、ASPack 2.33 の $n = 2$ であれば、Recall の値が 97 % となっている。これは 34 検体（検体数 # の 97 %）に対してアンパッキングに成功したことを意味している。

表 1 の $n = 1$ を見ると、19 個のパッカーに対して、90 % 以上の割合でアンパッキングに成功している。$n = 1$ のため、これらに対しては一意に OEP を特定できたことを意味している。$n = 8$ を見ると 23 個のパッカーに対しては 97 % の割合でアンパッキングに成功している。つまり、ソートされたアドレスの 8 番めまでを調べれば OEP にたどりつける。他のシステムとの比較については文献 [13] を参照されたい。また、ASProtect 1.70 に対する Recall を改善する方法も文献 [13] で提案している。

表 2　汎用アンパッキングシステムのオーバーヘッド

パッカー（検体のパッキングに使用）	サイズ（KB）	システムの適用無しの終了時間（msec）	システムの適用有無の時間差（msec）	（%）
ASPack	29	103	5	4.9
ASProtect	176	153	5	3.3
exe32 pack	30	103	-3	-2.9
Exe Stealth	69	99	13	13.1
Ezip	69	101	4	4.0
FSG	24	100	-1	-1.0
Mew11 SE	24	102	5	4.9
MoleBoxPro	94	119	12	10.1
mpress	26	109	-1	-0.9
nPack	29	98	9	9.2
NsPack	25	100	12	12.0
Packman	24	94	9	9.6
PECompact	26	107	-2	-1.9
PESpin	47	130	3	2.3
Petite	31	99	4	4.0
PKLITE32	111	36	6	16.7
RLPack	24	97	4	4.1
SimplePack	25	98	0	0.0
tElock	38	107	5	4.7
Themida	1184	1220	41	3.4
Upack	22	107	5	4.7
UPX	26	93	8	8.6
WinUpack	22	103	12	11.7
WWPack32	36	103	5	4.9
yoda's protector	44	6399	33	0.5

本システムがどの程度のパフォーマンスオーバーヘッドを生み出すか計測した。計測では、Intel Core i7 3.4 GHz の CPU を搭載した PC を用いた。この上で、汎用アンパッキングシステムを使用しない状態で検体を実行して、プロセスの起動から終了までの時間を計測した。これを 5 回繰り返して平均を求めた。次に、汎用アンパッキングシステムを使用した状態で検体を実行し、プロセスの起動から終了までの時間を計測した。こちらも 5 回の平均を求めた。この検体には Trojan.Panddos を選び、唯一 PKLITE32 のみ、この検体のパッキングに失敗していたため、Trojan.Usugelgen3 を代わりに使用した。

表 2 の 3 列めの「システムの適用無しの終了時間（以下、オリジナル時間）」はシステムを使用しない状態で計測した時間を示している。4 列めの「システムの適用有無の時間差」は、システム有りで計測した時間とオリジナル時間との差を示しており、5 列めは時間が何パーセント増加したかを示している。表から全体的に汎用アンパッキングシステムが与える時間が非常に小さいことが分かる。特に、exe32 pack の他、いくつかのパッカーにおいてシステムを適用したときの方がプロセス終了までの時間が短い。

プロセス終了までの時間の差が最も大きかったのは Themida に対してであり、41 ミリ秒増加している。これは Themida のアンパッキングルーチンが非常に頻繁に発生させるためであり、書込／実行された命令の抽出に 27 ミリ秒かかっている。残りの 14 ミリ秒は書込／実行された命令をソートする時間（OEP の最有力候補を求める時間含む）である。他のパッカーに対しては書込／実行された命令をソートする時間は 2 ミリ秒以下で完了している。汎用アンパッキンシステムが与えるオーバーヘッドは非常に小さいといえる。最後に、全 753 検体に対してシステムを適用したときのプロセス終了までの時間の合計は 1061 秒で、1 検体あたり平均 1.41 秒だった。解析者は、1 検体を解析する際、1〜2 秒ほど待てば OEP の候補が得られることが分かる。

5　むすび

本稿では、我々が研究開発してきた汎用アンパッキングシステムについて概説した。出現するマルウェアの数は増加の一途をたどっているため、マルウェア解析の効率化は急務である。汎用アンパッキングシステムを用いることで、解析の大幅な効率化が可能となる。

今後もマルウェアの脅威に対抗すべく、我々は研究を遂行していく。単に解析の効率化だけでなく、マル

ウェアの収集、解析、対策の導出に関して逐次精査し、解決すべき課題に取り組んでいく。なお、本稿で説明した内容、成果は主に文献 [10]（パッカー特定）及び文献 [11]（汎用アンパッキング）で発表している。

【参考文献】

1 M. F. Oberhumer, L. Molnar, and J. F. Reiser, “UPX: Ultimate Packer for eXecutables,” available at http://upx.sourceforge.net/, (Last access: April 21st, 2016).
2 Oreans Technologies, “Themida,” available at http://www.oreans.com/themida.php, (Last access: April 21st, 2016).
3 P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, “PolyUnpack: Automating the Hidden-Code Extraction of Unpack- Executing Malware,” Proceedings of the 22nd Annual Computer Se- curity Applications Conference (ACSAC’06), pp.289–300, 2006.
4 M.G. Kang, P. Poosankam, and H. Yin, “Renovo: A hidden code extractor for packed executables,” Proceedings of the 5th ACM workshop on Recurring Malcode (WORM’07), New York, NY, USA, pp.46–53, ACM, 2007.
5 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 (CCS’08), New York, NY, USA, pp.51–62, ACM, 2008.
6 Y. Kawakoya, M. Iwamura, and M. Itoh, “Memory behavior-based automatic malware unpacking in stealth debugging environment,” Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE’10), pp.39–46, 2010.
7 H.C. Kim, T. Orii, K. Yoshioka, D. Inoue, J. Song, M. Eto, J. Shikata, T. Matsumoto, and K. Nakao, “An Empirical Evaluation of an Unpacking Method Implemented with Dynamic Binary Instrumentation,” IEICE Trans. Inf. & Syst., vol.94-D, no.9, pp.1778–1791, 2011.
8 L. Martignoni, M. Christodorescu, and S. Jha, “OmniUnpack: Fast, Generic, and Safe Unpacking of Malware,” Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC’07), pp.431–441, 2007.
9 F. Guo, P. Ferrie, and T.C. Chiueh, “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, pp.98–115, Springer-Verlag, 2008.
10 R. Isawa, T. Ban, S. Guo, D. Inoue, and K. Nakao, “An Accurate Packer Identification Method using Support Vector Machine,” IEICE Trans. Fundamentals, vol.E97-A, no.1, pp.253–263, Jan. 2014.
11 R. Isawa, D. Inoue, and K. Nakao, “An Original Entry Point Detection Method with Candidate-Sorting for More Effective Generic Unpacking,” IEICE Trans. on Info. & Syst., vol.E98-D, no.4, pp.883–893, April 2015.
12 StarForce Technologies Ltd., “ASPack Software,” http://www.aspack.com/asprotect32.html, (Last access: April 21st, 2016).
13 XenProject, “TheXenProject,” available at http://www.xenproject.org/, (Last access: April 21st, 2016).
14 R.H.O.S. Community, “Kvm: Kernel-based virtual machine,” available at http://www.linux- kvm.org/, (Last access: April 21st, 2016).
15 F. Bellard, “QEMU.” http://www.qemu.org/, (Last access: April 21st, 2016).
16 Intel Corporation, “Pin - a dynamic binary instrumentation tool,” available at http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, (Last access: April 21st, 2016).
17 N. Nethercote and J. Seward, “Valgrind: A framework for heavy-weight dynamic binary instrumentation,” SIGPLAN Not., vol.42, no.6, pp.89–100, June 2007.
18 Intel Corporation, “Intel 64 and ia-32 architectures software developer's manual,” available at http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, 2014.
19 R. Lyda and J. Hamrock, “Using entropy analysis to find encrypted and packed malware,” IEEE Security and Privacy, vol.5, no.2, pp.40–45, March 2007.
20 P.G. John Bryson, “Secure hash standard (shs) (federal information processing standards publication 180-4),” available at http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, 2012.
21 Oracle, “Oracle VM VirtualBox,” available at https://www.virtualbox.org/, (Last access: April 21st., 2016).
22 J. Han, M. Kamber, and J. Pei, “Data Mining: Concepts and Techniques,” Third Edition, Morgan Kaufmann, 2011.
23 B. Croft, D. Metzler, and T. Strohman, “Search Engines: Information Retrieval in Practice,” 2009.

伊沢亮一　（いさわ　りょういち）

サイバーセキュリティ研究所
サイバーセキュリティ研究室
主任研究員
博士（工学）
マルウェア解析、ネットワークセキュリティ

班　涛　（ばん　とう）

サイバーセキュリティ研究所
サイバーセキュリティ研究室
主任研究員
博士（工学）
機械学習、ネットワークセキュリティ