Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

動的エミュレーションと静的解析を併用したバイナリコードの解析手法

泉田大宗[†1]　橋本政朋[†1]　森彰[†1]

本稿では軽量の動的エミュレーションと静的解析を組み合わせたマルウェアの振る舞い解析を自動化する手法について述べる。動的解析により実行時の情報を得るとともに、エミュレーションでは実行されなかった制御フローを静的解析を用いて調べることができる。　静的解析ではバイナリコードから制御フローグラフ (CFG) を作成するが、間接ジャンプ/コールの行き先アドレスのような未解決な行き先に対し記号的評価を行い、 定数値に帰結させることで順次 CFG を拡大する手法を用いている。記号的評価手法は静的単一代入 (SSA) 形式に変換した上で定数伝播とメモリアクセス解析を行っている。 実際のマルウェアサンプルを用いた実験を行い、本手法の有効性を示した。

A Method for Analyzing Malware using Lightweight Emulation and Static Analysis of Binary Code

Tomonori IZUMIDA,[†1] Masatomo HASHIMOTO[†1] and Akira MORI[†1]

We present an automated method for analyzing malware behaviors based on a combination of lightweight emulation and static analysis of binary executables. The emulator gathers run-time information while static analysis reveals control flows not explored by the emulator. The static analyzer incrementally converts binary code into static single assignment (SSA) form in which indeterminate values such as destination addresses of indirect jumps/calls and return addresses are symbolically evaluated for further conversion by constant propagation and by memory read/write tracing. The results of experiments conducted on malware samples collected in a real environment are presented to demonstrate the capability of the method.

1. 背　　景

近年コンピュータウィルスなどのマルウェアによる被害が深刻化しつつあり、日々大量にインターネット上で流れるバイナリコードに対し、マルウェアであるかを判別するツールの需要が増している。現在多くの場面で用いられている検知ツールは、既知のマルウェアから採取したバイナリパターンを用いて迅速な判別を行っているが、全く未知のマルウェアの場合には対処が遅れてしまうという問題が存在する。そのため未知のバイナリコードを解析して、悪意のある振る舞いを行うマルウェアであるかを判別するための手法の研究が必要とされている。

未知のマルウェアの解析方法には大別して、動的な手法と静的な手法が存在する。動的な手法では、実際の実行環境を模擬した仮想サンドボックス環境を用意し、その中でバイナリを実行させて、振る舞いを観察する。仮想環境を実環境に近づければ、より検出の精度を上げることができるが、完全なオペレーションシステムを用意し、実行を繰り返す度に環境を元に戻す必要があるため、オーバーヘッドが大きい。また、すべての実行パスを網羅的に検査できるわけではないので、悪意のある振る舞いを行うようなバイナリを見逃してしまう可能性がある。

一方で静的手法は、バイナリコードを逆アセンブルして制御フローグラフ (CFG) を構成し、得られたグラフを用いて抽象解釈などの手法を用いて網羅的な検査を行うことができる。しかし、そのためにはバイナリコードの完全な逆アセンブルが可能でなくてはならないが、間接ジャンプの行き先を静的に解決することは、決定不能問題を含むために一般には困難であり、なんらかのヒューリスティックを必要としている。また、実際のマルウェアの中にはこうした解析を防ぐために、逆アセンブルを行いにくくするための難読化処理を施していたり、プログラム本体を暗号化しているものもあるため、静的手法のみでのマルウェア検知には自ずと限界が存在する。

本稿では、軽量の実行環境エミュレータを用いた動的解析ツールと、静的単一代入形式を用いた静的解析ツールを組み合わせたマルウェア解析手法について説明を行う。筆者ら[8),9)]による Alligator は、Intel IA32 CPU シミュレータと PE 形式ファイルのローダ、Win32 API の軽微なエミュレーションから構成される実行環境エミュレータである。通常の仮想

†1 独立行政法人 産業技術総合研究所
National Institute of Advanced Industry and Technology

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

実行環境とは異なりフルサイズの OS を用意する必要は無く、エミュレーション中の対象プログラムの API 呼び出しを捕捉し、代わりに簡易な OS のエミュレーションを行うことで、仮想的な実行を可能にする。Alligator では、各 API 呼び出しや、予め定めた不正なアドレスへのアクセスなどのイベントを監視することができ、それらのイベントが特定の振る舞いパターンを持つかどうかを以て、悪意のあるプログラムであるかを判定する。悪意のある振る舞いパターンの定義集合を変更することで、様々な種類のセキュリティポリシーに基づく判定を下すことができる。Alligator の行う Win32 API のエミュレーションは簡易なため、その返り値などの値に従って制御を変更するようなプログラムでは、本来観察対象となる悪意のある振る舞いを実行するとは限らない。そのため、事前に静的解析を行い、各分岐命令において、どの分岐を選択すれば、より適切な観察が可能になるかを調査し、エミュレーションが分岐命令にさしかかった時に、その時の実行文脈に関わらずに適切な分岐を行うことで、判定精度を上げている。実際に、Alligator は 2004 年から 2006 年の間に流行したコンピュータウィルスやインターネットワームの 95%を判定することができた。

Alligator で行っていた静的解析手法は単純なものであったが、近年のマルウェアの多様化に対応するために、新たな静的解析手法の研究を行った。本稿で説明する Syman は、動的エミュレータと協調しながら、実行中の命令アドレスからの制御フローを静的に解析可能な限り調査を行うツールである。難読化が施されているマルウェアに対応するために、各機械命令をより低レベルの中間表現に分解し、その上で記号的な評価を行うことで、間接ジャンプの行き先などを可能な限り静的に解決している。静的に解決のできない制御フローに関しては、Alligator を実行して動的に実行時情報を収集し、新たに判明した制御フローに対して、繰り返し Syman で解析を行うことで、反復的にプログラム全体の制御フローグラフ (CFG) を得る事ができる。例として、プログラム本体が暗号化されているようなマルウェアの場合、まず復号ルーチン部の静的解析が行われるが、復号後の本体部分に関しては解析できていないか、誤った解釈での解析が行われている。この場合、Syman は Alligator に制御を移し、エミュレーションで復号ルーチンを実行した後、復号されたプログラムの実行に入る前に再び Syman に制御が戻り、復号後の制御の解析を開始する。このように動的手法と静的手法を組み合わせる事で、年々複雑化するマルウェアにも対応可能な解析手法を考案した。

また解析の結果、最終的に得られた CFG に対して、派生分類学的手法を用いてマルウェアの自動分類を行う研究も行っている。

以降では、2 章で本解析手法の概説と、特に静的解析ツール Syman について説明を行い、3 で静的解析で用いる記号的評価法について概要を述べる。4 章で実在のマルウェアサンプルを用いた実験を元に、本手法の有用性について議論し、5 章で今後の課題などを述べる。

2. 解析手法の概説

動的な解析手法では、分岐命令によって選択されなかったの行き先のように、全ての制御フローを把握できるわけではない。一方で、静的な解析手法では間接ジャンプの行き先アドレスを完全に決定したり、暗号化されたプログラム部分まで解析を行うことは不可能である。本手法では、動的手法と静的手法を組み合わせて両者の難点を相補する試みを行っている。二つの手法を組み合わせるためには、如何にしてそれぞれの制御を切り替えるかが問題となる。以下に本手法での処理の流れを概説する。まず、解析対象となる実行ファイルをエミュレータの仮想メモリに読み込み、仮想実行環境の準備を行ったのち、開始エントリアドレスから静的解析を行う。この静的解析で得られた CFG を参照しながら、エミュレータを一命令ずつステップ実行を行う。常に現在の命令アドレスを監視し、もし次の述べる事象が起これば、その時点の実行コンテキストをもとに、再び静的解析を行った後、新たに得られた CFG を参照してステップ実行に戻る。

- 命令アドレスが CFG で想定したアドレスから逸脱した場合。これは静的解析では行き先アドレスが解決できなかった、もしくは、誤ったアドレスに解決してしまった場合に起こりうる。例えば、Win32 API の呼び出しアドレスを GetProcAddress 関数を用いて順次取得してテーブルに納めた後、そのテーブルを参照して API 呼び出しを行うような場合、こうしたアドレスを静的に解決する事は困難なので、静的解析の段階では行き先不明と判定されていることが多い。
- エミュレーション中に書き換えられた可能性のあるメモリ領域の命令を実行しようとしている場合。命令アドレス自体は静的解析で得られたアドレスと一致しているが、静的解析後のエミュレーションによって、命令自体が書き換えられてしまっている可能性があるときは、静的解析は間違った命令列を解釈してしまっていると考えられる。この判定を行うために、当手法では IA32 アーキテクチャのメモリページ管理機構を利用している。静的解析が完了し、エミュレーションに制御を渡す前に、各ページのダーティビットをクリアしておく。エミュレーション中はステップ実行ごとに命令アドレスを含むページのダーティビットを確認し、ダーティビットがセットされていた時は、なんらかの自己書き換えがあったものと判断する。ページ単位の判定なので厳密ではないが、実用的には有効である。

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

2.1 バイナリコードの静的解析

バイナリコードを静的に解析するにはいくつかの難点が存在する。それらは対象がバイナリコードであるが故の一般的な問題と、マルウェアのようなプログラムを対象にしているために起こる問題がある。

2.1.1 バイナリコードを静的解析する難しさ

バイナリコードの場合、間接ジャンプ/コール命令のように、制御フローの行き先アドレスが直接的にプログラムに記述されているとは限らない。そのため、行き先アドレスを決定するために、値解析が必要となる。しかし、バイナリコードの値のフローはレジスタだけではなく、メモリに対する読み書きも含むため、メモリ上のデータの解析手法も必要になる。こうした値解析には CFG が必要であるが、その CFG を構築するために値解析が必要になるというジレンマが生じる。

2.1.2 マルウェアを静的解析する難しさ

マルウェアの中には C/C++言語といった高級言語を用いず、直接アセンブラ言語で記述されているため、関数やサブルーチンといった構造を持たないものがある。さらには IA32 の機械命令には関数呼び出しやスタックなど構造化されたプログラムを支援するための機構が導入されているが、こうした機械命令が本来の意図とは異なる使用が行われる場合もある。例えば、`CALL` 命令は次の命令アドレスを復帰アドレスとしてスタックに積んだ後にジャンプを行うが、ジャンプ先でスタックから復帰アドレスを取り出し、プログラムが実際にロードされたアドレスを取得するような使用例があり、`RET` 命令はスタックの先頭に積まれたデータを復帰アドレスと看做して、そのアドレスにジャンプするが、いくつかのマルウェアでは、スタックを操作する事で単なる間接ジャンプ命令として使う事例も存在する。すなわち、静的解析中に `CALL` 命令や `RET` 命令が現れたとしても、それをもってサブルーチンと判断する事はできない。また、スタック操作自体も必ずしも `PUSH` 命令や `POP` 命令を使わずに、直接 `ESP` レジスタの値からスタック上のメモリを書き換えるようなプログラムもあるため、静的解析を抽象的なデータ構造としてスタックを用いる事は適切ではない。

2.1.3 Syman の静的解析手法

Syman の静的解析手法は、以上の観点をふまえた上で、IA32 の各命令をより低レベルな代入命令の記号式の列に解釈して解析を行う**図 1** に記号式の定義を、**図 2** に解釈の例を示す。変数には値変数とメモリ状態変数の二種類が存在する。値変数は値のビットサイズで型付けされ、添字の数字で表している (図では 32 ビット)。ビットサイズが自明の場合、この添字を省略する。*src* や *dest* といった変数は、一つの機械命令を解釈するにあたり便宜上導

$$
\begin{aligned}
\mathrm{expr}_{sz} \ :: & = \mathrm{int}_{sz} \mid \mathrm{var}_{sz} \mid \mathrm{expr}_{sz} + \mathrm{expr}_{sz} \mid \mathrm{expr}_{sz} - \mathrm{expr}_{sz} \mid \mathrm{expr}_{sz} * \mathrm{expr}_{sz} \ldots \\
& \quad \mid \mathrm{expr}_{sz} \& \mathrm{expr}_{sz} \mid \mathrm{expr}_{sz} \| \mathrm{expr}_{sz} \mid \sim \mathrm{expr}_{sz} \ldots \\
& \quad \mid \mathrm{Ld}_{sz}(M, expr_{32}) \\
\mathrm{ControlExpr} \ :: & = \mathrm{Jump}(\mathrm{expr}_{32}) \mid \mathrm{Call}(\mathrm{expr}_{32}) \mid \mathrm{Ret}(\mathrm{expr}_{32}) \mid \mathrm{Branch}(\mathrm{expr}_1, \mathrm{expr}_{32}, \mathrm{expr}_{32}) \ldots \\
\mathrm{MemoryExpr} \ :: & = M \mid \mathrm{St}(M, \mathrm{expr}_{32}, \mathrm{expr}_{sz})
\end{aligned}
$$

図 1 記号式
Fig. 1 Definition of Symbolic Expressions

IA32 命令	解釈
`ADD EAX, 1234[EDX,ESI]`	$src_{32} \leftarrow \mathrm{Ld}_{32}(\mathrm{M}, \mathrm{EDX}_{32} + \mathrm{ESI}_{32} + 1234)$ $\mathrm{EAX}_{32} \leftarrow \mathrm{EAX}_{32} + src_{32}$
`CALL EBX`	$dest_{32} \leftarrow \mathrm{EBX}_{32}$ $\mathrm{ESP}_{32} \leftarrow \mathrm{ESP}_{32} - 4$ $\mathrm{M} \leftarrow \mathrm{St}_{32}(\mathrm{M}, \mathrm{ESP}_{32}, NextIP)$ $\leftarrow \mathrm{Call}(dest_{32})$
`RET`	$\mathrm{ESP}_{32} \leftarrow \mathrm{ESP}_{32} + 4$ $dest_{32} \leftarrow \mathrm{Ld}_{32}(\mathrm{M}, \mathrm{ESP}_{32})$ $\leftarrow \mathrm{Ret}(dest_{32})$

図 2 機械命令の解釈の例
Fig. 2 Example: Translations of Instructions

入される一時的変数であり、別の命令の解釈からは参照されることはない。メモリアクセスは $var_{sz} \leftarrow \mathrm{Ld}_{sz}(M, addr_{32})$ と、$M \leftarrow \mathrm{St}_{sz}(M, addr_{32}, expr_{sz})$ という二つの代入文で記述される。前者は現在のメモリ状態 M でアドレス $addr$ から sz ビットの値を読み出して変数 var に代入する事を表し、後者は現在のメモリ状態 M に対してアドレス $addr$ に sz ビットの値 $expr$ を書き込んで、メモリ状態を更新することを表している。**制御文**は制御フローを記述する文で、左辺が無く、右辺には制御の行き先を示す制御式が置かれた代入文で表される。制御式は可読性を高めるために、疑似関数を用いて $\mathrm{Jump}(\mathrm{expr}_{32})$ や $\mathrm{Call}(\mathrm{expr}_{32})$ などと記述されるが、Syman ではこうした疑似関数は特別な意味を持たず、単に行き先アドレスを表す式を示すに過ぎない。

Syman は、指定された命令アドレスから順次機械命令を読み込み、代入文へ変換して行く。制御文が現れるまで読み込んだ代入文の列を**基本ブロック**と呼ぶ。基本ブロックの制御文の示す行き先アドレスが定数値の場合、すなわち即値ジャンプ/コールである場合は、続けてそのアドレスから基本ブロックを読み込んで行く。行き先アドレスが定数値でない場合は、未決定であることを記録しておく。即値アドレスを可能な限り辿って CFG を作成した後、未決定である行き先を、その CFG を用いた記号的評価を行って、即値アドレスに解決できるか試みる。即値アドレスに解決できた場合は、そこからまた基本ブロックを読み込ん

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

で行くことで CFG を拡大する。このように、1) 即値アドレスを辿って可能な限り CFG を作成し、2) 未解決の行き先アドレスを、その CFG を用いて記号的評価を行い、即値アドレスへの解決を試みるというプロセスを反復的に適用して、CFG を可能な限り拡大して行く。本方式で用いる記号的評価方法については、3 節で述べる。

3. 記号的評価方法

Syman では CFG を作成する際に未決定の飛び先アドレスを定数値アドレスに帰結させるため、単一静的代入 (Static Single Assignment, SSA) 形式[5] を用いた需要主導型定数伝播 (demand-driven constant propagation)[14] を用いた記号的評価を行う。SSA 形式とは、コンパイラ設計で用いられる中間表現であり、各変数が一度のみ代入文で定義されるように CFG を変換したものである。CFG を SSA 形式に変換する時、制御フローが合流する地点で、ある変数 v の定義が複数存在する場合には、Φ **関数**と呼ばれる疑似関数を用いて $v \leftarrow \Phi(v, \ldots, v)$ という代入文 (以降では Φ **文**と呼ぶ) を挿入することで、変数の定義を一つにまとめる。

CFG のどのノードに、どの変数のための Φ 文を挿入するかを決定するために支配辺境 (dominance frontier) を用いたアルゴリズムが知られている。適切に Φ 文が挿入された後に、各代入文ごとに変数の名前を添字を加えるなどして変更することで SSA 形式が得られる。SSA 形式は、変数の定義が一意に定まり、それゆえ use-def 鎖が明示的であるため、定数伝播などコンパイラの最適化の技術が適用されやすい。

3.1 需要主導型定数伝播

本手法では、需要主導型の定数伝播を用いて記号式を定数式に帰結させる。需要主導型の定数伝播とは、記号式中に現れる変数を随時その定義式で置き換えていくことで、最終的に (可能であれば) 定数値に帰結させる手法である。

例として、$\mathrm{ESP}_{42} \leftarrow \mathrm{ESP}_{41} + 4; \mathrm{ESP}_{43} \leftarrow \mathrm{ESP}_{42} - 4;$ という定義となっていた時に式 $\mathrm{ESP}_{43} - 4$ を評価すると、まず変数 ESP_{43} が、その定義式である $\mathrm{ESP}_{42} - 4$ に置き換えられ、$\mathrm{ESP}_{42} - 8$[⋆1] を得る。次に変数 ESP_{42} を定義 $\mathrm{ESP}_{41} + 4$ で置き換える事で、$\mathrm{ESP}_{41} - 4$ を得る。このように順次変数を定義で置き換えていくことで評価を行う。

需要主導型の手法を採用する理由は二つある。一つは、間接ジャンプの行き先アドレスは、局所的に定まることが多いという知見を得ているため、需要主導型を用いる方が効率が良い場合が多い。第二に次節で説明するメモリ状態の遡行アルゴリズムとの相性が良いためである。

変数の定義が Φ 文である場合は、注意が必要になる。SSA 形式の中で Φ 文 $v \leftarrow \Phi(v_0, \ldots, v_n)$ が現れるのは、主に分岐で分かれた処理が合流する地点か、ループの先頭の地点である場合である。分岐の合流の場合はそれぞれ変数 $v_0, \ldots, v_n$ の評価を行えば、分岐の時点で再び合流するので、それまで時に得られた式 $expr_0, \ldots, expr_n$ から、$\Phi(expr_0, \ldots, expr_n)$ を得る。本来 Φ 関数は代数的な意味を持たないが、本手法では束代数における結合演算と同様の代数処理を行う。

ループの先頭である場合、同様に $\Phi(expr_0, \ldots, expr_n)$ を得るが、v がループ不変でない場合、式 $expr_0, \ldots, expr_n$ の中に v が含まれる。このときは便宜的に不動点演算子 μ を導入して、$\mu X.\Phi(expr_0, \ldots, expr_n)[v/X]$ とする。ただし現状では不動点演算子 μ に対する代数的な処理は行わないため、あくまで値がループ依存であることを示すにとどまる。

3.2 メモリアクセス解析

バイナリコードの静的解析の難しさに、データ値がレジスタ変数だけはなく、任意にメモリ上に読み書きされうるために、メモリアクセスも解析しなくてはならないことがある。Syman では、データを表す値変数に加えてメモリ状態を表すメモリ状態変数を導入し、メモリ状態の変更を制御フローとは逆向きに遡ることで、メモリアクセスの解析を行っている。

評価対象の記号式にメモリ状態 M のアドレス $addr$ から読み込んだ値を表す $\mathrm{Ld}(M, addr)$ という部分式が現れた場合、メモリ状態変数 M の use-def 鎖を制御フローとは逆向きに辿りながら、$addr$ と最初に一致するアドレスへのデータの書き込みを捜索する。記号式の定義によりメモリ状態変数 M の定義代入文は、$M \leftarrow \mathrm{St}(M', addr', value)$ か、$M \leftarrow \Phi(M_0, \ldots, M_n)$ のどちらかに限られる。

前者の場合は、$addr$ と $addr'$ が一致するか比較を行う。具体的には記号式 $addr - addr'$ が定数式 0 に帰結されるかを本章で説明している手法を用いて記号的に評価を行う。このときこの評価において新たなメモリアクセス解析が起こりうることに注意しなくてはならない。ループの中の値を評価している場合、アドレス比較による新たな評価スレッドが発生した場合、再び同じ値の評価を繰り返して評価が停止しない可能性がある。このため、ループの中でのメモリアクセス解析は、アドレスの比較を最も内側のループの中までしか行わない。そのために本来は一致していたはずの二つのアドレスが一致しないという判定になる場合がある。これは本手法において評価が失敗するひとつの要因である。二つのアドレスが一致した場合は値 $value$ を答えとして返す。一致しなかった場合は、メモリ状態変数 M'

⋆1 記号式に含まれる演算子については、代数的な処理を施す事ができるものとする。

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

$$
\begin{aligned}
hModule &\leftarrow \mathrm{Ld}_{32}(M, \mathrm{ESP}+4)\\
lpProcName &\leftarrow \mathrm{Ld}_{32}(M, \mathrm{ESP}+8)\\
\mathrm{EAX} &\leftarrow \mathrm{Kernel32::GetProcAddress}(hModule, lpProcName)\\
dest &\leftarrow \mathrm{Ld}_{32}(M, \mathrm{ESP})\\
\mathrm{ESP} &\leftarrow \mathrm{ESP}+12\\
&\leftarrow \mathrm{Ret}(dest)
\end{aligned}
$$

図 3 API ブロックの例
Fig. 3 An Example of API Block

の定義に遡る。

後者の場合は、前節と同様にループ不変の場合はそのまま、Φ 関数の各引数について評価を進める。ループの先頭である場合も同様に行うが、メモリアクセスの場合、ループで依存した参照なのかどうかを判定することは難しい。ループの中のアドレス比較がすべて失敗した場合は、該当するメモリ書き込みはループ内ではなかったと判断して、捜索を続ける。

最終的に初期メモリ状態までたどり着いた場合、もしそのときの *addr* の評価が定数式であったなら、実際のエミュレーションの仮想メモリから該当するアドレスのデータを読み出して返す。それ以外の場合は失敗とみなす。よって本手法のメモリアクセス解析は必ずしも成功するとは限らない。それゆえに誤った答えを返したり、失敗におわる場合がある。そのため、Syman は間違った CFG を作成する可能性がある。静的解析である以上完全な値解析は不可能なので、自ずと存在する限界ではあるが、2 章にあるように Syman の作った CFG から Alligator の実行が逸脱した時点で、新たに CFG を作ることで、この難点を補っている。

3.3 API ブロック

Alligator は Win32 API DLL をメモリに読み込む際に、その DLL で定義された関数のエントリアドレスのテーブルを管理する。上述の記号的評価により定数値に定まった行き先アドレスが、API 関数のエントリアドレスである場合には、基本ブロックを読み込む代わりに API 呼び出しを抽象化した *API* **ブロック**を CFG に加える。

図 3 は GetProcAddress 関数の API ブロックである。前半にはスタックから引数を読みだすための命令が並んでいる。またブロックの最後には Win32 API 関数の慣習に基づき、引数の数だけスタックの値を巻き戻してから、*RET* 命令を実行している。API 関数の返り値自体はその API 名を持った擬似関数で表現される。基本的にこの擬似関数が記号的評価中に代数的に扱われることはないが、いくつかの制御フローに影響を及ぼす API に関しては、特別な評価を行うことがある。この例である GetProcAddress 関数もその一つである。記号的評価の結果、行き先アドレスが GetProcAddress 関数の返り値であると判明した場合、さらに関数の引数を解析し、API 名が静的に解決された場合は、その API 関数のエントリアドレスと看做して、API ブロックを作成する。なお、この引数の解析を行うとき、関数のシグネチャからデータの型が定められる。Null 終端文字列型であった場合には、メモリアクセス解析を一バイトずつ、Null 値を返すまで反復することで、ある程度の解析が行うことができる。

4. 事例研究

本手法の有効性を示すため、いくつかの事例研究を紹介する。

4.1 難読化が施されたマルウェアに対する解析

ここでは従来の解析手法では解析が困難な難読化が施されたマルウェアサンプルに対しての解析事例を二つの例で説明する。

最初の例は Rustock[⋆1] である。Rustock では、暗号化された本体部を復号するルーチンの中の複数の場所で、RET 命令を間接ジャンプ命令として利用している。これは単純な難読化の例であるが、バイナリコード解析に利用されることが多い IDAPro のようなデバッガでも解析を行うことが難しい。**図 4** に、実際に Syman によって解析されたコード片を示す。アドレス 0040117E や 00490123B での RET 命令は、事前にスタックを操作することで、単なる間接ジャンプ命令として利用されている。Syman ではスタックや関数呼び出しなど、抽象化された概念は想定せず、あくまでメモリ操作とジャンプ命令という低レベルな解釈で解析を行うために、こうした難読化にも対応することが可能である。なお、Syman は Rustock のの暗号化された本体部を解析することができないため、復号ルーチンの解析を行った時点で終了する。当手法では、その後 Alligator を起動して、実際に復号を行った後、本体部の実行前に再び Syman による解析を行っている。

次の例は、Troj.Obfus.Gen[⋆1] である。Troj.Obfus.Gen は約 17.5KB の大きさを持つトロイの木馬系のマルウェアである。WININET.DLL などいくつかの Win32 API ライブラリを読み込んで、API を利用して、インターネット上にあるから別ファイルをダウンロードして実行を行う。このマルウェアは単純な文字列走査によって使用する API 関数名を察知されないように、文字列をプログラム中に隠蔽している。**図 5** は、実際のバイナリーコード辺であり、GetProcAddress 関数の第二引数である文字列領域に “Sleep” という API 関数名を一文

⋆1 名前はマルウェアスキャナ BitDefenderhttp://www.bitdefender.com による

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

Disassembled Code		SSA Form
401172	MOV SS:[ESP], 4015A9	$M_{106} \leftarrow \mathrm{St}_{32}(M_{105}, \mathrm{ESP}_{208}, 004015A9)$
401179	PUSH 40123B	$\mathrm{ESP}_{209} \leftarrow \mathrm{ESP}_{208} - 4$ $M_{107} \leftarrow \mathrm{St}_{32}(M_{106}, \mathrm{ESP}_{209}, 40123B)$
40117E	RET	$dest_{58} \leftarrow \mathrm{Ld}_{32}(M_{107}, \mathrm{ESP}_{209})$ $\mathrm{ESP}_{210} \leftarrow \mathrm{ESP}_{209} + 4$ $\leftarrow \mathrm{Ret}(dest_{58})$
40123B	RET	$dest_{59} \leftarrow \mathrm{Ld}_{32}(M_{107}, \mathrm{ESP}_{210})$ $\mathrm{ESP}_{211} \leftarrow \mathrm{ESP}_{210} + 4$ $\leftarrow \mathrm{Ret}(dest_{59})$
4015A9 ...	INC ESP ...	$\mathrm{ESP}_{212} \leftarrow \mathrm{ESP}_{211} - 1$ ...

図 4 難読化が施された制御の例 (Rustok)
Fig. 4 Obfuscated Control Flow (Rustok)

```
402F7C   MOV DS:1433[EBX]{8}, 'e'
402F85   MOV DS:1435[EBX]{8}, 'p'
402F92   MOV DS:1431[EBX]{8}, 'S'
402F9C   MOV DS:1432[EBX]{8}, 'l'
402FA6   MOV DS:1436[EBX]{8}, '\0'
402FB0   MOV DS:1434[EBX]{8}, 'e'
402FD2   LEA EDI, DS:1431[EBX]{8}
402FD8   MOV SS:4[ESP]{32} ,EDI
402FDC   CALL DS:1637[EBX]{32}
```

図 5 API 関数名文字列 “Sleep” の隠蔽例 (Troj.Obfus.Gen)
Fig. 5 Hiding API Name String “Sleep” (Troj.Obfus.Gen)

字づつ順不同にメモリに書き込むことで、実行ファイルの単純な走査では見つけられなくしている。Syman ではメモリ状態を遡行することで、メモリ上に読み書きされた値の解析も可能なため、静的にこのような文字列を解析することが可能である。実際に Troj.Obfus.Gen 自体は暗号化されていないため、そのプログラムコード全体の約 89%を静的手法のみで解析することに成功した。Troj.Obfus.Gen のプログラム中の間接ジャンプの中で、Syman によって行き先の解析に失敗したものはただ一つだけであった。これは、Troj.Obfus.Gen がヒープに確保したメモリ領域にインターネットから別ファイルをダウンロードして書き込んだ後に、その別ファイルにあるエントリーポイントへのジャンプであった。このようなジャンプの解決は静的な手法だけでは不可能であるので、Alligator のような動的手法と組み合わせることでより完全に近い解析が行われることが期待できる。

4.2 マルウェアの自動分類と検索

本節ではマルウェアの振る舞いの系統派生学的解析を行った結果について述べる。

まず 58 種のマルウェアサンプルに対し、本手法を適用し、マルウェアの CFG を作成した。次に、得られた CFG のうち API ブロックだけをその API 名でラベル付けし、その他の基本ブロックを空白ノードと置き換える抽象化を行った後、CFG の支配木を作成した。これは、どのような制御フローパターンで API が呼ばれるかを表す粗い抽象化である。こうして得られた支配木に対して、木構造の差分を相互に計算した。これにはプログラムソースコードの抽象文法木の比較のためのプロジェクト[7] のシステムを応用した。このシステムでは、ある木構造を持ったデータに、最小コストの変更を加え、他方の木構造に変換されるかを算定する。この算定結果へ適切に距離を導入することで、木構造の類似度を測定することができる。

図 6 は、実在の 58 種のオンラインゲームワームの相互類似度から作成された樹状図である。各サンプルは実行ファイルの MD5 ハッシュ値でラベル付けされている。

手作業での調査により、密接度の高い郡には、同じタイプのマルウェアだけが含まれていることを確認した。また、マルウェアでありながら、ウィルス検知ソフトでは検知をまぬがれたいくつかのサンプルが、同種の既知ウィルス郡に含まれていることも見つかった。このことは、未知のマルウェアの同定や分類に上記手法が有効である可能性を示しているが、今後のさらなる調査研究を必要としている。

さらに木構造の差分から得られる距離が近ければ、マルウェアの振る舞いも近いという知見は、大量のマルウェアサンプルの中から、より振る舞いが近いサンプルを見つけ出すことにも応用出来ることを示唆している。一年以上にかけて収集した 950 種のマルウェアサンプルの中から、より振る舞いの近似度が高いもの検索するシステムのプロトタイプを作成した。

あるサンプルの MD5 ハッシュ`a1273bc593` でラベルされたサンプルで検索した実行結果が **図 7** である。5 つの候補が近似度の高さに従って並べられている。各候補には `BitDefender` による同定名が併記されている。事実 `a1273bc593` も `Trojan.PWS.OnlineGames.ZWU` と同定されるため、この結果は有効なものとなっている。マルウェアの類似度によるの検索システムは、日々のマルウェアサンプルの収集、分類、新種発見に役立つものになることが期待される。

4.3 ベンチマーク

この章では上記の実験における動作ベンチマークを列挙する。実験は OS に Linux kernel 2.6.24、Quad-Core Intel Xeon CPU (3.0GHz) と 16GB RAM の PC を使用した。

- Rustock: 復号ルーチンの解析に 79 秒かかった。
- Troj.Obfus.Gen: 全体の解析に 1588 秒かかった。
- オンラインゲームワームの樹形図作成: ひとつのワームにつき約 20 分から 50 時間
- 類似度検索: ひとつのサンプル検索に 518 秒

現在 `Syman` は Python 言語で実装されているため処理が遅くなっている。アルゴリズムの改善とともに、解析速度の向上の余地はまだ多く残されている。

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

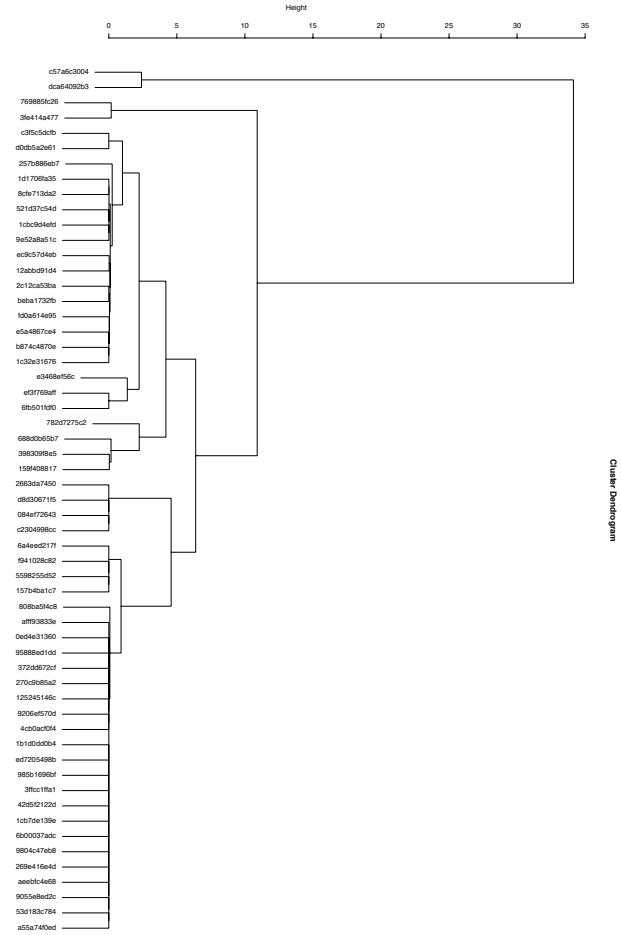


図 6 オンラインゲームワームの樹状図
Fig. 6 Cluster Dendrogram of Online Game Worms

5. ま と め

本稿ではマルウェア解析のために仮想環境かでのエミュレーションによる動的解析と、静的解析を相互に組み合わせた手法を提示した。マルウェアを対象としたバイナリコードの静

```
#find_similar -n 5 ~/dom/ a1273bc593

candidates:
1. "b5a4e1d227" (dist=0.000000) Trojan.PWS.OnlineGames.ZWU
2. "4aa7ac48ee" (dist=0.167832) Trojan.PWS.OnlineGames.ZWU
3. "bea5eb9245" (dist=0.181818) Trojan.PWS.OnlineGames.ZWU
4. "9b5ef9a200" (dist=0.717949) Trojan.Generic.2598621
5. "4f73c73ccc" (dist=0.717949) Trojan.Generic.2310394
```

図 7 マルウェア検索の出力
Fig. 7 Malware Search Session

的解析ツール Syman で制御フローを解析し、静的に解析不能な部分にはエミュレータによる動的解析ツール Alligator でエミュレーションを行うことで、実行時情報を収集することができる。Syman は、各機械命令を低レベルに記号的解釈し、得られた CFG の SSA 形式を用いた値解析を行うことによって、間接ジャンプの行き先アドレスを解決し、CFG を拡大していく。バイナリコードに対しいかなる仮定も置かずに解析を行うので、難読化されたマルウェアでも解析が可能である。本手法を用いた解析の結果得られた CFG を用いたマルウェアの分類法についても実験を行った。

現時点では、Alligator と Syman の連携は不十分であり、静的解析の結果、得られた CFG を用いて更なる静的解析を行い、その結果をエミュレータの実行にフィードバックする仕組みが必要と考えている。解析速度や精度の向上などまだ本研究には改善する余地は多いが、将来的にこれまでにない新規なマルウェア対策方式の端緒となることを目標としている。

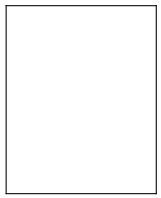
参 考 文 献

1) Ananian, C.S.: The Static Single Information Form, Technical Report MIT-LCS-TR-801, Laboratory for Computer Science, Massachusetts Institute of Technology (1999).
2) Balakrishnan, G. and Reps, T.: Analyzing memory accesses in x86 executables., *Proceedings of International Conference on Compiler Construction (CC2004)*, Springer-Verlag, pp.5–23 (2004).
3) Ballance, R.A., Maccabe, A.B. and Ottenstein, K.J.: The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages, *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp.257–271 (1990).
4) Brumley, D.: Analysis and Defense of Vulnerabilities in Binary Code, PhD Thesis, School of Computer Science, Carnegie Mellon University (2008).
5) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Effi-

Vol.2010-DPS-142 No.36
Vol.2010-CSEC-48 No.36
2010/3/5

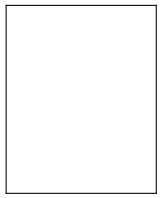
ciently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems*, Vol.13, No.4, pp.451–490 (1991).
6) Gal, A., Probst, C.W. and Franz, M.: Java Bytecode Verification via Static Single Assignment Form, *ACM Transaction on Programming Languages and Systems*, Vol.30, No.4, pp.1–21 (2008).
7) Hashimoto, M. and Mori, A.: Diff/TS: A Tool for Fine-Grained Structural Change Analysis, *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)* (2008).
8) Mori, A.: Detecting Unknown Computer Viruses – A New Approach –, *Lecture Notes in Computer Science*, Vol.3233, pp.226–241 (2004).
9) Mori, A., Izumida, T., Sawada, T. and Inoue, T.: A tool for analyzing and detecting malicious mobile code, *Proceedings of the 28th International Conference on Software Engi neering (ICSE2006)*, pp.831–834 (2006).
10) Reif, J.H. and Lewis, H.R.: Symbolic Evaluation and the Global Value Graph, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp.104–118 (1977).
11) Reif, J.H. and Lewis, H.R.: Efficent Symbolic Analysis of Programs, *Journal of Computer and System Sciences*, Vol.32, No.3, pp.280–313 (1986).
12) Reps, T. and Balakrishnan, G.: Improved memory-access analysis for x86 executables, *Proceedings of International Conference on Compiler Construction (CC2008)*, Springer-Verlag (2008). Paper accompanying "unifying" invited talk at ETAPS 2008.
13) Tarjan, R.E.: Fast algorithms for solving path problems, *Journal of the ACM*, Vol.28, pp.594–614 (1981).
14) Tu, P. and Padua, D.: Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers, *Proc. 9th International Conference on Supercomputing (9th ICS'95)*, Barcelona, Spain, ACM Press, pp.414–423 (1995).

(平成？年？月？日受付)
(平成？年？月？日採録)



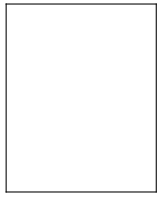
泉田　大宗

修士 (理学) 京都大学 (1996). 博士後期課程単位認定退学. 2002 年より産業技術総合研究所. ソフトウェア工学 (バイナリコード解析) 等に興味を持つ。



橋本　政朋

博士（理学）京都大学（1998 年）. 東京工業大学大学院情報理工学研究科リサーチアソシエイトを経て 2001 年産業技術総合研究所入所. ソフトウェア工学（ソースコード解析）, プログラミング言語（理論と設計）等に興味を持つ. 日本ソフトウェア科学会会員.



森　　彰（正会員）

平成 7 年京都大学大学院工学研究科情報工学専攻博士課程修了（工学博士）。英国オックスフォード大学訪問研究員、米国カリフォルニア大学サンディエゴ校ポスドク、北陸先端科学技術大学院大学助手を経て、平成 13 年産業技術総合研究所入所。現在同所主任研究員。ソフトウェア工学の基礎、ユビキタスコンピューティング、ソフトウェアセキュリティ技術などの研究に従事。