2-8　未知ウィルス検知のための新手法と実装

2-8 Detecting Unknown Computer Viruses –A New Approach–

森　　彰　　澤田寿実　　泉田大宗　　井上　直
MORI Akira, SAWADA Toshimi, IZUMIDA Tomonori, and INOUE Tadashi

要旨

本稿では、x86 プロセッサ上での Win32 実行ファイル形式のコンピュータウィルスを、パターン定義に依存することなく未知の状態で検知する新しい技術について報告する。具体的には、コードシミュレーションによる暗号復号とコードの静的解析による API 関数呼び出しレベルでの振る舞い解析を行うことにより、ファイル感染や無差別メール送出といったウィルス特有の振る舞いを検知するものである。禁止すべき振る舞いをセキュリティポリシーとして別途定義しておくことにより、様々な振る舞いを検知することが可能である。この方式に基づいて開発されたツールは近年に蔓延したウィルスのほとんどを未知の状態で検知することに成功している。

We give an overview of the tools to detect computer viruses without relying on "pattern files" that contain "signatures" of previously captured viruses. The system combines static code analysis with code simulation to identify malicious behaviors commonly found in computer viruses such as mass mailing, file infection, and registry overwrite. These prohibited behaviors are defined separately as security policies at the level of API library function calls in a state-transition like language. The current tools target at Win32 binary viruses on Intel IA32 architectures and experiments have shown that they can detect most email viruses that had spread in the wild in recent years.

[キーワード]
未知ウィルス，静的コード解析，コードシミュレーション，セキュリティポリシー，仮想実行時環境
Unknown computer viruses, Static code analysis, Code simulation, Security policy, Virtual runtime environment

1　はじめに

今日の社会では、個人の通信から、企業活動、更に社会インフラまでもがコンピュータネットワークに深く依存しつつあり、コンピュータウィルスにより甚大な被害がもたらされる危険性がますます増大してきている。対ウィルスソフトを導入することがコンピュータ利用における常識として認知されつつあるが、従来の方式では既知のウィルスのパターンを収集したデータベースを用い、照合を行うことによりウィルスかどうかを判別するという方法に基づいているため、未知のウィルスの検知は原理的に困難であった。

近年発見されるウィルスはますます高度になっており、感染力が増大する一方で、亜種の頻繁な出現とそれに対応するためのパターン抽出の手間が増大するという問題が生じている。ひとたびウィルスが進入し実行されれば、一瞬の間に大きな被害をもたらす可能性があることから、未知のウィルスに対してもその危険性を判別し被害を未然に防止する精度の高い検知技術が必要とされている。

このような背景の下、本研究では、x86 プロセッサ上での Win32 実行ファイル形式（Windows プラットフォームで動作する通常のプログラム形式）のコンピュータウィルスを対象としてこれらを未知状態で検知するための技術を開発して

きた。具体的にはコードシミュレーションによる暗号復号とコード静的解析による API 関数呼び出しレベルでの振る舞い解析を行うことにより、ファイル感染や無差別メール送出といったウィルス特有の振る舞いの検出を行うツールの開発である。このツールによって、禁止すべき振る舞いをポリシーとして定義しておくことにより、パターン定義に依存することなく、未知の状態でウィルス特有の有害な振る舞いを検出することが可能となっている(図 1)。

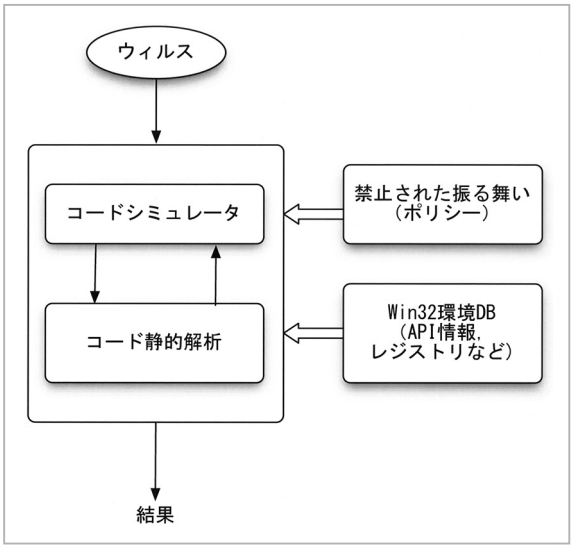

図1 未知ウィルス検知の枠組み

これまでの実験の結果、このツールを用いて実際に蔓延したウィルスの大部分を未知の状態で検出可能であることを確認できた。これは、サンプルとなるウィルスの解析を通じて有効と考えられるポリシーを定義・実装した後、無害プログラムに対する誤認検査を経て実際の検知実験を行うという作業の繰り返しであったが、その過程で近年のウィルスに共通して用いられる「対アンチウィルスソフト」の仕掛けが多く見られることが分かった。

本稿では作成されたツールの基本原理を説明するとともに、上で述べた「対アンチウィルス」手法の代表的なものについて説明し、これらに対する本ツールの対処法を述べる。

なお、本研究で x86 上の Win32 実行プログラム形式のウィルスに的を絞った理由は、それらが最も一般的かつ検知の困難な形式であることと、この形式に対応することで得られた技術は他のプロセッサや OS に対しても有効だと考えられるからである。

2 従来のアンチウィルス技術

現在多くの商用のアンチウィルスプログラムに採用されているウィルス検知の手法は、パターン(シグネチャ)マッチングあるいはスキャナと呼ばれる手法を基礎としている。この手法では既に知られているウィルスのバイナリコードの特定の部分を 16 進数文字列(これをパターンあるいはシグネチャと呼ぶ)としてデータベースに登録しておき、検査したいファイルと照合することで、それがウィルスであるかどうかを判定する。この方法に関しては一般に以下の問題点がある。

- パターンデータベースに登録されていない未知ウィルスを検知することができない。
- 無害なファイルをウィルスと誤認することがないように、ウィルスを一意に特徴付けるパターンを用意することが困難[1]。
- ウィルスコードを一部修正しただけで(亜種の多くがそうである)既存のパターンとの照合ができなくなる。極端な場合使用するコンパイラを変更して再コンパイルするだけで知られているパターンと照合されない場合もある。

パターンの照合に関しても単純な文字列の照合ではなく、より一般化された文字列パターンを指定できる正規表現を用いたものや、ファイル構造、プログラム構造に応じたパターン照合も考えられているものの、基本的には構文的な照合によっており、能力的な限界があるのが実情である。

未知ウィルスに対する対策としては、疑わしい実行形式ファイルを隔離されたコンピュータ上で実行させ、その動作を観察することによって危険性を判別する方式(これを動的保護と呼ぶ)が一部で採用されているが、プログラムが実際に実行した動作しか観察できないことから、特定の条件下(例えば特定の日時)でのみ危険な動作をするウィルスを確実には検出できないという問題がある。また、特定のプログラム構造の出現を見るといった、より一般的なパターンを

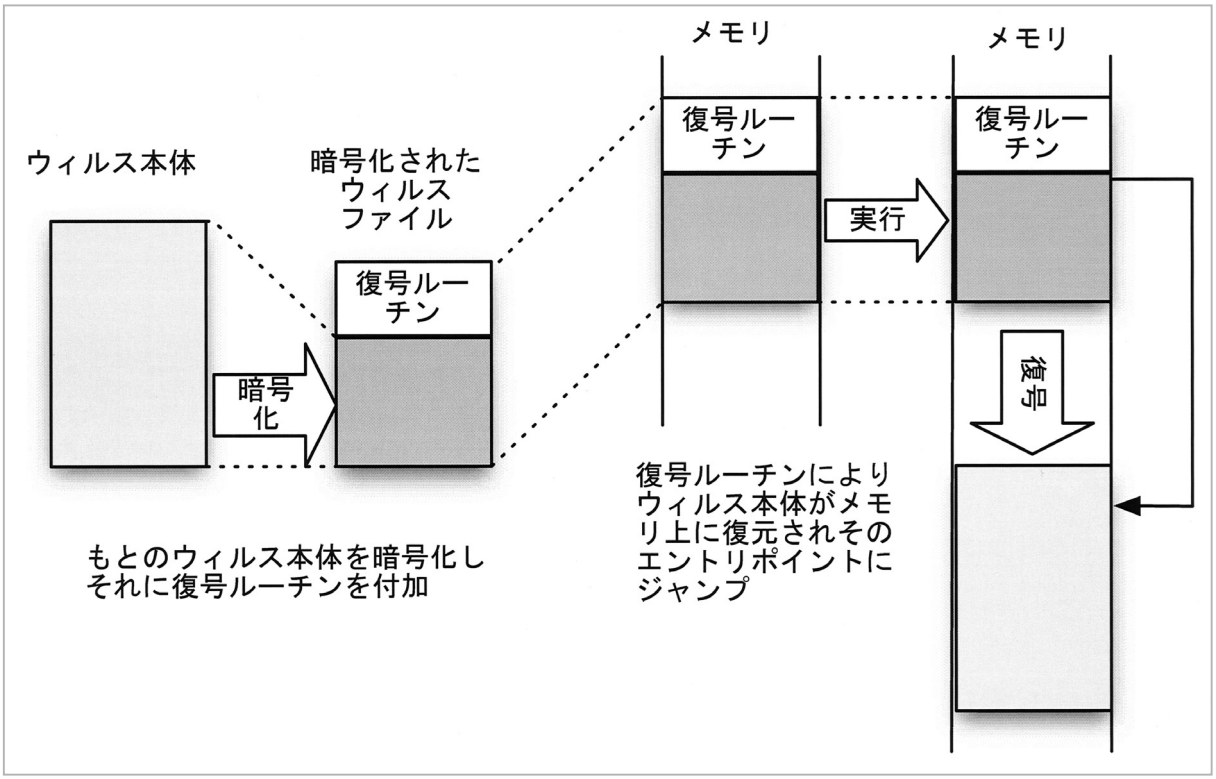

図2　自己暗号化ウィルス

利用する（ヒューリスティックスキャンと呼ばれる）方法も知られているものの、悪意のない有用プログラムをウィルスと誤認する率が高くなるという問題のあることが指摘されている。

1　あるアンチウィルスソフトベンダーによれば、登録されているウィルスは亜種を含めて２万個以上あるとも言われている。

3　自己暗号化、自己変形ウィルス

未知であるか既知であるかにかかわらず、今日のウィルス検知において克服しなければならない問題として、自己暗号化さらには自己変形を行うウィルスへの対応がある。これらは元々ウィルスパターンの作成を困難にし、パターン照合に基づく検知を困難なものにするねらいで考案された手法であるが、未知ウィルス検知においては更に困難な状況を生み出すことが予想される。

自己暗号化ウィルスとは、ウィルス本体を暗号化したデータにこれをメモリ上で展開する復号コードを付加した構造を持つウィルスである（実行の様子については図２を参照のこと）。実際に悪害をもたらすウィルスコードが暗号化されているため、プログラムコードを調べただけではその振る舞いを知ることができないという効果があり、特にパターン照合においてはパターンを設定できる範囲が復号コードの部分に限られるため、パターン作成が一層困難になるという状況が生じる。

自己暗号化の仕組みを更に発展させ、「定まったパターンが一切ない」ウィルスを実現するために考え出されたのが自己変形ウィルスの手法である。これは暗号化の方法そのものを実行の度に変化させていくことで復号コードを毎回変化させようとするものである（図３）。実際には使用できる暗号化アルゴリズムの種類は物理的な制約上有限になるため、「定まったパターンが一切ない」ウィルスを実現することは非常に困難ではあるものの、ネットワークから暗号アルゴリズムをダウンロードするなどして、より多くの変種を生み出そうとする試みなどが知られている。こうした自己変形ウィルスを従来のパターン照合で検知するには、複雑で長大なパターンを用意しなければならず、このことが処理効率

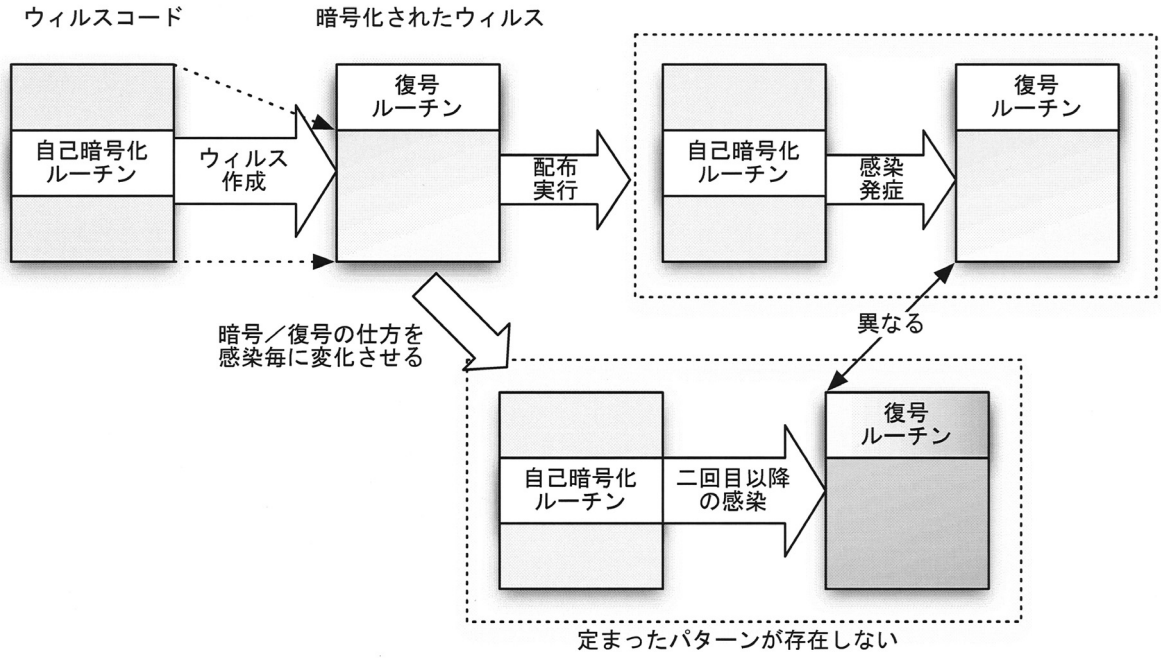

図3 自己変形ウィルス

の大幅な低下を招くことが確かめられている。

4 未知ウィルス検知の手法

与えられた未知の実行プログラムが悪意を持ったウィルスであるかどうかを検査する手法は大別して

- （隔離された環境などで）実際にプログラムを実行させて危険な動作をするか観察する（動的保護、**2** で既述）。
- プログラムそのものを解析して悪意ある動作をするプログラム部分を特定する（静的解析）。

の二つの手法に大別される。本研究開発においては、後者の静的解析の手法に基づいた検知手法を採用した。この理由としては、先に述べた動的保護の技術的限界に加えて、ウィルスの被害を防ぐために基本的にすべてのコンピュータ上で常時監視活動を行わなければならない点が挙げられる。

一方静的解析をベースとしたウィルス検知のためには、以下のような問題点を解決しなければならない。

⑴ 自己暗号化、自己変形への対応

先に述べたとおり、今日のウィルスは自分自身を暗号化することで単純なファイル検査やパターン検査により悪意が発覚しないような仕組みがプログラムされているものがほとんどである。このようなウィルスの危険性を未知状態で検知するには暗号化を解読するか、ウィルスの実行を追跡して復号を実際に行わせるかしかない。

⑵ 悪意ある動作を同定するための手法

与えられた実行プログラムが悪意ある動作（ファイル破壊、機密文書メール送出など）を行うかどうか判別するには、プログラムがオペレーティングシステムに対して依頼する特殊な関数呼び出し（Windows 上では API 関数呼び出し、Unix 上ではシステムコール）を解析することが必要である。Windows や Unix のような一般オペレーティングシステムにおいては、ファイル操作のようなシステム資源に対する操作を一般プログラムが直接行えないよう保護されており、例えウィルスであってもオペレーティングシステムへの関数呼び出しによってしか悪意ある動作を実行することができない。具体的には悪意ある動作を特徴付ける一連の関数呼び出しを同定するための仕組みが必要となる（図 4）。

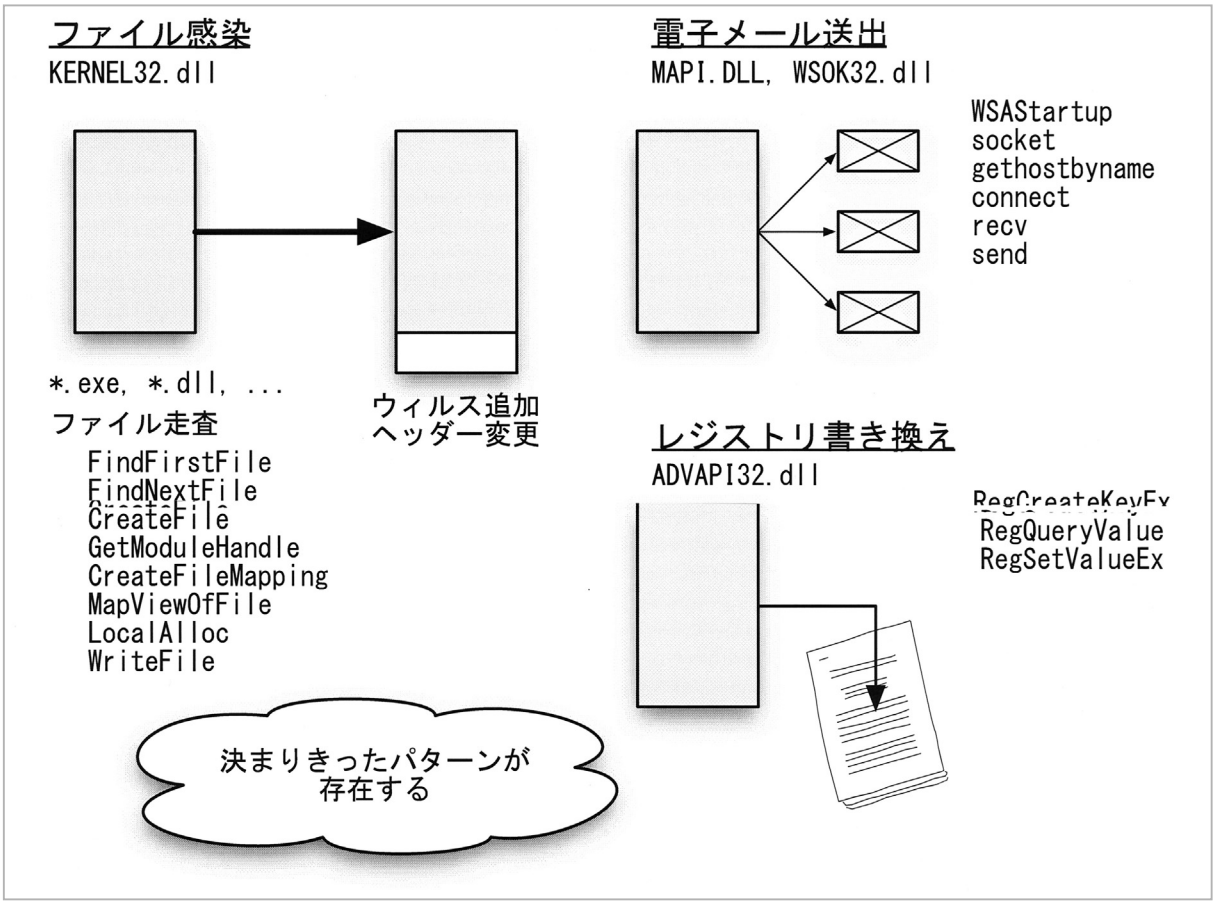

図4 API 呼び出し解析によるウィルスパターンの同定

本研究では、これらの課題に対してコードシミュレーションと呼ばれる技術を利用することでコード静的解析の弱点を補い、実機環境でコードを実行させることなく実行時のコードの振る舞いを解析することとした。例えば自己暗号化ウィルスの場合は、シミュレータ上でウィルスコード自身に暗号の復号を行わせた上で、得られた情報を元にコードの静的解析を通じウィルスかどうかの判定を行うことになる。

また、図 4 にあるように、ウィルス固有の振る舞いに特定の API 関数呼び出しパターンが存在することを踏まえ、コード解析を行いながらこうしたパターンを同定することとした。この際、何を持ってウィルスであると判断するための情報が事前に必要であるが、これについては API 呼び出しパターンを元に、禁止すべき振る舞いをポリシーとして記述しておくことで解決した。このポリシー記述は構文的な情報のみに依存していた従来のウィルスパターンの記述を意味的な振る舞いの記述にまで拡張したものとみなすことができ、従来のパターンに基づく方式には存在しない広範な検知を可能にするための鍵となる部分である。

以下ではこれらの方針に基づいて作成された検知ツールの取る方式について、より詳細な説明をする。

4.1 コードシミュレーション

上で述べたコードシミュレーションとは、x86 アーキテクチャの CPU の内部構造（レジスタ、メモリ、フラグ等）の機械命令の実行による変化を忠実に模倣することである。開発したシミュレータは Windows、Linux 両方のプラットフォームで動作し、ステップ実行、ブレイクポイント設定、メモリダンプなどの基本的なデバッガ機能を持っている。

先に述べたとおり、自己暗号化さらには自己変形を行うウィルスを静的解析により検知するには、プログラムコードを解析するだけでは不十分であり、それが実際に実行する命令の一つ

一つをシミュレートしていくことによって、ウィルス自身が行う復号動作や悪意ある動作を追跡することが必要である。

4.2 PE ローダ機能

ウィルス検知のためには、単に機械命令をシミュレートするだけでは不十分である。プログラムには、実行コードそのもののほかに利用される外部定義関数等の付加情報がファイル内の様々な場所に格納される。また実行コードをメモリ上のどの番地に配置するか、最初に実行の制御をどこの番地に渡すかといった情報も格納されている。したがって、ウィルス検知のためにはファイルを走査しこれらの情報を取り出して、シミュレータ内のメモリの適切な番地に実行コードをロードする機能が必要になる。こうした本来 OS が行うローダ機能をシミュレータ自身で行う必要がある。本研究では Win32 環境を対象とするため、そこでの実行形式である PE (Portable Executable) と呼ばれるバイナリファイル形式を処理する機能を実装してシミュレータに統合している。

4.3 外部 API 関数呼び出し処理

先に述べた API 関数は外部ライブラリとして提供され、アプリケーションプログラムはそこに格納されている API 関数を利用する。API 関数のコードをシミュレートするには完全な実機環境が必要であり、また、ウィルス検知に必要がない関数が大部分である。したがって、そうした関数呼び出しを行う命令に遭遇した場合、シミュレータはそれを忠実にシミュレートすることはせず、API 関数呼び出しが発生したことを記録するにとどめ、そのサブルーチンコールから戻った状態へと遷移する。これを行うには以下の処理が必要である：

・実行時に定まる外部関数の番地が格納されている番地を記憶しておくこと、すなわち
  a ローダがプログラムをロードする時に割り当てる番地
  b ウィルスコード自体が LoadLibrary（ライブラリを動的にロード）、GetProcAddress（ロードされたライブラリに含まれる関数の番地を取得）などの API 関数を用いて自ら実行時に割り当てる番地

  の両方を、その格納番地と関数の名前とともに保持しておくこと。
・スタックに積まれた引数を取り除くこと [2]
・ポリシー検査処理（ポリシー違反検知のための状態機械の駆動、後述）

実際の処理に関しては、API 関数の数が非常に多いためこれらの処理をシミュレータのコードの中に埋め込んでいてはシステムの発展性を保つのが困難である。このため、シミュレータが API 関数呼び出しに遭遇した場合は、必要な処理を行うダミーの関数（スタブ関数と呼ぶ）を呼び出すこととし、各 API 関数に対応したスタブ関数をまとめたライブラリを別途用意することとした。現在 Windows のシステム情報（API 関数の引数のバイト数など）を機械的に処理し、スタブ関数のひな形を自動生成することが可能となっており、個々に詳細な対応が必要な場合、他には影響を与えずに個別に対応することが可能である。

2 Windows 環境では、スタック上の引数をライブラリ関数が取り除くというきまりがあるためである。なかには例外もあり個別に対処する必要がある。

4.4 仮想実行環境

上で述べたとおり、ウィルス検知の核となる振る舞い同定のための API 関数に関しては、

・実行ファイル/ライブラリのロード機構
・スタブ関数
  ・スタックから引数を降ろす処理
  ・ポリシー検査処理（状態機械の駆動）

を用意することにより、実際に外部ライブラリコードを（仮想）実行することなくプログラムの解析が可能となっている。しかし、深い解析を行うためには、より詳細な実行環境に関する情報や、プログラム実行に伴って生じる副作用を仮想的に引き起こす必要が生じる。開発した検知ツールでは、前者は Windows 仮想環境データベースを用意することで、後者はスタブ関数内の実装で対応している。

具体的には、

・Windows 仮想環境データベース
  ・レジストリ

・シェル環境変数
・実行時情報
・ヒープ領域：HeapAlloc などの API 関数によって割り当てられる、メモリ内に確保される動的作業領域
・ファイル：CreateFile などの API 関数によって生成あるいはオープンされるディレクトリやファイル

などを処理している。

レジストリに関しては、Windows レジストリ自体が巨大なデータベースであり、使途不明なキーが大多数を占めていることもあり、標準的なプログラムで参照されるものやウィルスに悪用される可能性の高いものだけをデータベース化している。スタブ関数についても、標準的な Windows 環境で用意される実行時ライブラリは 1000 個を超える一方、プラットフォーム SDK として準備される API 関数の数も莫大である。これらすべてについて、引数のバイト数や成功時の返り値などを埋めていくのは非常に困難なことである。悪いことにこれらの API 関数では、成功時にゼロを返すものやゼロ以外の値（文字数やエラーコードなど）を返すものが首尾一貫せずに混在しており、頭の痛い問題である。現状では、使用頻度の高いライブラリ 100 個程度について、その中で定義されている API 関数のためのスタブ関数のひな型を自動生成し、特殊な返り値に関しては種々のドキュメントを参照しながら人手で入力を行うようにしている。

実行時の情報として、ヒープ領域はプログラムごとに仮想ヒープ領域を確保し、その中で仮想メモリ領域を割り当てるようになっている。ファイルについては、ファイル構造体のみを作成・管理し、実際にファイルを作成することは行わない。いずれの場合も該当するスタブ関数の中で構造体の処理を行い、その情報は後続のポリシー検査において用いることが可能である。また、先にも述べたように、本ツールでは CPU による命令実行の部分だけを忠実にシミュレートすることを基本としている。したがって、OS によって行われる様々な処理は別途対応が必要になる。実行ファイルをロードする PE ローダの機能はそうしたものの一つであるが、これ以外にも、

・ページングを含むメモリ管理
・例外割り込み
・スレッド管理

を適切に処理することが必要である。例外割り込みは、Windows プラットフォーム固有の SEH（Structured Exception Handling）と呼ばれる機構を仮想的に処理すればよいが、ページングは基本的なアクセス制御を実現するためにも必要不可欠であり、本ツールでも基本的なページングとそれに伴う例外割り込みの発生処理を行うようにしている。スレッド管理については、TEB（Thread Environment Block）と呼ばれるスレッド固有の情報を格納したメモリ領域の管理のほかに、同期や排他制御を行うマルチスレッド実行機構を CPU シミュレータの上に実装する必要がある。TEB については公開されている情報に関しては対処済みであるが、マルチスレッドに関しては現在のツールでは単に生成された順番にスレッドのコードを解析しているにとどめている。振る舞い同定のためには、現状ではこれで十分であると考えているが、複数のスレッドが協調して一つの有害タスクを実行するような場合は、細粒度のポリシー検査が行えない可能性がある。より詳細なスレッド実行のエミュレーションについては今後の課題である。

4.5　コード静的解析

コードの流れに従ってシミュレートしていただけでは、エミュレーションに基づく実行時保護と同じ問題を抱えることになる。すなわち「たまたま」実行されたコードの振る舞いだけを基にウィルスかどうかの判断を行うことになる。一方で、実行される可能性のあるすべてのコード片を検査することは、自己暗号化ウィルスのように自分で自分を書き換えるコードがある場合は非常に困難である。

開発の当初は分岐命令ごとにバックトラックを行うことを考えていたが、CPU の状態やメモリ内容のスナップショットをバックアップすることは効率が悪く、また、ループ構造への対応が非常に困難であった。現在では、まずコード静的解析により注意の必要な API 関数呼び出し命令を先読みし、後にこれらの API 関数呼び出しを含む実行パスにシミュレーションを優先的

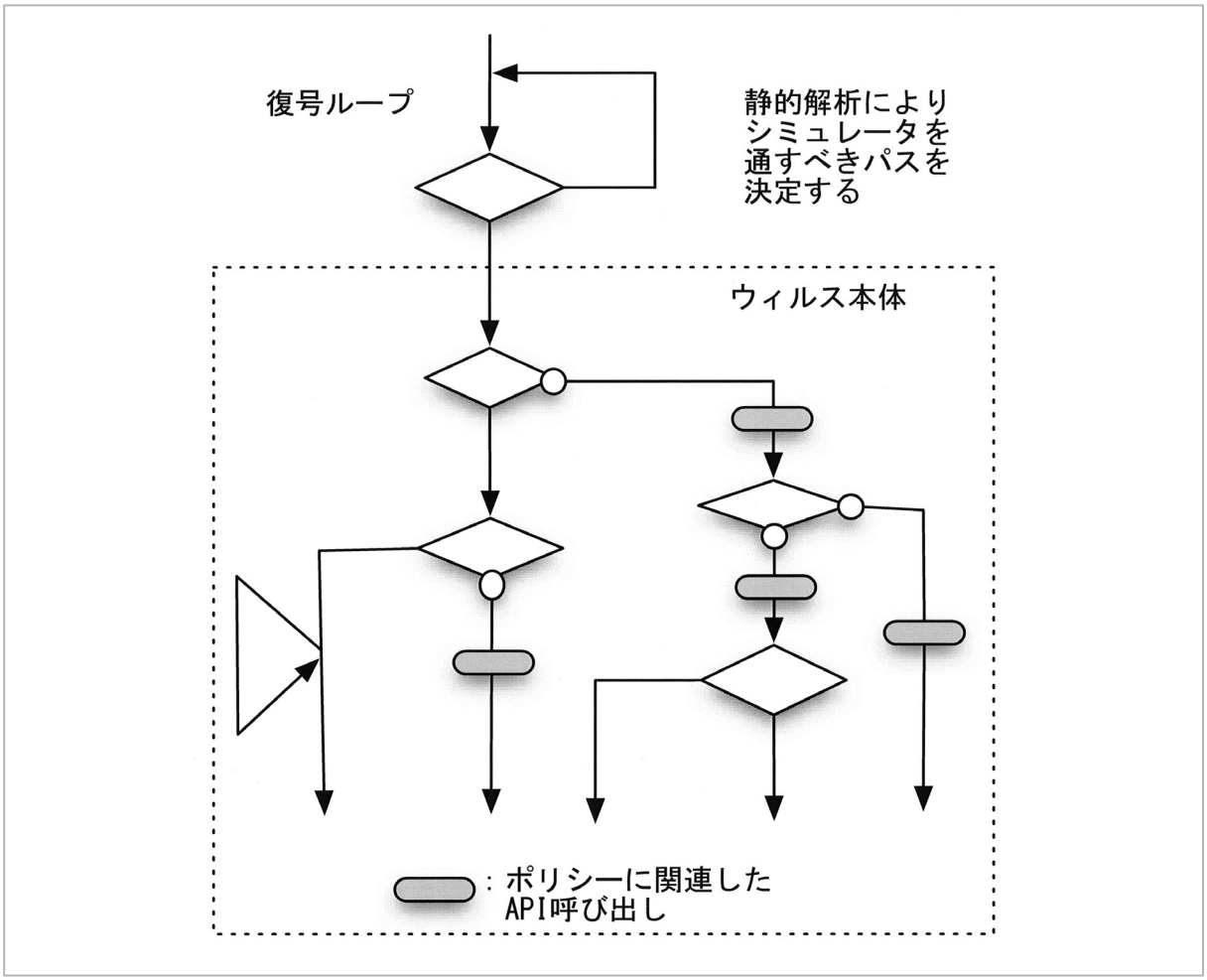

図5 シミュレータと静的解析の併用

に通すように制御を行っている(図5参照)。

基本的にはコード静的解析の行うことは逆アセンブラが行うことと同じである。すなわち、

(1) 与えられた番地から始めてメモリ上から1バイトずつデータを取り出す。
(2) 取り出したデータを機械命令セットに参照し、命令の種類(オペコード)や引数(オペランド)を決定する。
(3) 命令に応じた、しかるべき長さのバイト数分をスキップする。

という処理の繰り返しが基本であるが、本研究では分岐命令とその飛び先によって区切られるコード断片を単位に、既に解析が完了したかをマークしておくことで、シミュレータが自己コード書き換えにより新たに発生した(マークの付いていない)コードに遭遇した場合に、再びコード静的解析器を呼び出すことができるようにしている。こうすることで、動的に変化するコード片に対しても効率よく検査が行えるようになっている。

4.6 ポリシー検査システム

既に述べたように、ポリシーとはウィルス固有の禁止すべき危険な振る舞いを想定したものである。具体的にはメール無差別送信やファイル感染といったシナリオを単位として、API関数呼び出しに対応した粒度でその振る舞いが既述される。そしてポリシー定義を基に、コードシミュレーションとコード静的解析の結果を参照しながら、コード内に指定された振る舞いが存在するかどうかを調べるのがポリシー検査の役割である。ポリシー定義は基本的には注目したい挙動(主として特定のAPI関数呼び出し)をイベントとし、加えて付加的な情報(API呼び出しの場合は引数列)を入力パラメータとした状態遷移機械として定義する。例えば、ある種のメ

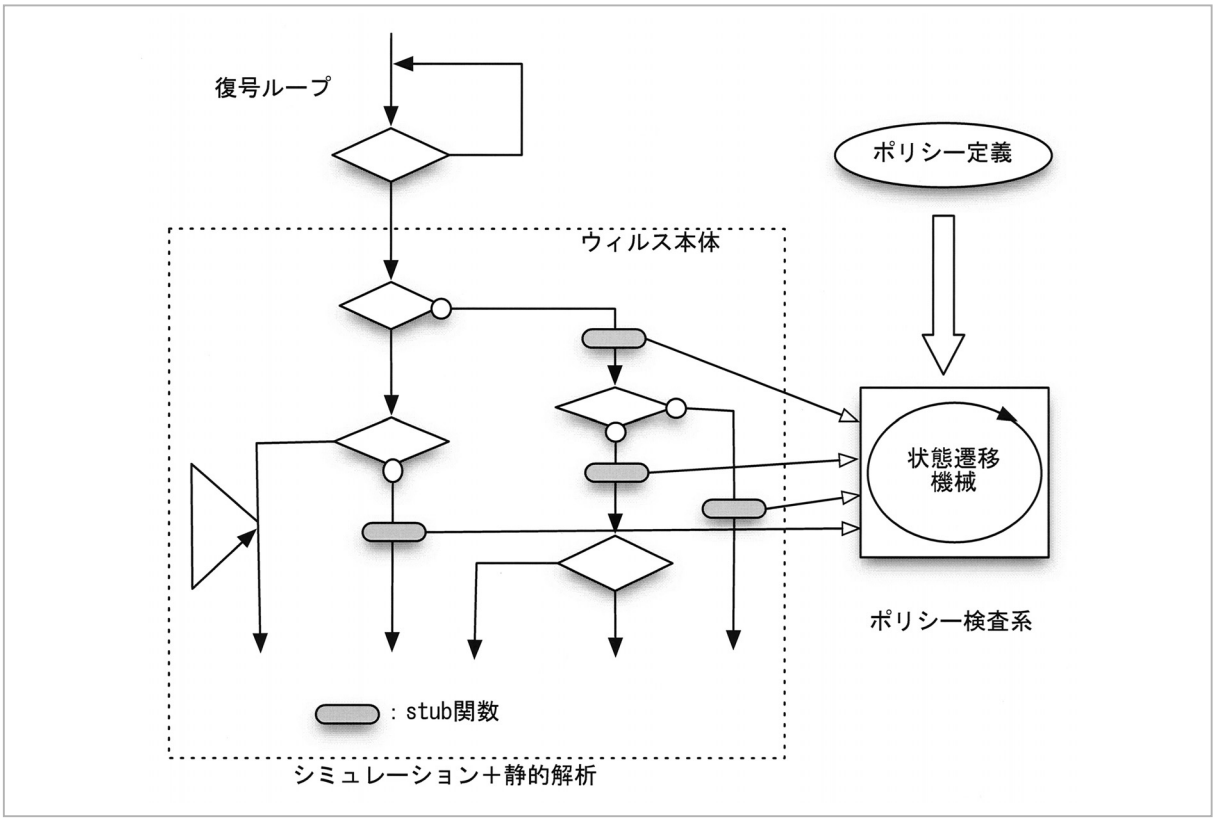

図6　ポリシー検査の仕組み

ール無差別送信に対応するポリシー定義は次のようになる：WSOCK32.dll ライブラリの send 関数が呼び出されたときに、その第 2 引数（メッセージに相当する）の To 属性（あて先に相当）が所定のファイルから得られた無差別なものであり、かつ第1引数（ソケットに相当）の接続先が所定のレジストリから得られたデフォルトのメールサーバアドレスである場合、現在の状態が「ソケット接続が完了した」状態であれば、次状態は「無差別メール送信の発生が確定した」状態へ遷移する、というような形の記述内容である。

ポリシー検査では、シミュレータと静的解析から得られる情報を基にポリシー定義の記述どおりに状態遷移を行い、それが受理状態に到達したかどうかを監視することで行われる（図 6 参照）。実行順序を考慮した状態遷移機械に基づく処理を行うため、ツールの実装に当たっては、シミュレータの制御の下にスタブ関数経由でポリシー検査系を駆動することとした。静的解析の結果を基にシミュレータの分岐動作を強制する場合に、検査すべき API 関数呼び出しへの経路が優先されるため、このポリシー検査とポリシー定義との対応が崩れる可能性がある。ただし、これまでに試験を行った 200 個余りのウィルスの検査ではそのような状況は生じていない。

5　対アンチウィルス技法とそれへの対処

本章では現行のウィルスが用いる様々な「対アンチウィルス技法」の概要とそれに対する本研究で開発した検知ツールの対処法について述べる。

5.1　API 関数呼び出し隠ぺい

本研究で開発したツールは、プログラムから OS に対しての API 関数呼び出しに基づく振る舞いの同定が核となっている。このため、API 関数呼び出しを見逃すことはウィルス検知の失敗につながる。一般にどの API 関数を用いるかが分かるだけで、そのプログラムがどのような動作をするかをおおまかに知ることが可能であるため、近年のウィルスでは様々な隠ぺい手法を用いる傾向がある。

最も基本的な隠ぺい方法は、API 関数の番地

を GetProcAddress などの API 関数を用いて実行時に取得するものである。一般のプログラムでは実行形式ファイル中に前もって使用する実行時ライブラリと関数に関する情報が宣言されており、ローダがプログラムをメモリ上へ展開する際に関数のアドレスが決定されるのに対して、実行時結合では実際にプログラムを実行させなければどのような API 関数が用いられるか不明である。

このような手法に対し、本研究ではローダによるアドレス結合とプログラム自身の GetProcAddress を用いた結合の双方に仮想的なアドレスを割り当てることで、後にどの関数が呼び出されたかを知ることを可能にする機構を実装している。しかし、近年の多くのウィルスでは、ロードされた実行時ライブラリをメモリ上で走査することで、GetProcAddress すら用いずに API 関数の実行時アドレスを取得することが一般的となってきている。ライブラリの先頭アドレスさえ分かれば Windows プラットフォームでの実行ファイル形式である PE フォーマットに従って、ロードされたライブラリのメモリ上イメージからすべての API 関数の実行時アドレスを所得することが可能である。さらに次のような手順も多く見受けられる：

(1) 利用したい API 関数が定義されているライブラリ (foo.dll とする) をロードする。これはあらかじめリンクしていても実行時に LoadLibrary API 関数を用いてロードしてもよい。
(2) ロードされたライブラリの中で定義されている他の関数[3] (Func とする) のアドレスを取得する。これも上述のようにあらかじめリンクしておいても、GetProcAddress を使って取得してもよい。
(3) Func のアドレスから 16bit 境界でライブラリのメモリイメージを前方に走査し、PE 形式イメージの先頭にあるマジックワード“MZ”を発見することで、ライブラリの先頭アドレスを取得する[4]。

このようなやや複雑な手順を踏むことで、どのライブラリを走査しているかを分からなくするものも多く見られる。このような挙動は、実際のライブラリファイルを用意した上で仮想実行を行う必要が生じるため、安易なエミュレーションでは振る舞いを同定することが困難になるという効果をねらったものと考えられる。このような特殊な手法が多くのウィルスに共通してみられるのは、ウィルス作成者同士の情報交換が盛んに行われていることや、“zine”と一般に称される様々な地下オンライン情報誌で紹介された手法を多くの作者がまねていることに起因していると思われる。

本研究のツールにおいても、既述のようにライブラリファイルをロードする PE ローダ機能をエミュレートするようにして、上述のような方法で関数アドレスを所得した場合でも、それがどの API 関数のアドレスであるかが分かる仕組みが作り込まれている。また、実際の Window プラットフォームのライブラリファイルは使用せず、代わりに個々のライブラリごとに作成した PE 形式の疑似ライブラリファイルを用意しており、これには上述のようなウィルスの行うメモリ走査に対応できるだけの情報を含んだものとなっており、個々の API 関数の実行本体コードは含んでいない[5]。また、このようにロードされた実行時ライブラリを走査して API 関数アドレスを所得するということは、通常のプログラムによっては全く必要のない振る舞いであると考えられるので、これをポリシーとして定義しウィルスとして検知することも可能となっている (**6** の IAC policy)。

3　ウィルスの立場からはなるべく関係のない一般的な関数が望ましい。
4　Windowsプラットフォームでは 16 ビットの境界で実行ファイルがロードされる。
5　ライブラリが外部に公開している名前と番地等の情報のみを持っている。

5.2　対デバッガ／エミュレータ対策

多くのアンチウィルス会社では、未知ウィルス検知の手法としていわゆるヒューリスティックスキャン技術を採用している。これは、隔離された実行環境でウィルスと疑わしきファイルの実行を監視することが中心となっている。このためには、実機での実行環境を完全にエミュレートする必要があるが、OS レベルでのデバッガ機能を用いたのでは、以下の述べるように完

全にエミュレートできない場合が生じる。

5.2.1 SEH（Structured Exception Handling）の悪用

ここではその代表的なものとして、Windowsプラットフォームにおける SEH（Structured Exception Handling）について説明する。

SEH は例外処理を統一して効率よく行うために考えだされた仕組みであり、CPU の割り込みレベルの上位の OS レベルで処理されている。

具体的にはセグメントレジスタ FS の 0 番地に現在のスレッドにおける例外処理ハンドラ構造体へのポインタが格納されることになっている。この構造体はスタック領域に格納され、通常 dword（4 バイト）2 組で構成される。最初が次の例外処理ハンドラ構造体へのポインタ、次の dword がハンドラコードのアドレスとなっている。このように例外処理ハンドラ構造体をポインタでつなぐことで大域的にどのハンドラが例外処理を引き受けるかを実行時に動的に決定することができるようになっている。

通常は OS レベルでのデバッガ解析を行っていても特に問題は生じないが、ウィルスによっては意図的に不法なアドレスのメモリへアクセスを行ったり、ゼロ割り算を行うことで例外割り込みを強制的に発生させるものが近年多く見られる。このとき意図的に引き起こす例外に対応した SEH ハンドラとして自身の暗号化を復号するルーチンやペイロード（ウィルスの行う害のある挙動）実行を行うコードを登録しておく。SEH は Windows OS によって処理されるので、OS を通じてデバッガ機能を実現している場合は SEH 処理が OS に横取りされ、例外処理が解析対象のプログラムのスレッドとは別のデバッガアプリケーションのスレッドで処理される結果となる。このためプログラム中に記載されたとおりの振る舞いが発生せず、結果としてウィルス検知に失敗することとなる。

通常の条件分岐による制御フローの移行は、すべての実行経路を追うことで原理的に解析可能であるが、SEH のような特殊なメカニズムで飛び先までが動的に決定されるような場合は、仮想実行機構に対応する処理が実装されていなければ解析は不可能である。この点で SEH 処理は本研究のツールにおいても必須の機能であるため、コードシミュレータの例外割り込み処理と連携して SEH 処理を正しく行えるようになっている。ただし、SEH 処理に関しては、様々なデータがスタック上に積まれ、また、連鎖的に割り込みハンドラが起動される場合のスタックの畳み込みや、try-finally 形式のクリーンアップ処理のための終端ハンドラなどの扱い等、実際の詳細が明らかでない（公開されていない）OS 側の処理が付随するため、新たな悪用の方法が出現した場合は新たに対処を迫られる可能性を否定できないのが現状である。

5.2.2 その他の対デバッガ／エミュレータ手法

現在利用されかつ有効な、デバッガ／エミュレータの存在を知る方法として以下のものがある。

- OS を通じてデバッガ機能が実現されている場合に、デバッガの存在を判別する方法。
  - a KERNEL32.DLL ライブラリの IsDebuggerPresent関数を利用。
  - b セグメントレジスタ FS の 0x20 番地に格納されるフラグの値を利用。
  - c デバッガによってセグメントレジスタ FS の値が通常の Windows 実行環境と大きく異なった値に設定されることを利用。
- エミュレーション環境の下での実行かどうかを判別する方法
  - a 非常に大きな（最大 4 GB）番地のアドレスへのアクセスを行い、これが成功するかどうかをチェック。
  - b 特定の定義ファイルがシステムディレクトリに存在することや、特定のレジストリキーが設定されていることを利用。
  - c BIOS を通じて得られるシステム情報が明らかに標準的でないことを利用。
  - d CPU の仕様書に明記されていない命令仕様を利用。代表的な例では、AAM 命令がある。この命令は、実際には任意の 1 バイトのオペランドを除算数として取るため、仕様書にある 10 以外の除算数を扱うことが可能である[6]。

開発した検知ツールでは、ページングによるメモリ管理の実装や、コードシミュレータ及び仮想実行環境における対応により、上記のいずれの方法によっても実機以外の環境で実行され

ていることを検査対象のプログラムに知られることはなくなっている。ただし、今後出現するかもしれない新たな手法に追随する必要があるのはSEHと同様である。

一方、上述のような振る舞いを一般のプログラムではあり得ないものとみなすこともできる。これらをポリシーとして定義しウィルス検知に用いることも可能である。例えば、IsDebuggerPresent関数を用いるプログラムを受け入れないというポリシーは通常の範囲では受け入れることのできるものと考えられる(**6** のanti-debugger/emulation policy)。

6　しかしこの事実はいまだにIntelのマニュアルには記載されていない。

6　ポリシー

以上述べたような道具立てで、現存するほぼすべてのウィルスと実用的なプログラムの解析が可能である。問題は、どのプログラムを悪質なものとして検知するかであり、この点でどのようなポリシーを定義するかが重要になる。現在実装されているポリシーのうち主要なものを以下に挙げて説明する。

- Mass mail policy：レジストリ情報を基にSMTPサーバアドレスと無作為送信先アドレスを取得し、メッセージを送信するというシナリオを検査。実際にmass mailかどうかを調べるためにSMTPプロトコルのシミュレーションも行う。ソケット(WSOCK32.DLL、WS2_32.DLL)を用いるものや、WININET.DLLやMAPI.DLLを用いる等使用するライブラリによるバリエーションが存在する。
- Registry modification policy：システム設定、特に自動起動するプログラムを登録するようなレジストリキーへの書き込みを検査。保護すべきレジストリキーのprefixリストを別途定義しておき、レジストリ設定に関するAPI関数スタブの中でこのリストとの照合を行うことで処理される。
- File modification policy：書き込みが禁止されたディレクトリ／ファイルへの書き込み検査。処理方法はレジストリの場合と同様に禁止されたprefixリストによる。
- File infection policy：ファイルに自分自身を書き込むような振る舞いを検査。ディレクトリ走査によりファイルハンドラを取得し、メモリ上で展開・修正を行った後にファイルに書き戻すようなシナリオが定義される。ファイル感染には様々な方法があるため、現在は典型的なシナリオのみを検査している。
- Process scan policy：実行中のプロセスを列挙しプロセスIDを取得するシナリオを検査。実際にはプロセスへの感染を行うところまで定義する必要があると考えられるが、通常のプログラムが行いそうにない行為ということで一つのポリシーとしている。
- Self-code modification policy：自己のメモリイメージ中のヘッダ、特にimport tableを書き換えるような振る舞いを検査。自己解凍ルーチンのような一般的な自己コード書き換えは、一般のプログラムが行うジャンプテーブルの書き換え等との区別が困難なため、特殊なパターンに特化して定義している。
- Anti-debugger/emulation policy：**5.2** で説明したようなデバッガやエミュレーション環境を判別しようとする特殊な振る舞いを検査。
- Out of bounds execution(OBE) policy：プログラムがそのヘッダ内で仕様を宣言したセクション自体のアドレスでコードが実行されるかを検査。非常に一般的なポリシーであるが、通常のコンパイラやアセンブラで作成されたプログラムでは起こりえない振る舞いであるため、独立したポリシーとして実装されている。一部の実行可能圧縮ファイル生成ツールによるファイルをウィルスとして検知する可能性がある。
- External execution policy：CreateProcessやShellExecuteなどにより外部プログラムを起動するかどうかを検査。
- Illegal address call policy(IAC)：**5.1** で説明したようなライブラリ関数のメモリイメージを直接走査して取得されたアドレスへ

の呼び出しがあるかを検査。

・Self duplication policy：実行中の自分自身をコピーするような振る舞いを検査。多くのウィルスに共通の振る舞いであるため、独立したポリシーとして実装されている。

・Network connection policy：FTP、HTTP等の通信を行うかを検査。Mass mail policyのようにアドレスの検査は行わない。FTPプロトコル、HTTPプロトコルのエミュレーションが必要。

これらのポリシーは、すべてスタブ関数から呼ばれ状態機械の駆動を行うプログラムとしてライブラリの形でC++言語によって記述されている。より高度なポリシー定義のための記述言語とそれを用いたポリシー処理系については現在設計中である。ポリシーを定義するため、検査対象となるプログラムの仮想実行に伴って動的に変化する状態を抽象化して、自動変換可能な語彙にまとめる作業が最も困難であり、これについては継続した地道な作業が行われている。

7　実験結果

6 で説明したポリシーを用いて、近年実際に蔓延したウィルス約200個に対する検知実験を行ったところ、すべての検知に成功した。このうちIAC policy、OBE policy、self-duplication policyで95％以上のウィルスを検知することができた。IAC policyをはずしてもOBE policyやregistry/file modification policyなどによって検知されるため、これらのポリシーによる全体の検知率は85％を超える。

実験に用いたサンプルの8割以上は、事前の解析を全く行っていないウィルスであるため、未知状態で検知されたものと考えてよく、十分な検知性能を持つことを確かめることができた。また、先に蔓延して大きな被害をもたらしたMyDoomやBagleやNetskyなどのウィルスについてはその亜種も含めてすべてを未知状態で検知できており、実環境での有効性も確かめることができた。

正常なプログラムをウィルスであると誤認してしまう誤報の問題であるが、本研究で用いた手法は決定的なアルゴリズムによって実現されていることから、実際にポリシーで定義されている振る舞いがないのにもかかわらず、あると判定されることは生じない。この意味で本研究開発における手法では誤報というものは存在せず、それはすべてポリシー定義の強弱の問題に帰着すると言うことが可能である。つまり、OBEポリシーのような一般的なポリシーの場合は、害悪のあるプログラムでなくともウィルスと判定されることはあるが、これはポリシー検査が間違っているのではなく、そういった強いポリシーを適用していることから生じる結果に過ぎない。このように何を善しとして何を悪しきとするかを実行時ポリシーとして正確に定義できることこそが、コードの外見からの構造をルールベースとした既存のヒューリスティック手法との大きな違いであり、高い検知性能を実現する中核技術となっている。

開発されたツールは、独立したプログラム検査ツールとして利用できるほか、メールサーバ上でウィルスフィルタとして運用しサーバ利用者全員を保護したり、メールクライアントに組み込んで個々のユーザを保護したりといった様々な利用法が可能である。

8　まとめと今後の課題

現状のツールはようやく実用化の入り口まで来たという段階である。ツール自体はすべての機能が独立して拡張可能なように実装されており、ポリシーごとにC++で検査プログラムを実装しなければならないことを除けば、試験運用とそれに伴う機能追加・調整は問題なく行うことが可能である。

一方で、処理速度や微調整において改善の余地があり、実用化に向けては実環境での運用実験を通じた処理速度、検出性能、負荷性能、運用性の向上が必要であることも明らかになっている。具体的には以下のような点で改善の必要があると考えている。

・データ構造やアルゴリズムの洗練化。またスレッド管理のように簡易な実装が行われている部分については再設計が必要である。

・ポリシー記述がC++プログラムによっているため、一般ユーザによるカスタマイズが

困難なものとなっている。ポリシー記述言語処理系の設計と実装が必要。

・プログラムの振る舞いを同定するために必要なレジストリ値やファイル構造、動的ライブラリ等の仮想実行環境に関するデータに遺漏があるため、正しい検出処理ができない場合がある。Windows プラットフォームの欠陥に起因して人海戦術的にデータを収集せざるを得ない部分があり、相当程度の労力と期間が必要。

・高負荷化での性能分析を行い、その結果を踏まえた処理高速化を行うことが必要。

・現状のポリシー定義は実際に蔓延したウィルスの解析を通して得られた知見に基づいたものである。未知ウィルス検知システムとは言え、ウィルス作成者によって新たなトリックが考案された場合には、それに対応した汎用ポリシーを新たに定義する必要がある。将来を見据えた包括的なポリシーセットを定義しておくことが必要。

将来の目標としては、数百人規模のユーザを持つメールサーバ上で未知ウィルス検出ツールを稼働させることにより、メールシステムのスループットを損なうことなく、未知ウィルスの95％以上を検知可能にすることを想定している。実際にはどのようなウィルスがこれから出現するかに依存するため、現在はシステムの洗練化と併せて標準的なポリシーを制定する作業を監視している。今後1年間の具体的な目標は以下のとおりである。

・新たに設計するポリシー記述言語を用いて、各ポリシーが平均して20行程度で記述できること。それまでに出現したウィルスを処理するのに十分な仮想実行環境データベースが整っていること。

・予見されるウィルスに対して有効と考えられる汎用ポリシーセットが20個程度のポリシーによって定義されていること。数百人規模のユーザを持つメールサーバ上で未知ウィルス検出の実験が行えること。

・未知ウィルス検出に起因するスループットの低下がユーザに意識されないようなレベルに抑えられ、かつ未知ウィルスのおよそ95％以上を検出可能であること。システム運用に関するドキュメントが存在し、ツールを広く実用できる状況にあること。

謝辞

本研究は平成13年度から15年度にかけて通信・放送機構（現情報通信研究機構）の委託研究として実施されたものである。研究の遂行に当たっては機構よりあらゆる面で多大なご支援及びご指導を頂き、最終的に一定の成果を出すことができた。ここに記して感謝の意を表させていただく。

参考文献

未知ウィルスの検知のための技法に関する論文／文献はほとんど存在しない。以下は一般的な有害ソフトウェアに関する解説書[1]及び（少し古い内容であるが）ウィルス作成技法についての解説[2]である。

1 Roger Grimes, "Malicious Mobile Code : Virus Protection for Windows", O'Reilly & Associates Inc, 2002.

2 Mark Ludwig, "The Giant Black Book of Computer Viruses", Second Edition, Lexington & Concord Partners, Ltd., 2000.

また、未知ウィルス検知に関連して米国で出願されている特許に以下のものがある。

3 米国特許庁特許開示文書，特許番号6，357，008，Dynamic heuristic method for detecting computer viruses using decryption exploration and evaluation phases.

4 米国特許庁特許開示文書，特許番号5，696，822，Polymorphic virus detection module.

森　彰（もり　あきう）
独立行政法人産業技術総合研究所グループリーダ　工学博士
形式仕様技術、ユビキタス・コンピューティング、セキュリティ

泉田大宗（いずみだ　とものり）
独立行政法人産業技術総合研究所テクニカルスタッフ
計算機科学

澤田寿実（さわだ　としみ）
株式会社 SRA 先端技術研究所シニア研究員
形式仕様技術、セキュリティ

井上　直（いのうえ　ただし）
株式会社 SRA 先端技術研究所チーフ研究員
コンピュータウイルス検知、侵入検出システム