情報処理学会論文誌 Vol.57 No.2 756–768 (Feb. 2016)

推薦論文

ブランチトレース機能を用いたシステムコール呼び出し元識別手法

大月 勇人[1,a)] 瀧本 栄二[1] 齋藤 彰一[2] 毛利 公一[1]

受付日 2015年5月10日, 採録日 2015年11月5日

概要：近年，マルウェアの脅威が問題となっており，その対策のためにはマルウェアの挙動を正確に解析することが重要である．最近のマルウェアは，他のプロセスに感染し，正規のプロセスやスレッドに自身のコードを実行させる．そのため，プロセスが発行するシステムコールをトレースする観測手法では，感染されたプロセスが持つ挙動とマルウェアの動作を区別して観測することは困難である．そこで，本論文では，マルウェアが感染したメモリ領域を識別し，その領域からシステムコールが発行されたことを検出可能にする手法を提案する．我々が開発しているシステムコールトレーサである Alkanet にブランチトレース機能の 1 つである Branch Trace Store を用いて呼び出し元を取得する機能を実現し，マルウェアから発行されたシステムコールを識別できることを確認した．

キーワード：マルウェア解析，システムコールトレース，ブランチトレース，仮想計算機モニタ

Identification of System Call Invoker by Branch Trace Facilities

Yuto Otsuki[1,a)] Eiji Takimoto[1] Shoichi Saito[2] Koichi Mouri[1]

Received: May 10, 2015, Accepted: November 5, 2015

Abstract: Malware has become a major security threat on computers. Malware analysis is important to enhance countermeasure for malware. Some recent malwares hide in other processes. Even if a benign process is running, the executed codes may be malicious. System call tracing, is one of conventional methods for observing malware, focuses on process or thread. The method cannot distinguish system calls invoked by the above-mentioned malwares from other system calls. Therefore, we propose a method for finding malicious regions in a memory space and detecting system calls invoked by the regions. In this paper, we describe a method for identifying a system call invoker by branch trace store. We have implemented our proposed method in our system call tracer Alkanet. We confirmed that our method could distinguish system calls invoked by malwares from other system calls.

Keywords: malware analysis, system call tracing, branch tracing, virtual machine monitor.

1. はじめに

近年，マルウェアの脅威が問題となっており，その対策が急務である．マルウェア対策のためには，マルウェアが持つ機能や挙動を正確に解析することが重要である．マルウェアの解析手法は，人手でマルウェアのコードを読み解く静的解析と，マルウェアを実際に実行して挙動を観察する動的解析に大別される．静的解析は，マルウェアの機能を網羅的に解析できるが，コードの難読化やパッキングなどの解析を妨害する技術の影響を受けやすく，解析に時間がかかる．一方で，動的解析は，比較的短時間で解析が可能であるが，マルウェアの動作を他のアプリケーションの

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[2] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466–8555, Japan
[a)] yotuki@asl.cs.ritsumei.ac.jp

本論文の内容は 2014 年 10 月のコンピュータセキュリティシンポジウム 2014 にて報告され，同プログラム委員長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．

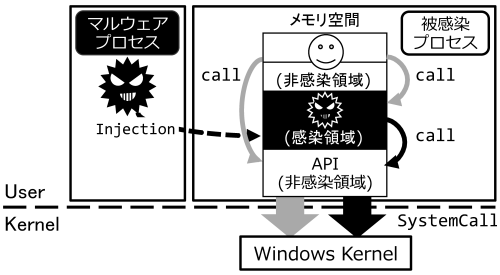

図 1 メモリ感染型マルウェア
Fig. 1 Memory-infecting malware.

動作と混交せずに解析する必要がある．我々は，動的解析をベースとして，短時間でより精度の高い解析が可能なマルウェア解析技術の確立を目指している．

マルウェアの代表的な動的解析手法であるシステムコールトレースや API トレースは，マルウェアが発行したシステムコールや呼び出した API をトレースし，そのログを解析することで，マルウェアの挙動を明らかにする．これまでに，我々は仮想計算機モニタ（VMM）をベースとするシステムコールトレーサ Alkanet [1] を開発し，動的解析を妨害するアンチデバッグ機能を持つマルウェアをスレッド単位で短時間に解析可能とするなどの成果を得た．同様の解析システムには，TTAnalyze [2]，EtherTrace [3]，Nitro [4] などがあり，これらも発行されたシステムコールや API をプロセスもしくはスレッド単位で解析することができる．

プロセスやスレッドを単位とした解析は，マルウェアが一般のアプリケーションと同様にプロセスとして動作する場合に有効である．しかし，近年のマルウェアには，**図 1** のように，他のプロセスに感染し，そのメモリ空間内に潜んで動作するものが存在する．本論文では，このような他のプロセスのメモリ空間へ侵入し，悪意あるコードの実行を他のプロセスに行わせるタイプのマルウェアをメモリ感染型マルウェアと定義する．被感染プロセスのメモリ空間では，マルウェアのコードが存在する領域（感染領域）と正常な領域（非感染領域）が混在し，どちらの領域も被感染プロセスのスレッドによって実行される．そのため，当該スレッドが発行するシステムコールは，感染領域を経由したものと非感染領域のみを経由したものが混交する．したがって，従来のプロセス・スレッド単位でマルウェアを識別する手法では解析が困難である．そこで，本論文では，我々が開発しているシステムコールトレーサ Alkanet をベースとして，さらにシステムコールが感染領域を経由して発行されたか否かをメモリ領域単位で識別可能とする手法を提案する．提案手法の技術的課題は，(1) 監視対象のシステムコールが発行されるまでの関数呼び出しの階層の取得と，(2) 感染領域の識別の 2 点となる．

課題 (1) に対する既存の解決手法として，スタックに積まれた戻りアドレスを利用したスタックトレースがある．しかし，スタックトレースは，プロセスのスタック領域がユーザ空間に存在するため，マルウェアによりスタックを改竄・偽装された場合に，正確な関数呼び出し階層を取得できない [5]．したがって，課題 (1) を解決するためには，マルウェアが改竄できない情報に基づいて関数呼び出し階層を取得する必要がある．提案手法では，近年のプロセッサに搭載されているブランチトレース機能の 1 つである BTS（Branch Trace Store）に着目し，実際に実行された分岐命令の情報から，スタックトレースと同等の関数呼び出し階層を取得する．以下，本論文では，この手法を BTS トレースと呼称する．ただし，BTS は VMM から仮想計算機（VM）内のプロセスやスレッドを観測するような用途を想定していない．また，BTS にはプロセスやスレッドを区別する機能はない．したがって，VMM として動作し，VM 内のマルウェアの動作を観測する Alkanet に，BTS トレースの機能を実現するためには，これらが実装上の課題となる．

課題 (2) については，マルウェアによるメモリの読み書きを 1 バイト単位で追跡する細粒度のテイント解析による解決が考えられる [6], [7]．しかし，その実現には，シングルステップ実行による命令単位のフックやエミュレータを用いたハードウェア拡張が必要となるため，オーバヘッドが大きくなる．提案手法では，Windows のメモリ管理用のデータ構造 VAD（Virtual Address Descriptor）および PTE（Page Table Entry）から，実行ファイルがマッピングされている領域や，ページのアクセス権限に関する情報を取得可能であることに着目する．また，ファイル作成やメモリの確保などにはシステムコールが必要であるため，これらのシステムコールを解析することで，マルウェアが作成したファイルや確保したメモリ領域を取得することができる．したがって，これらの情報を組み合わせることで，感染領域と非感染領域を識別し，課題 (2) を解決する．

以上から，提案手法の課題を解決し，Alkanet にシステムコールの呼び出し元がマルウェアであることを識別する機能を実現した．本論文では，提案手法と有効性，およびスタックトレースとの比較を行った結果について述べる．既存研究に対する本論文の貢献を以下に示す．

- BTS を用いて，VM 上のプロセスやスレッドを VMM からトレースする手法を提案したこと
- スタックトレースに代わり，スタックを偽装された場合でも正しい関数呼び出し階層を取得可能な BTS トレースを提案したこと
- 発行されたシステムコールと，VAD，PTE が持つ情報から，メモリ上に存在するマルウェアの範囲を追跡可能であることを確認したこと

以下，2 章で Alkanet と提案手法の概要について述べる．3 章で BTS による分岐記録手法，4 章で関数呼び出し階層の取得法，5 章で感染領域の識別手法を述べる．6 章では BTS トレースでスタックトレースと同等の情報を得られる

ことを示し，7 章では BTS トレースのスタック偽装耐性について述べる．8 章では感染領域を識別可能であることを示し，9 章で性能評価について述べる．そして，10 章で考察，11 章で関連研究について述べる．

2. Alkanet と呼び出し元識別機能の概要

2.1 Alkanet

Alkanet [1] は，VMM である BitVisor [8] をベースとするマルウェア動的解析システムである（**図 2** 参照）．Alkanet は，VMM ベースとすることで，多くのマルウェアに搭載されているアンチデバッグ機能を回避できるという特徴を有している．Alkanet は，マルウェアの多くが攻撃対象としている Windows のシステムコールをトレースすることができる．システムコールを観測することにより，マルウェアの挙動を機能単位で抽出し，挙動の理解を容易にしている．マルウェアの実行環境であるゲスト OS には，32 ビット版 Windows XP Service Pack 3 を用いている．システムコールのフックは，システムコールの出入口にハードウェアブレイクポイントを設定することで実現している．フック後は，レジスタやメモリの内容を解析してシステムコールの種類や引数を取得し，ログに保存する．Alkanet が生成したログはロギング用 PC から IEEE 1394 を用いて取得し，ログ解析ツールを用いて特徴的な挙動を抽出したレポートを得ることができる．

2.2 呼び出し元識別機能

2.1 節で述べた Alkanet に，システムコールのフック時にシステムコールの呼び出し元を識別する機能を追加する．当該機能は，下記の 3 つの機能で構成される．

(1) BTS を用いて VM 上のスレッドを観測する．
(2) 分岐記録から関数呼び出し階層を取得する．
(3) 感染領域を識別する．

機能 (1) は，1 章で述べた実装上の課題を解決するための機能であり，機能 (2) と (3) はそれぞれ 1 章で述べた技術的課題 (1) と (2) を解決するための機能である．

機能 (1) は，Intel 製プロセッサに搭載されているブランチトレース機能 BTS を用いて，VM 内で動作するマルウェアが実行した分岐命令の情報を取得する．機能 (2) は，機能 (1) で取得した分岐情報から，スタックトレースと同等の関数呼び出し階層を抽出し，BTS トレースを実現する．

機能 (3) は，Windows のメモリ管理用のデータ構造である VAD や PTE を用いて，マップされているファイルや，動的に生成された領域の情報を取得することで実現している．これらの情報とシステムコールの解析結果をあわせることで，マルウェアが作成した実行ファイルがマッピングされている領域や，マルウェアが書き込みを行ったメモリ領域などの感染領域を検出可能となる．以上から，感染領域からのシステムコール呼び出しの識別を可能とする．

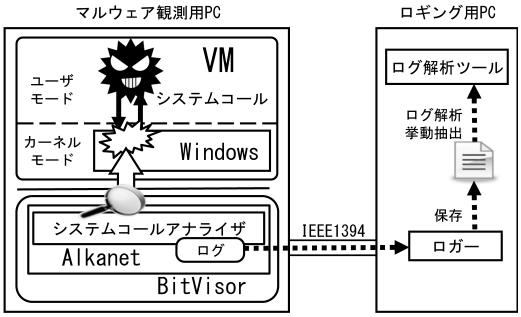

図 2 Alkanet の構成
Fig. 2 Overview of Alkanet.

Alkanet は，関数呼び出し階層の取得や VAD や PTE の情報の収集をシステムコールフック時に毎回行い，ログに記録する．挙動観測終了後，ログ解析ツールにより，感染領域の識別とマルウェアが発行したシステムコールの選別を自動的に行う．

3. BTS による VM 上のスレッドの観測

本章では，BTS の概要と，VM 上で動作するスレッドを VMM から BTS を用いて観測する手法について述べる．これは，2.2 節で述べた機能 (1) に相当する．

3.1 BTS

Intel 製のプロセッサには，発生した分岐の情報を MSR（Model Specific Register）に保存する LBR（Last Branch Record）やメモリ上に保存する BTS，分岐命令のたびにデバッグ例外を発生させる BTF（Single Step on Branches）といったブランチトレース機能が搭載されているものがある．提案手法では，分岐情報を保存する 2 つの機能のうち，保存できる分岐数を指定できる BTS を用いる．

BTS は，分岐命令の実行時に分岐元アドレス，分岐先アドレス，分岐予測の成否の 3 つの情報を指定されたカーネル空間内のバッファに記録する機能である．BTS は，Debug Store save area（DS save area）と呼ばれるデータ構造と，いくつかの MSR を用いてプロセッサのコアごとに設定が可能である．DS save area は，バッファの仮想アドレスやサイズなどを保持し，IA32_DS_AREA MSR に当該データ構造を指す仮想アドレスを設定する．BTS は，IA32_DEBUGCTL MSR の TR ビット（第 6 ビット）と BTS ビット（第 7 ビット）をセットすることで有効にすることができる．BTS_OFF_OS ビット（第 9 ビット），BTS_OFF_USR ビット（第 10 ビット）は，セットするとそれぞれ特権モード時，非特権モード時の分岐情報が記録されなくなる．さらに，BTINT ビット（第 8 ビット）をセットすることで，バッファがすべて埋まると任意の割込みを発生させることが可能となる．なお，BTINT ビットがセットされていない場合は，バッファはリングバッファとして扱われ，古い記録から上書きされる．

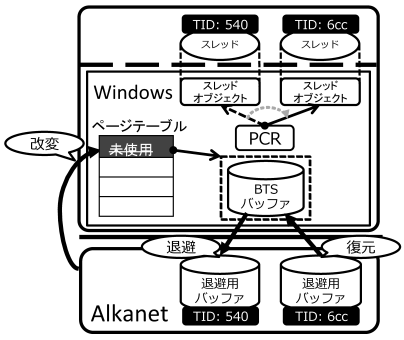

図 3 BTS 用バッファの管理
Fig. 3 Management of BTS buffers.

3.2 VM 上で発生する分岐の取得

BTS は，VMM から VM 上のプロセスを観測するような用途を想定していない．これは，BTS バッファとして，観測対象の動作中に有効な仮想メモリ領域が必要なためである．すなわち，VMM が VM 上のプロセスを観測するためには，VM 上のカーネル内に DS save area やバッファを確保する必要がある．さらに，マルウェアを観測するという観点から，BTS 関連の制御が観測対象である VM から透過的に行われる必要がある．そこで，Alkanet は，Windows 内部のデータ構造を書き換えて，BTS バッファ用のページと物理メモリ領域の設定を行うことで，VMM による透過的な BTS 利用を実現する．

図 3 に，提案手法における BTS バッファの管理機能の構成を示す．本節では，図中の BTS バッファについて述べ，退避用バッファについては 3.3 節で述べる．Alkanet は，Windows のカーネル空間内で未使用の仮想メモリ領域に対応する PTE を書き換えて，BTS バッファ用に確保した物理メモリ領域を指すように設定する．さらに，Windows が物理アドレスを管理するデータ構造である MmPfnDatabase を書き換えて，BTS バッファ用の物理メモリ領域を使用できない領域として認識されるようにする．

確保したメモリ領域を BTS バッファとして設定し，BTS を有効化するために VM 内の MSR を設定する．Alkanet は，Intel 製プロセッサの仮想化支援機能である Intel Virtualization Technology を利用している．そのため，VM 上の IA32_DEBUGCTL は，VMCS（Virtual-Machine Control Structure）で設定可能である．IA32_DS_AREA は，VMCS に設定項目がないため物理 MSR に設定する．以上の設定により，Windows 上で発生した分岐の情報を BTS バッファに記録することが可能となる．また，本論文では，システムコールの呼び出し元を特定する目的で BTS を使用するため，OS 内部で発生する分岐については記録しない．したがって，BTS_OFF_OS ビットをセットし，ユーザモードで発生する分岐のみを記録する．なお，バッファの確保や MSR の設定は，プロセッサのコアごとに行う．

3.3 スレッドごとの分岐記録の取得

BTS にはスレッドやプロセスを区別する機能はないため，複数のスレッド・プロセスで発生した分岐の情報が混交してしまう．マルウェアが実行した分岐命令の情報を取得するために，スレッドごとに分けて分岐情報を記録する機能を実現する．具体的には，Alkanet は，BTS 用のバッファをスレッドごとに管理し，コンテキストスイッチが発生した場合にバッファの退避および復元を行う．

当該機能の実現には，Windows で発生するコンテキストスイッチをフックする必要がある．Windows では，PCR（Processor Control Region）というデータ構造内に現在動作中のスレッドオブジェクトのアドレスを持つメンバがある．Alkanet は，ハードウェアブレイクポイントを用いて当該メンバの書き換えを検出することでフックを実現する．

Alkanet は，スレッドごとに退避用バッファを自身のメモリ領域に確保し，スレッド ID（TID）をもとにスレッドと紐付けて管理する．Windows における TID は他のプロセスの持つスレッドも含めて重複しないため，TID のみでスレッドを一意に識別可能である．ただし，スレッドの終了後に TID が再利用される場合があるため，バッファ復元時には，プロセス ID（PID）やスレッドオブジェクトのアドレスなどを確認し，バッファ退避時のスレッドと現在のスレッドが同一であることを確認する．図 3 では，Windows 上で TID 540 のスレッドから TID 6cc のスレッドへの切替えが発生している．このとき，Windows 内部にある BTS バッファには TID 540 のスレッドが実行した分岐が記録されている．Alkanet は，BTS バッファにある分岐記録を TID 540 の退避用バッファへ退避し，TID 6cc の退避用バッファの分岐記録を BTS バッファに復元する．以上により，別のスレッドの動作と混交せずに，各スレッドについて分岐記録を保持していくことが可能となる．

4. 関数呼び出し階層の抽出

3.2 節，3.3 節により，Windows 上で動作するスレッドが実行する分岐命令の記録が可能となる．これを用いて関数呼び出し階層を抽出することで，システムコールの呼び出し元となる関数のアドレスを特定する．本章では，分岐記録から関数呼び出し階層を得る手法を述べる．これは，2.2 節で述べた機能 (2) に該当する．

図 4 は，関数呼び出しのフローとこのときの BTS バッファを例示したものである．ただし，実際の BTS バッファには call 命令や ret 命令以外の分岐命令による分岐も記録されている．また，分岐命令そのものは記録されていないが，分岐元アドレスを参照することで取得できる．

図 4 では，func B が感染領域内の関数であり，func A からシステムコールが発行されたことを想定している．Alkanet は，func A から発行されたシステムコールをフック後，BTS バッファから当該スレッドが実行した分岐命

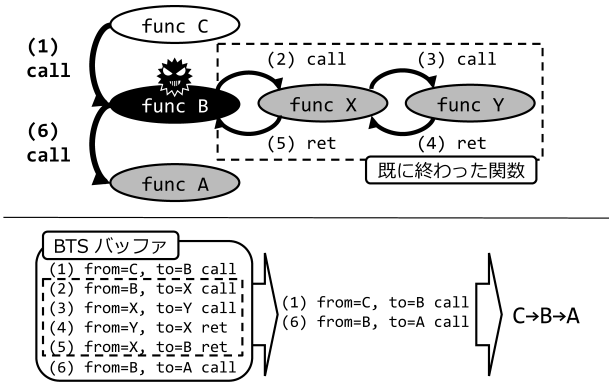

図 4 BTS から関数呼び出し階層の取得
Fig. 4 Extracting a call hierarchy from BTS.

令の情報を取得する．この分岐記録には，すでに終わった関数呼び出しも含まれている．そこで，分岐記録内の call 命令と ret 命令による分岐の対応付けを行い，まだリターンされていない関数呼び出しのみを取り出すことで，現在実行中の関数呼び出し階層を取得する．図 4 の例では，分岐 (3) と分岐 (4)，および分岐 (2) と分岐 (5) が対応付けられるため，分岐 (1) と分岐 (6) が残る．これにより，func C から func B を経由して，func A が呼び出されたことが分かる．これはスタックトレースで得られる情報と同等である．

func B から呼び出される図中に灰色で示す関数（func X，func Y，func A）は，それら自身はマルウェアの関数ではない．しかし，感染領域内の関数である func B から呼び出されているため，これらの関数もマルウェアの動作を構成するものである．したがって，func B だけでなく，func X，func Y，func A から発行されるシステムコールは，すべてマルウェアによるシステムコールとして検出する．

5. 感染領域の識別

4 章で述べた手法により，システムコールに至るまでに経由した関数のアドレスの取得を可能にする．ただし，実際に図 4 の func B が感染領域内にあることを知るためには，メモリ空間に存在する感染領域を特定する機能が必要である．本章では，本機能を実現する手法について述べる．なお，本機能は，2.2 節で述べた機能 (3) に該当する．

Alkanet は，マルウェアが生成したプロセスやスレッドを追跡する機能を持つ [1]．本論文では，これに加えて感染領域を識別可能とする機能を追加する．マルウェアのコードがメモリ上に存在するとき，ファイルとしてマッピングされる場合と実行時に生成される場合が考えられる．本章では，これらの領域の検出方法をそれぞれ述べる．

5.1 マッピングされた実行ファイル

Windows は，VAD と呼ばれるデータ構造を用いて，プロセスが確保したメモリ領域を管理している [9]．VAD は，プロセスがメモリを確保するときやファイルをマッピングするときに作成され，管理する範囲やマッピングされているファイルの情報などを保持する．各 VAD は，同じプロセスのアドレス空間の一部を管理する他の VAD と連結され，平衡二分探索木を構成する．また，プロセスオブジェクトは，自身のアドレス空間を表す VAD ツリーのルートノードへのポインタを持つ．したがって，VAD ツリーをたどることで，プロセスのメモリ空間にマッピングされているファイルの情報を取得することができる．

さらに，マッピングされているファイルがマルウェアであることを確認するため，マルウェアが作成したファイルの情報を取得する．プロセスが，ファイルを作成するためには NtCreateFile システムコールが必要であり，ファイルの読み書きには NtReadFile システムコールや NtWriteFile システムコールが必要である．したがって，これらのシステムコールを解析することで，マルウェアが作成したファイルを取得することができる．以上から，VAD から得たメモリマップと，発行されたシステムコールから得たマルウェアが作成したファイルの情報からマルウェアのファイルがマッピングされている領域を特定できる．

5.2 動的に生成されたコード

マルウェアは，パッキングされたコードの展開や，他プロセスへのコードインジェクションを行うなど，しばしば動的にコードを生成する．マルウェアが動的にコードを生成する場合，NtAllocateVirtualMemory システムコールなどを用いて，コード生成先のメモリ領域を確保する．その領域は，実行可能かつ書き込み可能である必要があるため，必要に応じて，NtProtectVirtualMemory システムコールを用いて領域の保護属性を変更する．また，別のプロセスへのコード挿入の場合は，NtWriteVirtualMemory システムコールが必要である．したがって，これらのシステムコールを追跡することで，マルウェアが動的に確保したメモリ領域や別のプロセスへ挿入したコードの範囲を取得することが可能である．

ただし，すでに確保されているメモリ領域やファイルがマッピングされている領域にコードが書き込まれた場合，この挙動を Alkanet では観測できない．これは，プロセス内でのメモリ書き込み自体にはシステムコールを必要としないためである．そこで，PTE の保護属性に着目し，書き込みが行われた領域をページ単位で検出する．ページが書き込み可能であることは，PTE の Writable ビット（第 1 ビット）から判定でき，実際に書き込まれたことは Dirty ビット（第 6 ビット）によって判定できる．これらの情報から，少なくともページに実際に書き込みがされたことを確認できる．したがって，上記の特徴を持つページに存在する関数が関数呼び出し階層に含まれていれば，実行時に生成されたコードが実行されたとして検出する．

表 1 評価用 PC の構成
Table 1 Specification of PC for evaluation.

CPU	Intel Core 2 Quad Q6600 2.4 GHz
ゲスト OS	Windows XP SP3（32-bit）
Memory	4 GB

なお，マルウェアがコード生成後に書き込み可能属性を取り除くことが考えられる．この場合でも，NtProtectVirtualMemory システムコールを発行する必要があるため，Alkanet でこの挙動を捕捉することができる．

6. スタックトレースとの比較評価

提案手法の有効性を検証するために，評価用プロトタイプを作成して評価を行った．具体的には，BTS トレースがスタックトレースと同等の関数呼び出し階層を取得できることを確認し，マルウェアから発行されたシステムコールを識別可能であることを示す．本章では，その評価結果を述べる．

6.1 評価環境

本評価は，**表 1** に示す評価用 PC で行った．また，評価用プロトタイプは，提案手法である BTS トレースに加え，評価用にスタックトレースを用いたシステムコール呼び出し元識別機能も追加実装したものである．さらに，両トレースの結果を比較する機能も追加している．なお，上記の評価環境およびプロトタイプは，7 章，8 章，9 章の評価でも共通である．

6.2 評価手法と検体

メモリ感染型マルウェアが正規プロセスのメモリ空間に侵入する方法の 1 つに，DLL（Dynamic Link Library）をロードさせる方法がある．そこで，指定された DLL を読み込んで実行するコマンド rundll32.exe に，DLL タイプのマルウェアをロードさせることで，メモリ感染型マルウェアに感染された状況を再現する．本評価では，BTS トレースがスタックトレースと同等の結果を得られることを確認し，上記の状態の rundll32.exe の挙動からマルウェアの挙動を区別できることを示す．

検体は，MWS Datasets 2014 [10] に含まれる CCC DATAset 2013 に活動が記録されているマルウェアのうち，DLL 検体を選択した．マルウェアのファイル名は，アンチウイルスソフトでの検出名から Conficker.dll とした．

6.3 ログエントリ

図 5 は，6.4 節で述べる Conficker.dll の解析で得たログの一部である．“[]” 内の数字はスタックフレームの深さを示し，スタックから取得した戻りアドレスとスタックフレームについての情報が続く．戻りアドレスについては，

```
[00] 7c94d1fc (API: NtDelayExecution+0xc,
     Writable: 0, Dirty: 0,
     VAD: {7c940000--7c9dc000, ImageMap: 1,
     File: "\WINDOWS\system32\ntdll.dll"}),
     From: 7c94d1fa, Valid: YES, SP: 7ed24
<略>
[02] 7c802455 (API: Sleep+0xf,
     Writable: 0, Dirty: 0,
     VAD: {7c800000--7c933000, ImageMap: 1,
     File: "\WINDOWS\system32\kernel32.dll"}),
     From: 7c802450, Valid: YES, BP: 7ed7c
[03] 10003898 (API: -, Writable: 0, Dirty: 0,
     VAD: {10000000--10018000, ImageMap: 1,
     File: "\Conficker.dll"}),
     From: 10003892, Valid: YES, BP: 7ed8c
[04] 1000401b (API: -, Writable: 0, Dirty: 0,
     VAD: {10000000--10018000, ImageMap: 1,
     File: "\Conficker.dll"}),
     From: 10004016, Valid: YES, BP: 7f184
[05] 7c94118a (API: LdrpCallInitRoutine+0x14,
     Writable: 0, Dirty: 0,
     VAD: {7c940000--7c9dc000, ImageMap: 1,
     File: "\WINDOWS\system32\ntdll.dll"}),
     From: 7c941187, Valid: YES, BP: 7f1a4
<略>
[10] 7c80aeec (API: LoadLibraryW+0x11,
     Writable: 0, Dirty: 0,
     VAD: {7c800000--7c933000, ImageMap: 1,
     File: "\WINDOWS\system32\kernel32.dll"}),
     From: -, Valid: UNKNOWN, BP: 7f888
[11] 1001792 (API: -, Writable: 0, Dirty: 0,
     VAD: {1000000--100b000, ImageMap: 1,
     File: "\WINDOWS\system32\rundll32.exe"}),
     From: -, Valid: UNKNOWN, BP: 7f89c
<略>
```

(a)

(b)

図 5 Conficker.dll によるシステムコール
Fig. 5 System call invoked by Conficker.dll.

下記の情報を出力する．

API マッピングされているファイルのシンボル情報から得た API 名とその API の先頭アドレスからのオフセットを示す．シンボル情報が取得できなかった場合には “-” を表示する．

Writable ページが書き込み可能か否かを示す．

Dirty ページがすでに書き込まれているか否かを示す．

VAD VAD の管理するアドレス範囲，実行ファイルをマッピングしているか否か（ImageMap），マッピングしている場合はそのファイル名（File）を示す．

さらに，スタックトレースと BTS トレースの結果を階層の深さに基づいて対応付けを行い，From と Valid を追加した．From は，BTS より取得した分岐元アドレスを示す．また，Valid は，分岐元アドレスとスタックトレースで取得した戻りアドレスの整合性を示す．整合なら YES，不整合なら NO を出力する．なお，本プロトタイプでは，BTINT の機能を使用していないため，分岐記録が上書きされて整合性の確認ができないケースが発生する．その場合には，UNKNOWN と表示する．

図 5 の [00] のエントリでは，戻りアドレスが 0x7c94d1fc で，この値はスタック内の 0x7ed24 から取得した．戻り先のページには，書き込み可能属性はなく（Writable: 0），0x7c940000 から 0x7c9dc000 の範囲を管理する VAD に属する．この領域には，実行ファイルがマッピングされており（ImageMap: 1），そのファイルは “\WINDOWS\system32

\ntdll.dll" である．そして，この戻りアドレスは，ntdll.dll 内の NtDelayExecution という API の先頭アドレスから +0xc の位置である．戻りアドレスに対応する呼び出し元アドレス From は 0x7c94d1fa である．なお，当該呼び出し元アドレスにある call 命令は，"call dword ptr [edx]" (0xff12) である．戻りアドレスは，呼び出し元アドレスに当該 call 命令の命令長 2 バイトを足した値となっており，整合している（Valid: YES）といえる．

6.4 評価結果

図 5 に，本評価で取得したログを示す．図 5 の [11]，[10] は，rundll32.exe が LoadLibraryW API を呼び出したことを示す．さらに，[05]，[04] より，LoadLibraryW API の内部ルーチンである LdrpCallInitRoutine から，Conficker.dll が呼び出されていることが分かる．したがって，rundll32.exe が LoadLibraryW API でロードした DLL は，Conficker.dll であることが分かる．続いて，[04]～[00] より，Conficker.dll は Sleep API を呼び出し，NtDelayExecution システムコールの発行に至ったことが分かる．これにより，当該システムコールは，Conficker.dll から呼び出されたものであるといえる．

図 5 の [05]～[00]（図中 (a)）において，BTS より取得した分岐元アドレスと，スタックトレースにより取得した戻りアドレスが整合している（Valid: YES）．一方，[11]，[10] を含む深い位置にある戻りアドレス（図中 (b)）については，分岐記録が上書きされていたため，確認できなかった（Valid: UNKNOWN）．この課題は，BTINT による通知と BTS 用のバッファの再確保によって解決が可能である．以上から，バッファサイズの問題はあるものの，BTS トレースは，スタックトレースと同等の結果を得ることが可能であり，システムコールの呼び出し元を取得することが可能であることが確認できた．

7. スタック偽装耐性の評価

本章では，BTS トレースが，スタックトレースと異なり，スタックを偽装された場合でも正しい関数呼び出し階層を得られることを示す．

7.1 スタック偽装方法

本節では，マルウェアがシステムコールの呼び出し元を偽装・隠蔽するためにスタックを偽装する方法を述べる．**図 6** に，マルウェアが API X を利用して，システムコールの発行に至った場合のスタックの一部を示す．図 6 では，func A，B，C が感染領域にある関数であり，API X と func Y は，非感染領域に存在する．Windows XP Service Pack 2 以降では，Windows のすべての DLL と実行可能ファイルにおいて FPO（Frame-Pointer Omission）は無効になっている [11]．したがって，API X は必ずフレームポインタをスタックに積むため，API X を呼び出した func A への戻りアドレスはスタックトレースで必ず取得可能である．以上から，感染領域内を経由してシステムコールが発行されたことを隠蔽や偽装するためには，下記に該当する戻りアドレスを隠蔽する必要がある．

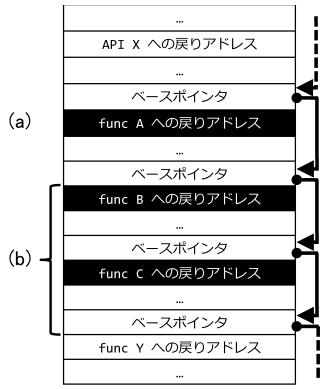

図 6 マルウェアが API を呼び出したときのスタック
Fig. 6 Stack when malware called API.

(a) API X を呼び出した関数（func A）への戻りアドレス
(b) マルウェア内部での関数呼び出しで積まれた戻りアドレス（func B，C への戻りアドレス）

(a) の隠蔽を実現する方法として，非感染領域に存在する FPO が有効な関数を経由して API を呼び出す方法や，偽の戻りアドレスをスタックに積んだ後，call 命令以外の分岐命令によって API を呼び出す方法が考えられる．なお，後者の場合，マルウェアが動作を継続するために，偽の戻りアドレスを経由していずれは本来の呼び出し元へ戻れるようにしておく必要がある．(b) の隠蔽を実現する方法として，FPO やフレームポインタの改竄により，func B，C のスタックフレームをスキップさせる方法や，まったく別のスタックに切り替える方法が考えられる．

7.2 サンプルプログラムの動作

BTS トレースの有効性を検証するために，呼び出し元を偽装して API を呼び出す機能を持つ FPOTest.dll を作成した．FPOTest.dll は，ロードされるとマイドキュメントに "out.txt" を作成し，"Hello, World!" を出力する．本評価は，6 章の評価と同様に FPOTest.dll を rundll32.exe にロードさせることで行う．

FPOTest.dll は，FPO オプション（/Oy）[12] を有効にしてコンパイルした．これにより，7.1 節の (b) の隠蔽を実現する．さらに，7.1 節の (a) の隠蔽は，偽の戻りアドレスをスタックに積む手法を用いて実現する．FPOTest.dll は，FPOTest.dll 内に定義した ApiCallWrapper 関数を経由して Windows API を呼び出す．API 呼び出し時の動作の流れを下記に示す．

(1) FPOTest.dll をロードした rundll32.exe 内を探索し，ret 命令を発見する．

```
[00] 7c94d09c API: NtCreateFile+0xc
     VAD: "\<略>\ntdll.dll" (Valid: YES)
[01] 7c8109a6 API: CreateFileW+0x1b6
     VAD: "\<略>\kernel32.dll" (Valid: YES)
[02] 1001c58  API: -
     VAD: "\<略>\rundll32.exe" (Valid: NO)
     bts[0681] from =10001094 (-, "\<略>\FPOTest.dll")
               to   =100010a0 (-, "\<略>\FPOTest.dll")
[03] 7c95c4da API: LdrpRunInitializeRoutines+0x205
     VAD: "\<略>\ntdll.dll" (Valid: NO)
     bts[0242] from =1000104f (-, "\<略>\FPOTest.dll")
               to   =10001060 (-, "\<略>\FPOTest.dll")
[04] 7c956351 API: LdrpCheckForLoadedDll+0x4a1
     VAD: "\<略>\ntdll.dll" (Valid: NO)
     bts[0241] from =10001143 (-, "\<略>\FPOTest.dll")
               to   =10001040 (-, "\<略>\FPOTest.dll")
[05] 7c9564b3 API: LdrLoadDll+0x110
     VAD: "\<略>\ntdll.dll" (Valid: NO)
     bts[0240] from =100011f6 (-, "\<略>\FPOTest.dll")
               to   =10001120 (-, "\<略>\FPOTest.dll")
[06] 7c801bbd API: LoadLibraryExW+0xc8
     VAD: "\<略>\kernel32.dll" (Valid: NO)
     bts[0238] from =7c941187 (LdrpCallInitRoutine+0x11,
                              "\<略>\ntdll.dll")
               to   =100011f0 (-, "\<略>\FPOTest.dll")
<略>
```

図 7 FPOTest.dll による NtCreateFile システムコール
Fig. 7 NtCreateFile system call invoked by FPOTest.dll.

(2) 本来の戻りアドレスをスタックに積む．
(3) API に渡す引数をスタックに積む．
(4) (1) で発見したアドレスをスタックに積む．
(5) jmp 命令で API へ遷移する．
(6) API 実行後，(4) で積んだ戻りアドレスにより，rundll32.exe 内の ret 命令へ遷移する．
(7) rundll32.exe 内の ret 命令は，(2) で積んだ戻りアドレスを用いて，ApiCallWrapper へ戻る．
(8) ApiCallWrapper から呼び出し元へ戻る．

7.3 評価結果

図 7 は，rundll32.exe により NtCreateFile システムコールが発行されたときの関数呼び出し階層である．図 7 は，6.3 節で述べた ImageMap や Writable などの一部の項目を省略しているが，BTS から得られた関数呼び出し階層を示すために分岐記録の情報を追加している．Valid は，6.3 節で述べたものと同様に，スタックの戻りアドレスと BTS トレースにより取得した呼び出し元アドレスとの比較結果を示す．ただし，Valid: NO の場合，対応付けた分岐記録についての情報もあわせて出力する．図 7 では，`[02]` 以降の戻りアドレスにおいて，Valid: NO となっており，分岐記録に存在する call 命令による分岐から期待される値と一致していない．`[02]` の戻りアドレスは，`bts[0681]` の分岐記録と対応付けられている．`bts[0681]` は，BTS バッファの 681 番目の分岐記録である．各分岐記録には，分岐元アドレス（from）と分岐先アドレス（to）を出力している．各アドレスについては，戻りアドレスと同様に API 名とファイルの情報を付加している．

`bts[0681]` の分岐記録は，FPOTest.dll 内部の 0x10001094 から，FPOTest.dll 内部の 0x100010a0 への遷移を示す．スタックトレースで得た `[02]` の戻りアドレスである 0x1001c58 は，rundll32.exe 内のアドレスであり，FPOTest.dll の ApiCallWrapper が呼び出し元偽装のためにスタックに積んだものである．`[03]` 以降のスタックトレースの結果でも，FPOTest.dll が呼び出されたことは確認できなかった．一方，BTS トレースで取得した関数呼び出し階層では，FPOTest.dll が出現しており，FPOTest.dll を経由してシステムコールに至ったことが分かった．したがって，スタックトレースは，FPOTest.dll により偽装された関数呼び出し階層を取得してしまったのに対し，BTS トレースは正しい呼び出し階層を取得できている．以上から，BTS トレースは，スタックトレースと異なり，スタックを偽装された場合でも正しい関数呼び出し階層を取得可能であることが確認できた．

8. 感染領域識別機能の評価

本評価では，実際に他プロセスのメモリに侵入する挙動を持つマルウェアを用いて，5 章で述べた感染領域の識別手法の有効性を示す．具体的には，メモリ感染型マルウェアが他プロセスに感染する挙動を観測し，その感染領域から発行されたシステムコールを識別可能であることを確認する．

8.1 評価手法と検体

本評価で用いたマルウェア検体 c13.exe は，マルウェア対策人材育成ワークショップ（MWS）組織委員会が提供準備を進めている研究用データセット「動的活動観測 2014」[13] 内で観測されていたものと同等のものである．当該検体は，多くのアンチウイルスソフトウェアで PlugX として検出されたものである．PlugX は，Windows の持つ実行ファイルである svchost.exe を起動し，メモリにマップされたエントリポイントを改竄することで，挿入したコードを実行させる [14]．実行ファイルそのものの改竄や新たなスレッドの挿入を行わないため，従来のプロセスやスレッドを単位とする観測手法では，正規のコードとマルウェアのコードを区別できず，無害なファイルが実行されただけにみえる．

PlugX の挙動を解析するためには，(1) svchost.exe プロセス内に挿入された感染領域を識別し，(2) その領域から発行されるシステムコールを識別する必要がある．当該感染領域は，PlugX が svchost.exe プロセスに対して行ったメモリ領域の確保や，当該領域への実行権限の付与やコードの書き込みに相当する挙動を観測することで検出できる．また，関数呼び出し階層に感染領域内の関数が含まれることを確認することで，システムコールの識別も可能である．なお，これらの識別ができれば，PlugX が挿入した感染領域から拡大する感染領域も追跡・識別が可能である．したがって，本評価では，上記 (1)，(2) の 2 点を確認事項

```
11008: (32c.584) starter.exe NtCreateProcessEx
       File:<略>\svchost.exe, Pid:5c8, Name:svchost.exe
11068: (32c.584) starter.exe NtCreateThread
       Pid:5c8, Name:svchost.exe, Tid:5b0, <略>, Suspended:TRUE
11115: (32c.584) starter.exe NtAllocateVirtualMemory
       Pid:5c8, Name:svchost.exe, <略>
       BaseAddress:0x90000, AllocationSize:0x1d000
11128: (32c.584) starter.exe NtWriteVirtualMemory
       Pid:5c8, Name:svchost.exe,
       BaseAddress:0x90000, BufferLength:0x1c7f5, <略>
-------------------------------------------------------
12139: (5c8.5b0) svchost.exe NtAllocateVirtualMemory
       Pid:5c8, Name:svchost.exe, <略>
  <略>
  [03] 90234 (API: -, Writable: 1, Dirty: 1,
       VAD: {90000--ad000, ImageMap: 0}),
       From: 90231 (Valid: YES), BP: 7fea0
  [04] 90038 (API: -, Writable: 1, Dirty: 1,
       VAD: {90000--ad000, ImageMap: 0}),
       From: - (Valid: UNKNOWN), BP: 7ff94
```

図 8 svchost.exe へのコードインジェクション
Fig. 8 Code injection into svchost.exe process.

とし，これらを提案手法で識別可能であることを示す．

8.2 評価結果

検体 c13.exe は，Temp フォルダに starter.exe を作成し，この starter.exe を PID 32c のプロセスとして起動した．**図 8** の破線より上は，starter.exe が svchost.exe を起動し，その内部にコードインジェクションを行う挙動を示し，破線より下は感染された svchost.exe の挙動を示す．図中の行頭の数字は，ログの番号を示す．先頭に表示しているログ番号 11008 のログでは，“(32c.584)” は，PID 32c，TID 584 であることを示し，“starter.exe” はプロセス名，“NtCreateProcessEx” が発行されたシステムコールの名前である．その後は，引数から得た情報が続く．当該ログの場合，“`<略>\svchost.exe`” が PID 5c8 のプロセスとして起動されたことを示す．続く 11068 の NtCreateThread システムコールを用いてスレッドを作成しているが，この段階では作成された TID 5b0 のスレッドは，中断状態である（“Suspended:TRUE”）．破線より下に示したログ 12139 には，6 章で示したログと同様に，システムコールフック時の関数呼び出し階層も表記している．

11115 や 11128 のログは，起動された svchost.exe プロセスのメモリ空間にメモリ領域の確保と書き込みを行っていることを示している．11115 のログは，starter.exe が NtAllocateVirtualMemory システムコールを用いて，svchost.exe のメモリ空間上のアドレス 0x90000 から 0x1d000 バイトの領域を確保していることを示す．さらに，11128 のログは，11115 で確保されたメモリ領域に書き込みが行われたことを示す．また，上記の領域に対して，実行権限と書き込み権限を付与するために，NtProtectVirtualMemory システムコールが複数回発行されていたことも確認した．以上のことから，提案手法は当該領域を感染領域として検出する．このほか，さらなるメモリ確保や書き込み，メモリの保護権限の変更などが行われた後，NtResumeThread システムコールによって svchost.exe の TID 5b0 のスレッドが動作を開始する．なお，svchost.exe が起動されるまでに当該実行ファイルを PlugX が改竄するような挙動はなかった．以上から，上記の挙動は PlugX の持つ svchost.exe に感染する挙動であり，挿入された感染領域は 0x90000 から 0xad000 であることが確認できた（確認事項 (1)）．

破線より下のログ 12139 が示す NtAllocateVirtualMemory システムコールは，starter.exe が発行した 11068 のシステムコールにより作成されたスレッド（TID 5b0）から発行されたものである．前述したように実行ファイルである svchost.exe には改竄された痕跡はなく，スレッドも通常のプロセス生成の一環で作成されたものである．したがって，プロセス単位およびスレッド単位でのマルウェアの区別では，このシステムコールをマルウェアによるものであると判定することは困難である．

関数呼び出し階層の `[03]` および `[04]` のエントリは，0x90000 から 0xad000 のメモリ領域にある関数を経由したことを示している．このメモリ領域は，11115 や 11128 のログが示すようにマルウェアである starter.exe により確保・書き込みされた感染領域である．また，PTE から得た情報から，書き込み可能な領域であり（Writable: 1），実際に書き込みが行われた領域である（Dirty: 1）ことも確認できる．少なくとも `[03]` は Valid: YES であるため，この領域を経由してシステムコールが発行されてきたことは確実である．したがって，確認事項 (2) についても確認できた．以上から，システムコールの結果と VAD や PTE から得られるメモリ領域の情報から，感染領域を識別可能であることが確認できた．したがって，提案手法は，感染領域から発行されたシステムコールを識別可能である．

9. パフォーマンスの評価

BTS の使用がパフォーマンスへ与える影響を確認するために，6 章で用いたプロトタイプを PCMark05 [15] の System Test Suite を用いて評価を行った．システムコールトレース機能のみの Alkanet は，Windows を直接ハードウェア上で実行した場合の 75%程度のスコアであった．これに対し，提案手法を適用した Alkanet は，HDD 系のテストを除き，10%以下のスコアであった．すなわち，提案手法を適用した Alkanet では，処理時間が通常の 10 倍相当となった．6 章で用いたプロトタイプは，スタックトレースによる戻りアドレスの取得機能および，その結果を提案手法と比較する機能を有している．そのため，上記の性能低下には提案手法のオーバヘッド以外にこれらの機能のオーバヘッドも含んだものとなっている．しかし，このスコアは改変なしの QEMU [16] と同程度の性能であった．すなわち，QEMU にシステムコールトレースやテイント解析の機能を追加した解析システムより高速に動作可能であるといえる．

10. 考察

10.1 誤検知や見逃しに関する考察

提案手法で誤検知・見逃しが発生するケースとして，正しい関数呼び出し階層が得られないケースと，ページ内に感染領域と非感染領域が混在するケースの 2 つがあげられる．以下，それぞれのケースについて考察する．

10.1.1 正しい関数呼び出し階層が得られないケース

BTS トレースは，典型的な関数の呼び出しおよび関数からのリターンが call 命令と ret 命令によって行われることに着目している．しかし，特殊なプログラミングテクニックやコンパイラによる最適化などにより，call 命令と ret 命令が対応付けられないケースが発生する．当該ケースでは，正しい関数呼び出し階層が得られない，あるいは関数呼び出し階層自体が存在しないため，誤検知や見逃しが発生しうる．実際に，特殊なフローを多く含む ROP (Return-Oriented Programming) [17] やシェルコードを含む検体で検証したところ，スタックトレース，BTS トレースともに期待する関数呼び出し階層が得られなかった．

7.3 節で述べた評価においても，図 7 に示したシステムコールの次のシステムコールをフックした際に，call 命令の呼び出し元と ret 命令の戻り先が一致しないペアが検出された．これは，ApiCallWrapper の戻りアドレス偽装機能により，API の呼び出しが jmp 命令で行われること，API 内の ret 命令で呼び出し元に直接戻らないこと，呼び出し元へ戻るために ret 命令が余分に実行されることが原因である．しかし，この場合，call 命令と ret 命令の齟齬は，API から FPOTest.dll へ戻るフローにおいて発生する．FPOTest.dll からシステムコールに至るまでの関数呼び出し階層は問題なく取得できるため，FPOTest.dll から発行されたシステムコールの識別自体は可能である．

しかし，感染領域からシステムコールに至るまでの関数呼び出しにおいて上記のような齟齬が発生した場合には，正しい関数呼び出し階層が得られない可能性がある．call 命令と ret 命令の不一致が発生する要因として，カーネルからのコールバックや，意図的なスタックの変更，関数呼び出し・リターン以外を目的とする call 命令・ret 命令の使用，他の分岐命令を用いた関数呼び出し・リターンなどがあげられる．カーネルからのコールバックは，開始と終了時にカーネルとプロセス間の遷移を検出することで対処が可能である．一方，それ以外のものは，ユーザ空間で行われ，かつ，さまざまな実装が考えられるため，検出は困難である．

提案手法の本来の目的は，システムコールの呼び出し元がマルウェアであることを判定することにある．したがって，必ずしも厳密な関数呼び出し階層を得る必要はない．そこで，分岐記録の中から，メモリ領域間のフローを取得し，関数呼び出し階層を補正することで上記課題の解決を図る．シンボルが提供されている実行ファイルや DLL については，関数間の遷移を取得できる．一方，シンボルのない実行ファイルや動的に生成された領域では，VAD の範囲間での遷移を取得する．取得した関数間や VAD の範囲間のフローと BTS トレースで得た関数呼び出し階層との矛盾を検出し，補正を図る．また，感染領域と非感染領域間のフローを検出できるため，少なくともマルウェアからそれ以外に処理が移ったことを検出できると考えられる．

10.1.2 ページ内に感染領域と非感染領域が混在するケース

提案手法が感染領域として検出する最小粒度は，1 ページ単位である．そのため，単一ページの中にマルウェアのコードとそうでないコードが混在すると，誤検知が発生する可能性がある．当該ケースは，正規の実行ファイルがマップされている領域を書き換えられた場合に発生する．この課題については，実行されたページにあるコードを取得し，ファイルのマップ時などに取得した書き換え前コードと比較することで正確な範囲を検出できると考えられる．

ただし，既存のコードを書き換えてシステムコールを発行しうるコードを挿入し，動作させることは困難である．したがって，このようなケースはないと考えられる．

10.2 他の解析システムへの適用可能性

本論文では，Alkanet を拡張し，BTS トレース機能と感染領域識別機能を実現し，有効性を検証したが，提案手法としては Alkanet に依存したものではない．Intel VT を用い，システムコールフック時にスタックトレースを行う解析システムであれば，提案手法を適用することで，スタックが改竄された場合における信頼性の向上が期待できる．

ただし，提案手法を実装するときには，下記の 4 つの機能はゲスト OS に依存した実装が必要である．

- バッファに使用する物理メモリ領域を使用不可としてゲスト OS に認識させる機能
- コンテキストスイッチをフックする機能
- 実行中にプロセスのメモリマップを取得する機能
- 発行されたシステムコールからメモリ領域の確保や，保護属性の変更，他プロセスのメモリ空間に書き込みなどを検出する機能

本論文では，Windows XP Service Pack 3 を対象とする実装を述べたが，そのデータ構造やシステムコールの大部分は，他のバージョンの Windows でも共通である．そのため，バージョン間の細かな差異を吸収することで，他のバージョンの Windows でも提案手法の適用は可能である．また，その他の OS についても，上記の機能を実装することで提案手法を実現できる．

10.3 パフォーマンスのさらなる改善方法

3.1 節で述べたように，Intel 製プロセッサには BTS の同種の機能として LBR がある．LBR を活用して ROP の

検出を行う kBouncer [18] のオーバヘッドが数%であることから，LBR は BTS と比べて性能低下が小さく，高速な解析に向いていると考えられる．また，Haswell アーキテクチャ以降のプロセッサでは，EN_CALLSTACK オプションにより，リターンしていない関数呼び出しの記録のみを LBR に残すことができる．以上から，BTS の代わりに EN_CALLSTACK を有効にした LBR を用いることで，システムコールの呼び出し元識別手法の高速化が期待できる．ただし，記録できる分岐の数が MSR の個数に制約されるため，深さが 16 までの関数呼び出しまでしか取得できない．そのため，関数呼び出し階層が 16 階層を超える場合における誤検出や見逃しについて検討する必要がある．

10.4 マルウェアへの露顕性に関する考察

BTS の設定は，MSR を介して行われるため，マルウェアが MSR を確認することができれば提案手法が検知される可能性がある．これについては，MSR に対する読み書きを Intel VT の機能でフック可能なことを利用し，BTS が使用する MSR の値をゲスト OS から隠蔽することで対処できる．また，BTS のバッファはカーネル空間に確保されるため，ユーザモードで動作するマルウェアからは検出できない．以上から，提案手法がマルウェアに検出されることはない．

11. 関連研究

11.1 呼び出し元を識別する手法

システムコールを発行した実行ファイルや DLL を識別するマルウェア動的解析システムとして，MAT（Module-based Analysis Tool）[19] がある．MAT は，システムコールの呼び出し元を得る方法にスタックトレースを用いているため，スタックの改竄された場合に正しい呼び出し元を得られないという課題を有する．MAT の本体は，Windows のドライバとして実装されている．提案手法のベースとなる Alkanet は，Windows には手を加えず，VMM として実現している．また，MAT は，ファイルおよびプロセス，モジュールのマッピングの範囲を追跡する機能を持つ．しかし，動的に生成されたコードの領域や暗黙的にマッピングされるファイルは想定しておらず，十分ではない．提案手法は，発行されたシステムコールだけでなく，VAD や PTE から情報を補完することで，これらについても追跡を可能にする．

CXPInspector [20] や IntroLib [21] は，VMM からページの実行権限を操作することで，実行ファイルや DLL などのメモリ領域間の遷移をフックする手法を実現している．具体的には，ある実行ファイルがマップされている領域を実行中は，その領域に含まれるページのみ実行を許可し，それ以外の領域のページを実行不可に設定する．これによって，領域の範囲外への遷移が発生すると，ページフォルトが発生するため，VMM でフックが可能となる．VMM は，遷移先の実行ファイルの領域を取得し，各ページの権限を再設定し，実行を再開させる．CXPInspector および IntroLib ともに，フック時に発生した分岐の情報を取得するために LBR を活用している．

egg [6] や API Chaser [7] は，1 バイト単位のテイント解析によりマルウェアの存在するメモリ領域を明確に区別し，命令単位での動作の監視を可能とする．これにより，マルウェアからの API 呼び出しを検出できるため，API トレースを実現する．なお，egg は，ページフォルトハンドラを用いたメモリアクセスフックとトラップフラグを用いたシングルステップ実行によって上記の機能を実現する．一方，API Chaser は，エミュレータを用いて命令やメモリ，ディスクなどのハードウェアを拡張することにより実現している．

11.2 コントロールフローの検証手法

提案手法では，BTS に記録された分岐情報をもとに，実際に実行された関数呼び出し階層を再現している．既存研究には，実行された分岐命令からコントロールフローを再現することで，攻撃を検出する手法を提案するものがある．

ROPdefender [22] は，call 命令実行時にシャドウスタックにも戻りアドレスを積み，ret 命令実行時にスタックの戻りアドレスとシャドウスタックにある戻りアドレスを比較することで，戻りアドレスの改竄や ROP による攻撃を検出する．この機能は，Intel Pin [23] を用いて実行時にコード挿入を行うことで実現する．マルウェアの解析を目的とする提案手法とは目的が異なるが，call 命令や ret 命令の列から関数呼び出し階層を取得する点が類似している．

CFIMon [24] は，実行時のコントロールフローが静的解析と事前のトレーニング時で得たコントロールフローから外れていないかチェックを行う．CFIMon では，トレーニング時および実行時において，実際に実行された分岐情報を取得するために BTS を使用している．一方，提案手法は，システムコールの呼び出し元を識別するために BTS から得られる分岐記録を活用する．

kBouncer [18] は，特定の API とシステムコールをフックし，ブランチトレース機能の 1 つである LBR を用いてフローをチェックすることで，ROP の検知・緩和を目的とする．具体的には，ret 命令の戻り先が call 命令の直後であることを検証することで，ROP によって API が呼び出されたことを検出する．また，API コール時とシステムコール発行時の 2 段階で検証することで，LBR に記録できる分岐数の制約を緩和している．本論文では，マルウェアの動作を観測するために，BTS を用いた．また，kBouncer が Windows のドライバとして動作する点も提案手法と異なる．提案手法では，VMM から VM 上で発生する分岐を記録する目的で BTS を活用する．

12. おわりに

本論文では，メモリ感染型マルウェアを解析可能とすることを目的として，感染領域を経由して発行されたシステムコールを識別する手法を提案した．提案手法は，VM 上で動作するマルウェアを VMM から BTS を用いてトレースし，実際に発生した分岐命令の情報に基づいてスタックトレースと同等の機能を実現する．さらに，発行されたシステムコールと VAD や PTE から感染領域を識別する．これらにより，感染領域を経由してシステムコールに至ったことを検出可能とする．評価では，DLL タイプのマルウェア，スタックを偽装するプログラム，実際のマルウェアを用いて，それらが実現されていることを確認した．今後の課題として，call 命令と ret 命令が必ずしも対応付かない場合における関数呼び出し階層の取得手法，感染領域と非感染領域間のフローに着目した手法，LBR の EN_CALLSTACK オプションの活用の検討がある．

参考文献

[1] 大月勇人ほか：マルウェア観測のための仮想計算機モニタを用いたシステムコールトレース手法，情報処理学会論文誌，Vol.55, No.9, pp.2034–2046 (2014).

[2] Bayer, U. et al.: TTAnalyze: A Tool for Analyzing Malware, *EICAR 2006* (2006).

[3] Dinaburg, A. et al.: Ether: Malware Analysis via Hardware Virtualization Extensions, *Proc. CCS '08*, pp.51–62, ACM (online), DOI: 10.1145/1455770.1455779 (2008).

[4] Pfoh, J. et al.: Nitro: Hardware-based System Call Tracing for Virtual Machines, *Proc. IWSEC'11*, pp.96–112, Springer-Verlag (2011).

[5] Butler, J. et al.: Bypassing 3rd Party Windows Buffer Overflow Protection, Phrack 62, Volume 0x0b, Issue 0x3e, Phile #0x05 of 0x10, available from ⟨http://www.phrack.org/issues.html?issue=62&id=5#article⟩ (accessed 2015-01-21).

[6] 丹田 賢：仮想環境に依存せず詳細な解析能力を持つ動的解析システムの設計と実装，株式会社フォティーンフォティ技術研究所（オンライン），入手先 ⟨http://www.ffri.jp/assets/files/research/research_papers/Dynamic_malware_analyzer_without_virtual_environments_ja.pdf⟩ (2011).

[7] Kawakoya, Y. et al.: API Chaser: Anti-analysis Resistant Malware Analyzer, *Research in Attacks, Intrusions, and Defenses*, Lecture Notes in Computer Science, Vol.8145, pp.123–143, Springer Berlin Heidelberg (online), DOI: 10.1007/978-3-642-41284-4_7 (2013).

[8] Shinagawa, T. et al.: BitVisor: A Thin Hypervisor for Enforcing I/O Device Security, *Proc. VEE '09*, pp.121–130, ACM (online), DOI: 10.1145/1508293.1508311 (2009).

[9] Dolan-Gavitt, B.: The VAD Tree: A Process-eye View of Physical Memory, *Digital Investigation*, Vol.4, pp.62–64 (online), DOI: 10.1016/j.diin.2007.06.008 (2007).

[10] 秋山満昭ほか：マルウェア対策のための研究用データセット—MWS Datasets 2014，情報処理学会研究報告コンピュータセキュリティ（CSEC），Vol.2014-CSEC-66, No.19, pp.1–7 (2014).

[11] Glaister, A.：シンボルを使用したデバッグ，Microsoft（オンライン），入手先 ⟨http://msdn.microsoft.com/ja-jp/library/bb694540(v=vs.85).aspx⟩（参照 2015-05-09）.

[12] Microsoft: /Oy（フレームポインターの省略）（オンライン），入手先 ⟨http://msdn.microsoft.com/ja-jp/library/vstudio/2kxx5t2c.aspx⟩（参照 2015-05-09）.

[13] 寺田真敏ほか：研究用データセット「動的活動観測 2014」の検討，コンピュータセキュリティシンポジウム 2014 論文集，Vol.2014, No.2, pp.1121–1125 (2014).

[14] 株式会社インターネットイニシアティブ：Internet Infrastructure Review（IIR）Vol.21，株式会社インターネットイニシアティブ（オンライン），入手先 ⟨http://www.iij.ad.jp/company/development/report/iir/pdf/iir_vol21.pdf⟩ (2013).

[15] Futuremark Corporation: Futuremark–Legacy Benchmarks, available from ⟨http://www.futuremark.com/benchmarks/legacy⟩ (accessed 2015-05-29).

[16] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *Proc. ATEC '05*, pp.41–46, USENIX Association, (2005).

[17] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86), *Proc. CCS '07*, pp.552–561, ACM (online), DOI: 10.1145/1315245.1315313 (2007).

[18] Pappas, V. et al.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, *Proc. SEC '13*, pp.447–462, USENIX Association (2013).

[19] Jianming, F. et al.: Malware Behavior Capturing Based on Taint Propagation and Stack Backtracing, *Proc. TrustCom2011*, pp.328–335 (2011).

[20] Willems, C. et al.: Hypervisor-based, hardware-assisted system monitoring, *Virus Bulletin Conference* (2013).

[21] Deng, Z. et al.: IntroLib: Efficient and transparent library call introspection for malware forensics, *Digital Investigation*, Vol.9, Supplement, No.0, pp.S13–S23 (online), DOI: 10.1016/j.diin.2012.05.013 (2012).

[22] Davi, L. et al.: ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks, *Proc. ASIACCS '11*, pp.40–51, ACM (online), DOI: 10.1145/1966913.1966920 (2011).

[23] Luk, C.-K. et al.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Proc. PLDI '05*, pp.190–200, ACM (online), DOI: 10.1145/1065010.1065034 (2005).

[24] Xia, Y. et al.: CFIMon: Detecting violation of control flow integrity using performance counters, *Proc. DSN2012*, pp.1–12 (2012).

推薦文

ハイパーバイザで BTS を利用する仕組みを詳細に記述しており，また，プロトタイプ実装による評価を行うとともに提案手法の制限についても丁寧に論証しており推薦するにふさわしい．

（コンピュータセキュリティシンポジウム 2014
プログラム委員長 井上大介）

大月 勇人 （学生会員）

1988 年生．2011 年立命館大学情報理工学部情報システム学科卒業，2013 年同大学大学院理工学研究科博士前期課程情報理工学専攻修了．同年同大学院情報理工学研究科博士後期課程情報理工学専攻に入学，現在に至る．オペレーティングシステム，仮想化技術，コンピュータセキュリティ等に興味を持つ．

瀧本 栄二 （正会員）

1976 年生．1999 年立命館大学理工学部情報学科卒業，2001 年同大学大学院理工学研究科博士前期課程修了，2005 年同研究科博士後期課程単位取得退学，同年（株）ATR 適応コミュニケーション研究所専任研究員，2010 年立命館大学情報理工学部情報システム学科助手，現在に至る．主にシステムソフトウェア，無線通信に関する研究に従事．博士（工学）．

齋藤 彰一 （正会員）

1993 年立命館大学理工学部情報工学科卒業．1995 年同大学大学院博士前期課程修了．1998 年同大学院博士後期課程単位習得中退．同年和歌山大学システム工学部情報通信システム学科助手．2003 年同講師，2005 年同助教授．2006 年名古屋工業大学大学院助教授，2007 年同准教授，現在に至る．オペレーティングシステム，インターネット，セキュリティ等の研究に従事．博士（工学），ACM，IEEE-CS 各会員．

毛利 公一 （正会員）

1994 年立命館大学理工学部情報工学科卒業，1996 年同大学大学院理工学研究科修士課程情報システム学専攻修了，1999 年同研究科博士課程後期課程総合理工学専攻修了．同年東京農工大学工学部情報コミュニケーション工学科助手，2002 年立命館大学理工学部情報学科講師，2004 年同大学情報理工学部情報システム学科講師，2008 年同准教授，2014 年同教授となり，現在に至る．博士（工学）．オペレーティングシステム，仮想化技術，コンピュータセキュリティ等の研究に従事．電子情報通信学会，ACM，IEEE-CS，USENIX 各会員．