ブランチトレース機能を用いたシステムコール呼出し元アドレス取得手法

大月 勇人† 瀧本 栄二† 齋藤 彰一‡ 毛利 公一†

† 立命館大学
525-8577 滋賀県草津市野路東 1-1-1
yotuki@asl.cs.ritsumei.ac.jp, {takimoto, mouri}@cs.ritsumei.ac.jp

‡ 名古屋工業大学
466-8555 愛知県名古屋市昭和区御器所町
shoichi@nitech.ac.jp

あらまし 最近のマルウェアには，他のプロセスのメモリ上に潜んで動作するものや複数のモジュールで構成されるものが存在する．このようなマルウェアに対して，従来のプロセスやスレッドを単位として挙動を観測する手法では個々の動作の区別が困難である．この課題の解決のために，システムコールトレーサである Alkanet は，システムコールフック時にスタックトレースを行い，呼出し元となったコードまで特定する．ただし，当該手法では，マルウェアにスタックを改竄された場合に呼出し元を正確に取得できない．そこで，本論文では，CPU に搭載されているブランチトレース機能を活用した正確な呼出し元の取得手法とその有効性について述べる．

Identifying of System Call Invoker by Branch Trace Facilities

Yuto Otsuki† Eiji Takimoto† Shoichi Saito‡ Koichi Mouri†

†Ritsumeikan University
1-1-1 Nojihigashi, Kusatsu, Shiga 525-8577 Japan
yotuki@asl.cs.ritsumei.ac.jp, {takimoto, mouri}@cs.ritsumei.ac.jp

‡Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya, Aichi, 466-8555 Japan
shoichi@nitech.ac.jp

Abstract

Recent malware infects other processes. Another one consists of two or more modules and plugins. It is difficult to trace these malware because traditional methods focus on threads or processes. We are developing Alkanet, a system call tracer for malware analysis. To trace the malware, Alkanet identifies the system call invoker by stack trace. However, if malware has falsified its stack, Alkanet cannot identify it correctly. In this paper, we describe a method for identifying a system call invoker by branch trace facilities. We consider the effectiveness of branch trace facilities for malware analysis.

1 はじめに

マルウェアの脅威が問題となっているが，その対策には，マルウェアの挙動を調査する必要がある．特に，最近のマルウェアには，他のプロセスのメモリ上に潜んで動作するものや複数のモジュールで構成されるものが存在するが，こ

のようなマルウェアに対して，従来のプロセスやスレッドを単位として挙動を観測する手法では個々の動作の区別が困難である．以上の背景から，我々は，システムコールトレースによってマルウェアを解析する Alkanet[1, 2] を開発している．Alkanet は，システムコールフック時にスタックトレースを行い，システムコールを呼び出した実行ファイルやコードまで識別する．これにより，上記のようなマルウェアでも個々の動作を区別して観測が可能である．

ただし，当該手法では，マルウェアにスタックを改竄された場合に呼出し元を正確に取得できない．この課題を解決するためには，マルウェアに改竄されない情報に基づいて呼出し元を取得する必要がある．そこで，CPU のブランチトレース機能を用いることで，実際に実行された分岐の情報が得られることに着目した．CPU が記録した分岐情報に基づいてスタックトレースと同等の関数呼出し階層を得ることで，マルウェアにスタックを改竄された場合でも正しい呼出し元を取得することが可能となる．本論文では，ブランチトレース機能を活用した呼出し元の取得手法とその有効性について述べる．

2 Alkanet

Alkanet は仮想計算機モニタ (VMM) ベースのマルウェア動的解析システムである．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．また，Alkanet は，マルウェアが多く出回っている Windows のシステムコールをトレースすることができる．観測単位をシステムコールとすることにより，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．取得したシステムコールトレースのログからは，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．

Alkanet の全体構成を図 1 に示す．Alkanet は， VMM である BitVisor[3] の拡張機能として実装している．BitVisor は，ホスト OS を必要とせず，ハードウェア上で直接動作するハイパーバイザ型の VMM である．Intel 製 CPU の仮想化支援機能である Intel VT (Intel Virtualization Technology) を利用しており，Windows を修正なしで実行できる．

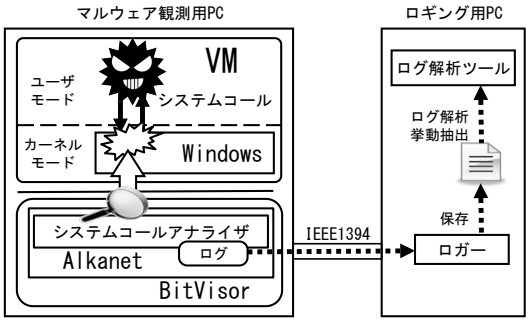

図 1: Alkanet の全体構成

マルウェアの実行環境であるゲスト OS には，32bit 版 Windows XP Service Pack 3 を用いている．この環境におけるシステムコールは，通常 sysenter 命令によってカーネルモードへ遷移し，sysexit 命令によってユーザモードへ復帰する．システムコールのフックは，これらの命令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．フック後に，レジスタやメモリ内容を解析することで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 のログは，ロギング用 PC から IEEE 1394 を用いて取得し，その後に解析することができる．

我々の先行研究 [2] では，システムコールフック時にスタックトレースを行い，システムコールに至るまでに経由したアドレスを取得することができることを示した．さらに，VAD (Virtual Address Descriptor) や PTE (Page Table Entry) を用いて，取得したアドレスにマップされているファイルの情報や，そのアドレスが動的に生成された領域内であるかなどの情報を取得することができる．これにより，マルウェアの実行ファイルや，マルウェアが書込みを行ったメモリ領域を経由したシステムコール呼出しを識別可能であることを示した．

3 BTS

Intel 製の CPU 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 (Branch Trace Store)，分岐命令の度にデバッグ例外を発生させる BTF (Single Step on Branches) などのブランチトレース機能が搭載されているものがある．本論文では，分岐情報を保存する 2 つの機能のうち，保存できる分岐数に実質制限のない BTS を用いて，システムコールの呼出し元アドレスの取得を行った．以下，BTS について述べる．

BTS は，分岐命令の実行の度に分岐元アドレス，分岐先アドレス，分岐予測の成否の 3 つの情報を CPU がメモリ上のバッファに記録する機能である．バッファの仮想アドレスやサイズなどは，IA32_DS_AREA MSR に設定する．BTS は，IA32_DEBUGCTL MSR の TR bit (第 6bit) と BTS bit (第 7bit) をセットすることで有効にすることができる．BTS_OFF_OS bit (第 9bit)，BTS_OFF_USR bit (第 10bit) 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 bit (第 8bit) をセットすることで，バッファが全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．この機能を応用すれば，実質無制限に分岐情報を記録できる．なお，BTINT bit 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．

4 BTS を用いた呼出し元取得

VMM である Alkanet が，BTS を用いてシステムコールの呼出し元を取得するためには，以下の技術的課題が存在する．

- VM 上で発生する分岐を取得する方法
- スレッド毎に分岐記録を取得する方法
- 取得した分岐記録からシステムコールの呼出し元を得る方法

本章では，上記課題の解決方法を述べる．

4.1 VM 上で発生する分岐の取得

ゲスト OS 上で実行されるユーザプログラムで発生する分岐を記録する機能を実現するには，VM 上に前述の MSR や分岐記録を保存するバッファを設定する必要がある．そこで，Alkanet から VM の MSR やページテーブルを設定する．

具体的には，VM 上の IA32_DEBUGCTL は，VMCS (Virtual-Machine Control Structures) 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．ここで，IA32_DS_AREA に設定するバッファなどの仮想アドレスは，ゲスト OS のページテーブルで解決可能なアドレスである必要がある．したがって，ゲスト OS である Windows が使用していない仮想メモリ領域を示すページテーブルを Alkanet から書き換えることで設定を行う．さらに，BTS に使用する物理メモリ領域を Windows が使用しないように，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，使用できない物理メモリ領域として認識されるようにする．

なお，本論文では，システムコールの呼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する必要はない．したがって，BTS_OFF_OS bit 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．

4.2 スレッド毎の分岐記録の取得

BTS にはスレッドやプロセスを区別する機能はないため，そのままでは複数のスレッド・プロセスで発生した分岐の情報が混交してしまう．マルウェアのコントロールフローを取得するためには，スレッド毎に分岐情報を記録する必要がある．そこで，Alkanet は，BTS 用のバッファをスレッド毎に管理し，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．なお，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しているスレッドオブジェクトのアドレスを保持するメンバがある．そこで，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換えを検出することでフックを実現する．

4.3 システムコールの呼出し元取得

スレッド毎に記録された分岐情報を用いて，スタックトレースと同等の情報を生成することで，システムコールの呼出し元を特定できる．具体的には，記録された分岐情報の分岐元アドレスに存在する命令を確認し，call 命令による分岐を取り出すことで関数の呼出し履歴を作成する．ただし，BTS には，既にリターンされた関数への call 命令も記録されている．そこで，分岐記録内の call 命令と ret 命令の対応付けを行い，まだリターンされていない関数のみを取り出し，現在実行中の関数呼出し階層を生成する．これによって，スタックトレースと同等，かつ実際に実行された正確な関数呼出し階層を得ることができる．

5 機能評価

本手法の実現可能性および有効性を検証するために，プロトタイプを作成して評価を行った．本章では，その評価結果について述べる．

5.1 DLL 検体

複数のモジュールで構成されるマルウェアについて，モジュール毎に挙動を取得できることを示すために，先行研究 [2] では，rundll32.exe を用いて DLL タイプのマルウェアをロードし，rundll32.exe の挙動からマルウェアの挙動を区別できるかを確認した．本評価では，先行研究と同様の評価を行い，BTS より生成した関数呼出し階層とスタックトレースの結果とを比較し，同等の結果を得られることを確認する．

MWS Datasets 2014 [4] 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，解析した．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．

図 2 は，本評価で取得したログから，StackTrace の項目のみを抜き出したものである．当該項目 の 1 行目はシステムコールフック時のスタックの情報である．2 行目以降は，スタックから取得した戻りアドレスと，それを含むスタックフレームについての情報である．“[]” 内の数字はスタックフレームの深さを示す．また，戻りアドレスについては，そのアドレスに存在する API 名やファイル名，書込みの可否，既に書き込まれたか否かなどについて表示している．各項目の詳細は，文献 [2] を参照されたい．

本論文では，BTS より生成した関数呼出し階層とスタックトレースの結果を，深さに基づいて対応付けを行い，ログに 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 VALID，不整合になら INVALID 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，UNVALIDATED と表記する．

図 2 の [11]，[10] は，rundll32.exe が LoadLibraryW を用いて Conficker.dll をロードしたことを示す．同様に，[05]〜[00] より，その初期化処理の中で Conficker.dll が Sleep API，SleepEx API，NtDelayExecution のスタブを経由し，NtDelayExecution システムコールを発行したことがわかる．ここで，[05]〜[00] 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)．一方，[11]，[10] を含むこれより深い位置にある戻りアドレスについては，分岐記録が上書きされたため，確認できなかった (UNVALIDATED)．この課題は，BTINT による通知と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 からスタックトレースと同等の出力を得ることが可能であり，システムコールの呼出し元を取得することが可能であることが確認できた．

5.2 PDF 検体

一般にシェルコードでは，自身の存在するアドレスを取得する目的で call 命令を使用する場合や，ROP (Return Oriented Programming) が用いられる場合がある．このようなケースで

```
StackTrace:
  SP: 7ed24, StackBase: 80000, StackLimit: 74000
  [00] <- 7c94d1fc (API: NtDelayExecution+0xc, Writable: 0, Dirty: 0,
          VAD: {7c940000--7c9dc000, ImageMap: 1, File: "\WINDOWS\system32\ntdll.dll"}),
          From: 7c94d1fa, Valid: VALID, SP: 7ed24
  [01] <- 7c8023f1 (API: SleepEx+0x51, Writable: 0, Dirty: 0,
          VAD: {7c800000--7c933000, ImageMap: 1, File: "\WINDOWS\system32\kernel32.dll"}),
          From: 7c8023eb, Valid: VALID, SP: 7ed28
  [02] <- 7c802455 (API: Sleep+0xf, Writable: 0, Dirty: 0,
          VAD: {7c800000--7c933000, ImageMap: 1, File: "\WINDOWS\system32\kernel32.dll"}),
          From: 7c802450, Valid: VALID, BP: 7ed7c
  [03] <- 10003898 (API: -, Writable: 0, Dirty: 0,
          VAD: {10000000--10018000, ImageMap: 1, File: "\Conficker.dll"}),
          From: 10003892, Valid: VALID, BP: 7ed8c
  [04] <- 1000401b (API: -, Writable: 0, Dirty: 0,
          VAD: {10000000--10018000, ImageMap: 1, File: "\Conficker.dll"}),
          From: 10004016, Valid: VALID, BP: 7f184
  [05] <- 7c94118a (API: LdrpCallInitRoutine+0x14, Writable: 0, Dirty: 0,
          VAD: {7c940000--7c9dc000, ImageMap: 1, File: "\WINDOWS\system32\ntdll.dll"}),
          From: 7c941187, Valid: VALID, BP: 7f1a4
  <省略>
  [10] <- 7c80aeec (API: LoadLibraryW+0x11, Writable: 0, Dirty: 0,
          VAD: {7c800000--7c933000, ImageMap: 1, File: "\WINDOWS\system32\kernel32.dll"}),
          From: -, Valid: UNVALIDATED, BP: 7f888
  [11] <- 1001792 (API: -, Writable: 0, Dirty: 0,
          VAD: {1000000--100b000, ImageMap: 1, File: "\WINDOWS\system32\rundll32.exe"}),
          From: -, Valid: UNVALIDATED, BP: 7f89c
  <省略>
```

図 2: Conficker.dll の挙動観測時のログ

は，通常の関数呼出しと異なり，call 命令と ret 命令が対応付かないフローが多く存在すると考えられる．スタックトレースおよび本論文で述べた関数呼出し階層手法は，関数呼出しにおける call 命令と ret 命令の対称性に基づいているため，上記のようなイレギュラーなフローでは呼出し元アドレスの取得が正常に行えない可能性がある．本評価では，このようなシェルコード内のイレギュラーなフローに対し，スタックトレースによって得られるフローと，BTS から生成した関数呼出し階層について比較を行う．

MWS Datasets 2014 [4] に含まれる D3M 2013 に活動が記録されているマルウェア検体の中から PDF 検体を選択し，解析した．本検体は，先行研究 [2] で述べた PDF 検体と同一であり，スタックトレースを用いてシェルコードによるシステムコールを識別できることを確認している．本評価では，シェルコードが展開されるアドレスが異なったものの，先行研究で示した挙動と同様の動作を示した．図 3 に取得したログの一部を示す．

図 3 では，書込み可能 (Writable: 1) かつ既に書き込まれた (Dirty: 1) 領域 ([03]) から VirtualProtect API，VirtualProtectEx API，NtProtectVirtualMemory のスタブを経由 ([02]〜[00]) し，NtProtectVirtualMemory システムコールが発行されたことを示している．ここで，[02]〜[00] においては，Valid: VALID となっており，BTS より生成した関数呼出し階層と整合する．[03] は，From のアドレス 2f900a9 から想定される戻り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 2f900c7 が整合せず，INVALID となっている．

そこで，実際のフローを確認するために，BTS に記録された分岐記録と分岐元アドレスに存在する命令を別途取得し，解析を行った．図 3 の [03] に該当する分岐とその直前の分岐を図 4 に示す．図中の各行は，BTS に記録された分岐元アドレス (from) および分岐先アドレス (to) と，その分岐を発生させた命令を示す．図 4 より，2f900a9 から 2f900ae への call 命令による分岐の後 (図中 1 行目)，2f900c5 から VirtualProtect API の途中である 7c801ad9 への jmp 命令を用いた分岐 (図中 2 行目) が発生してい

```
Stacktrace:
  SP: f601ff8, StackBase: 130000, StackLimit: 11d000
  [00] <- 7c94d6dc (API: NtProtectVirtualMemory+0xc, Writable: 0, Dirty: 0,
          VAD: {7c940000--7c9dc000, ImageMap: 1, File: "\WINDOWS\system32\ntdll.dll"}),
          From: 7c94d6da, Valid: VALID, SP: f601ff8
  [01] <- 7c801a81 (API: VirtualProtectEx+0x20, Writable: 0, Dirty: 0,
          VAD: {7c800000--7c933000, ImageMap: 1, File: "\WINDOWS\system32\kernel32.dll"}),
          From: 7c801a7f, Valid: VALID, SP: f601ffc
  [02] <- 7c801aec (API: VirtualProtect+0x18, Writable: 0, Dirty: 0,
          VAD: {7c800000--7c933000, ImageMap: 1, File: "\WINDOWS\system32\kernel32.dll"}),
          From: 7c801ae7, Valid: VALID, BP: f60201c
  [03] <- 2f900c7 (API: -, Writable: 1, Dirty: 1,
          VAD: {2f90000--2f91000, ImageMap: 0}),
          From: 2f900a9, Valid: INVALID, BP: f602038
```

図 3: PDF に含まれたシェルコードの挙動観測時のログ

```
from= 2f900a9, to= 2f900ae  call 2f900ae
from= 2f900c5, to=7c801ad9  jmp EBX
```

図 4: 記録された分岐情報と分岐命令

る．2 行目の jmp 命令は，API の先頭数バイトをスキップすることで API フックを回避する動作であると考えられる．したがって，この jmp 命令による分岐は，call 命令を用いていないものの，実質関数呼出しであることがわかる．

今回のケースでは，本来のシステムコールの呼出し元アドレスは 2f900c5 である．スタックトレースより取得した戻りアドレスの方が正確な値を示している理由は，シェルコードが自らスタックに戻りアドレスを積んだためであると考えられる．本論文で述べた関数呼出し階層生成手法は，call 命令や ret 命令による分岐のみを対象とした．そのため，本手法では，今回のシェルコードにおける正しい呼出し元アドレスを取得できなかった．ただし，図 4 のように BTS には正しいフローが記録されている．したがって，一般的な関数呼出しに限定せずに関数呼出し階層を生成することで，この課題を解決できると考えられる．

6 パフォーマンス評価

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，Intel Core 2 Quad Q6600 2.4GHz，メモリ 4GB を搭載した評価用 PC で PCMark05[5] の System Test Suite を用いて評価を行った．本評価では，評価用 PC 上で Windows を実行する Native と，システムコールトレースのみ行う Alkanet (Normal)，システムコールフック時にスタックトレースも行う Alkanet (ST)，BTS を用いてスタックトレースの検証を行う Alkanet (BTS) の 3 種類の Alkanet でゲスト OS として Windows を実行し，比較を行う．

Native，Alkanet (Normal)，Alkanet (ST) の総合スコア (単位 PCMarks) は，それぞれ 5557，4171，3266 であった．Native のスコアを 100 として正規化すると，Alkanet (Normal) が 75，Alkanet (ST) が 59 である．一方，Alkanet (BTS) は，テストに含まれる Web Page Rendering の項目が失敗したため，総合スコアが算出されなかった．当該項目では Internet Explorer を起動してテストを行うが，オーバヘッドによる遅延からプロセス間通信のタイムアウトが発生したことが失敗の原因と考えられる．

各項目を個別で確認すると，Alkanet (BTS) は HDD 系のテストを除き，Native の 10%以下の性能となっており，性能低下が大きくなっていた．この原因としては，BTS そのもののオーバヘッドに加えて，下記の処理のオーバヘッドが考えられる．

1. コンテキストスイッチのフックおよびそれに伴うバッファの退避・復元
2. 分岐記録に基づく関数呼出し階層の生成およびスタックトレースとの比較

上記 2 は，記録された分岐元アドレス全てに対して，call 命令や ret 命令が存在するか確認する

ため，性能低下が大きいと考えられる．また，上記1について，今回のプロトタイプではバッファの退避・復元の度にバッファサイズ分のメモリコピーが発生する．したがって，IA32_DS_AREA に設定する仮想アドレスや対応する PTE を調整することで，バッファコピーのオーバヘッドを削減することができると考えられる．

なお，参考までに何も変更を加えていない QEMU 上で計測した結果，HDD 系のテストを除き，多くが Native の 10%程度のスコアとなっていた．したがって，本論文で述べた手法は QEMU と同程度のパフォーマンスであった．

7 関連研究

kBouncer[6] は，特定の API 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．すなわち，マルウェアの観測や解析を目的としたものではない．本論文では，マルウェアの動作を観測するために，BTS の活用を検討した．また，kBouncer が Windows のドライバとして動作する点も本手法と異なる．本手法では，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．

CXPInspector[7] 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 などの特定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．CXPInspector はフック時に発生した分岐を取得するために，LBR を活用している．本手法では，システムコール発行に至るまでの関数呼出し階層を取得する目的で BTS を活用している点が異なる．

8 考察

8.1 システムコールの呼出し元取得

BTS を用いることで，スタックトレースと同等の結果を得ることが可能であることが確認できた．ただし，関数呼出し階層の生成手法には課題が残る．5.2 節で述べたシェルコード以外でも，call 命令と ret 命令が必ずしも対応付かないケースが確認されている．具体的には，例外に代表される大域脱出やカーネルからのコールバックが挙げられる．これらは，意図的にスタックを操作することで，数階層前の call 命令による戻りアドレスや予め登録されたアドレスへジャンプする．そのため，直近の call 命令の戻りアドレスと ret 命令の戻り先が一致せず，本論文で述べた手法ではスタックトレースと同等の出力を得ることは困難となる．

本手法の目的は，システムコールの呼出し元がマルウェアであることを判定することにある．本論文では BTS からスタックトレースと同等の結果を得ることで目的の達成を試みた．そこで，前述の課題を解決した上で目的を達成するために，マルウェアが API やシステムコールを呼び出す時に，マルウェアから Windows が提供する DLL に制御が移ることに着目する．具体的には，BTS に記録された分岐から，マルウェアのメモリ領域とそれ以外のメモリ領域間のフローを生成することで上記課題の解決を図ることができる．

8.2 パフォーマンス

BTS は網羅的に分岐を記録できるが，一方で，6 章で述べたように性能低下が大きい．また，ループ等が存在する場合など，記録するデータ量が増加する課題もある．以上から，高速な解析が必要な場合は活用が難しい．

kBouncer[6] 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，LBR は，分岐命令の種類によって記録するかどうかを選択可能である．Haswell アーキテクチャ以降の CPU では，EN_CALLSTACK オプションにより，リターンしていない関数呼出しの記録のみを 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出し元取得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 までの関数呼出しまでしか取得できない．また，前述の大域脱出やシェルコードで確認された課題について

は，同様に課題となると考えられる．以上から，スタックトレースと併用し，それぞれの内容を検証することを検討しなければならない．

8.3 BTS の有効な用途

BTS は，LBR と異なり保持できる分岐数に制限がないため，マルウェアのコントロールフローを網羅的に観測することが可能である．したがって，短時間での解析ではなく，より細粒度の解析での活用が期待される．具体的には，コントロールフローの類似性に基づいたマルウェアの分類や検出の手法が挙げられる．また，マルウェアのコントロールフローを取得することで，アンパックが難しいマルウェアや独自のローダを必要とするマルウェアなど，静的解析が難しいマルウェアでも内部構造の把握が可能となる．これにより，新しい特徴を備えたマルウェアの理解に役立てることができる．

さらに，実機で分岐記録を取得できることから，エミュレータを用いた従来の手法と比べると，命令エミュレーション上の齟齬によりマルウェアに検出される恐れがない．また，各分岐間は連続で実行されることから，その間に実行された命令列の取得も可能である．これにより，従来ではシングルステップ実行で取得していた命令単位でのトレースを，フック回数を抑えつつ行うことが可能となる．以上から，BTS は細粒度での解析の安定性や高速化に寄与できる．

9 おわりに

本論文では，BTS を用いて関数呼出し階層を取得する方法について述べ，スタックトレースで取得した戻りアドレスと比較を行った．これにより，BTS を用いることで，マルウェアに改竄され得るスタックに頼らずにスタックトレースと同等の情報が得られることが確認できた．したがって，本手法は，システムコールフック時に呼出し元を取得する方法として有効である．ただし，call 命令と ret 命令が必ずしも対応付かないため，分岐記録から関数呼出し階層を生成する手法については課題がある．また，BTS は，使用時のパフォーマンスへの影響が大きく，高速な解析に不向きであった．一方で，マルウェアのコントロールフローを網羅できる利点を活用したより細粒度の解析での活用が期待される．今後の課題として，マルウェアのメモリ領域とそれ以外のメモリ領域間のフローに着目した手法や，LBR の EN_CALLSTACK オプションの活用の検討がある．

参考文献

[1] Otsuki, Y., Takimoto, E., Kashiyama, T., Saito, S., Cooper, E. and Mouri, K.: Tracing Malicious Injected Threads Using Alkanet Malware Analyzer, *IAENG Transactions on Engineering Technologies*, Lecture Notes in Electrical Engineering, Vol. 247, Springer Netherlands, pp. 283–299 (2014).

[2] 大月勇人，瀧本栄二，齋藤彰一，毛利公一：Alkanet におけるシステムコールの呼出し元動的リンクライブラリの特定手法，コンピュータセキュリティシンポジウム 2013 論文集，Vol. 2013, No. 4, pp. 753–760 (2013).

[3] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y. and Kato, K.: BitVisor: a thin hypervisor for enforcing i/o device security, *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ACM, pp. 121–130 (2009).

[4] 秋山満昭，神薗雅紀，松木隆宏，畑田充弘：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).

[5] Futuremark Corporation: Futuremark - Legacy Benchmarks, http://www.futuremark.com/benchmarks/legacy (2014, accessed 2014-08-25).

[6] Pappas, V., Polychronakis, M. and Keromytis, A. D.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, USENIX Association, pp. 447–462 (2013).

[7] Willems, C., Hund, R. and Holz, T.: Hypervisor-based, hardware-assisted system monitoring, *Virus Bulletin Conference* (2013).