Vol.2015-CSEC-69 No.2
Vol.2015-IOT-29 No.2
2015/5/21

分岐トレース支援機能を用いたカーネルルートキット検知手法の提案

赤尾 洋平[1,a)]　山内 利宏[1]

概要：計算機システムへの攻撃の中でも，カーネルルートキットを用いた攻撃の脅威が深刻である．攻撃にカーネルルートキットが用いられると，攻撃の検知が困難になり，対処が遅れ，計算機の被害が拡大する．このような背景から，これまでに様々なカーネルルートキットの検知手法が提案されている．しかし，既存のカーネルルートキット検知手法にはいくつかの問題点がある．本稿では，既存手法の問題点に対処するために，多くのカーネルルートキットが通常とは異なる分岐を発生させることに着目し，分岐トレース支援機能を用いてカーネル内で発生する分岐を監視することでカーネルルートキットを検知する手法を提案する．提案手法を用いることで，既存手法では検知が困難なカーネルルートキットを早期に検知できる．また，提案手法は，カーネルの拡張性を低下させず，異なる OS やバージョンへの移植性が高い．

1. はじめに

計算機システムへの攻撃の中でも，ルートキットを用いた攻撃の脅威が深刻な問題である [1]．ルートキットとは，攻撃の痕跡，攻撃プログラムの存在，および自身の存在を隠蔽する機能を持つ不正なプログラムである．例えば，ルートキットは，オペレーティングシステム（以降，OS）上で動作しているプロセス一覧を出力する際，不正なプロセスのプロセス名が出力されないように出力を改ざんする．これにより，計算機の利用者は不正なプロセスの検知が困難となる．このように，攻撃にルートキットが用いられた場合，攻撃の検知が困難になり，攻撃への対処が遅れ，計算機の被害が拡大する．

ルートキットには，ユーザレベルで動作するユーザルートキットとカーネルレベルで動作するカーネルルートキットが存在する [2]．ユーザルートキットは，ユーザレベルで動作するプログラムを改ざんする．このため，同じユーザレベルで動作するプログラムから改ざんされているか否かの検査が可能であり，比較的検知が容易である [1]．一方，カーネルルートキットは，OS のカーネルを改ざんし，カーネル内で攻撃者の用意した処理を実行することで OS から出力される情報を改ざんする．カーネルルートキットに感染した場合，OS から出力される情報は信用できず，OS から出力される情報に基づいて攻撃を検知する手法ではカーネルルートキットの検知は困難である．このため，カーネルルートキットは，ユーザレベルのプログラムでの検知は困難である．

このような背景から，カーネルルートキットを検知する様々な手法が提案されている．池上ら [3] は既存手法において，カーネルルートキットを早期に検知不可であること，カーネルの拡張性を低下させること，および多種の OS やバージョンへの移植性が低いことのうち，いずれかの問題点が存在することを挙げ，これらの問題点を同時に解決に対処するために，カーネルスタックの比較によりカーネルルートキットを検知する手法を提案した．しかし，池上らの手法には，カーネルスタックに情報を積まない jmp 命令などの分岐命令を用いるカーネルルートキットを検知できないという問題点がある．

そこで，本稿では，多くのカーネルルートキットが通常とは異なる分岐を発生させることに着目し，近年の Intel CPU に搭載されている分岐トレース支援機能を用いてカーネル内で発生する分岐を監視することでカーネルルートキットを検知する手法を提案する．分岐トレース支援機能を用いることで，カーネルスタックに情報を積まない jmp 命令などの分岐命令を用いるカーネルルートキットを検知できる．本稿では，提案手法の設計，実現方式，および評価結果について述べる．

2. カーネルルートキット

2.1 カーネル内の処理流れの改ざん

カーネルルートキットは，カーネルを改ざんすることにより，攻撃の痕跡，攻撃プログラムの存在，および自身の

[1] 岡山大学 大学院自然科学研究科
[a)] akao@swlab.cs.okayama-u.ac.jp

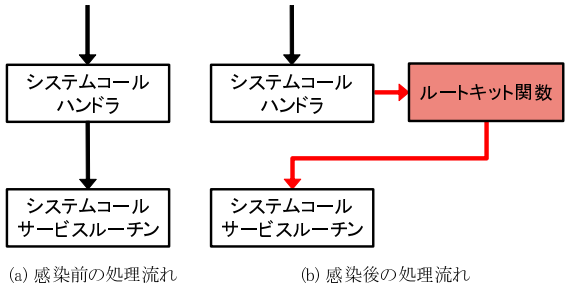

図 1 カーネルルートキットに感染した際の処理流れの変化

存在を隠蔽する．改ざん対象には，システムコールテーブル，システムコールハンドラ，割り込みハンドラ，および動的なデータ領域がある．カーネルルートキットに感染したカーネルの特徴として，カーネル内の処理流れが通常とは異なる場合が多いことが挙げられる．文献 [4] は，カーネルルートキットのうち，96%のカーネルルートキットがカーネルの処理流れを変化させることを示している．一例として，システムコール処理を改ざんするカーネルルートキットに感染した際のカーネルの処理流れを**図 1** を用いて説明する．正常なシステムコール処理では，はじめにシステムコールハンドラが実行され，その後，発行されたシステムコールに対応するシステムコールサービスルーチンが実行される．一方，カーネルルートキットに感染した際は，通常の処理流れとは異なり，システムコールハンドラから攻撃者の用意した関数（以降，ルートキット関数）に処理が移り，不正な処理が実行される．その後，発行されたシステムコールに対応するシステムコールサービスルーチンが実行される．

2.2 既存のカーネルルートキット検知手法

文献 [4],[5] の手法は，周期的にカーネルメモリの完全性を検査し，カーネルルートキットを検知する．これらの手法は，周期によってはカーネルルートキットの検知が遅れる問題がある．文献 [6] の手法は，CPU の Performance Monitoring Unit（PMU）の一部である Hardware Performance Counters（HPCs）を用いてテストプログラムの実行命令数を検査することでカーネルルートキットを検知する．この手法も，テストプログラムを実行する周期によってはカーネルルートキットの検知が遅れる問題がある．文献 [7] の手法は，監視対象の OS で取得したファイル一覧と CD から起動した OS で取得したファイル一覧を比較し，カーネルルートキットを検知する．この手法では，ファイル一覧を比較するために計算機を一時的に停止する必要がある．文献 [8] の手法は，カーネルモジュールを追加する際，カーネルモジュールのバイナリにカーネルルートキットの特徴がないかを検査し，カーネルルートキットを検知する．この手法では，カーネルルートキットと類似した機能をもつ正規のカーネルモジュールをカーネルルートキットと誤検知する可能性がある．文献 [9] の手法は，仮想マシンモニタを用いてカーネルモジュールのバイナリがシステムに登録されたものと一致するかを検査し，カーネルルートキットを検知する．この手法では，追加するカーネルモジュールをあらかじめシステムに登録しておく必要がある．文献 [10] の手法は，仮想マシンモニタを用いてレジスタやメモリへのアクセスを監視し，カーネルルートキットを検知する．このアクセスの監視は，OS のメモリ構造に依存している．池上ら [3] の手法は，カーネルスタックに積まれた情報を基にカーネルルートキットを検知する．この手法では，カーネルスタックに情報を積まない jmp 命令などの分岐命令を用いるカーネルルートキットは検知できない．

2.3 既存手法の問題点

2.2 節で述べた既存手法には，以下のいずれかの問題点が存在する．

(問題 1) カーネルルートキットを早期に検知不可
カーネルルートキットを検知するまでの時間が長引くと，攻撃が長時間行われることになり，計算機の被害が拡大する．

(問題 2) カーネルの拡張性の低下
カーネルモジュールの追加が制限された場合，カーネルの拡張性が低下する．

(問題 3) 異なる OS やバージョンへの移植性の低さ
OS 構造に依存した検知手法である場合，異なる OS やバージョンへの移植が困難である．

(問題 4) カーネルスタックに情報を積まない分岐命令を用いるカーネルルートキットを検知不可
池上ら [3] の手法は，カーネルスタックに積まれた情報を参照し，正常な場合のカーネルスタックと比較することでカーネルルートキットを検知する．この手法は，(問題 1) から (問題 3) に対処している．しかし，この手法は，カーネルスタックに情報を積まない jmp 命令などの分岐命令を用いるカーネルルートキットを検知できない．

3. 分岐トレース支援機能を用いたカーネルルートキット検知手法の設計

3.1 目的

2.3 節で述べた問題に対処するために，分岐トレース支援機能を用いたカーネルルートキット検知手法を提案する．提案手法の目的を以下に示す．

(目的 1) カーネルルートキットを早期に検知
カーネルルートキットに感染後，検知までの時間が長引くと，攻撃が長時間行われることになり，計算機の被害が拡大する．このため，カーネルルートキットを早期に検知する手法の実現を目指す．

(目的 2) カーネルの拡張性の維持
カーネルモジュールの追加の制限によりカーネルの拡張性が失われると，利便性が失われる．このため，カーネルの拡張性を制限しない手法の実現を目指す．

(目的 3) 異なる OS やバージョンへの移植性の向上
検知手法が OS 構造に依存したものである場合，異なる OS やバージョンへの移植が困難である．このため，OS 構造への依存度が低く，異なる OS やバージョンへ移植しやすい手法の実現を目指す．

(目的 4) カーネルスタックに情報を積まない命令を用いるカーネルルートキットの検知
カーネルスタックに情報を積まない命令を用いるカーネルルートキットを検知できない場合，攻撃者は jmp 命令などのカーネルスタックに情報を積まない分岐命令を用いることで検知手法を回避できる．このため，カーネルスタックに情報を積まない命令を用いるカーネルルートキットを検知できる手法の実現を目指す．

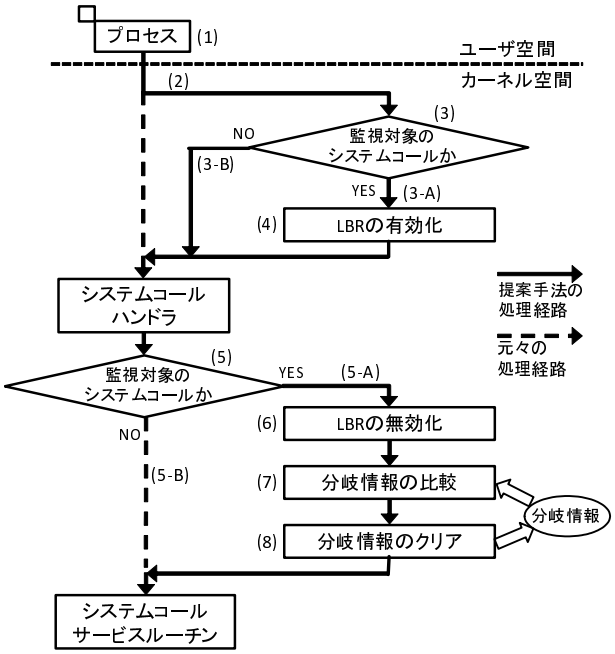

図 2 提案手法の全体図

3.2 設計方針

提案手法は，カーネルルートキットに感染後，カーネル内で通常とは異なる分岐が発生することに着目し，カーネル内で発生する分岐を監視することでカーネルルートキットを検知する．提案手法で検知するカーネルルートキットは，池上らの手法 [3] と同様に，システムコール処理を改ざんするカーネルルートキットである．ただし，池上らの手法 [3] では検知できないシステムコールサービスルーチンに分岐命令を埋め込むカーネルルートキット [15] も提案手法では新たに検知対象とする．

(目的 1) を達成するために，提案手法は，システムコール発行のたびに提案手法の処理を呼び出し，分岐を監視する．これにより，カーネルルートキットに感染後，最初に監視対象のシステムコールが発行されるタイミングでカーネルルートキットを検知できる．また，(目的 2) を達成するために，カーネルモジュールの追加を制限しない．さらに，(目的 3) と (目的 4) を達成するために，分岐を監視する手段として，OS 構造への依存度が低く，カーネルスタックに情報を積まない jmp 命令などによる分岐命令を監視できる Last Branch Record[11]（以降，LBR）を用いる．

LBR は，call や jmp などの分岐命令が発行された際の分岐元命令アドレスと分岐先命令アドレスの組（以降，分岐情報）を専用のレジスタに記録する機能であり，近年の Intel CPU に導入されている分岐トレース支援機能である．LBR は，LBR フラグと呼ばれるフラグをセットすることで有効化され，有効化されている間のみ分岐情報を記録する．LBR により保持できる分岐情報は 16 個であり，それ以降に記録される分岐情報は，循環的に上書きされる．

LBR を用いて分岐情報を記録することで，カーネル内で発生する分岐を監視できる．分岐の監視に LBR を用いる利点を以下に示す．

(利点 1) OS 構造への依存度の低さ
LBR は CPU の機能であり，OS と分離している．このため，LBR による分岐の監視は OS への依存度が低い．

(利点 2) カーネルスタックに情報を積まない分岐命令を監視可能
LBR を用いると，カーネルスタックに情報を積まない分岐命令に対応する分岐情報を記録できる．このため，カーネルスタックに情報を積まない jmp 命令などによる分岐も監視可能である．

なお，分岐を監視する方法として，LBR を用いる方法の他に，CPU エミュレータを用いる方法 [12] や分岐トレース支援機能の 1 つである Branch Trace Store[11]（以降，BTS）を用いる方法がある．しかし，これらの方法には，計算機の性能を大きく低下させるという問題がある [12][13]．一方，LBR による分岐情報の記録のオーバヘッドは非常に小さく，計算機の性能を大きく低下させることなく分岐情報を記録できる [14]．

3.3 基本方式

提案手法の全体図を**図 2** に示し，以下で処理の流れを述べる．

(1) ユーザ空間のプロセスがシステムコールを発行
(2) システムコールハンドラへの移行処理をフック
(3) 発行されたシステムコールが監視対象のシステムコールであるか否かを判定し，以下の処理を実行
  (A) 発行されたシステムコールが監視対象のシステムコールである場合，(4) へ移行

(B) 発行されたシステムコールが監視対象のシステムコールでない場合，提案手法は何もせず，システムコールハンドラへ処理を移行

(4) LBR を有効化し，システムコールハンドラへ処理を移行

(5) 発行されたシステムコールが監視対象のシステムコールであるか否かにより，以下の処理を実行

(A) 発行されたシステムコールが監視対象のシステムコールである場合，システムコールサービスルーチンへの移行処理をフックし，(6) へ移行

(B) 発行されたシステムコールが監視対象のシステムコールでない場合，発行されたシステムコールに対応するシステムコールサービスルーチンへ処理を移行

(6) LBR を無効化し，分岐情報の記録を終了

(7) LBR スタックに記録されている分岐情報の個数と正常な場合の分岐情報の個数を比較し（分岐情報の比較処理については，3.4 節で述べる），カーネルルートキットに感染したと判断した場合，結果をログに出力

(8) LBR スタックに記録されている分岐情報をクリアし（クリアすることにより，次回以降の処理で分岐情報が記録されていない状態から分岐情報の記録を開始できる），発行されたシステムコールに対応するシステムコールサービスルーチンへ処理を移行

提案手法は，オーバヘッドを削減するため，カーネルルートキットに改ざんされやすいシステムコールが発行された場合にのみ分岐を監視する．監視対象のシステムコールは，文献 [3] で改ざんされやすいシステムコールとして挙げられている exit()，fork()，read()，write()，open()，close()，execve()，ioctl()，readlink()，stat64()，lstat64()，getuid32()，および getdents64() の 13 個である．

3.4 分岐情報の比較

3.4.1 検知方法と記録される分岐情報

提案手法は，図 2 の (7) において，正常時に記録される分岐情報とカーネルルートキット感染時に記録される分岐情報を比較することにより，カーネルルートキットを検知する．

提案手法において，LBR により記録される分岐情報は，以下の 4 通りに分類できる．

(1) カーネルルートキットに感染していない場合
LBR が有効になっている間に発生する分岐は，システムコールハンドラのフック関数からシステムコールハンドラへの分岐と，システムコールハンドラからシステムコールサービスルーチンのフック関数への分岐の 2 個である．このため，2 個の分岐情報が記録される．

(2) カーネルルートキットに感染している場合
システムコール処理を改ざんするカーネルルートキットに感染した場合，カーネルルートキットの処理による分岐が発生する．このため，カーネルルートキットに感染していない場合と異なり，3 個以上の分岐情報が記録される．

(3) プロセスがトレースされている場合
gdb などのデバッガや strace などのトレーサによりプロセスがトレースされており，トレースされていることを表すフラグが立っている場合，システムコールハンドラ内からトレースの処理に移行する．トレースの処理とは，デバッガやトレーサからアタッチされるための処理である．プロセスがトレースされている場合，LBR によりトレースの処理による分岐情報が記録される．このため，3 個以上の分岐情報が記録される．

(4) 割り込みが発生した場合
LBR が有効になっている間に割り込みが発生した場合，割り込み処理による分岐情報が記録される．このため，LBR が有効になっている間に割り込みが発生した場合も 3 個以上の分岐情報が記録される．割り込みが発生した場合への対処は，今後の課題とする．

3.4.2 比較方法

3.4.1 項で述べたように，カーネルルートキットに感染していない場合に記録される分岐情報は 2 個であり，それ以外の場合に記録される分岐情報は 3 個以上である．このため，記録された分岐情報が 2 個のとき，カーネルルートキットに感染していないと判断できる．しかし，記録された分岐情報が 3 個以上のときは，上記の 3 通りが存在する．このため，記録された分岐情報が 3 個以上であっても，カーネルルートキットに感染していると断定できない．

そこで，プロセスのトレースは，計算機を利用するにあたり必須の機能ではなく，gdb などのデバッガや strace などのトレーサを用いて利用者が意図的に行うものであることに着目する．分岐情報が 3 個以上記録されている際，利用者が意図的にトレースしているプロセスであるか否か判断し，カーネルルートキットに感染しているか否かを判断する．利用者が意図的にトレースしているプロセスであるか否かの判断は，利用者にプロセスの実行プログラム名とプロセス ID を提示することで達成できる．これは，gdb などのデバッガや strace などのトレーサは，実行プログラム名またはプロセス ID を用いてトレース対象のプロセスを決定するためである．

上記の方法を用いた分岐情報の比較（図 2 の (7)）の流れを**図 3** に示し，以下で述べる．

(1) LBR により記録された分岐情報の取得

(2) 3 個以上の分岐情報が記録されているかの判断

(A) 3 個以上の分岐情報が記録されていない場合，カーネルルートキットに感染していないと判断

(B) 3 個以上の分岐情報が記録されている場合，(3) に移行

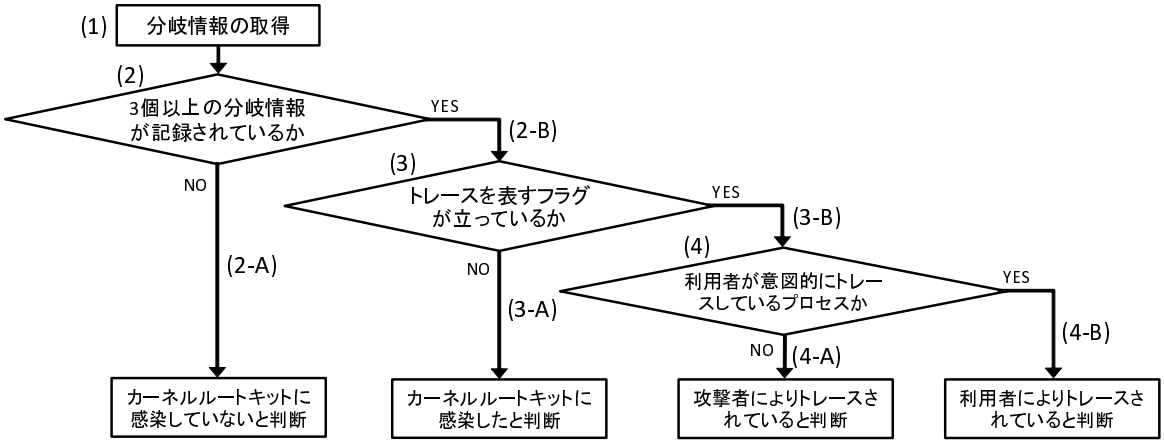

図 3 分岐情報の比較の流れ

(3) トレースを表すフラグが立っているかの判断
  (A) フラグが立っていない場合，カーネルルートキットに感染したと判断
  (B) フラグが立っている場合，(4) に移行
(4) 利用者が意図的にトレースしているプロセスか否かの判断
  (A) 利用者が意図的にトレースしているプロセスでない場合，攻撃者によりトレースされていると判断
  (B) 利用者が意図的にトレースしているプロセスである場合，利用者によりトレースされていると判断

以上のように，LBR により記録された分岐情報の個数を比較することでカーネルルートキットを検知できる．ただし，以上に述べた方法を用いると，利用者が意図的にトレースしているプロセスは提案手法の監視下から外れることになる．しかし，複数のプロセスが動作している環境において，トレースされているプロセス以外のプロセスがカーネルルートキットによる分岐を発生させることにより，カーネルルートキットを検知できる．利用者が動作しているプロセスのすべてをトレースすることは考えにくく，利用者に意図的にトレースされているプロセスを提案手法の監視下から外しても問題ないと考える．

3.5 検知可能なカーネルルートキット

提案手法は，システムコールハンドラのフック関数からシステムコールサービスルーチンのフック関数が呼び出されるまでの分岐を監視するため，この範囲内で分岐を発生させるカーネルルートキットを検知できる．このため，提案手法は，池上らの手法 [3] と比較し，システムコールハンドラに分岐命令を埋め込むカーネルルートキットを新たに検知できる利点がある．ただし，システムコールハンドラを改ざんするカーネルルートキットのうち，システムコールサービスルーチンの処理後に実行されるシステムコールハンドラのコードに分岐命令を埋め込むカーネルルートキットは検知できない．また，カーネルルートキットの中には正規のシステムコールサービスルーチンを呼び出さないものも存在する．この場合，正規のシステムコールサービスルーチンはフックされず，提案手法に処理が移らないことにより，分岐情報の比較が行われないため，提案手法では検知できない．しかし，池上らの手法 [3] のように，システムコールが発行された際，前回のシステムコールにおいて対応するシステムコールサービスルーチンが呼ばれたか否かを確認する機構を導入することで正規のシステムコールサービスルーチンを呼び出さないカーネルルートキットに対処できる．

提案手法が検知できないカーネルルートキットについては，周期的にカーネルを検査する手法 [4],[5],[6] と併用することで対処できる．

3.6 提案手法への攻撃

提案手法は，カーネルレベルで動作する．このため，同じカーネルレベルで動作するカーネルルートキットは，提案手法への攻撃が可能である．提案手法への攻撃への対処として，文献 [5] のように，提案手法が参照するアドレスが書き換えられていないことを一定の周期で検査する方法や，提案手法を仮想マシンモニタを用いて実装する方法がある．

また，提案手法が LBR を用いることをカーネルルートキットの作成者が知っていた場合，ルートキット関数内で LBR を無効化し，LBR スタックの値を適切に書き変えることで提案手法を回避できる問題がある．この問題への対処として，Shadow LBR[16] を用いる方法がある．Shadow LBR とは，仮想マシンモニタを用いてマルウェアによる LBR の無効化を防止する手法であり，LBR を操作するための RDMSR 命令と WRMSR 命令が実行される際，ゲスト OS から仮想マシンモニタへ処理が移ることを利用している．Shadow LBR を用いることで，カーネルルートキットが LBR を操作することによる提案手法の回避を防止できる．

3.7 期待される効果

(効果 1) カーネルルートキットを早期に検知可能
提案手法は，監視対象のシステムコールが発行される度に分岐情報を比較する．これにより，カーネルルートキットに感染後，はじめに監視対象のシステムコールが発行されるタイミングでカーネルルートキットを検知できる．

(効果 2) カーネルの拡張性の維持
提案手法は，カーネルルートキット感染後に記録される分岐情報の変化を検知することでカーネルルートキットを検知する．このため，カーネルモジュールの追加を制限せず，カーネルの拡張性を制限しない．

(効果 3) 異なる OS やバージョンへの移植性の高さ
LBR は CPU の機能であり，LBR による分岐情報の記録は OS 構造と独立している．このため，LBR により記録される分岐情報を用いる手法は，異なる OS やバージョンへ移植しやすい．

(効果 4) カーネルスタックに情報を積まない分岐命令を用いるカーネルルートキットの検知
LBR を用いることでカーネルスタックに情報を積まない分岐命令を監視できるため，カーネルスタックを参照する手法に比べ，カーネルスタックに情報を積まない jmp 命令などの分岐命令を用いるカーネルルートキットも検知できる．

4. 実現方式

4.1 実現における要件

3 章に述べた提案手法を 32 ビット環境の Linux 2.6.32 に LKM として実現する．提案手法を実現するために，以下の技術的な要件を満たす必要がある．

(要件 1) システムコールハンドラ呼び出しのフックとシステムコール番号の取得
LBR を有効化する機構に処理を移すために，システムコールハンドラ呼び出しをフックする必要がある．また，監視対象システムコールであるか否かを判定するために，システムコール番号を取得する必要がある．

(要件 2) システムコールサービスルーチン呼び出しのフック
分岐情報を比較する機構に処理を移すために，システムコールサービスルーチン呼び出しをフックする必要がある．

(要件 3) 分岐情報の取得
提案手法は，LBR により記録される分岐情報を用いてカーネルルートキットを検知する．このため，LBR を用いて分岐情報を取得する必要がある．

(要件 4) プロセスの情報の取得
プロセスがトレースされている場合，プロセスの実行プログラム名とプロセス ID を取得し，ログとして利用者に提示する．このため，プロセスがトレースされているか否かを示すフラグ，プロセス ID，および実行プログラム名を取得する必要がある．

4.2 システムコールハンドラ呼び出しのフックとシステムコール番号の取得

システムコールハンドラへの移行処理のフックは，提案手法の導入時に SYSENTER_EIP_MSR レジスタの値を書き換えることで実現する．SYSENTER_EIP_MSR レジスタは，システムコールが発行された際に実行されるシステムコールハンドラのアドレスを格納するレジスタである．SYSENTER_EIP_MSR レジスタの値を提案手法のフック関数のアドレスに書き換えることで，システムコール発行の度に提案手法のフック関数に処理を移すことができる．

システムコール番号は，システムコール発行時に EAX レジスタに格納される．このためシステムコールハンドラのフック関数内で EAX レジスタを参照することでシステムコール番号を取得できる．

4.3 システムコールサービスルーチン呼び出しのフック

システムコールサービスルーチンへの移行処理のフックは，提案手法の導入時にシステムコールテーブルに格納されているシステムコールサービスルーチンのアドレスを提案手法のフック関数のアドレスに書き換えることで実現する．書き換え対象は，監視対象のシステムコールに対応するシステムコールサービスルーチンのアドレスである．これにより，監視対象のシステムコールに対応するシステムコールサービスルーチンが呼び出される時だけ，提案手法のフック関数に処理を移すことができる．

4.4 分岐情報の取得

LBR を有効化するには，MSR_DEBUGCTLA MSR レジスタの 0 ビット目（LBR フラグ）をセットする．また，無効化するには，LBR フラグをリセットする．分岐情報は LBR スタックと呼ばれるレジスタに 16 個まで保存され，それぞれ 0〜15 の位置番号で表される．最新の分岐情報が記録されている位置番号は，MSR_LASTBRANCH_TOS レジスタの下位 4 ビットに格納される．このため，MSR_LASTBRANCH_TOS レジスタの下位 4 ビットを参照することで最新の分岐情報が記録されている位置番号を取得する．また，LBR スタックの値を読み取ることで分岐情報を取得する．分岐情報のクリアは，LBR スタックの値をすべて 0 に書き換えることで実現する．

4.5 プロセスの情報の取得

プロセスがトレースされている場合，プロセスの実行プログラム名とプロセス ID を取得し，ログとして利用者に提示する．これを実現するために必要な情報を以下に示す．

(1) プロセスがトレースされているか否かを示すフラグ
(2) 実行プログラム名
(3) プロセス ID

上記の情報は，プロセス制御ブロックから取得できる．Linux 2.6.32 では，プロセス制御ブロックは thread_info 構造体と task_struct 構造体から成る．プロセスがトレースされていることを示すフラグは，thread_info 構造体の flags メンバに格納されている．プロセスがトレースされているか否かの判断は，thread_info 構造体の flags メンバを参照することで実現する．また，プロセスの実行プログラム名とプロセス ID は，それぞれ task_struct 構造体の comm メンバと pid メンバに格納されている．プロセスの実行プログラム名とプロセス ID は，task_struct 構造体の comm メンバと pid メンバを参照することで取得する．

5. 評価

5.1 評価の目的と評価環境

評価の目的と評価項目を以下に示す．

(1) カーネルルートキットの検知実験
提案手法を導入した環境にカーネルルートキットを感染させ，提案手法がカーネルルートキットを検知できるか否かを評価する．また，感染前と感染後に記録される分岐情報の変化を確認する．

(2) オーバヘッドの評価
提案手法の導入によるシステムコール 1 回あたりのオーバヘッドを測定する．また，提案手法の導入による実 AP の性能への影響を評価するためにカーネルの make 処理時間の変化を測定する．

評価環境は，CPU は Intel Core i5-3470 3.2GHz，メモリは 4.0GB，カーネルは Linux 2.6.32-5（32bit）である．

5.2 カーネルルートキットの検知実験

提案手法を導入後，カーネルルートキットを検知できることを示す．本実験では，カーネルルートキットとして，KBeast[17] を用いる．KBeast は，システムコールテーブルに格納されているアドレスを改ざんするカーネルルートキットである．

提案手法を導入した環境に KBeast を感染させると，カーネルルートキットを検知したことを示すログが出力されることを確認した．また，KBeast に感染前と感染後に記録されていた分岐情報を出力した結果を**図 4** に示す．図 4 の分岐情報は，番号が若い順に最新の分岐情報が記録されている．図 4 により，KBeast に感染前の分岐情報は 2 個記録されており，KBeast に感染後の分岐情報は 16 個記録されていることがわかる．以上により，提案手法を導入することでカーネルルートキットを検知できた．

```
1 :FROM 0xc10030f4  TO 0xf7cab4a3
2 :FROM 0xf7cab0aa  TO 0xc1003078
3 :FROM 0x00000000  TO 0x00000000
4 :FROM 0x00000000  TO 0x00000000
5 :FROM 0x00000000  TO 0x00000000
6 :FROM 0x00000000  TO 0x00000000
7 :FROM 0x00000000  TO 0x00000000
8 :FROM 0x00000000  TO 0x00000000
9 :FROM 0x00000000  TO 0x00000000
10:FROM 0x00000000  TO 0x00000000
11:FROM 0x00000000  TO 0x00000000
12:FROM 0x00000000  TO 0x00000000
13:FROM 0x00000000  TO 0x00000000
14:FROM 0x00000000  TO 0x00000000
15:FROM 0x00000000  TO 0x00000000
16:FROM 0x00000000  TO 0x00000000
```

(a) KBeastに感染前の分岐情報

```
1 :FROM 0xf7cb504d  TO 0xf7cab4a3
2 :FROM 0xf7cb501d  TO 0xf7cb5038
3 :FROM 0xc113c76c  TO 0xf7cb501b
4 :FROM 0xc113c764  TO 0xc113c754
5 :FROM 0xc113c764  TO 0xc113c754
6 :FROM 0xc113c764  TO 0xc113c754
7 :FROM 0xc113c764  TO 0xc113c754
8 :FROM 0xc113c764  TO 0xc113c754
9 :FROM 0xc113c764  TO 0xc113c754
10:FROM 0xc113c764  TO 0xc113c754
11:FROM 0xc113c764  TO 0xc113c754
12:FROM 0xc113c764  TO 0xc113c754
13:FROM 0xc113c764  TO 0xc113c754
14:FROM 0xc113c764  TO 0xc113c754
15:FROM 0xc113c764  TO 0xc113c754
16:FROM 0xc113c764  TO 0xc113c754
```

(b) KBeastに感染後の分岐情報

図 4 KBeast に感染前と感染後の分岐情報

表 1 open() と getdents64() の処理時間（単位: μs）

システムコール	導入前	導入後	オーバヘッド
open()	0.39	1.18	0.79
getdents64()	0.07	0.85	0.78

表 2 read() の処理時間（単位: μs）

システムコール	ファイルサイズ	導入前	導入後	オーバヘッド
read()	1KB	0.24	1.01	0.77
	100KB	4.26	5.06	0.80

5.3 オーバヘッドの評価

5.3.1 システムコール 1 回当たりのオーバヘッド

提案手法の導入前と導入後でシステムコール 1 回当たりの処理時間を測定することにより，システムコール 1 回当たりのオーバヘッドを測定した．測定対象のシステムコールは，提案手法が監視対象としている open()，getdents64()，および read() である．open() と getdents64() の 1 回あたりの処理時間は，それぞれ 1000 回実行し，処理時間の平均を算出することで測定した．read() の 1 回あたりの処理時間は，1KB のファイルと 100KB のファイルをそれぞれバッファに読み込む処理を 1000 回行い，処理時間の平均を算出することで測定した．

open() と getdents64() の測定結果を**表 1** に示す．また，read() の測定結果を**表 2** に示す．表 1 と表 2 から，システムコール 1 回あたりのオーバヘッドは 0.77μs から 0.80μs であることがわかる．これは，池上らの手法 [3] における open()，getdents64()，および read() の 1 回あたりのオーバヘッド（CPU は Pentium 4，メモリは 4GB の環境において，0.01μs から 0.37μs）と比較すると，提案手法のオーバヘッドは大きい結果となった．しかし，周期的にカーネルメモリの完全性を検査する手法 [5] におけるオーバヘッド（CPU は Pentium 4，メモリは 503.9MB の環境において，mprotect()1 回あたり 7.88μs）と比較すると小さく，提案手法の導入によるシステムコール 1 回当たりのオーバヘッドは，十分に小さいといえる．池上らの手法 [3] と比較してオーバヘッドが大きくなった理由は，提案手法ではシステムコールのフックによるオーバヘッドに加え，LBR

表 3 カーネルの make 処理時間（単位: s）

導入前	導入後	オーバヘッド
2146.43	2162.49	16.06（0.74%）

により分岐情報を取得するためのレジスタの読み書きによるオーバヘッドが発生するためであると考えられる．

5.3.2 カーネルの make 処理時間の変化

提案手法の導入によるカーネル（Linux 2.6.0）の make 処理時間の変化を測定した結果を**表 3** に示す．提案手法の導入によるカーネルの make 処理時間のオーバヘッドは 16.6s（0.74%）であり，提案手法の導入による実 AP の性能への影響は小さいことがわかる．

6. おわりに

カーネル内で発生する分岐を LBR を用いて監視することでカーネルルートキットを検知する手法を提案した．提案手法は，監視対象のシステムコールが発行されるたびに分岐情報を比較する．これにより，カーネルルートキットに感染後，早期に検知できる．また，提案手法は，カーネルモジュールの追加を制限しない．さらに，LBR による分岐情報の記録は OS 構造と分離しているため，異なる OS やバージョンへの移植性が高い．加えて，LBR を用いることで，提案手法は，カーネルスタックに情報を積まない jmp 命令などの分岐命令を用いるカーネルルートキットを検知できる．

評価では，提案手法を導入することでカーネルルートキットを検知できることを示した．また，提案手法の導入による性能への影響は小さいことを示した．

今後の課題として，割り込みが発生した場合への対処と検知可能なカーネルルートキットを増加させることがある．

参考文献

[1] ITmedia Inc. : ここまで来ている，ルートキットの危険性 : ハードウェアの力を借りて見えない脅威に対策せよ，入手先〈http://www.atmarkit.co.jp/ait/articles/1211/01/news001.html〉（参照 2015-02-02）．
[2] ASCII.jp : 知っているようで知らない？ルートキットのすべて，入手先〈http://ascii.jp/elem/000/000/618/618802/〉（参照 2015-02-02）．
[3] 池上祐太，山内利宏 : カーネルスタックの比較によるカーネルルートキット検知手法の提案，情報処理学会論文誌，Vol.55，No.9，pp.2047–2060（2014）．
[4] Petroni, Jr., N.L and Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks, Proceedings of the ACM Conference on Computer and Communications Security (CCS'07), pp.103-115 (2007).
[5] 小倉寛之，大山恵弘，岩崎英哉 : カーネルレベルルートキット検知システムの構築，情報処理学会研究報告，Vol.2008-OS-108，No.35，pp.51-58（2008）．
[6] Wang，X. and Karri，R.: NumChecker: Detecting Kernel Control-flow Modifying Rootkits by Using Hardware Performance Counters，Proc. 50th Annual Design Automation Conference，No.79，pp.1-7 (2013).
[7] Wang, Y-M., Beck, D., Vo, B., Roussev, R. and Verbowski, C: Detecting Stealth Software with Strider GhostBuster , Proc. 2005 International Conference on Dependable Systems and Networks, pp.368-377 (2005).
[8] Kruegel, C., Robertson, W. and Vigra, G.: Detecting Kernel-Level Rootkits Through Binary Analysis, Proc. 20th Annual Computer Security Applications Conference (ACSAC'04), pp.91-100 (2004).
[9] Litty, L., Lagar-Cavilla, H.A. and Lie, D.: Hypervisor Support for Identifying Covertly Executing Binaries, Proc. 17th USENIX Security Symposium, pp.243-258 (2008).
[10] 秋月康志，今泉貴史 : ベアメタルハイパーバイザを用いたカーネルレベルルートキット検知システムの実現，情報処理学会研究報告，Vol.2010-IOT-9，No.7，pp.1-6（2010）．
[11] IA-32 インテル アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル 下巻 : システム・プログラミング・ガイド，入手先〈http://www.intel.co.jp/content/dam/www/public/ijkk/jp/ja/documents/developer/IA32_Arh_Dev_Man_Vol3_i.pdf〉(accessed 2015-01-26).
[12] Portokalidis, G., Slowinska, A. and Bos, H.: Argos: an Emulator for Fingerprinting Zero-Day Attacks, Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06), pp.15-27（2006）．
[13] Soffa, M. L., Walcott, K. R. and Mars, J.: Exploiting Hardware Advances for Software Testing and Debugging (NIER Track), Proc. 33rd International Conference on Software Engineering (ICSE'11), pp.888-891（2011）．
[14] Pappas, V., Polychronakis, M. and Keromytis, A.: Transparent ROP Exploit Mitigation using Indirect Branch Tracing, Proc. 22nd USENIX Security Synposium, pp.447-462 (2013).
[15] Prochazka, B., Vojnar, T. and Drahansky, M. : Hijacking the Linux Kernel, Proc. Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10), pp.85-92 (2010).
[16] Deng, D., Xu, D., Zhang, Z. and Jiang, X.:IntroLib: Efficient and transparent library call introspection for malware forensics, Proc. 12th Annual Digital Forensics Research Conference, pp.S13-S23 (2012).
[17] KBeast, available from〈http://packetstormsecurity.com/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html〉(accessed 2015-01-25).