情報処理学会論文誌 Vol.55 No.9 2047–2060 (Sep. 2014)

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

池上 祐太[1,a)] 山内 利宏[1,b)]

受付日 2013年12月2日, 採録日 2014年6月17日

概要：標的型攻撃でカーネルルートキットを使用する事例が増加している．カーネルルートキットに感染した場合，標的型攻撃の検知までに要する時間が長引き，計算機への被害が拡大する可能性がある．攻撃による被害の抑制には，カーネルルートキットの早期検知が重要である．しかし，既存のルートキット検知手法は，カーネルルートキットを早期に検知できるものが少なく，カーネルの拡張性を制限するという問題がある．そこで，カーネルルートキットに感染前と感染後の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する．提案手法では，カーネルルートキットに改ざんされる可能性の高いシステムコールの発行後に呼び出される正規のシステムコール処理ルーチンの呼び出し前に，処理をフックし，その時点のカーネルスタックの情報をホワイトリストと比較する．また，正規のカーネルモジュールの情報を事前にホワイトリストに登録しておくことで，提案手法による誤検知を防止する．本論文では，提案手法の設計，Linux を対象とした実現方式，および評価結果を報告する．

キーワード：ルートキット，カーネルスタック，オペレーティングシステム，セキュリティ

Proposal of Kernel Rootkits Detection Method by Comparing Kernel Stack

Yuta Ikegami[1,a)] Toshihiro Yamauchi[1,b)]

Received: December 2, 2013, Accepted: June 17, 2014

Abstract: Recently, there is a case which attacker uses kernel rootkits on a target attack is increasing. If a system infected a kernel rootkit, the time required for until the detection of a target attack is prolonged. Moreover, the damage to a system may spread. Therefore, early detection of kernel rootkits is important. However, there are few kernel rootkit detection methods which can detect kernel rootkits earlier. In addition, existing methods limit the extensibility of the kernel. This paper proposes a kernel rootkits detection method which compares a kernel stack before infected a kernel rootkit with that after infected a kernel rootkit. This method compares the white list with a kernel stack before the system call service routine which invoked after system call that is more likely to be tampered with kernel rootkits. Also, the proposed method prevents false positive by registered information of the legitimate kernel module. In this paper, we report design of the proposed method, implementation for Linux, and evaluation results.

Keywords: rootkit, kernel stack, operating system, security

1. はじめに

近年，標的型攻撃の検知を困難にするため，ルートキットを使用する攻撃事例が増加している [1]．ルートキットとは，計算機へ侵入した痕跡，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などを持つプログラムである．上記の機能により，ルートキットに感染した場合でも，管理者には，計算機は正常な場合と同様の振舞いをしているように見える．このため，管理者は，正常な場合の振舞いとルートキットに感染した場合の振舞いの差異に気づか

[1] 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan
[a)] en421603@s.okayama-u.ac.jp
[b)] yamauchi@cs.okayama-u.ac.jp

ず，標的型攻撃の検知までに要する時間が長引く可能性が高くなる．攻撃の検知が遅れると計算機への被害が拡大する恐れがあるため，被害の抑制には，ルートキットの早期検知が重要である．

ルートキットの中には，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．カーネルルートキットを作成するには，OS の仕組みについて深い知識と理解が必要である．しかし，2007 年以降，ルートキットの数は急激に増加し，1 日に約 2,000 個のルートキットが作成されている [2]．これは，SpyEye [3] や Zeus [4] のようなマルウェア作成ツールの発展のためである．これらの現状から，カーネルルートキット（以降，ルートキットと略す）を検知する様々な方式が提案されている [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]．ルートキットを検知するタイミングとして，(1) ルートキットに感染前に検知，(2) ルートキットの実行時または実行直後に検知，(3) ルートキットの実行後に検知の 3 つのタイミングが存在する．ルートキットに感染前に検知する手法として，文献 [10], [11], [12], [13], [14] がある．しかし，これらの手法は，ルートキットの感染前に検知できたとしても，カーネルの拡張性を制限するという問題がある．

そこで，本論文では，カーネルスタックの比較により，ルートキットの実行時または実行直後にルートキットを検知し，カーネルの拡張性を制限しない手法を提案する．提案手法の目的は，多くの種類のルートキットを検知することではなく，できるだけ早くルートキットを検知し，計算機への被害を最小限に抑制することである．提案手法は，ルートキットに感染後，最初の監視対象システムコール発行時，または，その次のシステムコール発行時にルートキットを検知できるため，計算機への被害を最小限に抑制できる．また，正規のカーネルモジュールの誤検知を防止できる．本論文では，既存のルートキット検知手法の問題点を述べ，問題点を解決する提案手法の設計，Linux (x86) を対象とした実現方式，および評価結果を述べる．本論文による貢献は以下の 2 点である．

(1) ルートキットの実行時または実行直後にルートキットを検知し，計算機への被害を最小限に抑制する手法を提案した．

(2) ルートキットの検知において，オーバヘッドが小さく，導入の容易な手法を提案した．

2. ルートキットの改ざん手法と既存のルートキット検知手法

2.1 ルートキットの改ざん手法

ルートキットは感染のために，特定の領域を改ざんすることで，自身の作成した関数を呼び出したり，特定のプログラムを隠蔽したりする．以下は，ルートキットが自身の関数の呼び出しやプログラムの隠蔽を達成するための改ざん手法である．

(1) システムコールテーブルに格納されたアドレスの改ざん

システムコールテーブルに格納されている正規のシステムコール処理ルーチンのアドレスをルートキット関数のアドレスに改ざんする．正規のシステムコール処理ルーチンとは，システムコールハンドラからシステムコールテーブルを参照して呼ばれるルーチンである．たとえば，open システムコールの場合，sys_open である．システムコールテーブルに格納されているアドレスが改ざんされた状態でシステムコールを発行すると，ルートキット関数が呼び出される．

(2) 正規のシステムコール処理ルーチンの呼び出しの横取り

多くのルートキットは，ルートキット関数から正規のシステムコール処理ルーチンを呼び出し，正しいシステムコールの処理へ移行させる．しかし，ルートキット関数から正規のシステムコール処理ルーチンを呼び出さず，システムコールを改ざんすることもできる．

(3) システムコールテーブルのアドレスの改ざん

システムコールハンドラから参照されるシステムコールテーブルのアドレスをルートキットが用意したシステムコールテーブルのアドレスに改ざんする．これにより，システムコールを発行すると，ルートキットが用意したシステムコールテーブルが参照される．

(4) IDT（Interrupt Descriptor Table）の改ざん

IDT のエントリをルートキット関数のアドレスに改ざんする．IDT が改ざんされた状態で割込みが発生すると，ルートキット関数が呼び出される．

(5) IDTR（Interrupt Descriptor Table Register）の改ざん

IDTR をルートキットが用意した IDT のアドレスへ改ざんする．これにより，割込みが発生すると，ルートキットが用意した IDT が参照される．

(6) SYSENTER_EIP_MSR の改ざん

SYSENTER_EIP_MSR に格納されているアドレスをルートキットが用意したシステムコールハンドラのアドレスに改ざんする．これにより，SYSENTER 命令によりシステムコールを発行した場合，ルートキットが用意したシステムコールハンドラが呼び出される．

(7) カーネル関数の改ざん

カーネル関数の先頭コードをルートキット関数へのジャンプ命令に改ざんする．これにより，改ざんされたカーネル関数が呼び出されると，ルートキット関数が呼び出される．

(8) 動的なデータ領域の改ざん

カーネルによって変更される動的なデータ領域を改ざんする．たとえば，Linux では，カーネルモジュールを連結循環のリンクリストで管理している．ルートキットは，リンクリストのポインタを改ざんすることで特定のカーネルモジュールを切り離し，特定のカーネルモジュールを隠蔽できる．

(9) Return-Oriented Programming の利用

Return-Oriented Programming [16] とは，正規のコードに存在する ret 命令に着目し，それらを組み合わせることで任意のコードを実行する攻撃である．この攻撃は，カーネルコードにも適用でき，カーネルの完全性検査（2.2.1 項で説明），Exec-Shield [17]，および NX ビットによるコード実行時の防止を回避 [18] できる．

提案手法では，これらのルートキットのうち，(1)，(2)，(3) を検知対象とする．これは，文献 [19], [20] において，(1)，(2)，(3) のルートキットの割合が高いことを示しているためである．また，提案手法を拡張することで，(1)，(2)，(3) 以外のルートキットも検知できると考える．検知対象のルートキットと提案手法の拡張についての詳細は，5 章で述べる．

2.2 既存のルートキット検知手法

2.2.1 カーネルの完全性検査

文献 [5], [6], [7] の手法は，指定した周期で，ルートキットに感染前のカーネルメモリと現在のカーネルメモリを比較し，差異がないか検査する手法を提案している．文献 [5] は，PCI カードを用いて，異なる計算機から監視対象の計算機のメモリを周期的に検査する．この手法では，検査の周期によっては，ルートキットの検知が遅れる可能性がある．文献 [6] は，システムコール発行のたびに，保存したカーネルメモリとその時点のカーネルメモリを比較する．しかし，1 回のシステムコール発行につき，区切ったサイズ 1 つ分しかカーネルメモリを比較できない．このため，頻繁にシステムコールを発行しない環境では，検知が遅れるという問題がある．文献 [7] は，仮想化技術を利用し，監視ゲスト OS から監視対象ゲスト OS のカーネルメモリを周期的に検査する．このため，検査の周期により，ルートキットの検知が遅れる可能性がある．また，文献 [5], [6], [7] は，カーネルメモリの完全性を検査するため，正規のカーネルモジュールを追加できないという問題がある．

文献 [8], [9] は，仮想化技術を利用し，ハイパーバイザからゲスト OS を監視し，ルートキットからカーネルを保護する手法を提案している．この手法は，カーネルの保護領域の改ざんを試みるカーネルモジュールをルートキットとして検知する．しかし，ルートキットと類似した処理を行う正規のカーネルモジュールをルートキットとして誤検知する．文献 [9] は，事前に許可したカーネルコードのみ実行できるため，正規のカーネルモジュールもルートキットとして誤検知する．また，文献 [8], [9] は，OS によりメモリ構造が異なるため，多種の OS やバージョンへの対応が難しい．

2.2.2 カーネルメモリへの書き込みを禁止することによる検知

文献 [10], [11], [12] は，仮想化技術を利用し，ゲスト OS のカーネルメモリへの書き込みを禁止する検知手法を提案している．ゲスト OS の仮想アドレスに対応するページテーブルエントリの WR ビットをクリアする．WR ビットがクリアされたカーネルメモリ領域へ書き込みがされた場合，ページフォルトが発生する．このページフォルトを検知することで，ルートキットの感染を検知できる．しかし，カーネルメモリ領域の書き込みを禁止することで，正規のカーネルモジュールを追加できないという問題がある．また，OS によりメモリ構造が異なるため，多種の OS やバージョンへの対応が難しい．

2.2.3 バイナリ検査

文献 [13] は，カーネルモジュールのロード時に，カーネルモジュールのバイナリを検査し，ルートキットを検知する手法を提案している．この手法は，ルートキットと類似したバイナリコードが正規のカーネルモジュールに存在する場合，ルートキットとして誤検知する．また，事前にルートキットが使用する可能性の高いバイナリコードを登録する必要があるため，登録したバイナリコード以外のバイナリコードを使用するルートキットは，検知できない．文献 [14] は，仮想化技術を利用し，ゲスト OS へ導入されるカーネルモジュールのバイナリをハイパーバイザから検査する手法を提案している．しかし，計算機へ導入するすべての正規のカーネルモジュールを事前にデータベースへ登録する必要がある．

2.2.4 クロスビュー検知

文献 [15] は，ルートキットに感染した OS で取得したファイル一覧と CD ブートした OS で取得したファイル一覧を比較する．クロスビュー検知とは，ユーザレベルで取得したファイル一覧とファイルシステムやレジストリをスキャンし，取得したファイル一覧を比較する手法である．ルートキットに感染していない OS で取得したファイル一覧を比較に用いるため，ルートキットによるファイル隠蔽を検知できる．しかし，文献 [15] の手法では，利用者の任意のタイミングで 1 度計算機を停止し，検査する必要がある．このため，検知が遅れるという問題がある．

2.3 既存のルートキット検知手法の検知タイミング

2.2 節で述べた手法について，ルートキットを検知するタイミングの早期なものから以下に示す．

(1) ルートキットに感染前に検知

(2) ルートキットの実行時または実行直後に検知

(3) ルートキットの実行後に検知

ルートキットに感染前に検知する手法は，文献 [10], [11], [12], [13], [14] があり，ルートキットの実行時または実行直後に検知する手法は，文献 [8], [9] がある．これらの手法は，正規のカーネルモジュールをルートキットと誤検知するという問題がある．文献 [5], [6], [7], [15] の手法は，周期的に検査する手法，システムコール発行のたびに部分的に

メモリを検査する方法，および利用者の決定したタイミングで検査する手法であり，ルートキットの実行前に検知する可能性がある．しかし，ルートキットの実行後に検知する可能性もあり，被害が発生する前の適切なタイミングで検知することは難しい．このため，ルートキットによる被害を確実に防ぐことはできない．

2.4 既存のルートキット検知手法の問題点

これまでに述べた既存のルートキット検知手法には，以下のいずれかの問題が存在する．

(問題 1) ルートキットを早期に検知不可能

ルートキットによる被害が出る前に検知することが最も重要である．文献 [8], [9], [10], [11], [12], [13], [14] 以外の検知手法では，利用者が設定した周期やタイミングで検知手法を動作させる必要がある．このため，ルートキットの検知が遅れる可能性がある．

(問題 2) カーネルの拡張性の制限

文献 [15] 以外の検知手法では，カーネルの完全性を検査，カーネルメモリへの書き込みの禁止，およびバイナリの検査をするため，正規のカーネルモジュールやルートキットに類似した正規のカーネルモジュールを追加できない．

(問題 3) 多種の OS やバージョンへの対応の困難さ

文献 [5], [13], [14], [15] 以外の検知手法は，OS の種類やバージョンに依存するため，適応性が低い．

3. 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設計

3.1 考え方

事前調査として，2.1 節の (1)，(2)，(3) のルートキット（以降，手法 1 のルートキット，手法 2 のルートキット，および手法 3 のルートキットと略す）の感染時に現れる特徴を調査した．ルートキットは計算機に感染すると，システムコール発行後に呼び出される正規のシステムコール処理ルーチンの呼び出しをフックし，自身の関数を呼び出す．自身の関数の処理後，正規のシステムコール処理ルーチンを呼び出すか，もしくは正規のシステムコール処理ルーチンを呼び出さずに処理を終える．

図 1 に，open システムコールを例として，システムコール発行時の処理の流れとカーネルスタックについて示す．カーネルスタックは，カーネルモードに移行時から使用される．システムコールハンドラの処理により，カーネルスタックには，退避されたレジスタの値，セグメントセレクタ，およびシステムコールからの戻りアドレスなどが積まれる．その後，システムコールハンドラから call 命令でシステムコールテーブルを参照し，正規のシステムコール処理ルーチン（図 1 では，sys_open）が呼び出される．call 命令により，正規のシステムコール処理ルーチンからの戻りアドレスがカーネルスタックに積まれる．正規のシステムコール処理ルーチンは，ret 命令を使用し，call 命令によりカーネルスタックに積まれた戻りアドレスへ処理を移行する．このように，システムコール発行後のカーネル内での正規のシステムコール処理ルーチンを呼び出す場合，必ずカーネルスタックが使用される．

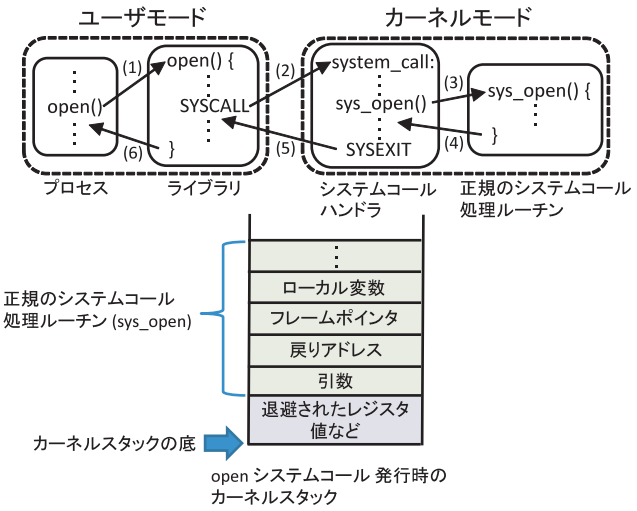

図 1 open システムコール発行時の処理の流れとカーネルスタック

Fig. 1 Flow and kernel stack when open system call was invoked.

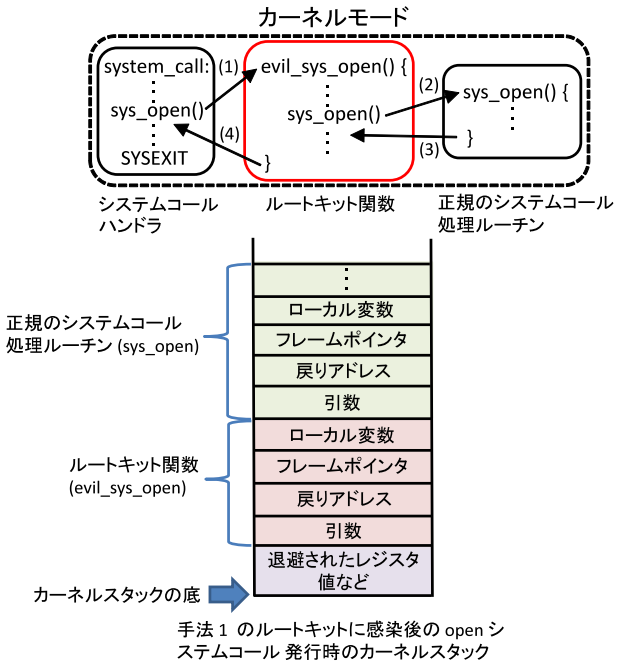

図 2 手法 1 のルートキットに感染後の open システムコール発行時の処理の流れとカーネルスタック

Fig. 2 Flow and kernel stack after infected by rootkit method 1 when open system call was invoked.

図 2 に，手法 1 のルートキット（open システムコールを改ざんする場合）に感染後の open システムコール発行時の処理の流れとカーネルスタックについて示す．ルートキット関数は，感染時にシステムコールテーブルに格納されたアドレスをルートキット関数のアドレスに書き換え

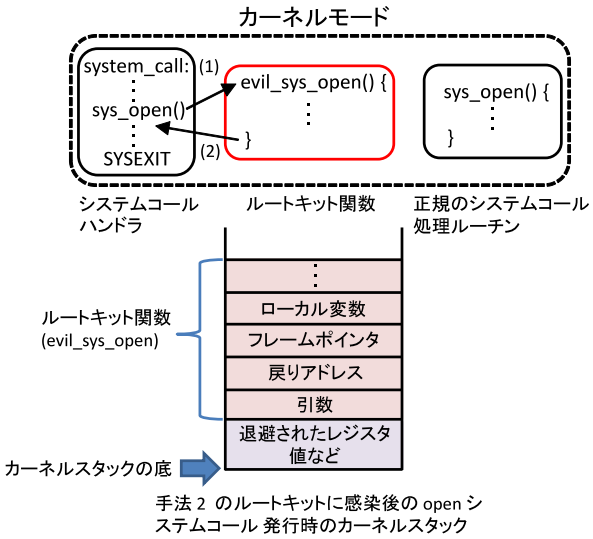

図 3 手法 2 のルートキットに感染後の open システムコール発行時の処理の流れとカーネルスタック

Fig. 3 Flow and kernel stack after infected by rootkit method 2 when open system call was invoked.

る．図 2 のように，システムコールハンドラは，システムコールテーブルに格納されたルートキット関数のアドレスを参照し，正規のシステムコール処理ルーチンの呼び出しと同じ手順でルートキット関数を呼び出すため，カーネルスタックにはルートキット関数の情報が必ず積まれる．手法 1 のルートキットは，ルートキット関数から正規のシステムコール処理ルーチンを呼び出すため，カーネルスタックは，図 2 のように積まれる．

図 3 に，手法 2 のルートキット（open システムコールを改ざんする場合）に感染後の open システムコール発行時の処理の流れとカーネルスタックについて示す．図 2 と同様に，ルートキット関数は，システムコールハンドラから呼び出されるため，カーネルスタックにはルートキット関数の情報が必ず積まれる．手法 2 のルートキットは，ルートキット関数から正規のシステムコール処理ルーチンを呼び出さないため，カーネルスタックに正規のシステムコール処理ルーチンの情報は積まれない．また，手法 3 のルートキットは，正規のシステムコール処理ルーチンを呼び出すものと呼び出さないものがあるため，図 2 と図 3 の両方の積まれ方が考えられる．

このように，手法 1，2，3 のルートキットに感染した場合，システムコールハンドラは，システムコールテーブルに格納されたルートキット関数のアドレスを参照し，正規のシステムコール処理ルーチンの呼び出しと同じ手順でルートキット関数を呼び出すため，カーネルスタックにはルートキット関数の情報が必ず積まれる．

また，文献 [19], [20] やルートキットの調査により，ルートキットに改ざんされる可能性の高いシステムコールを明らかにした．詳細は，3.6 節で述べる．

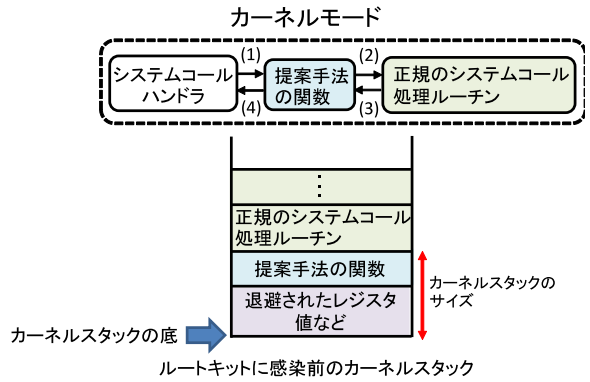

図 4 ルートキットに感染前の提案手法の処理の流れ

Fig. 4 Flow of proposed method before infected by rootkits.

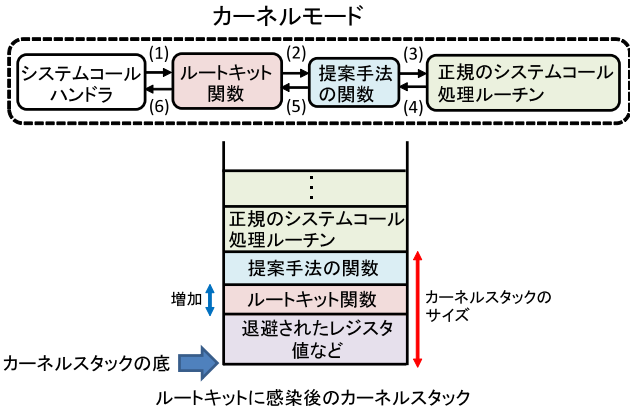

図 5 ルートキットに感染後の提案手法の処理の流れ

Fig. 5 Flow of proposed method after infected by rootkits.

ルートキットに感染後のシステムコール処理において，正規のシステムコール処理ルーチンの呼び出しがルートキットにフックされること，または正規のシステムコール処理ルーチンが呼ばれず，その代わりにルートキット関数が呼ばれることを検知し，2.4 節で述べた問題を解決する．

3.2 基本方式

カーネルスタックの比較により，ルートキットの実行時または実行直後にルートキットを検知する．また，カーネルの拡張性を制限しないという特徴がある．**図 4** と**図 5** に，提案手法導入後のルートキットに感染前と感染後の処理の流れを示す．

提案手法は，ルートキットに感染前に，システムコールテーブルに提案手法の関数のアドレスを登録する．この処理により，図 4 のように，監視対象システムコールの発行後に呼び出される正規のシステムコール処理ルーチンの呼び出し前に，処理をフックし，その時点のカーネルスタックの情報（以降，カーネルスタックの情報と略す）を取得する．次に，取得したカーネルスタックの情報とルートキットに感染前のカーネルスタックの情報を比較する．ルートキットに感染している場合，図 5 のように，ルートキットは，システムコールテーブルに登録されている提

案手法のアドレスを正規のシステムコール処理ルーチンであると判断し，提案手法がルートキット関数によりフックされる．これにより，ルートキット関数から提案手法が呼び出される．正規のシステムコール処理ルーチンの呼び出し前にルートキット関数が呼び出されるため，ルートキットに感染前のカーネルスタックよりカーネルスタックのサイズが増加する．これにより，正規のシステムコール処理ルーチンの呼び出し前に，カーネルスタックのサイズの増加を検知できる．また，図 2 のように，ルートキットに感染後の正規のシステムコール処理ルーチンからの戻りアドレスは，システムコールハンドラのアドレスからルートキット関数のアドレスへと変化する．これより，比較に用いるカーネルスタックの情報は，カーネルスタックのサイズと正規のシステムコール処理ルーチンからの戻りアドレスを用いる．

ルートキットに感染前のカーネルスタックの情報は，事前にホワイトリストとして登録する．手法 1 のルートキットは，現在のカーネルスタックの情報とホワイトリストを比較することで，ルートキットに感染後，最初に監視対象システムコールが発行された時点で検知できる．このため，計算機への被害を最小限に抑制できる．また，手法 2，3 のルートキットに感染すると，正規のシステムコール処理ルーチンが呼び出されない，または正規のシステムコール処理ルーチンが呼ばれたとしても提案手法が呼び出されない．そこで，システムコール発行の直後の処理（図 1 の (2) の処理）をフックし，前回に発行されたシステムコールが監視対象システムコールか否かを確認する．監視対象システムコールが前回に発行されていた場合のみ，提案手法が呼び出されたか検査する．前回の監視対象システムコール発行時に，提案手法が呼び出されていない場合，ルートキットにより処理をフックされていたことが分かる．これにより，正規のシステムコール処理ルーチンを呼び出さないルートキットを次のシステムコールの発行時に検知できる．システムコールを 1 回しかフックしないため，計算機への被害が少ないうちにルートキットを検知できる．上記の処理により，（問題 1）を解決できる．

（問題 2）を解決するために，監視対象システムコールをフックする正規のカーネルモジュールを導入する場合，事前に正規のカーネルモジュールを動作させ，カーネルスタックの情報をホワイトリストとして取得する．当該カーネルモジュールを導入した状態でのカーネルスタックの情報をホワイトリストとして登録するため，正規のカーネルモジュールの誤検知を防止でき，カーネルの拡張性を制限しない．

（問題 3）を解決するために，提案手法を LKM（Loadable Kernel Module）で実現する．提案手法は，カーネルのシンボルテーブル（System.map）を参照し，システムコールをフックする．このため，System.map を取得可能であれば，OS のバージョンへの依存性は低い．

各問題への対処から，提案手法への要件は以下のようになる．

（**要件 1**） システムコール発行と正規のシステムコール処理ルーチンの呼び出しの監視

（**要件 2**） カーネルスタックの情報の取得

（**要件 3**） ホワイトリストの作成

3.3 利用形態

提案手法の利用形態は，ホワイトリスト作成期間と実運用期間の 2 つに分類される．ホワイトリスト作成期間は，監視対象システムコールがすべて発行されるまでの期間である．ホワイトリスト作成期間でのルートキットの感染は想定しない．ホワイトリストは，提案システムが導入された後，最初の各監視対象システムコール発行時に作成するため，作成期間は短い．なお，頻繁に発行されない監視対象システムコールについては，監視対象システムコールを呼び出すプログラムを実行することで，対処できる．また，監視対象システムコールをフックする正規のカーネルモジュールを導入する場合，提案手法の導入前にカーネルモジュールを導入し，ホワイトリストに登録する．実運用期間は，ホワイトリスト作成期間の終了から，ルートキットを検知する期間である．

3.4 提案手法の構成

提案手法の全体像を図 **6** に示す．提案手法は，以下の 4 つの機構により実現する．

- システムコールフック機構
- システムコール呼び出し確認機構
- カーネルスタック情報取得機構
- カーネルスタック比較機構

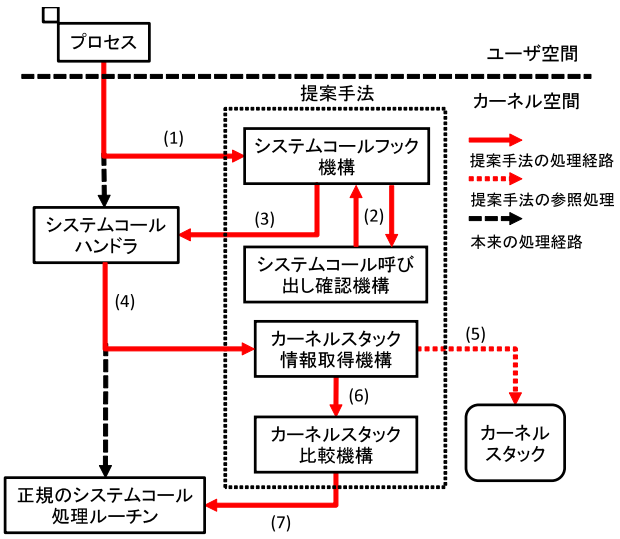

図 6 提案手法の全体像

Fig. 6 Architecture of proposed method.

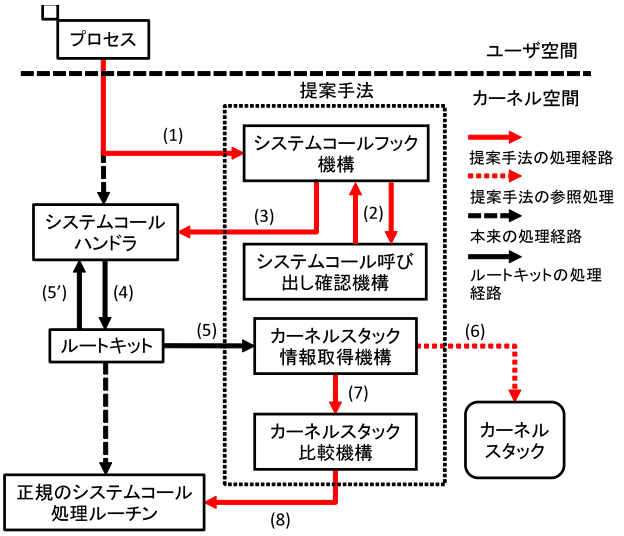

図 7 提案手法とルートキット処理の流れ
Fig. 7 Architecture of proposed method and flow of rootkits.

提案手法の処理の流れを以下に示す．

(1) システムコール発行をフック
(2) システムコール呼び出し確認機構を呼び出し
(3) システムコールハンドラを呼び出し
(4) 正規のシステムコール処理ルーチンをフック
(5) カーネルスタックの情報を取得
(6) カーネルスタックを比較
(7) 正規のシステムコール処理ルーチンを呼び出し

また，提案手法とルートキットの処理の流れを図 **7** に示す．正規のシステムコール処理ルーチンを呼び出すルートキット（手法 1，3）に感染した場合，システムコールハンドラから (4)，(5)，(6)，(7) と処理され，カーネルスタック比較機構でルートキットを検知する．正規のシステムコール処理ルーチンを呼び出さないルートキット（手法 2，3）に感染した場合，(5) ではなく，(5′) の処理へ移るため，カーネルスタック情報取得機構が呼び出されない．その後，次のシステムコール発行時に，システムコール呼び出し確認機構により，ルートキットを検知する．

なお，本論文では，提案手法への攻撃やシステムの管理者による不正は想定しない．（要件 1）には，システムコールフック機構，システムコール呼び出し機構，およびカーネルスタック比較機構により対処する．（要件 2）と（要件 3）には，カーネルスタック情報取得機構により対処する．

3.5 システムコールフック機構

システムコールフック機構は，システムコール発行後に呼び出されるシステムコールハンドラの呼び出し前に，処理をフックし，システムコール番号を取得する．取得したシステムコール番号は，システムコール呼び出し確認機構で使用する．

Linux 2.6 以降では，システムコールの発行に標準で SYSENTER 命令を使用する．このため，システムコールフック機構は，SYSENTER_EIP_MSR をシステムコールフック機構のアドレスに書き換えることで，システムコール処理をフックする．SYSENTER_EIP_MSR の書き換えは，提案手法の導入時に行う．これにより，システムコール発行のたびにシステムコールフック機構へ処理が移行する．

3.6 システムコール呼び出し確認機構

システムコール呼び出し確認機構は，前回に発行されたシステムコールのシステムコール番号を参照し，監視対象システムコールか否かを検査する．監視対象システムコールの場合，前回に発行されたシステムコールにおいて，カーネルスタックの情報を取得できているか検査する．カーネルスタックの情報を取得できていない場合，正規のシステムコール処理ルーチンが呼び出されていない，または正規のシステムコール処理ルーチンが呼ばれたとしても提案手法が呼び出されていないことが分かる．これにより，手法 2，3 のルートキットを次のシステムコール発行時に検知できる．

監視対象システムコールは，文献 [19], [20] とダウンロードしたルートキットの調査から，exit()，fork()，read()，write()，open()，close()，execve()，ioctl()，readlink()，stat64()，lstat64()，getuid32()，および getdents64() の 13 個に決定した．文献 [19] は，10 種類のルートキットを解析し，ルートキットがフックするシステムコールを分析している．文献 [20] は，9 種類のルートキットを実行させ，ルートキットがフックするシステムコールを分析している．

3.7 カーネルスタック情報取得機構

カーネルスタック情報取得機構は，システムコールハンドラによる正規のシステムコール処理ルーチンの呼び出しをフックし，カーネルスタックの情報を取得する．正規のシステムコール処理ルーチンの呼び出しのフックは，事前にシステムコールテーブルを書き換え，提案手法のアドレスを登録することで，実現できる．取得するカーネルスタックの情報は，カーネルスタックのサイズと正規のシステムコール処理ルーチンからの戻りアドレスである．カーネルスタック情報取得機構の処理の流れを以下に示す．

(1) 正規のシステムコール処理ルーチンの呼び出し前にフック
(2) 現在の EBP レジスタの値を取得

現在の EBP レジスタの値を取得する．EBP レジスタとは，スタックフレームの底のアドレスを格納するレジスタである．

(3) 正規のシステムコール処理ルーチンからの戻りアドレスを取得

戻りアドレスは，フレームポインタの 4 バイト上位のアド

レスへ格納されている．このため，EBP レジスタの値に，4 バイト加算することで，正規のシステムコール処理ルーチンからの戻りアドレスを取得できる．

(4) thread_info 構造体のアドレスを取得

続いて，カーネルスタックのサイズを取得するため，カーネルスタックと共有してメモリを割り当てられている thread_info 構造体のアドレスを取得する．カーネルスタックと thread_info 構造体に割り当てられているメモリ領域の大きさは，THREAD_SIZE マクロで定義されている．このため，thread_info 構造体のアドレスに，THREAD_SIZE（実装した環境では，8,192 バイト）を加算することで，カーネルスタックの底のアドレスを取得できる．

(5) カーネルスタックの底のアドレスを取得

(6) カーネルスタックのサイズを取得

カーネルスタックの底のアドレスから，現在の ESP レジスタを減算することで，カーネルスタックのサイズを取得できる．ESP レジスタとは，スタックフレームの先頭のアドレスを格納するレジスタである．

3.8 カーネルスタック比較機構

カーネルスタック比較機構は，取得したカーネルスタックの情報とホワイトリストを比較する．手法 1 のルートキットに感染前と感染後のカーネルスタックの比較を図 8 に示す．ルートキットに感染後のカーネルスタックは，ルートキットに感染前のカーネルスタックと比べ，ルートキット関数の分だけサイズが大きくなる．また，カーネルスタックに積まれている提案手法からの戻りアドレスがルートキット関数のアドレスを指すように変化する．比較結果が異なる場合，ルートキットに感染していると判断し，利用者への警告としてログを出力する．これにより，手法 1 のルートキットを，ルートキットに感染後の最初のシステムコール呼び出し時に検知できる．

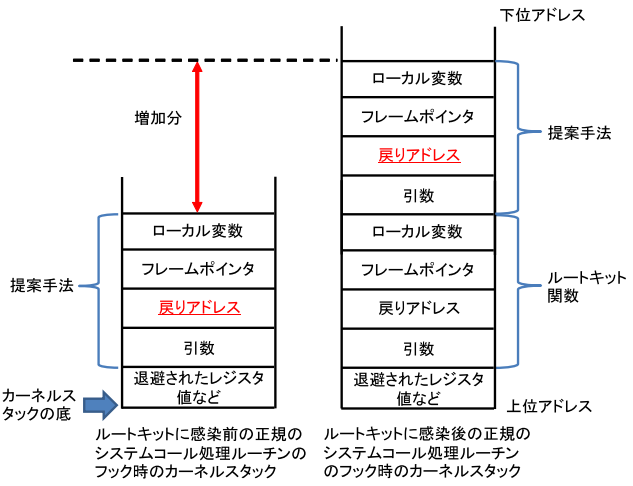

図 8 カーネルスタックの比較

Fig. 8 Comparison of kernel stack.

3.9 期待される効果

提案手法の導入により期待される効果を以下に示す．

(1) ルートキットの実行時または実行直後に検知可能

手法 1 のルートキットは，現在のカーネルスタックの情報とホワイトリストを比較することで，ルートキットに感染後，最初に監視対象システムコールが発行された時点で検知できる．また，手法 2，3 のルートキットは，監視対象システムコール発行のその次のシステムコールの発行時に，検知できる．このため，提案手法は，ルートキットの感染後，最初の監視対象システムコール発行時，または，その次のシステムコール発行時に検知できるため，感染を受けた計算機の被害を最小限に抑制できる．

(2) カーネルの拡張性を制限しない

監視対象システムコールをフックする正規のカーネルモジュールを導入する場合，事前に正規のカーネルモジュールを動作させ，カーネルスタックの情報をホワイトリストとして取得する．当該カーネルモジュールを導入した状態でのカーネルスタックの情報をホワイトリストとして登録するため，正規のカーネルモジュールの誤検知を防止でき，カーネルの拡張性を制限しない．

(3) OS のバージョンによる依存性が低い

提案手法が OS に依存するのは，カーネルのシンボルテーブルのみである．Linux では，カーネル導入時にシンボルテーブルを生成し，特定のディレクトリに保存する．提案手法は，シンボルテーブルのみ取得できればよいため，OS のバージョンへの依存性は低い．

(4) 既存のルートキット検知手法との併用による検知対象の拡張

提案手法は，カーネルスタックの情報を比較し，ルートキットを検知する．既存のルートキット検知手法において，カーネルスタックを比較するものは著者らの調査では見つからなかった．このため，提案手法を既存のルートキット検知手法と併用することにより，検知対象を拡張できる．

4. 評価

4.1 評価の目的と評価環境

評価の目的と評価の観点を以下に示す．

(1) ルートキットの検知

提案手法の導入により，ルートキットを検知できることを示し，ルートキットに感染前と感染後のカーネルスタックの変化を明らかにする．また，ルートキットの検知にかかる時間を明らかにする．

(2) 正規のカーネルモジュールの導入による誤検知

提案手法の動作中に，正規のカーネルモジュールを正常に追加できるか明らかにする．また，提案手法の動作中に，監視対象システムコールをフックする正規のカーネルモジュールが正しく動作するか明らかにする．

(3) 提案手法と既存のアンチウイルスソフトとの併用
提案手法と既存のアンチウイルスソフトを併用して動作できるか明らかにする．
(4) 提案手法の導入によるオーバヘッドの測定
提案手法は，システムコールをフックしているため，システムコール単位での処理のオーバヘッドを測定する．また，提案手法の導入によるアプリケーションプログラム（以降，AP と略す）の性能への影響を明らかにする．
(5) ホワイトリストの増加による性能への影響
提案手法のホワイトリストの増加によるシステムコールのオーバヘッドを測定し，性能への影響を明らかにする．
(6) 既存のルートキット検知手法との比較
提案手法と既存のルートキット検知手法を比較し，提案手法の優位性を明らかにする．

(2)，(3)，(4)，および (5) の評価環境は，CPU は Intel Pentium 4 3.60 GHz，メモリは 2.0 GB，カーネルは Linux 2.6.32-5，ディストリビューションは，Debian 6.0.7 を用いた．また，(4) の評価において，ホワイトリストのエントリ数は，監視対象システムコールの数と同じ 13 個である．

4.2 ルートキットの検知実験

実験で使用したルートキットは，2.1 節で示したすべての改ざん手法に対応している．2.1 節の (2)，(3)，(5)，(6)，(9) の改ざん手法を用いるルートキットは，入手できなかったため，同様の改ざん手法を用いるルートキットを自作した．検知実験で使用するルートキットと検知結果を**表 1** に示す．Hook_getuid は，getuid32() をフックし，正規のシステムコール処理ルーチンを呼び出さないように作成した．Hook_SCT は，システムコールハンドラを書き換え，ルートキットが用意したシステムコールテーブルを呼び出し，read() の正規のシステムコール処理ルーチンをフックするように作成した．Hook_IDT は，IDT の 0x80 番目に格納されたエントリを改ざんし，ルートキットの処理を呼び出すように作成した．Hook_IDTR は，IDTR を書き換え，ルートキットの処理を呼び出すように作成した．Hook_MSR は，SYSENTER_EIP_MSR を書き換え，ルートキットの処理を呼び出すように作成した．ROP1-rootkit は，Linux のカーネルイメージから ROP（Return-Oriented Programming）に利用できる命令が格納されたアドレスを探し，そのアドレスを呼び出すことで，ROP を行うように作成した．ROP2-rootkit は，システムコールテーブルに格納された open() の正規のシステムコール処理ルーチンのアドレスを改ざんした後，ROP を行うように作成した．また，EnyeLKM 1.3 は，2 種類の改ざん手法を使用する．

表 1 より，提案手法の導入により，提案手法が対象とするルートキットを検知できた．また，提案手法の導入により，ルートキットの感染から検知までの時間とルートキットに感染前と感染後のカーネルスタックの情報がどのように変化するか調査した．調査のため，KBeast と Hook_getuid のプログラムを改変し，ルートキットの感染時に「Rootkit: Inserted.」という文字列をログとして出力させる．提案手法がルートキットを検知した場合，「Rootkit detector: Detected a rootkit.」という文字列をログとして出力させる．

図 9 に提案手法が KBeast を検知した際のログの出力を示す．ログに表示された時間から，提案手法は，KBeast に感染してから，0.25 ms で KBeast を検知できた．**図 10** に提案手法が Hook_getuid を検知した際のログの出力を示す．ログに表示された時間から，提案手法は，Hook_getuid に感染してから，0.37 ms で Hook_getuid を検知できた．Hook_getuid の検知時間が KBeast よりも 0.12 ms 遅くなっ

```
[   47.464952] Rootkit detector: Start rootkit detection.
[   64.060113] Rootkit: Inserted.
[   64.060360] Rootkit detector: Detected a rootkit.
```

図 9 KBeast を検知した際のログ
Fig. 9 Logs when KBeast was detected.

表 1 提案手法の導入によるルートキットの検知実験の結果
Table 1 Results of rootkits detection experiment by importing proposed method.

ルートキット	2.1 節での分類番号	改ざん手法	検知の可否	検知したシステムコール
KBeast [21]	(1)	システムコールテーブルに格納されたアドレスの改ざん	○	read(), write(), getdents64()
EnyeLKM 1.3 [22]	(1)	システムコールテーブルに格納されたアドレスの改ざん	○	getdents64()
Hook_getuid	(2)	正規のシステムコール処理ルーチンの呼び出しの横取り	○	getuid32()
Hook_SCT	(3)	システムコールテーブルのアドレスの改ざん	○	read()
Hook_IDT	(4)	IDT の改ざん	×	
Hook_IDTR	(5)	IDTR の改ざん	×	
Hook_MSR	(6)	SYSENTER_EIP_MSR の改ざん	×	
EnyeLKM 1.3	(7)	カーネル関数の改ざん	×	
adore-ng-0.56 [23]	(8)	動的なデータ領域の改ざん	×	
ROP1-rootkit	(9)	Return-Oriented Programming の利用	×	
ROP2-rootkit	(1), (9)	システムコールに格納されたアドレスの改ざんと Return-Oriented Programming の利用	○	open()

```
[   1692.992344] Rootkit detector: Start rootkit detection.
[   2034.868356] Rootkit: Inserted.
[   2034.868728] Rootkit detector: Detected a rootkit.
```

図 10 Hook_getuid を検知した際のログ
Fig. 10 Logs when Hook_getuid was detected.

表 2 KBeast に感染前と感染後のカーネルスタックのサイズの変化
Table 2 Change of kernel stack size before infected by KBeast and after infected by KBeast.

変化したシステムコール	KBeast に感染前 (Bytes)	KBeast に感染後 (Bytes)
read()	96	156
write()	96	124
getdents64()	96	132

表 3 KBeast に感染前と感染後の提案手法の戻りアドレスの変化
Table 3 Change of return address before infected by KBeast and after infected by KBeast.

変化したシステムコール	KBeast に感染前	KBeast に感染後
read()	0xc10030fb	0xf7e1e7a2
write()	0xc10030fb	0xf7e527f3
getdents64()	0xc10030fb	0xf7e2abb7

たのは，監視対象システムコール発行のその次のシステムコールの発行時に検知するためである．

次に，KBeast に感染前と感染後のカーネルスタックのサイズの変化を**表 2** に示す．KBeast に感染前と感染後のカーネルスタックのサイズは，read() の場合は 60 バイト，write() の場合は 38 バイト，および getdents64() の場合は 36 バイト増加した．また，KBeast に感染前と感染後の戻りアドレスの変化を**表 3** に示す．KBeast に感染前と感染後で，正規のシステムコール処理ルーチンの呼び出し前の戻りアドレスが変化している．KBeast に感染後の正規のシステムコール処理ルーチンの呼び出し前の戻りアドレスは，KBeast の関数のアドレスである．これらの結果から，ルートキットの感染により，カーネルスタックのサイズの増加と正規のシステムコール処理ルーチンの呼び出し前の戻りアドレスが変化することが分かる．

4.3 正規のカーネルモジュールの導入による誤検知の評価

評価に使用するカーネルモジュールを以下に示す．

(1) kvm_intel
(2) kvm
(3) Dazuko

評価に用いるカーネルモジュールは，実際に計算機に導入する機会の多い KVM（Kernel-based Virtual Machine）のカーネルモジュールとして，kvm と kvm_intel を選択した．また，Dazuko [24] は，特定のファイルなどの書き込みを監視するカーネルモジュールであり，Linux や FreeBSD に対応している．Dazuko のバージョン 2 では，システムコールテーブルに格納されたアドレスを書き換え，open() などのシステムコールが発行された場合，Dazuko の関数が呼び出される．指定したファイルの呼び出しがあった場合，ユーザにそのことを伝える．これらの処理から，Dazuko はルートキットと類似した処理内容を行う正規のカーネルモジュールであり，評価に利用した．Dazuko の評価は，Dazuko のバージョン 2 が正常に動作できる Linux 2.4.22 のバージョンを使用した．

表 4 提案手法と既存のアンチウイルスソフトの併用の結果
Table 4 Results of combining proposed method and anti-virus software.

アンチウイルスソフト	併用の可否
ClamAV	○
avast! Linux Edition	○
Avira Antivirus	○

提案手法を動作させ，kvm_intel と kvm を導入したところ，誤検知せずカーネルへ導入できた．一方，Dazuko を導入したところ，提案手法は，Dazuko をルートキットとして検知した．これは，Dazuko が監視対象システムコールをフックするため，提案手法が誤検知を発生させた．そこで，事前に Dazuko を導入し，カーネルスタックの情報をホワイトリストとして登録した後，提案手法を導入すると，誤検知なく Dazuko をカーネルに導入できた．以上より，監視対象のシステムコールをフックしないカーネルモジュールは，誤検知なく導入でき，監視対象のシステムコールをフックするカーネルモジュールは，カーネルスタックの情報を事前にホワイトリストとして登録することで，誤検知を防ぐことができる．これにより，提案手法は，カーネルの拡張性を制限することなく導入できる．

また，ホワイトリストは，監視対象システムコール単位で登録する．このため，監視対象システムコールをフックするすべてのカーネルモジュールを導入する場合，カーネルモジュールを事前に導入し，その状態での各監視対象システムコール呼び出し時のカーネルスタックの情報をホワイトリストとして登録する．つまり，監視対象システムコール 1 つにつき，ホワイトリストが 1 つ増加する．これより，実運用においても，多くのホワイトリストを登録する必要性は小さいと考える．

4.4 提案手法と既存のアンチウイルスソフトとの併用実験

Linux 上で動作する ClamAV [25]，avast! Linux Edition [26]，および Avira Antivirus [27] のそれぞれと提案手法を併用できるか実験する．**表 4** に実験結果を示す．提案手法といずれのアンチウイルスソフトとの組合せにおいても，正常に併用できた．提案手法を導入し，ClamAV と avast! Linux Edition を手動スキャンしたところ，正常に動作した．Avira Antivirus において，リアルタイムスキャ

ンを実行したところ，正常に動作した．また，提案手法の動作を確認するために，KBeast と Hook_getuid を動作させたところ，提案手法により検知でき，正常に動作した．これにより，提案手法と既存のアンチウイルスソフトは，併用できることを明らかにした．

4.5 提案手法の導入によるオーバヘッドの測定

4.5.1 監視対象システムコールにおけるオーバヘッドの測定

提案手法がフックする監視対象システムコールのオーバヘッドを**表 5** に示す．また，**表 6** に，read() と write() のシステムコールにおいて，1 KB 分を入出力する場合と 100 KB 分を入出力した場合のオーバヘッドを示す．表 5 と表 6 のスコアは，各システムコールを 1,000 回実行し，1 回あたりの平均値を算出した．表 5 と表 6 から，1 回のシステムコールあたり，0.01 μs から 2.36 μs のオーバヘッドであることが分かる．また，ほとんどの場合で，0.4 μs 未満のオーバヘッドである．これにより，提案手法の導入による監視対象システムコールのオーバヘッドは，十分に小さいことが分かる．

4.5.2 提案手法の導入による AP の性能への影響

提案手法の導入による AP の性能への影響を評価するため，提案手法の導入前と導入後において，Web サーバの性能を測定し，比較した．評価に用いた Web サーバは，Apache 2.2.16 である．評価では，ApacheBench 2.3 を用いて，1 KB，10 KB，および 100 KB のファイルに対し，それぞれ 1,000 回アクセスした際のスループットを測定した．測定結果を**表 7** に示す．表 7 の括弧内の数値は，提案手法の導入前の性能を 100%としたときの性能比を示している．表 7 より，提案手法は，1 KB，10 KB，および 100 KB のファイルに対し，提案手法の導入前と比べ，1%しか性能が低下しなかったことが分かる．提案手法の導入により影響するオーバヘッドは，監視対象システムコールのフックによるオーバヘッドのみである．4.5.1 項の測定結果から，表 7 の数値は適当であると推察できる．これにより，提案手法の導入による AP の性能の低下は小さいことが分かる．

4.5.3 カーネルのビルド時間の測定

提案手法の導入によるカーネルのビルド時間の変化を測定した結果を**表 8** に示す．提案手法の導入前と導入後で，カーネルのビルド時間のオーバヘッドは，1.01 s（0.15%）となり，提案手法による AP の性能への影響は，十分に小さいことが分かる．

4.5.4 ホワイトリストの増加による性能への影響

ホワイトリストの増加による性能への影響を評価するため，ホワイトリストのエントリ数を 13（デフォルト），50，100 としたときの open()，close()，stat64() のオーバヘッドを測定した．評価結果を**表 9** に示す．表 9 の括弧内の数値は，ホワイトリストのエントリ数が 13 個の場合を 100%としたときの性能比を示している．表 9 のように，ホワイトリストの増加によるオーバヘッドは生じていない．これは，監視対象システムコールに対応するカーネルスタックの情報の比較は，配列参照で行うため，ホワイトリストを増加させても性能への影響はないからである．

なお，ホワイトリストは，監視対象システムコールごとに作成される．このため，監視対象システムコールが増加すると，提案手法が呼び出されるシステムコールが増加するものの，1 回の呼び出しオーバヘッドは小さいため，処理全体への影響は表 7 と表 8 が示すように小さい．

4.5.5 既存のルートキット検知手法との比較

提案手法で検知できるルートキットは，既存のルート

表 5 監視対象システムコールのオーバヘッド（単位：μs）
Table 5 Overheads of monitored system calls (μs).

システムコール	提案手法導入前	提案手法導入後	オーバヘッド
fork()+exit()	142.18	142.49	0.31 (0.22%)
fork()+execve()	487.64	490	2.36 (0.48%)
open()	1.61	1.62	0.01 (0.62%)
close()	0.27	0.29	0.02 (7.4%)
ioctl()	0.90	0.92	0.02 (2.2%)
readlink()	2.32	2.42	0.10 (4.3%)
stat64()	0.98	0.99	0.01 (1.02%)
lstat64()	1.01	1.02	0.01 (0.99%)
getuid32()	0.25	0.26	0.01 (4.0%)
getdents64()	0.28	0.29	0.01 (3.6%)

表 6 read() と write() のオーバヘッド（単位：μs）
Table 6 Overheads of read() and write() (μs).

システムコール	ファイルサイズ	提案手法導入前	提案手法導入後	オーバヘッド
read()	1 KB	1.80	1.86	0.06 (3.3%)
	100 KB	21.12	21.49	0.37 (1.8%)
write()	1 KB	0.90	0.95	0.05 (0.56%)
	100 KB	12.77	13.01	0.24 (1.9%)

表 7 Apache のスループット（単位：処理要求数/s）
Table 7 Throughputs in Apache Web server (Requests/s).

	提案手法の導入前	提案手法の導入後
1 KB File	4,409.85 (100%)	4,376.34 (99%)
10 KB File	4,163.95 (100%)	4,126.04 (99%)
100 KB File	1,164.66 (100%)	1,164.59 (99%)

表 8 提案手法の導入によるカーネルのビルド時間とオーバヘッド
Table 8 Overheads in building kernel by importing proposed method.

提案手法の導入前 (s)	提案手法の導入後 (s)	オーバヘッド
654.187	655.197	1.01 (0.15%)

表 9 ホワイトリストの増加によるシステムコールのオーバヘッド
Table 9 Overheads of system calls by increasing whitelist.

システムコール	ホワイトリストのエントリ数（個）	オーバヘッド（μs）
open()	13	1.62 (100%)
	50	1.62 (100%)
	100	1.62 (100%)
close()	13	0.29 (100%)
	50	0.29 (100%)
	100	0.29 (100%)
stat64()	13	0.99 (100%)
	50	0.99 (100%)
	100	0.99 (100%)

表 10 本評価と文献 [6] の評価環境
Table 10 Environmental evaluation of this evaluation and Ref. [6].

	提案手法	文献 [6]
CPU	Intel Celeron（2.53 GHz）	Intel Pentium 4（2.66 GHz）
メモリ	512 MB	503.9 MB
ディストリビューション	Ubuntu 7.04	Ubuntu 7.04
カーネル	Linux 2.6.20	Linux 2.6.20.3-ubuntu1

キット検知手法でも検知できる．たとえば，文献 [6] では，SYSENTER_EIP_MSR を書き換え，SYSENTER 命令をフックし，フック先のコードでシステムコールテーブルやシステムコールハンドラなどの完全性を検査する．文献 [6] は，カーネルメモリの本来不変であるほとんどの領域を検査するため，提案手法より検知できるルートキットの種類は多い．たとえば，文献 [6] では，提案手法では検知できない adore-ng を検知できる．しかし，文献 [6] は，監視するカーネルメモリを固定長に分割し，1 回のシステムコール発行につき，分割したサイズ 1 つ分しかカーネルメモリを比較できない．このため，オーバヘッドが大きいという問題がある．また，ルートキットの検知が遅れる可能性がある．そこで，提案手法と文献 [6] のオーバヘッドとルートキットの検知の即時性を比較し，提案手法の優位性を明らかにする．提案手法と文献 [6] を比較するため，文献 [6] と同等の性能を持つ計算機に提案手法を導入し，文献 [6] で記述されている評価方法を試した．本評価と文献 [6] の評価環境を**表 10** に示す．

表 11 に，カーネルのビルド時間のオーバヘッドの平均時間を示す．表 11 より，提案手法のオーバヘッドは，0.22 s（0.035%）であり，文献 [6] のオーバヘッド（7.9%）よりも十分に小さいといえる．

ルートキットの検知の即時性において，提案手法は，ルートキットに感染後，最初の監視対象システムコール発行時，または，その次のシステムコール発行時にルートキットを検知できる．これより，ルートキットに感染後，1 回，または 2 回のシステムコール内でルートキットを検知できる．一方，文献 [6] が採用しているカーネルメモリの分割サイズ（256 バイト）の場合，検査領域のカーネルメモリの分割個数は，10,697 個となるため，ルートキットの検知までの平均システムコール発行回数は，5,348.5 回となる．これより，提案手法は，文献 [6] より，ルートキットの検知の即時性に優れていることが明らかである．

表 11 提案手法の導入によるカーネルのビルド時間とオーバヘッド（文献 [6] と比較）
Table 11 Overheads in building kernel by importing proposed method (Comparison with Ref. [6]).

	導入前（s）	導入後（s）	オーバヘッド
提案手法	646.753	647.893	1.14 (0.18%)
文献 [6]	580	626	46 (7.9%)

5. 考察

5.1 提案手法が検知可能なルートキットについて

提案手法は，手法 1，2，3 のルートキットしか検知できないという問題がある．しかし，文献 [19], [20] において，手法 1，2，3 のルートキットの割合が高いことを示している．文献 [19] は，ハイパーバイザからゲスト OS 上で動作するルートキットを解析する手法を提案している．10 種類のルートキットのうち，6 種類のルートキットが手法 1，2，3 であることを示している．文献 [20] は，ハイパーバイザからゲスト OS 上のメモリを検査し，ルートキットがフックする可能性のある領域を収集する手法を提案している．9 種類のルートキットを実行させ，6 種類のルートキットが手法 1，2，3 であることを示している．また，提案手法の目的は，多くの種類のルートキットを検知することではなく，できるだけ早くルートキットを検知し，計算機への被害を最小限に抑制することである．提案手法は，ルートキットの感染後，最初の監視対象システムコール発行時，または，その次のシステムコール発行時に検知できるため，感染を受けた計算機の被害を最小限に抑制できる．提案手法の拡張として，割込みハンドラやファイルシステム内部関数のカーネルスタックの情報を検査することで，他の改ざん手法のルートキットを検知できると考える．また，4.4 節の評価のとおり，既存のアンチウイルスソフトと併用できるため，提案手法が検知できないルートキットを既存のアンチウイルスソフトにより検知できる可能性がある．既存のアンチウイルスソフトとの併用の際，オーバヘッドが問題点と考えられる．しかし，4.5 節の評価から，提案手法の導入によるオーバヘッドは十分に小さいため，既存のアンチウイルスソフトと十分併用できると考える．

5.2 提案手法への攻撃について

提案手法は，LKM で実装しているため，ルートキットから提案手法への攻撃や回避される可能性がある．提案手法への攻撃や回避への対処として，提案手法をハイパーバイザや文献 [28] のように SMM（System Management Mode）で実装することが考えられる．提案手法は，監視するシステムコールの発行直後にカーネルスタックを取得できればよいため，ハイパーバイザや SMM でも容易に実装できると推察できる．

5.3 OS の異なるバージョンへの提案手法の適用について

提案手法は，LKM で実現しているため，他のバージョンへの対応が容易である．4 章の評価では，ルートキットやカーネルモジュールが動作できるバージョンに対応させるため，提案手法を改修する必要があった．利用した Linux のバージョンは，2.6.32-5，2.6.22，2.6.20，2.6.12，2.4.22 である．この際，提案手法を改修した箇所は，システムコールの引数や SYSENTER_EIP_MSR を書き換える関数がバージョンにより異なっていたため，その箇所を改修する程度であった．このため，提案手法は OS のバージョンへの依存性は低いといえる．

また，Linux の 64 ビット環境のカーネルスタックのサイズや使用方法は，32 ビット環境と同様であり，提案手法を 64 ビット環境で実現できることを確認した．ただし，64 ビット環境の場合，データ幅が 64 ビットであり，使用されるレジスタも 32 ビット環境と異なるため，この違いに対応する必要がある．

5.4 異なる OS への提案手法の適用について

マルウェアに感染する多くのシステムは，Windows と Android 端末である．Windows において，カーネルレベルでの動作は，カーネルスタックを利用している．Android 端末においては，Android 用に最適化された Linux カーネル上に実現しているため，カーネルレベルでの動作は，カーネルスタックを利用している．このため，Windows と Android 端末の両者において，提案手法を適用できると推察できる．また，5.3 節で述べたように，Windows と Android 端末においても，32 ビット環境と 64 ビット環境の両者において提案手法を適用できると推察できる．

6. おわりに

既存のルートキット検知手法の問題点を 3 つ述べ，これらの問題を解決するために，カーネルスタックの比較によるルートキット検知手法を提案した．提案手法は，監視対象システムコールの発行後に呼び出される正規のシステムコール処理ルーチンの呼び出し前に，処理をフックし，現在のカーネルスタックの情報とホワイトリストを比較することでルートキットを検知する．提案手法は，ルートキットに感染後，最初の監視対象システムコール発行時，または，その次のシステムコール発行時にルートキットを検知できるため，計算機への被害を最小限に抑制できる．また，ルートキットと類似した正規のカーネルモジュールの誤検知を防止できる．

評価では，提案手法が対象とするルートキットを検知できた．検知の際，ルートキットに感染後のカーネルスタックのサイズは増加し，戻りアドレスが変化することを明らかにした．また，提案手法を導入した環境に，KVM と Dazuko を導入し，正規のカーネルモジュールの誤検知を評価した．KVM は，ホワイトリストを作成せず，正常にカーネルに導入できた．Dazuko については，事前に Dazuko を導入し，カーネルスタックの情報をホワイトリストとして登録することで，正常にカーネルに導入できた．これにより，提案手法は，誤検知を防ぐことができ，カーネルの拡張性を制限しないことを示した．提案手法と既存のアンチウイルスソフトとの併用では，3 種類のアンチウイルスソフトとのいずれの組合せにおいても，正常に併用できた．

性能評価として，提案手法の導入によるオーバヘッドをシステムコール単位や実際の AP を使用して測定した．監視対象システムコール 1 回あたり，0.01 μs から 2.36 μs のオーバヘッドであり，十分に小さいことを示した．Web サーバの性能は，提案手法の導入前と比べ，1%しか性能が低下しなかった．また，ホワイトリストを増加させても性能への影響がないことを示した．さらに，提案手法と既存のルートキット検知手法との比較では，オーバヘッドとルートキットの検知の即時性において提案手法が優れていることを示した．

今後の課題として，提案手法を Windows と Android に適用し，提案手法の有効性を評価することがある．

参考文献

[1] 国内で最も使われた標的型攻撃ツールは何だ!?—2012 年上半期の標的型攻撃分析，入手先 ⟨http://wp.techtarget.itmedia.co.jp/contents/?cid=12026⟩（参照 2013-11-29）．

[2] Root Out Rootkits, available from ⟨http://www.mcafee.com/us/resources/white-papers/wp-root-out-rootkits.pdf⟩(accessed 2013-11-29).

[3] Sood, A.K., Enbody, R.J. and Bansal, R.: Dissecting SpyEye – Understanding the design of third generation botnets, *The International Journal of Computer and Telecommunications Networking*, Vol.57, No.2, pp.436–450 (2013).

[4] Dr. Ibrahim, L.M. and Thanoon, K.H.: Detection of Zeus Botnet in Computers Networks and Internet, *International Journal of Information Technology and Business Management*, Vol.6, No.1, pp.84–89 (2012).

[5] Petroni, Jr., N.L., Fraser, T., Molina, J. and Arbaughi, W.A.: Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor, *Proc. 13th USENIX Security Symposium*, pp.179–194 (2004).

[6] 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の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).
[7] Petroni, Jr., N.L. and Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks, *Proc. 14th ACM Conference on Computer and Communications Security* (*CCS '07*), pp.103–115 (2007).
[8] Xiong, X., Tian, D. and Liu, P.: Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions, *NDSS Symposium 2011* (2011).
[9] Riley, R., Jiang, X. and Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing, *Proc. 11th International Symposium on Recent Advances in Intrusion Detection*, pp.1–20 (2008).
[10] Murakami, J.: A Hypervisor IPS based on Hardware Assisted Virtualization Technology, *Black Hat USA* (2008).
[11] 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).
[12] McAfee Deep Defender, available from ⟨http://www.mcafee.com/japan/products/deep-defender.asp⟩ (accessed 2013-11-29).
[13] Kruegel, C., Robertson, W. and Vigna, G.: Detecting Kernel-Level Rootkits Through Binary Analysis, *Proc. 20th Annual Computer Security Applications Conference* (*ACSAC '04*), pp.91–100 (2004).
[14] 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).
[15] 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).
[16] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), *Proc. 14th ACM Conference on Computer and Communications Security* (*CCS '07*), pp.552–561 (2007).
[17] Exec-Shield, available from ⟨http://www.redhat.com/magazine/009jul05/features/execshield/⟩ (accessed 2013-11-29).
[18] Hund, R., Holz, T. and Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms, *Proc. 18th USENIX Security Symposium*, pp.383–398 (2009).
[19] Riley, R., Jiang, X. and Xu, D.: Multi-Aspect Profiling of Kernel Rootkit Behavior, *Proc. 4th ACM European Conference on Computer Systems* (*EuroSys '09*), pp.47–60 (2009).
[20] Wang, Z., Jiang, X., Cui, W. and Ning, P.: Countering kernel rootkits with lightweight hook protection, *Proc. 16th ACM Conference on Computer and Communications Security* (*CCS '09*), pp.545–554 (2009)
[21] KBeast, available from ⟨http://packetstormsecurity.com/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html⟩(accessed 2013-11-29).
[22] EnyeLKM, available from ⟨http://www.fr33project.org/pages/projects/enyelkm.htm⟩(accessed 2013-11-29).
[23] adore-ng-0.56, available from ⟨http://huaidan.org/archives/3058.html⟩(accessed 2013-11-29).
[24] Dazuko, available from ⟨http://dazuko.dnsalias.org/wiki/index.php/Main_Page⟩(accessed 2013-11-29).
[25] ClamAV, available from ⟨http://www.clamav.net/lang/en/⟩(accessed 2013-11-29).
[26] avast! Linux Edition, available from ⟨http://files.avast.com/files/linux/avast4workstation_1.3.0-2_i386.deb⟩ (accessed 2013-11-29).
[27] Avira AntiVir, available from ⟨http://avira-antivir.en.malavida.com/linux/download⟩(accessed 2013-11-29).
[28] Zhang, F., Leach, K., Sun, K. and Stavrou, A.: SPECTRE: A Dependable Introspection Framework via System Management Mode, *Proc. 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (*DSN '13*), pp.1–12 (2013).

池上 祐太 （学生会員）

2013 年岡山大学工学部情報工学科卒業．同年同大学大学院自然科学研究科博士前期課程入学．現在，在学中．コンピュータセキュリティに興味を持つ．

山内 利宏 （正会員）

1998 年九州大学工学部情報工学科卒業．2000 年同大学大学院システム情報科学研究科修士課程修了．2002 年同大学院システム情報科学府博士後期課程修了．2001 年日本学術振興会特別研究員（DC2）．2002 年九州大学大学院システム情報科学研究院助手．2005 年岡山大学大学院自然科学研究科助教授．現在，同准教授．博士（工学）．オペレーティングシステム，コンピュータセキュリティに興味を持つ．2010 年度，2012 年度情報処理学会論文賞受賞．電子情報通信学会，ACM，USENIX 各会員．