情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 2　163–176 (June 2010)

VMMによるアプリケーションを意識したカーネル内の振舞い制御

尾上浩一[†1,*1]　大山恵弘[†2]　米澤明憲[†1]

カーネルレベルで稼働するマルウェア（カーネルレベルマルウェア）による攻撃は，システム全体に被害を与えたり，攻撃の検出が困難であったりすることから，その脅威は深刻である．これまでカーネルレベルマルウェアに対する様々なセキュリティシステムが提案されているが，保守的すぎるカーネル拡張の制限や適用時の実行時の性能低下に関して改善すべき点がある．本論文では，仮想マシンモニタ（VMM）を用いて，VM 内で稼働する OS カーネルの振舞いを制御するセキュリティシステム ShadowXeck を提案する．この制御は，読み込み専用のメモリ領域の保護と，OS カーネルにより発行された間接呼び出し命令や間接ジャンプ命令の制御によって実現される．ShadowXeck は，OS カーネルレベルよりも高い特権レベルの VMM による制御であるため，VM 内から ShadowXeck の振舞い制御機構を無効化することは困難である．我々は，AMD64 アーキテクチャ上で Xen を用いて ShadowXeck を実装し，既存のカーネルレベルマルウェアを用いた ShadowXeck による OS カーネルの振舞い制御の確認や実行時オーバヘッドの計測を行った．

Application-aware Control of Kernel Behavior through VMM-based Interposition

Koichi Onoue,[†1,*1] Yoshihiro Oyama[†2] and Akinori Yonezawa[†1]

Threat of kernel-level malware, malware running at the kernel-level, is serious because it compromises a whole operating system and is often capable of hiding itself. Various security systems have been proposed to protect operating systems or applications from kernel-level malware. However, these systems have drawbacks such as forbidding the use of any kernel extensions or significant performance degradation. In this paper, we propose ShadowXeck, a VMM-based security system that controls the behavior of OS kernels running in a VM (called target OS kernels). The control is achieved in two ways. First, it prevents modification to read-only kernel-level memory. Second, it applies application-aware control to indirect call instructions and indirect jump instructions issued by target OS kernels. Since ShadowXeck runs in the VMM layer, malware running on a target OS has difficulty in stopping or bypassing the protection. We implemented ShadowXeck by using Xen on the AMD64 architecture. We confirmed through experiments that ShadowXeck could protect a target OS kernel from attacks by actual kernel-level malware. We also measured runtime overheads imposed by ShadowXeck.

1. はじめに

コンピュータへの攻撃手法が多様化・複雑化し，その脅威がますます大きくなってきている．攻撃手法の 1 つである，カーネルレベルで稼働するマルウェア（カーネルレベルマルウェア）を利用した攻撃は，被害がシステム全体におよぶ点，利用者やアンチマルウェアツールによる検出が困難である点等から，その脅威は特に深刻である．カーネルレベルマルウェアは多くの場合，マルウェアの一部としてカーネルレベルルートキットを利用することで，悪意あるプログラムの隠蔽，バックドアの挿入や privilege escalation 等を実現しようと試みる．カーネルレベルマルウェアによる脅威の多くは OS カーネルの制御フロー（OS カーネルの振舞い）を改竄する攻撃であり，これらの攻撃を検出，防止，または解析するための手法が数多く提案されている[10),11),13),15),16),18)]．

しかし，既存手法においては以下の 2 つが課題となる．まず第 1 に，カーネル空間のデータ保護が保守的すぎる点があげられる．結果として，利用者が認証されたカーネルモジュールしか利用できない等，カーネルモジュール等を利用して OS カーネルを拡張する自由度が制限されてしまう．第 2 に，実行時のカーネル空間のデータ保護によって，システムの性能が低下するという点があげられる．実行時制御はアプリケーションよりも下位層の OS カーネルの実行コンテキスト中でマルウェアを検査するため，多くの場合，アプリケーションレベルの実行時検査よりも検査回数が増加する．Linux 等のようにカーネル空間をすべて

†1 東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo
†2 電気通信大学情報理工学部総合情報学科
Department of Informatics, Faculty of Informatics and Engineering, The University of Electro-Communications
∗1 現在，富士通研究所
Presently with Fujitsu Laboratories Ltd.

のプロセスで共有する場合には，すべてのプロセスに対して制御処理が適用されてしまい，性能低下の原因の 1 つとなる．また，CPU エミュレータを用いた既存手法の場合[11),18),22)]，すべての実行命令をエミュレートすることにより大きく性能が低下する．

本論文では，仮想マシンモニタ（VMM）を用いて，利用者が制御したいプログラム（被制御プログラム）のカーネルレベルの振舞いを制御するセキュリティシステム *ShadowXeck* を提案する．ShadowXeck では，以下の 2 つの手法により VM 内で稼働する OS カーネル（被制御 OS カーネル）の制御フローの安全性を保つ．まず 1 つは，被制御 OS カーネルのコード領域等の読み込み専用領域への書き込みを禁止することで，被制御 OS カーネルの制御フローの安全性を保つ．もう 1 つは，被制御 OS カーネルが間接操作命令（間接呼び出しと間接ジャンプ）を発行したときに，利用者が与えたセキュリティポリシに基づき，遷移先の実行アドレスを制御することで，被制御 OS カーネルの制御フローの安全性が保たれる．

被制御プログラムに対する制御は，被制御プログラムが稼働する仮想マシン（被制御 VM）とは異なる VM（制御 VM）から操作される．ShadowXeck による制御は，OS カーネルレベルよりも高い特権レベルで稼働する VMM による制御である．このため，ShadowXeck は，被制御 VM から ShadowXeck の振舞い制御機構を無効化できないという利点を持つ．

ShadowXeck では，カーネルモジュール等を用いた被制御 OS カーネルの拡張を制限しない．これにより，被制御 VM 内で拡張機能が有効になった後，被制御プログラム以外のプログラムに対してはその拡張機能が有効となる．他方，被制御プログラムのコンテキストでの被制御 OS カーネルの振舞い（間接操作命令）は ShadowXeck によって制御される．被制御プログラムのコンテキストでのみ VMM に発行された間接操作命令を制御させるため，我々は被制御 OS カーネル空間の部分的な多重化を用いる．これにより，被制御 OS カーネルによる間接操作命令の制御回数を削減することができる．

我々は，VMM として Xen[2)] を，被制御 OS として Linux を用いて，提案システム ShadowXeck の設計および実装を CPU アーキテクチャAMD64 上で行った．ShadowXeck の評価では，(1) カーネル空間の読み込み専用領域が保護されていることの確認と，(2) 被制御プログラムに関連する被制御 OS カーネルの振舞いがカーネル拡張に影響されないことの確認，(3) ShadowXeck により生じる実行時オーバヘッドの計測を行った．

以降，本論文は以下のように構成されている．まず，2 章でカーネル空間のデータ保護について述べ，3 章で ShadowXeck における脅威モデルについて述べる．そして，4 章で ShadowXeck に関して説明し，5 章で ShadowXeck の実装について述べる．さらに，6 章で ShadowXeck を評価し，関連研究を 7 章で述べる．最後に，8 章でまとめと今後の課題について述べる．

2. カーネル空間のデータの保護

カーネルレベルマルウェアによる攻撃は，OS カーネル上で稼働するすべてのプロセスに対して影響がおよぶ．さらに，アンチマルウェアツールによる検出や削除を迂回するため，カーネルレベルマルウェアは，アンチマルウェアツールを無効化したりカーネルレベルマルウェア自身を隠蔽したりする機能を有する場合が多い．カーネルレベルマルウエアの脅威に対抗するため，これまで様々なカーネルレベルマルウェアを検出・防止するシステムが提案されている[10),11),13)–16),18),21),23)]．

しかし，既存の手法は以下の 2 つの点が改善すべき課題としてあげられる．1 つ目は保守的なカーネルレベルマルウェア検出にともなう，OS カーネル拡張機能の利用の制限である．たとえば，Linux のカーネルモジュールは，システムを再起動する必要がないという利便性を持つため，OS カーネル拡張機能を実現する手段の 1 つとして有用である．しかし，カーネルレベルマルウェアを防止する既存手法を適用した場合，利用者が制限なくカーネルモジュールを利用することができなくなってしまう．たとえば，Patagonix[13)] では，システム内で有効にさせるカーネルモジュールは，あらかじめシステムに登録しておく必要がある．また，文献 10) の手法では，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正当性を検査する．しかし，このロード時の検査でカーネルモジュールが不当であると誤認されうる．たとえば，アンチマルウェアツールにカーネルモジュールが含まれ，カーネルレベルマルウェアのカーネルモジュールと類似した自己防衛機能を持つ場合である．この場合，マルウェアの一部であるカーネルモジュールとアンチマルウェアツールのカーネルモジュールを区別することは困難である．

2 つ目の課題は，カーネルレベルマルウェアによる攻撃検出にともなう，実行時オーバヘッドの発生である．OS カーネルの命令参照時やデータ参照時にカーネル領域のデータを制御する場合（リアルタイム制御），アプリケーションレベルの検査よりも検査回数が増加する．Linux のように，カーネル領域をすべてのプロセスで共有している場合には，被制御プログラムのコンテキストだけでなくシステム全体での制御となるため，これにより実行時オーバヘッドが大きくなってしまう．また，CPU の命令単位で制御，解析する既存の多くのシステムでは CPU エミュレータが利用されている[18),20),22),23)]．しかし，この場合には実行命令のエミュレーションによる大きなオーバヘッドが生じる．このため，既存のシステムでは，実行時オーバヘッドが考慮されない場合が多い．また，リアルタイム制御ではな

く，Copilot [14] のように，周期的にカーネル領域のデータの改竄を検査する場合には，周期間隔を広げれば実行時オーバヘッドを削減することができる．しかし，周期間隔を広げることによって，タイミング攻撃等が成功する可能性が増加してしまう．

提案システム ShadowXeck は，カーネルレベルの振舞いの中で被制御プログラムに関する振舞いのみを制御することで，既存手法の上記の 2 点を改善する．ShadowXeck では被制御 VM 内でのカーネルモジュール等の OS カーネル拡張機能の利用を制限しない．被制御 VM 内の OS カーネル拡張機能は被制御プログラム以外に対して有効となるが，被制御プログラムに関する振舞いは ShadowXeck によって制御される．また，被制御プログラム以外のコンテキストで OS カーネルが稼働しているときに VMM による制御が発生しないため，実行時オーバヘッドを削減できる．

3. 脅威モデル

ShadowXeck における脅威モデルは，カーネルレベルマルウェアが被制御 OS カーネルのメモリ領域を改竄し，被制御 OS カーネルの制御フローを変更する攻撃である．既存のカーネルレベルマルウェアの多くが制御フローを変更する攻撃である．たとえば，Linux における 26 の既存のカーネルレベルマルウェアの中で，25 のカーネルレベルマルウェアが制御フローを変更する機能を含む[⋆1]．ShadowXeck が対象とするアッタクベクタは以下の 2 つである．

- カーネルのコード領域等の読み込み専用領域を書き込み可能領域に変更し，データを改竄する攻撃．書き込み可能領域への変更は，ページテーブルの R/W ビットを変更したり，読み込み専用領域を含む物理ページを書き込み可能でメモリマップしたりすることで実現可能である．
- 間接呼び出し命令や間接ジャンプ命令のように，実行時に決定される被制御 OS カーネルの制御フローを改竄する攻撃．たとえば，被制御プログラムがファイル操作を実行したときに経由する仮想ファイルシステム（VFS）や/proc ファイルシステムに関連する関数ポインタを攻撃用コードにリダイレクトするような攻撃である．

ShadowXeck はカーネルレベルマルウェアを検出するシステムではなく，たとえカーネルレベルマルウェアが被制御 VM 内に存在したとしても，被制御プログラムの実行中は正当な OS カーネルの制御フローを維持させるシステムである．また，ShadowXeck は，利用者が指定した特定のプログラムがカーネルレベルマルウェアの影響を受けないようにさせるシステムである．たとえば，ShadowXeck を利用する 1 つのシナリオは，カーネルレベルマルウェアの攻撃対象となることが多いシステムユーティリティプログラム（ps や ls 等）を制御対象として指定し，これらプログラムへのカーネルレベルマルウェアの影響を無効化することである．また，別のシナリオとして，ユーザレベルで稼働する，サンドボックスシステムや侵入検知・防止システム等のセキュリティシステムを補完するために，ShadowXeck を利用することもできる．たとえば，システムコール捕捉に基づくセキュリティシステム[7),17)] はシステムコールの実行前後を制御できるが，OS カーネル内のシステムコール処理は制御できない．これらのセキュリティシステムの制御対象であるプログラムを ShadowXeck でも制御対象として指定することで，OS カーネル内のシステムコール処理も制御できるようになる．他方，すべての OS カーネルの動作を制御したい場合には，ShadowXeck は適当ではない．たとえば，キーロガーによるデータ漏洩をシステム全体で防ぎたい場合，すべてのキーボード処理に対する制御が必要となる．ShadowXeck ですべてのプログラムを指定することで OS カーネル全体への攻撃を防ぐことはできるが，ShadowXeck の 2 つの目標が損なわれてしまう．

本論文では，カーネルレベルマルウェアの攻撃対象となる OS を Linux とし，その攻撃手段はカーネルモジュールと特殊デバイスファイル（/dev/mem，/dev/kmem）を利用した攻撃を想定している[⋆2]．ShadowXeck における，カーネルモジュール等による被制御 OS のカーネル拡張は，被制御 OS カーネルの設計方針を逸脱しない範囲で許可する．たとえば，コード領域のように，OS カーネルが読み込み専用としている物理ページを書き込み可能に変更するような OS カーネル拡張は ShadowXeck では許可しない．

本論文で対象としていない攻撃は，OS カーネルがシステム全体で管理するデータを改竄する攻撃である．たとえば，プロセスリストやカーネルモジュールリストを改竄し，悪意あるプロセスやカーネルモジュールを隠蔽する攻撃である．また，ランキューを改竄し，被制御プログラムをスケジューリング対象からはずすような攻撃も本論文では対象としていない．これらのデータ改竄型の攻撃を緩和するために，我々は，Lycosid [9] や VMwatcher [8] のような，VM の内部と外部で取得したデータの違いによりデータの改竄を検出する仕組みと併用することを想定している，ただし，この仕組みは被制御 VM 全体に対して適用されるた

⋆1 文献 16) に記載されている 25+PhalanX のカーネルレベルマルウェア．

⋆2 /dev/mem や/dev/kmem は，既存の配布されている OS カーネルでは安全性の面から利用不可にされている場合が多い．

め，制御対象ではないプログラムに対する性能も低下する．また，return-into-libc のように，カーネルスタック上のデータを改竄する攻撃も対象としていない．我々は StackGuard [4] やスタック位置のランダム化 [12] と組み合わせることにより，この攻撃を緩和することを想定している．

4. 提案システム：ShadowXeck

4.1 概　　要

提案システム ShadowXeck の構成を**図 1** に示す．ShadowXeck における信頼すべき構成要素（TCB）は VMM と制御 VM である．ShadowXeck では，制御 VM から被制御 OS カーネルの振舞いに関する制御操作を実行する．

VMM 内と制御 VM 内の制御機構は以下の構成要素からなる．

- VMM 内コンポーネント SX-core：被制御 OS に関する情報（被制御 OS 情報）に基づき，プログラム単位で被制御 OS カーネルの振舞いを制御する．
- 制御コマンドユーティリティ：被制御 OS 情報の登録や制御の開始を SX-core に通知するためのコマンドを提供する．また，セキュリティポリシの雛形を生成するためのコマンドも含まれる．
- 制御デーモン・制御プログラム：制御デーモンが被制御プログラムごとに対応する制御プログラムを生成する．制御プログラムは，被制御プログラムの開始から終了時まで，

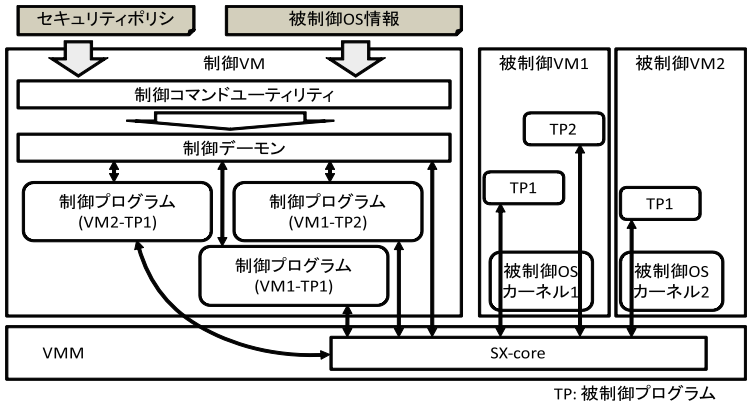

図 1　提案システム：ShadowXeck
Fig. 1　Proposed system: ShadowXeck.

実行ログの生成等の SX-core の補助処理を行う．

ShadowXeck は被制御プログラムごとに生成されたセキュリティポリシに基づいて，被制御 OS カーネルによって発行された間接操作命令を制御する．間接操作命令を制御するために，ShadowXeck は 2 つの操作モード（プロファイリングモード，制御モード）を提供している．利用者は，まず，プロファイリングモードで被制御プログラムを実行させ，SX-core と制御プログラムが連携し，間接操作命令に関する実行ログを記録する．その後，生成された実行ログを利用して利用者がセキュリティポリシを生成する．制御モードでは，ShadowXeck が生成されたセキュリティポリシに基づいて被制御 OS カーネルが発行した間接操作命令を制御する．セキュリティポリシは制御 VM で管理する．

ShadowXeck は被制御 OS 情報を用いてプログラム単位で制御する．被制御 OS 情報には，被制御 OS カーネルにおけるプロセスやシステムコール，間接操作命令に関連する情報が含まれる．プロセスやシステムコールに関連する情報は，SX-core による被制御プログラムの開始やプロセス生成の識別，システムコール手続きの開始・終了の捕捉のために用いられる．被制御 VM の外側から VM 内のプロセスやシステムコールを制御する方法に関しては，文献 25) で記載しているため，本論文では省略する．

4.2 OS カーネルの振舞い制御

OS カーネルの振舞いを制御するために，我々はメモリ上の読み込み専用領域の保護と間接操作命令の実行時制御を行う．まず，VMM により，被制御 OS カーネルの設計方針で読み込み専用領域となる，コード領域やシステムコールテーブル等の読み込み専用データの改竄を防止する．

読み込み専用領域が保護されたうえでも被制御 OS カーネルの振舞いを改竄する攻撃手法の 1 つに，機械語レベルの間接操作命令を改竄する方法が考えられる．我々が制御対象とする間接操作命令は間接呼び出し命令と間接ジャンプ命令であり，プログラミング言語レベルの関数ポインタ経由の関数呼び出し等に対応する．間接操作命令はレジスタまたはメモリ経由で次の命令の遷移先が決定される．攻撃者は遷移先が保存されているレジスタまたはメモリ上のデータを改竄することによって攻撃用コードにリダイレクトさせる等の攻撃を実現できる．この攻撃を無効化するために，我々は被制御 OS カーネルが間接操作命令を発行したときに遷移先を制御する．

4.3 被制御プログラムに関連する振舞いのみの制御

被制御プログラムに関連する OS カーネルが発行した間接操作命令のみ制御するため，**図 2** で示されているように，ShadowXeck は間接操作命令を含むカーネル領域を多重化す

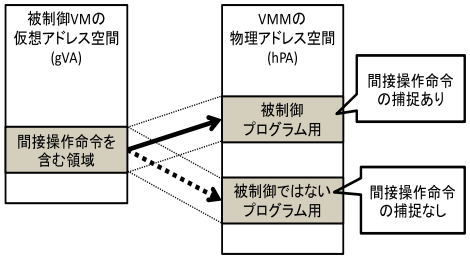

図 2　間接操作命令を含むカーネル領域の多重化
Fig. 2　Multiplexing kernel space including indirect operation instructions.

る．VMM が被制御 VM のメモリ操作を管理しているため，被制御 OS が操作する仮想アドレス（gVA）に対応する物理メモリ上のアドレス（hPA）への変換を VMM が制御できる．カーネル領域の多重化により，被制御プログラムとそれ以外のプログラムで同じ被制御 OS カーネルの間接操作命令を実行しても，異なる物理ページが参照される．我々は，被制御プログラムが間接操作命令を実行したときのみ VMM による捕捉が発生し，実行時制御が行われるように設定する．被制御プログラム以外のプログラムが間接操作命令を実行したときには実行時制御が発生しないため，被制御プログラム以外の性能低下を軽減させることができる．

ShadowXeck では，カーネル領域の中で間接操作命令を含む読み込み専用領域のみを多重化する．カーネル領域には，ランキューやプロセスリストのように，被制御 OS カーネルが提供する実行空間全体で共有すべきデータが含まれる．カーネル領域を多重化した後も，これらの共有データの整合性が保たれるようにするため，ShadowXeck ではカーネル領域の間接操作命令を含む領域のみを多重化する．これにより，ランキュー等のデータは被制御プログラムとそれ以外のプログラムのカーネル領域で共有させることができる．この部分的な多重化は，多重化されるメモリ領域を削減できるという利点も持つ．

4.4　セキュリティポリシ

4.4.1　文　　法

セキュリティポリシ文法を**図 3** に示す．ShadowXeck のセキュリティポリシでは，被制御プログラムに関連して発行された間接操作命令をどのように制御するかを指示する．

`defAction` に続く部分では，捕捉した間接操作命令がどの条件にも一致しなかった場合の対応処理（Action）を記述する．捕捉した間接操作命令に関する制御（IndirectSpec）に関

```
PolicyFile   →  defAction : Action+  IndirectSpec+
IndirectSpec →  currIP : virtAddr  CondAct+
CondAct      →  (destIP : (virtAddr | *), memLoc : (virtAddr | *), action : Action+)
Action       →  allow | fix(virtAddr) | raiseException | log
```

図 3　ポリシ文法
Fig. 3　Security policy syntax.

する記述では，間接操作命令位置（`currIP`）に基づいて条件と対応処理（CondAct）を指示する．条件部分では，間接操作命令の実行直後のインストラクションポインタ（`destIP`）と間接操作命令の実行時に参照されるメモリ位置（`memLoc`）を指示する．条件部分に`*`を記述した場合，その条件はつねに成立する．

対応処理の記述では，実行の継続（`allow`），実行ログの記録（`log`）を指示できる．また，間接操作命令の実行直後のインストラクションポインタを指定された仮想アドレス（*virtAddr*）に変更する指示（`fix`）も対応処理に含まれる．さらに，一般保護例外を発生させる指示（`raiseException`）も対応処理に含まれる．

4.4.2　セキュリティポリシの生成

ShadowXeck では，プロファイリングモードの実行ログをもとに，制御 VM の管理者または利用者が各被制御プログラムに対するセキュリティポリシを生成する．実行ログには以下の情報が含まれる．

- 間接操作命令発行時の仮想アドレス
- 間接操作命令実行後の遷移先の仮想アドレス
- 間接参照がレジスタ参照であるかメモリ参照であるかに関する情報

ShadowXeck のセキュリティポリシの記述者は OS カーネルに関する知識を必要とする．ShadowXeck では，記述者が生成されたログの内容を読みやすくするために，実行ログに含まれる仮想アドレスをカーネル内の関数名等のシンボルを用いて表示するコマンド `readlog` を提供している．このコマンドの引数にはログファイルと被制御 OS カーネルイメージファイルを与える．

さらに，記述者のセキュリティポリシの生成の手間を削減するために，我々はセキュリティポリシの雛形を生成するコマンド `mkpolicy` を提供する．`mkpolicy` には 4 つの引数（入力引数 3 つ，出力引数 1 つ）を与える．入力引数として，実行ログファイル，生成されるセキュリティポリシにおける `defAction` と CondAct の対応処理（Action）を与える．出力引数として，生成されるポリシの出力ファイル名を与える．`mkpolicy` により生成されるセ

キュリティポリシでは，レジスタ経由の間接操作命令に対し，memLoc の *virtAddr* が * として記述される．また，同一の currIP と destIP に対し，memLoc の *virtAddr* が異なる場合には，*virtAddr* を * と記述し，1 つの IndirectSpec としてまとめられる．雛形のポリシを生成後，記述者が特定の currIP に対し mkpolicy の引数として与えた Action と異なる対応処理を指示する場合には，生成された雛形ポリシを修正する．

ShadowXeck では，プロファイリングモードの実行ログで生成されなかったパターンが出現した場合には誤検知が生じうる．ただし，利用者は保守的なセキュリティポリシを生成することで，カーネルレベルマルウェアによる不正操作は防ぐことはできる．また，プロファイリングモードで十分稼働させたうえでセキュリティポリシを生成すれば，誤検知が生じることは少ないと我々は考えている．さらに，我々が想定している OS カーネルに関する知識の持つ記述者であれば，制御モードで生成される実行ログから適切にポリシを更新することは困難ではないとも考えている．

図 4 には，システムユーティリティプログラム netstat の間接操作命令を制御するためのセキュリティポリシ例の一部が示されている．このセキュリティポリシは，defAction と CondAct の対応処理に，各々allow log と allow を指定して雛形を生成した例である．カーネル内の関数 do_lookup（currIP: 0xffffffff8018f4b8）から呼び出される関数の対応処理は，currIP と memLoc から一意に遷移先（destIP）を決められるため，fix を利用して destIP を強制する．他方，カーネル内の関数 seq_read（currIP: 0xffffffff801a38f1）から呼び出される関数の対応処理は，currIP と memLoc から一意に destIP を決められないため，図 4 中の 4 つの destIP 以外の場合には一般保護例外を発生させるように raiseException を指示している．

```
defAction : allow log
    currIP: 0xffffffff8010bd6d
      (destIP: 0xffffffff8010bdfc, memLoc: *, action: allow)
      (destIP: 0xffffffff8010c420, memLoc: *, action: allow) ...
    ...
   currIP: 0xffffffff8018f4b8
      (destIP: 0xffffffff801b8280, memLoc: 0xffffffff80366f48,
       action: fix(0xffffffff801b8280))  # proc_root_lookup() ...
    ...
    currIP: 0xffffffff801a38f1
      # udp4_seq_show()
      (destIP: 0xffffffff802cf740, memLoc: *, action: allow)
      # tcp4_seq_show()
      (destIP: 0xffffffff802c82b0, memLoc: *, action: allow)
      # raw_seq_show()
      (destIP: 0xffffffff802ce380, memLoc: *, action: allow)
      # unix_seq_show()
      (destIP: 0xffffffff802e4920, memLoc: *, action: allow)
      (destIP: *, memLoc: *, action: raiseException)
    ...
```

図 4　netstat 用のセキュリティポリシ（抜粋）
Fig. 4　Security policy for netstat (extracted).

4.5 運　　用

4.5.1 制御用コマンドの提供

我々は制御 VM 内で以下のコマンドを提供する．制御操作は被制御 VM の外側で行われるため，たとえ攻撃者が被制御 VM を奪取できたとしても，攻撃者はこれらのコマンドを悪用することはできない．

- vconf：これは被制御 OS 情報を ShadowXeck に登録するためのコマンドである．ShadowXeck では OS カーネルのバイナリイメージごとに被制御 OS 情報を管理する．
- vcntl：これは vconf により登録済みの被制御 OS 情報と被制御 OS カーネルを関連付けるためのコマンドである．このコマンドは被制御 VM ごとに実行される必要がある．このコマンドが実行されると，OS カーネルの読み込み専用領域の保護が開始される．
- vstart：このコマンドは 2 つの操作モード（プロファイリングモード，制御モード）で被制御プログラムを操作するためのコマンドである．被制御プログラムの実行が開始されたときに間接操作命令とシステムコールが捕捉できるようになる．
- gen_indirect：これは被制御 OS カーネルのバイナリイメージから，間接操作命令に関する情報を収集するためのコマンドである．間接操作命令に関する情報には，間接操作命令が発行される仮想アドレスと間接操作命令のバイトコード列が含まれる．

4.5.2 被制御プログラムに関する制御の流れ

図 5 は，被制御 VM 内の被制御プログラム target_prog に関する被制御 OS カーネルの振舞い制御を開始するまでの流れを示している．まず，gen_indirect を用いて，被制御 OS カーネルのバイナリイメージ（targetOS.img）から間接操作命令に関する情報をファイル（output_indirect.txt）に出力する．次に，vconf を用いて被制御 OS カーネルのバイナリイメージごとの設定を行う．このコマンド引数には被制御 OS のバイナリイメージ（targetOS.img）と被制御 OS 情報を与える．被制御 OS 情報は，プロセス構造情報（targetOS_info.txt），システムコールに関する情報（syscall.txt）と間接操作命令に

```
[cvm] $ gen_indirect targetOS.img output_indirect.txt
[cvm] # vconf targetOS.img
          targetOS_info.txt syscall_info.txt output_indirect.txt
[cvm] # vcntl 1 targetOS.img
[cvm] # vstart profiling 1 target_prog output_prof.log
[cvm] $ mkpolicy output_prof.log allow && log allow policy.txt
[cvm] # vstart control 1 target_prog policy.txt
```

図 5　利用例
Fig. 5　Usage example.

関する情報（output_indirect.txt）からなる．さらに，vcntl を用いて被制御 VM ごとの設定を行う．このコマンド引数に被制御 VM の ID（VM ID:1）と targetOS.img を与え，被制御 VM と被制御 OS 情報を関連付ける．vcntl の実行時には，引数で与えられる OS カーネルのバイナリイメージから OS カーネルの読み込み専用に対応する仮想アドレス範囲情報も取得し，読み込み専用領域の保護が開始される．

上記の設定の終了後，2 つの操作モードで被制御プログラムに関連する被制御 OS カーネルの振舞い制御を開始する．まず，vstart を用いて，プロファイリングモード（profiling）で被制御プログラムが発行した間接操作命令に関する情報を収集する．このコマンドの引数に被制御 VM の ID，被制御プログラム名（target_prog），実行ログを記録する出力ファイル（output_prof.log）を与える．プロファイリングモード終了後，mkpolicy を用いて，output_prof.log から target_prog に対するセキュリティポリシの雛形をファイル（policy.txt）に出力し，policy.txt を適切に整形する．最後に，vstart を用いて，制御モード（control）の実行開始要求を発行する．これにより，VM ID 1 の被制御プログラムである target_prog が実行を開始したときに，セキュリティポリシ（policy.txt）に従って被制御 OS カーネルの振舞いが制御される．

4.6 特　　長

ShadowXeck は以下の特長を有する．

被制御 OS カーネルの振舞い制御の最適化　OS カーネル空間は，すべてのプロセスで共有しているため，単純にカーネルの振舞いの制御用コードを挿入すると，すべてのプロセスに対して適用されてしまう．ShadowXeck では，カーネル空間を多重化することで，被制御プログラムがカーネルレベルで稼働するときのみ制御用コードが実行される．

攻撃者による制御機構の無効化が困難　最も高い特権レベルで稼働する VMM が被制御 VM のメモリ操作を制御する．このため，攻撃者が被制御 VM 内からカーネル空間の読み込み専用領域や間接操作命令の捕捉を無効化するために，これらのデータを改竄することはできない．また，カーネル空間を多重化しているため，被制御プログラム以外のコンテキストで稼働している OS カーネルから，間接操作命令を捕捉するために必要なカーネル空間の書き換えを隠蔽できる．

被制御 OS 情報の自動生成　被制御 OS カーネルの読み込み専用領域や間接操作命令に関する情報は被制御 OS カーネルのバイナリイメージに依存する．これらの情報を生成する手間を軽減するために，我々は制御コマンドユーティリティを提供している．利用者は入力として被制御 OS カーネルのバイナリイメージを与えるだけで，制御コマンドユーティリティにより，これらの情報が自動生成される．

既存のプログラムとの互換性の維持　ShadowXeck は被制御 VM の外側から被制御 OS カーネルの振舞いを制御する．このため，被制御 OS カーネル[⋆1]や被制御プログラムのソースコードを修正する必要がない．

複数の VM 内で OS カーネルの振舞いを統一的に制御可能　同一の被制御 OS カーネルが複数の VM 内で稼働している場合には被制御 OS 情報を共有できる．さらに，同じ仮想環境を提供する複数の被制御 VM 内で被制御プログラムを稼働させる場合，プロファイリングモードの実行ログも共有でき，1 つの被制御 VM から取得できる実行ログより多くの情報を得ることができる．このため，適切かつ迅速なセキュリティポリシの生成やセキュリティポリシの共有が可能となる．

5. 実　　装

我々は，VMM として準仮想化を用いた Xen 3.0.3 を，被制御 OS として Linux 2.6.16 を用いて，CPU アーキテクチャ AMD64 上で ShadowXeck を実装した．

5.1 被制御 OS カーネル情報の取得

被制御 OS カーネルの振舞いを制御するために，被制御 OS カーネルに関する読み込み専用領域の範囲と間接操作命令に関する情報を取得する必要がある．これらの情報は被制御 OS カーネルに依存する．このため，利用者は被制御 OS カーネルのバイナリイメージを提供する必要がある．利用者から提供された被制御 OS カーネルのバイナリイメージから，我々は ShadowXeck に必要な情報を取得する．現在の ShadowXeck では，被制御 OS

⋆1 Linux 2.6.18 以前では，読み込み専用領域と書き込み可能領域を 4 KB のページ単位で分離するため，カーネルのメモリレイアウトを変更する必要がある．

カーネルの実行形式として ELF 形式を，CPU アーキテクチャとして x86 系をサポートしている．

読み込み専用領域の範囲に関する情報は，SX-core が読み込み専用領域の改竄を防止するために必要となる．読み込み専用領域の範囲に関する情報は，ShadowXeck では被制御 OS カーネルのバイナリイメージの ELF 形式のプログラムヘッダから，書き込み権限がないセグメントに関する仮想アドレスの範囲を取得する．

間接操作命令に関する情報は，SX-core が間接操作命令の捕捉の設定と捕捉時の間接操作命令のエミュレートのために必要となる．間接操作命令に関する情報を取得するために，ShadowXeck では被制御 OS カーネルのバイナリイメージから，間接呼び出し命令（call）と間接ジャンプ命令（jmp）の位置（仮想アドレス），命令のバイトコード列を取得する．

5.2　読み込み専用領域の改竄の防止

ShadowXeck では被制御 OS カーネルに関する読み込み専用領域の書き込みを禁止することで，被制御 OS カーネルの制御フローの改竄を防ぐ．読み込み専用領域への書き込みの禁止は SX-core が制御するため，被制御 VM からこの制御を無効化することはできない．読み込み専用領域への書き込みの禁止は，読み込み専用領域の仮想アドレス範囲への書き込み権限の付加と，読み込み専用領域の物理ページを書き込み可能にしたメモリマップの生成を禁止することで実現する．

5.3　カーネル空間の多重化処理

被制御プログラムがカーネルレベルで稼働しているときの間接操作命令の実行を制御するために，我々は被制御 OS カーネルのメモリ空間を部分的に多重化する．プロセスの最上位のページテーブルの仮想アドレスが設定される，CR3 レジスタ（CR3）の値に基づいて，ShadowXeck ではカーネル空間を多重化する．

多重化の初期化処理は被制御プログラムの実行開始時に行われる．被制御 VM 内で execve システムコールが実行されたとき，SX-core が被制御プログラム用の元のアドレス空間（ORIG_CR3）に対応する，新たなメモリ空間（NEW_CR3）を生成する．NEW_CR3 は，間接操作命令を含む物理ページのみ，部分的に多重化される．SX-core は，ORIG_CR3 と NEW_CR3 の組を多重化に関する情報として，被制御プログラムが終了するまで保持する．CR3 に NEW_CR3 が設定された場合，SX-core は被制御 OS カーネルによって発行された間接操作命令を捕捉できる．他方，ORIG_CR3 を含む他の CR3 から走査可能なカーネル空間の間接操作命令は捕捉されない．多重化の終了処理は，被制御プログラムの終了時に行われる．SX-core は exit または exit_group システムコールが実行された時点で，CR3 の値を多重化前の ORIG_CR3 の値に設定する．

NEW_CR3 への切換えは，SX-core が捕捉できる CR3 への書き込みのときに行われる．CR3 への書き込みが発生したときに，SX-core は，書き込まれる値が ORIG_CR3 であるか検査する．書き込まれる値が ORIG_CR3 である場合，SX-core は CR3 に NEW_CR3 を書き込む．

Linux の場合，コード領域や読み込み専用領域は被制御 VM の稼働中は常駐しているため，ユーザ空間の多重化[24] で必要であったスワップ操作に関する処理は行わない．また，Linux の場合には，x86 系の CPU アーキテクチャではカーネル空間の物理ページサイズが 4 MB または 2 MB であり，最小の物理ページサイズである 4 KB より大きい．しかし，VMM が被制御 OS のカーネル空間の物理ページサイズを 4 KB に変更して被制御 VM のメモリ空間を管理することにより，部分的な多重化の粒度を細かくすることができる．

5.4　被制御 OS カーネルの振舞い制御のための捕捉

被制御プログラムに関連するカーネルの振舞いを制御するために，SX-core が間接操作命令の実行位置とシステムコール手続きの開始・終了を捕捉する必要がある．

5.4.1　間接操作命令の捕捉

間接操作命令の捕捉は `vcntl` コマンドの実行時に初期化される．`vcntl` コマンドが発行されると，SX-core はまず被制御 VM を停止させる．その後，間接操作命令の仮想アドレス情報に基づき，停止時の被制御 VM のページテーブルを走査して多重化後の物理ページ内のデータを複製する．Linux はカーネル空間をすべてのプロセスで共有しているため，停止時の被制御 VM のページテーブルから間接操作を含む物理ページを取得できる．複製時には，SX-core が間接操作命令の実行を捕捉できるように，複製されたデータ中の間接操作命令の位置のバイナリコードを SX-core が捕捉できる命令（HLT 命令）で書き換える．最後に，複製されたデータを VMM 内の管理データとして登録し，被制御 VM を再開させる．SX-core が間接操作命令を捕捉したときには，登録されている間接操作命令に関する情報を用いて，書き換え元の命令をエミュレートする．

5.4.2　システムコール手続きの捕捉

ShadowXeck では，被制御プログラムの実行（execve），プロセスの生成（fork，clone 等），プロセスの終了（exit，exit_group）に関するシステムコールを捕捉する必要がある．被制御プログラムの実行の開始とプロセスの生成を捕捉するために，バイナリ書き換えの対象となるメモリ空間は被制御プログラム用のカーネル空間だけでなく，多重化元のカーネル空間も含まれる．以降では，被制御プログラム用のカーネル空間と多重化元のカーネル空間

を各々，target_kernel と orig_kernel と呼ぶ．

準仮想化を利用した Xen の CPU アーキテクチャAMD64 版（Xen-AMD64）では，VMM が SYSCALL 命令を捕捉するため，SX-core がシステムコール開始を制御できる．ShadowXeck ではシステムコール手続きの終了直後の被制御プログラムのレジスタやメモリ上の実行状態からシステムコールに関する実行状態を取得することが必要となる．SX-core がシステムコール手続きの終了時を捕捉するために，間接操作命令の捕捉の設定と同様に，我々は被制御 OS カーネルイメージのバイナリ書き換え[25] を導入している．

被制御プログラムの実行開始は，orig_kernel 内で execve の実行開始時にプログラム名を検査する．被制御プログラムであった場合には，orig_kernel のシステムコール手続きの終了位置にバイナリ書き換えを適用し，SX-core がシステムコール手続きの終了位置を捕捉できるようにする．この時点では，被制御 VM 内で被制御プログラム用のアドレス空間はまだ生成されていない．execve の終了の捕捉時に，SX-core がカーネル領域を多重化し，target_kernel を生成する．そして，SX-core は target_kernel に関する CR3 の値と orig_kernel に関する CR3 の値を組にして保存する．さらに，orig_kernel 内のシステムコール手続きの終了時の捕捉を無効化するために，システムコール手続きの終了位置の命令を元の命令に戻す．また，被制御 OS カーネルの制御開始以降で，被制御プログラムが execve を発行した場合には，実行開始時の処理と同様の処理を行い，target_kernel と orig_kernel，それらに関連する情報を更新する．

また，被制御プログラムが fork 等の実行によりプロセスを生成したときには，target_kernel と orig_kernel のシステムコール手続きの終了位置にバイナリ書き換えを適用し，SX-core がシステムコール手続きの終了位置を捕捉できるようにする．親プロセス（target_kernel）または子プロセス（orig_kernel）のどちらかの終了時を捕捉したときに，SX-core が子プロセス用に target_kernel を複製し，target_kernel の CR3 の値と子プロセスの orig_kernel の CR3 の値を組にして保存する．さらに，orig_kernel 内のシステムコール手続きの終了時の捕捉を無効化するために，システムコール手続きの終了位置の命令を元の命令に戻す．

exit 等によるプロセスの終了時には，SX-core が登録してある target_kernel と orig_kernel に関連した CR3 の組の値を削除する．さらに，target_kernel に関する最上位のページテーブルのみ解放する．

6. 評　　価

ShadowXeck を評価するため，被制御プログラムに関連する被制御 OS カーネルの振舞い制御の確認と間接操作命令の制御にともなうオーバヘッドを計測した．実験には CPU Dual-Core AMD Opteron 2.8 GHz が 2 つ，8 GB メモリ，1 Gbps NIC を有する物理計算機を用いた．制御 VM と被制御 VM は各々4 つの仮想 CPU を割り当て，各々1 GB メモリに設定した．

6.1　被制御 OS カーネルの振舞い制御の確認

6.1.1　読み込み専用領域の改竄の防止

Linux における，既存のカーネルレベルルートキットを用いたマルウェアの多くは，システムコールテーブルのエントリを改竄する．しかし，現在の Linux OS カーネルでは，これらのシステムコールテーブルを改竄するマルウェアは動作しない．たとえば，2.6.0 以降ではシステムコールテーブルのベースアドレスがエクスポートされなくなったため，カーネルレベルルートキットでエクスポートされたシステムコールテーブルのベースアドレスを利用しているマルウェアは有効でない．また，2.6.16 以降では，システムコールテーブル自体が読み込み専用領域に割り当てられているため，システムコールテーブルのエントリを仮想アドレス経由で直接書き換えることはできない．

しかし，現在の Linux OS カーネルの設計においても，システムコールテーブルのエントリを改竄することは可能である．このことは，システムコールテーブルを含む物理ページを異なる仮想アドレス空間に，書き込みを可能にしてメモリマップすることで実現できる．これを実証するために，我々は，任意の仮想アドレスに対応する物理ページを読み込み可能にしてメモリマップするカーネルモジュール（malLKM）を作成した．malLKM を用いることで，システムコールテーブルのエントリだけでなく，任意の読み込み専用領域を改竄することができる．たとえば，任意のカーネル空間のコード領域を改竄し，制御フローを変更することが可能である．

攻撃を模擬するため，ユーザレベルプログラムが発行する open システムコールを監視するために malLKM を利用した．我々は被制御 VM 内で malLKM をロードし，ユーザレベルプログラムが発行した open システムコールに関する情報を，malLKM が取得できることを確認した．次に，ShadowXeck の読み込み専用領域の改竄防止機能を有効にし，malLKM による open システムコールのリダイレクトを試みた．この場合には，malLKM のロード中に，システムコールテーブルを書き込み可能にするメモリマップを試みたところで SX-core が改竄を防止し，malLKM のロードに失敗した．これにより，ShadowXeck が読み込み専用領域を改竄する攻撃を防止できることを確認できた．

6.1.2　間接操作命令の制御

割込みディスクリプタテーブルや file_operations 構造体等の関数ポインタを含む領域は書き込み可能領域である．このため，6.1.1 項で述べた読み込み専用領域の改竄の防止では，攻撃者による書き込み可能領域の改竄を防止することはできない．ShadowXeck では，被制御プログラムごとに，発行される間接操作命令に関するセキュリティポリシに基づき，被制御 OS カーネルの振舞いを制御する．これにより，被制御プログラムのコンテキストでのカーネル内の制御フローは改竄の影響を受けない．

書き込み可能領域の改竄による攻撃を無効化できることを示すために，既存のカーネルレベルマルウェア adore-ng[1)] を用いた．adore-ng はメモリ上の/proc ファイルシステムやルートファイルシステムに関連する関数ポインタを不正に改竄する．被制御プログラムとして，3 つのシステムユーティリティプログラム ps，ls，netstat を用いた．

まず，被制御 VM に adore-ng を導入することで，攻撃者が指定するプロセス・ファイル・ポートを隠蔽できることを模擬した．隠蔽対象は各々，利用中のシェルプログラムのプロセス，ホームディレクトリの mal.db ファイル，nc プログラムで利用中のポート番号 2222 と設定した．この状態で，ps，ls，netstat を実行したが，各プログラムによって隠蔽対象の存在を検出することはできなかった．

次に，ShadowXeck による間接操作命令の制御を有効にして，同様の実験を行った．まず，プロファイリングモードで ps，ls，netstat に関する有効な間接操作命令の情報を収集した．次に，収集した情報から ps，ls，netstat に対する各々のセキュリティポリシを生成し，ps，ls，netstat に関する制御を開始した．ps，ls，netstat のセキュリティポリシに基づいて捕捉される間接操作命令位置（currIP）の数は，各々，54，57，46 であった．ps の場合，adore-ng により改竄される/proc ファイルシステムに関連する関数ポインタへ遷移させる間接操作命令位置（memLoc）で遷移先（destIP）が一意に決定できるため，セキュリティポリシでは対応処理（Action）として fix を指定した．ls の場合，ps の場合と同様，adore-ng により改竄されるルートファイルシステムに関連する関数ポインタへ遷移させる間接操作命令位置で遷移先が一意に決定できるため，対応処理として fix を指定した．netstat の場合，adore-ng により改竄される/proc ファイルシステムに関連する関数ポインタへ遷移させる間接操作命令位置で遷移先が一意に決定できないため，Action として raiseException を指定した．

その後，adore-ng を導入し，上述と同様の隠蔽対象に関する設定を行った．この状態で，ps，ls，netstat を実行し，ps と ls では adore-ng の隠蔽対象の存在を検出することができた．一方，netstat の実行時には一般保護例外が発生した．ShadowXeck を利用した場合，メモリ上の/proc ファイルシステムやルートファイルシステムに関連する関数ポインタは修正される．このため，セキュリティシステムの一部としてカーネルモジュールが利用された場合には，被制御プログラム以外のプログラムに対して，セキュリティシステムのカーネルモジュールが有効となる．しかし，ps と ls ではセキュリティポリシの fix により，間接操作命令の実行時の遷移先のアドレスが許可されたアドレスに強制された．一方，netstat ではセキュリティポリシの raiseException により，実行が失敗した．

ShadowXeck による読み込み専用領域の改竄の防止と間接操作命令の制御により，被制御プログラムに関連したカーネル内の振舞いを不正に変更することを困難にさせられることが確認できた．

6.2　間接操作命令の制御にともなうオーバヘッド

以下の既存のアプリケーションプログラムを用いて，実行時のオーバヘッドを計測した．

- システムユーティリティプログラム（ps，ls，netstat）：
  プロファイリングモードで各プログラムを実行し，各々に対するセキュリティポリシを生成した．制御モードにおける各プログラムの実行時間を計測した．
- ウェブサーバ（thttpd，Apache）：
  プロファイリングモードでは，各ウェブサーバに対し，起動・終了および静的コンテンツへの要求に関する処理を実行した．Apache では動的コンテンツ（CGI）への要求処理も実行した．これらの処理に基づき，各ウェブサーバに対するセキュリティポリシを生成した．制御モードでは，ApacheBench を用いてウェブサービススループットを計測した．ApacheBench は Pentium 4 3.2 GHz（HT 有効），2 GB メモリ，100 Mbps NIC を有する物理計算機上で実行し，被制御 VM の物理計算機とは同一 LAN に設置した．各ウェブサーバに対し，ApacheBench から要求数が 128 の場合に対する静的コンテンツ（ファイルサイズ（FS）：1 KB，100 KB）の要求を発行した．Apache に対しては，要求を送信した計算機情報を取得し，それを整形して返す CGI に対する実行時間も計測した．CGI の要求数は 128 とした．
- アンチウィルスツール（ClamAV）：
  clamscan と clamd/clamdscan を用いたウィルス検査の実行時間を計測した．clamscan はウィルス検査を実行するコマンドで，ウィルスデータベースをコマンド実行時に読み込む．一方，clamd はウィルスデータベースを起動時にあらかじめ読み込んでおき，clamdscan コマンドからのウィルス検査要求の処理を行うウィルス検査デーモンである．

表 1　実行時間（システムユーティリティプログラム）［ミリ秒］
Table 1　Execution time (system utility programs [ms]).

	ps	ls	netstat
ShadowXeck (target)	29	73	41
ShadowXeck (non-target)	17	62	35
Xen	14	61	30
Linux	14	32	24
QEMU	212	213	473

表 2　実行時間（ウェブサーバ）［ミリ秒/要求］
Table 2　Execution time (Web servers) [ms/req].

	thttpd		Apache		
	FS-1 KB	FS-100 KB	FS-1 KB	FS-100 KB	CGI
ShadowXeck (target)	0.73	9.39	0.74	9.36	11.57
ShadowXeck (non-target)	0.62	9.25	0.71	9.34	9.30
Xen	0.60	9.23	0.69	9.31	8.31
Linux	0.54	9.16	0.58	9.24	6.15
QEMU	5.20	95.38	9.24	105.62	348.57

プロファイリングモードでは，clamscan，clamd と clamdscan に対し，起動・終了およびウィルス検査処理を実行し，各プログラムに対するセキュリティポリシを生成した．制御モードでは，clamscan，clamd により，15 のファイル（ウィルスファイルは 5 つ）のウィルス検査に要する時間を計測した．

我々は以下の設定に対し，各アプリケーションを適用し，実行時間を計測した．

- ShadowXeck：アプリケーションを制御対象とする場合（target）と制御対象としない場合（non-target）に対し，各アプリケーションを実行．我々は，non-target に対しては，カーネル空間の多重化を利用することで，制御対象としないプログラムに関連した間接操作命令を捕捉しない．
- Xen：VMM 上で各アプリケーションを実行．
- Linux：物理計算機上で各アプリケーションを実行．物理メモリサイズを 1 GB に設定．
- QEMU[5]：マルウェア解析に利用される CPU エミュレータ上で各アプリケーションを実行．VM の設定は仮想 CPU を 4 つ，メモリサイズを 1 GB にした．

システムユーティリティプログラム，ウェブサーバ，アンチウィルスツールに対する実験結果は各々，**表 1**，**表 2**，**表 3** に示されている．ここで用いた被制御 OS カーネルの間接操作命令の数は 1,378（間接呼び出し：1,290，間接ジャンプ：88）であり，それらの間接操作命令を含むページ数（ページサイズ：4 KB）は 82 であった．また，コマンドプログラムに対する SX-core により捕捉された間接操作命令数は，ps では約 5,480 回，ls では約 1,150 回，netstat では約 1,480 回であった．

表 3　実行時間（アンチウィルスツール）［秒］
Table 3　Execution time (anti-virus tool) [s].

	clamscan	clamd/clamdscan
ShadowXeck (target)	8.70	0.31
ShadowXeck (non-target)	8.06	0.30
Xen	8.05	0.30
Linux	5.86	0.20
QEMU	111.12	2.43

表 1 より，コマンドラインプログラムに対する実行時間は，プログラムを制御対象とした場合（target），これらの実験を通し，Linux 上でアプリケーションを適用した場合に比べ，ShadowXeck を用いた場合にはある程度のオーバヘッドが生じるが，被制御 OS カーネルの振舞いを制御することができると我々は考えている．また，ShadowXeck では，制御対象としないプログラムに対する間接操作命令の捕捉による性能低下への影響を削減できることも実験を通じて確認できた．さらに，CPU エミュレータを用いた場合と比較して，非常に小さい実行時オーバヘッドで間接操作命令を制御できることも分かった．ShadowXeck は被制御 VM 内でのプログラムの実行開始時に SX-core が被制御プログラムであるか検査する．このため，CGI における制御対象としないプログラムの場合（non-taget）においても，CGI の実行開始時の検査にともないオーバヘッドが生じる．

7. 関 連 研 究

これまでカーネルレベルマルウェアを防止，検出，解析するために様々なシステムが提案されている．カーネル全体に対して機能が適用される既存のシステムとは異なり，ShadowXeck は，特定のプログラムに対してのみ間接操作命令の制御が適用される．

NICKLE[18] は認証されていないカーネルコードの実行を防止し，カーネルレベルマルウェアを無効化させるシステムである．NICKLE は QEMU 等の VMM とカーネル空間の多重化を利用し，命令参照とデータ参照で異なる物理ページを操作させ，命令参照の場合には認証されたコードのみを実行させる．ShadowXeck におけるカーネル空間の多重化は被制御プログラムであるかどうかに基づく．SecVisor[19] は，NX ビットを利用して，カーネル空間に対応する物理ページが実行可能かつ書き込み可能とならないようにし（X⊕W），

認証されたコード領域のみをカーネルモードで実行可能にさせる．Patagonix [13] は，NX ビットとカーネル空間で実行可能な認証済の実行コード領域に関するデータベースを用いて，VMM が被制御 OS カーネルの実行コードを制限する．これらのシステムとは異なり，ShadowXeck は読み込み専用領域の保護と被制御プログラムのコンテキストにおけるカーネルの振舞いを制御するシステムである．また，ShadowXeck では利用者はあらかじめ正当なカーネルコード領域をシステムへ登録する必要はない．ShadowXeck は被制御プログラムに基づきカーネルの振舞いを制御するため，被制御プログラム以外のプログラムにはカーネルモジュールが有効となり，カーネルモジュールの利便性は保たれる．

文献 10) では，ロード時にカーネルモジュールのバイナリを検査し，カーネルレベルルートキットを検出する手法を提案している．この手法を用いた場合，既存のセキュリティシステムに含まれるカーネルモジュールが，カーネルレベルマルウェアに含まれるカーネルモジュールと機能が類似していると，セキュリティシステムに含まれるカーネルモジュールが不正であると誤認されうる．システムの機能の一部であるカーネルモジュールだけで正当性を判断することは困難である．一方，ShadowXeck でのカーネルモジュールの利用は，カーネル内の読み込み専用領域の書き換えができない以外は制限されない．さらに，文献 10) によるカーネルモジュールの検査は，被制御 OS カーネルと同一実行空間で実行されるため，被制御 VM が奪取された場合，カーネルモジュールの検査が無効化されうる．同一実行空間で実行することによるマルウェア検出の無効化は chkroot [3] のような既存のシステムにもあてはまる．さらに，文献 10) による手法では，特殊デバイス（/dev/mem，/dev/kmem）を利用した攻撃は防げない．

Livewire [6] は被制御 OS 情報を用いて，被制御 VM の外側でセキュリティシステムを稼働させるシステムである．Livewire の応用例の 1 つとして，VMM 層での読み込み専用領域の保護があげられている．/proc ファイルシステムに関する関数ポインタの保護に関する適用例に関しては周期的な改竄検知があげられている．Copilot [14] は PCI カードを用いて，異なる物理計算機から別の物理計算機上におけるメモリ操作を周期的に監視し，カーネルレベルマルウェアによるカーネル空間の改竄を検出する．SBCFI [16] は，カーネル内の制御フローの改竄を検出するため，コード領域等に含まれる静的な制御フローと関数ポインタ等の動的な制御フローを含むメモリ領域を周期的に検査する．動的な制御フローの検査で，カーネルの各大域変数から到達可能な関数ポインタに関する情報を利用する際，SBCFI はカーネルのソースコードを必要とする．VMwatcher [8] は利用者の要求に応じて，被制御 VM の外側から再構築した VM 内のプロセスやファイルシステムの実行状態を用いて，マルウェアを検知するシステムである．これらの周期的な改竄検知の場合，周期が短ければ実行時オーバヘッドが増加し，周期が長ければ検知精度が低下しうる．他方，ShadowXeck では，関数ポインタを含む間接操作命令をリアルタイム検査することで周期的な検査の問題を改善している．

$\text{UCON}_{\mathcal{KI}}$ [22] は，被制御 OS カーネルの完全性を保持するために，被制御 VM の外側からアクセス制御を導入するシステムである．他方，ShadowXeck は読み込み専用領域の保護と被制御プログラムに関連した間接操作命令の制御によって，カーネルの振舞いを制御する．また，$\text{UCON}_{\mathcal{KI}}$ は利用者がカーネル内のアクセス制御に関するセキュリティポリシを一から作成する必要がある．他方，ShadowXeck はプロファイリングモードで収集した実行ログからセキュリティポリシの雛形を生成する仕組みを導入しているため，セキュリティポリシ記述の手間を軽減できる．また，$\text{UCON}_{\mathcal{KI}}$ のプロトタイプシステムは，CPU エミュレータを用いて OS カーネルのメモリへの書き込みを命令単位で制御するため，性能が著しく低下する．他方，ShadowXeck は VMM による捕捉を被制御プログラムに関連した間接操作命令の捕捉に制限することで性能低下を軽減している．

HookMap [20] は，アンチマルウェアシステムやシステムユーティリティプログラムをプロファイリングモードで実行し，被制御 OS カーネルによって実行される hook（間接操作命令）に関する情報を収集するシステムである．HookMap では，アンチマルウェアシステムのコンテキストで実行される間接操作命令は，カーネルレベルマルウェアによって改竄される可能性が高い hook であると見なす．HookFinder [23] や K-Tracer [11] は，カーネルレベルマルウェアの振舞いに基づき，カーネル内で改竄されうるデータを解析するシステムである．これらのシステムはカーネルレベルマルウェアを解析するシステムであるのに対し，ShadowXeck は被制御 OS カーネルの振舞いを制御するシステムである．

8. まとめと今後の課題

本論文では，VM の外側から内部で稼働する OS カーネルの振舞いを制御するセキュリティシステム ShadowXeck を提案した．OS カーネルの振舞い制御は，カーネル空間の読み込み専用領域の改竄防止と被制御プログラムに関連した間接操作命令の制御によって実現される．我々は VMM として準仮想化を利用した Xen を，被制御 OS として Linux を用いて，AMD64 アーキテクチャ上で ShadowXeck の設計および実装を行った．ShadowXeck の評価では，既存のカーネルレベルマルウェア等を利用し，読み込み専用領域の保護と被制御プログラムに関連した被制御 OS カーネルの振舞い制御が実現できていることを確認し

た．ウェブサーバとアンチウィルスツールの実行時オーバヘッドは，物理計算機上で実行した場合に比べ，各々，約 1%から約 88%，約 48%から約 55%であった．CPU エミュレータを利用した場合に比べ，ShadowXeck は非常に小さい性能低下で OS カーネルの振舞いを制御できた．

今後の課題としては，まず，被制御プログラムに対して正当なカーネルモジュールを有効にさせる仕組みを導入することがあげられる．カーネルモジュールはロード時にメモリ上の配置が決定するので，ロードごとに配置が異なる可能性がある．これに対処するため，相対アドレスで制御する仕組みを ShadowXeck に導入することを考えている．さらに，ロード時にカーネルモジュール中に含まれる間接操作命令を捕捉するためのバイナリ書き換えを行う仕組みも必要となる．別の今後の課題として，制御対象の間接操作命令間の順序関係等に関する記述もできるように，セキュリティポリシの記述力を高めることも検討している．

参 考 文 献

1) adore-ng: Announcing full functional adore-ng rootkit for 2.6 Kernel (2004). http://www.lwn.net/Articles/75991/
2) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles*, New York (2003).
3) chkroot: *chkroot — locally checks for signs of rootkit*. http://www.chkrootkit.org
4) Cowan, K., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. and Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proc. 7th USENIX Security Symposium*, San Antonio (1998).
5) Bellard, F.: QEMU CPU Emulator. http://fabrice.bellard.free.fr/qemu/
6) Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. 10th Annual Network and Distributed Systems Security Symposium*, San Diego (2003).
7) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.: A Secure Environment for Untrusted Helper Applications, *Proc. 6th USENIX Security Symposium*, San Jose (1996).
8) Jiang, X., Wang, X. and Xu, D.: Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction, *Proc. 14th ACM Conference on Computer and Communications Security*, Alexandria (2007).
9) Jones, S., Arpaci-Dusseau, A. and Arpaci-Dusseau, R.: VMM-based Hidden Process Detection and Identification using Lycosid, *Proc. 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle (2008).
10) Kruegel, C., Robertson, W. and Vigra, G.: Detecting Kernel-Level Rootkits Through Binary Analysis, *Proc. 20th Annual Computer Security Applications Conference*, Tucson, AZ (2004).
11) Lanzi, A., Sharif, M. and Lee, W.: K-Tracer: A System for Extracting Kernel Malware Behavior, *Proc. 16th Annual Network and Distributed System Security Symposium*, San Diego (2009).
12) Linux: Address space randomization in 2.6 (2005). http://lwn.net/Articles/121845/
13) Litty, L., Largar-Cavilla, H. and Lie, D.: Hypervisor Support for Identifying Covertly Executing Binaries, *Proc. 17th USENIX Security Symposium*, San Jose, CA (2008).
14) Petroni, N.L. Jr., Fraser, T., Molina, J. and Arbaugh, W.: Copilot — a Coprocessor-based Kernel Runtime Integrity Monitor, *Proc. 13th USENIX Security Symposium*, San Diego (2004).
15) Petroni, N.L. Jr., Fraser, T., Walters, A. and Arbaugh, W.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data, *Proc. 15th USENIX Security Symposium*, Vancouver, B.C. (2006).
16) Petroni, N.L. Jr. and Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks, *Proc. 14th ACM Conference on Computer and Communications Security*, Alexandria, VA (2007).
17) Provos, N.: Improving Host Security with System Call Policies, *Proc. 12th USENIX Security Symposium*, Washington, DC (2003).
18) 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*, Cambridge (2008).
19) Seshadri, A., Luk, M., Qu, N. and Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes, *Proc. 21st ACM Symposium on Operating Systems Principles*, Stevenson (2007).
20) Wang, Z., Jiang, X., Cui, W. and Wang, X.: Countering Persistent Kernel Rootkits Through Systematic Hook Discovery, *Proc. 11th International Symposium on Recent Advances in Intrusion Detection*, Cambridge (2008).
21) Wei, J., Payne, B., Giffin, J. and Pu, C.: Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense, *Proc. 2008 Annual Computer Security Applications Conference*, Boston (2008).
22) Xu, M., Jiang, X., Sandhu, R. and Zhang, X.: Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection, *Proc. 12th ACM Symposium on Access Control Models and Technologies*, Sophia Antipolis (2007).

23) Yin, H., Liang, Z. and Song, D.: HookFinder: Identifying and Understanding Malware Hooking Behaviors, *Proc. 15th Annual Network and Distributed System Security Symposium*, San Diego (2008).
24) 尾上浩一，大山恵弘，米澤明憲：アプリケーションデータを保護するための VMM に基づくアーキテクチャ，*IPSJ Transactions on Advanced Computing Systems*, Vol.2, No.3, pp.173–188 (2009).
25) 尾上浩一，大山恵弘，米澤明憲：システムコール制御に基づく仮想マシン間サンドボックスシステム，*IPSJ Transactions on Advanced Computing Systems*, Vol.2, No.1, pp.33–52 (2009).

(平成 21 年 10 月 2 日受付)
(平成 22 年 2 月 6 日採録)

尾上　浩一

1979 年生．2010 年東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程修了．博士（情報理工学）．現在，富士通研究所勤務．興味はオペレーティングシステムや仮想マシンモニタ等のシステムソフトウェア，セキュリティ．

大山　恵弘（正会員）

1973 年生．2001 年東京大学大学院理学系研究科情報科学専攻修了．博士（理学）．科学技術振興事業団研究員，東京大学大学院情報理工学系研究科助手を経て，現在，電気通信大学情報理工学部総合情報学科准教授．興味はシステムソフトウェア，セキュリティ，プログラミング言語，並列分散処理．

米澤　明憲（正会員）

1947 年生．1970 年東京大学卒業．1977 年 MIT 計算機科学科博士課程修了．Ph.D. 東京大学情報理工学系研究科教授．ACM フェロー，日本ソフトウェア科学会理事長を歴任．現在，産総研情報セキュリティ研究センター副センター長を兼務．第 12 期日本学術会議会員．マイクロソフト Trust-worthy Computing Academic Advisory Board メンバ．2008 年国際オブジェクト技術協会（AITO）Dahl-Nygaard 賞受賞．2009 年紫綬褒章受章．