Vol. 47 No. 4 情報処理学会論文誌 Apr. 2006

仮想マシンモニタにおけるデバイスドライバ安全性向上に関する提案

藤澤 一樹[†1] 橋本 正樹[†1] 宮本 久仁男[†1]
金 美羅[†1] 辻 秀典[†1,†2] 田中 英彦[†1]

近年，注目を集める仮想化技術はコスト削減の観点から普及が始まった．今後，仮想化支援機能を搭載した CPU より一層普及していくことが考えられる．特に仮想マシンモニタ（VMM：Virtual Machine Monitor または Hypervisor）というソフトウェアがパフォーマンスとセキュリティの点で注目を集める．VMM によって得られるセキュリティの効果は，VMM 上で稼動する Guest OS が使用するリソースを隔離できるということと，Guest OS の実行環境を防御可能になることが挙げられる．しかしながら，これらは VMM が安全であるという前提の基に成り立っており，VMM に脆弱性が存在すると，その上で動作する Guest OS が安全であってもシステム全体を脆弱性にしてしまう．そこで本稿では，Device Driver に起因する VMM の脆弱性を指摘し，これによって引き起こされる問題を事前に防ぐ手法を提案する．

Security enhancement of device drivers for virtual machine monitor

Kazuki Fujisawa,[†1] Masaki Hashimotoo,[†1] Kunio Miyamoto,[†1]
Mira Kim,[†1] Hidenori Tuji[†1,†2] and Hidehiko Tanaka[†1]

Now, Virtualization Technology which attracted attention came to be used from the viewpoint of cost cut. It is thought that it is used still more in future by a set of processor enhancements that improve traditional software-based virtualization solutions. In particular, software called Virtual Machine Monitor (VMM or Hypervisor) attracts attention at a point of performance and the security. VMM provide security advantage. VMM can isolate the resource that the Guest OS to operate on VMM uses and, VMM can defend the executing environment of the Guest OS in VMM. However, these make ends meet by a premise that VMM is secure. Even if the Guest OS on VMM is secure, if VMM has vulnerability to make system vulnerability. This paper points out that vulnerability of VMM is device driver. And technique is proposed to prevent a problem to happen for it beforehand.

1. はじめに

近年，注目を集める仮想化技術は，複数台のサーバを 1 台に集約すことによってサーバの運用効率を向上させ TCO（Total Cost of Ownership）の削減ができるという事から普及が始まった．今後，仮想化支援技術を搭載した CPU によって一層普及すると考えられる．仮想化技術の中でも特に仮想マシンモニタ（VMM：Virtual Machine Monitor または Hypervisor）と呼ばれるソフトウェアがパフォーマンスとセキュリティの点から注目を集めており，その代表的な例として VMware ESX Server や OSS（Open Source Software）の Xen[1] や Linux カーネルにマージされた KVM（Kernel Based Virtual Machine）[2] などが挙げられる．特に Xen は，全てのサーバ，デスクトップ，ラップトップ，携帯電話に VMM が使われる Ubiquitous Virtualization を目指しており，各社の Enterprise Server の仮想化ソフトのベースとして使われている．VMM は，その上で稼動する Guest OS が使用するリソースを隔離できるとともに，Guest OS の実行環境を防御することができるため，これらの機能をセキュリティ分野に適用した研究例が多く存在する．例えば，隔離機能を強化し VMM に Guest OS 間のポリシーベースの強制アクセス制御を持たせたもの[3] や，Guest OS のデバック機能を修正し防御対象システムのスナップショットを即座に取得できるシステム[4] などがある．しかしながら，これらの研究は，VMM が安全であるという前提に基づいているために，VMM に脆弱性が存在するとその上に載せた Guest OS が安全であってもシステム全体は脆弱に

†1 情報セキュリティ大学院大学
INSTITUTE of INFORMATION SECURITY
†2 株式会社情報技研
Institute of Information Technology, Inc.

なってしまう．

そこで本稿では，VMM 自体のセキュリティに焦点を合わせ，その安全性を向上させる手法を提案する．まず，VMM の脆弱性として考えられるのは，Device Driver の脆弱性を突いて VMM の特権を奪取することである．実装によるが，VMM は OS と同様に特権モードで Device Driver を動作させるため，Device Driver に脆弱性があった場合，VMM における特権命令を実行することができる．これによって Guest OS が停止されたり，新たな Guest OS が構築され踏み台にされたりするなどの問題が起きる．

現在の計算機における Device の機能が複雑になり，これらの機能を制御する Device Driver のコードが大きくなりつつある．そのために脆弱性が存在することが知られている．これら Device Driver が Trusted Virtual Machine Monitor（TVMM）として研究されている Terra[5] の中でも問題となることが指摘されており，Device Driver を Trusted Computing Base の一部として含めることができないということを述べている．

我々は，Terra の User Mode Driver と Next-Generation Secure Computing Base[6] を用いる方法と異なり，ハードウェアに依存せず Device Driver に起因する問題を防ぐために，Device Driver に対するリファレンスモニタを提案[7] し，Xen 上での提案アーキテクチャを示す．

本稿では，以下のように構成される第 2 節で，Device Driver と脆弱性について述べ，第 3，4 節で関連研究と本提案に至る議論について述べる．そして第 5，6 節にて提案アーキテクチャの概要と実装について説明し，最後に，まとめと今後の課題について述べる．

2. Device Driver と脆弱性

Device Driver が VMM の脆弱性になることを第 1 節にて述べた．以下に Device Driver の脆弱性について説明する．

- 入力値の取り扱い：Device Driver は，Systemcall を通じてユーザからデータを受け取る．このデータのサニタイズやサイズのチェックを行わないことによって整数オーバフローが発生し，権限の昇格やクラッシュ，DoS が起きる．
- 権限の取り扱い：Device Driver に記述しなければならない権限項目を忘れたことや，権限の設定ミスによって起きる問題ある．この脆弱性によって，権限の昇格や任意のコードの実行が可能となる．
- メモリの取り扱い：Device Driver はカーネル空間で動作する．そのため，確保したメモリを開放しないことによって，カーネルのメモリ空間を圧迫し，メモリリークを起こす．また，自分以外の別のアドレスに書き込んでしまうことによって，権限の昇格，クラッシュや DoS が起きる．さらに，初期化せずにメモリを扱ったため，カーネルが使用したデータを Device に送ってしまいデータリークが起きる．
- バッファの取り扱い：Device Driver に記述されたバッファのサイズを越えてデータを渡すことでバッファオーバフローが起きる．これによって権限の昇格，任意のコードの実行，クラッシュ，DoS が可能となる．NIC（Network Interface Card）のバッファに対してバッファオーバフロー攻撃を行うことができれば外部から特権を取得することが可能となる．

ここで挙げた Device Driver は我々が調査／分析を行った結果の中から，上位 4 項目にあたる脆弱性への対応を目標とし，これらの脆弱性に対して，Device Driver に対するリファレンスモニタを導入することでシステムを保護する．

3. 関 連 研 究

Device Driver の脆弱性によって起きる問題を回避しシステムの安全性を向上させるために様々な実装や研究が行われており，これらは大きく 4 つに分類することができる．それぞれの方法については以下に詳しく説明する．

3.1 バグの極小化

Device Driver そのものに存在する脆弱性を出来る限り排除することでシステムへの影響を小さくする手法である．主に SCAT（Static Code Analysis Tool）を用いてバグを発見する方法と DSL（Domain-Specific Language）を用いて正しく動作をすることを確かめる方法，ソフトウェアテストがある．Engler 等の研究[8] のように SCAT はソースコードを静的解析して脆弱性を発見する方法である．Microsoft は，Windows の Device Driver に最適化した Static Driver Verifier（SDV）[9] を提供している．

DSL は，C 言語等の汎用言語と異なり，ある用途に設計された言語で，ソースコードの検証が可能である．Device Driver の開発用として NDL[10] などがある．

ソフトウェアテストでは，静的に発見できないバグをプログラムを動作させることで動的に発見できる．ブラックボックステストやホワイトボックステストがある．また，Microsoft の Device Driver 署名[11] のよ

うな外部監査的なアプローチは有用である．

3.2 特権モード動作の局所化

通常，特権モードで動作する Device Driver を特権モードと非特権モードに分ける方法である．この方法は，パフォーマンス低下を最小限にして，Device Driver によるシステムに対する影響を小さくする．この手法は Microsoft の Windows Vista で Kernel Mode Driver と User Mode Driver[12] として実装され，3D ゲーム等の高パフォーマンスの要求に対応できる．

3.3 特権モードからの排除

一般的に micro-kernel における Device Driver は，ユーザ空間で動作しており，特権モードから Device Driver を排除している．また，VMM 自体に Device Driver を持たせないことで Device Driver の脆弱性による影響を排除する手法がある．この手法はケンブリッジ大学で開発された VMM である Xen[13] に実装されている．Fig.1 に示すように Xen 自体は Device Driver を持たなく，管理ドメイン（dom 0）の Device Driver を利用する．dom 0 は Xen を管理する特権ドメインであり，非特権ドメイン（dom U）とは区別される．dom U から Device へのアクセス要求があった場合は，必ず Frontend Driver（FED）が Xenbus を通って Backend Driver（BED）にアクセスし，dom 0 の Device Driver が Device へアクセスするようにしている．同様に，IBM Research によって開発された Secure Hypervisor（sHype）[3] や Terra においても VMM 自体が Device Driver を持つことは好ましくないことが述べられている．また，micro-kernel である L4 を VMM とし，QEMU をサービスとすることで，Guest OS を動作させる L4VM がある．この L4VM を用いて通常の OS の Device Driver を Reuse する研究[14] がある．

3.4 隔　　離

micro-kernel であれば，サービスとして Device Driver を動作させ，他のサービスから隔離することが可能である．そのため Device Driver の故障によって OS がクラッシュすることはない．しかし，monolithic-kernel であると Device Driver の故障が OS をクラッシュさせる．そこで，OS の拡張機能の保護を目的とした Nooks[15] を用いて，Device Driver を隔離し，OS をクラッシュから保護し，故障した Device Driver のリカバリを可能[16] とした．

4. 不完全な保護方法

第 3 節にて，Device Driver によって起きる問題を回避する手法について実装方法や研究について述べた．現存する保護手法の問題と，提案手法にいたる議論について本節では述べる．

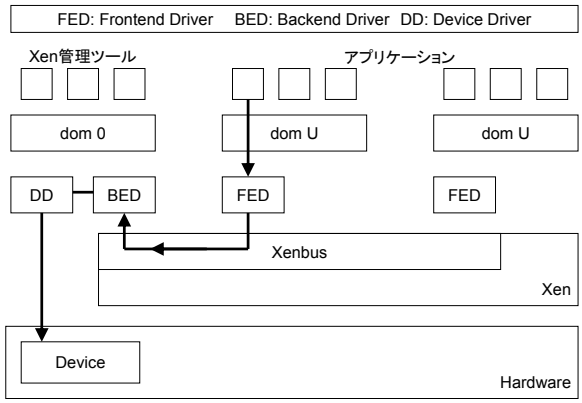

図 1 Xen アーキテクチャ
Fig. 1 Xen Architecture

一般にセキュリティ対策は，ソフトウェア開発工程である設計，実装，テスト，運用というサイクル全てで行うことが重要である．しかしながら，全ての段階でセキュリティ対策を行うことは難しく，行った場合も全ての脆弱性を無くすことは困難である．そのため運用時にセキュリティ対策をすることが多い．第 3 節で述べた既存の研究も，実装・テスト・運用に分けることができる．

実装工程において，SCAT は有効である．しかし，これらのツールは，あらかじめ決められたルールに違反しているかどうかを検査しているためルールに存在しないものは検出できない．SDV は，WDM（Windows Driver Model）と，その後継にあたる WDF（Windows Driver Foundation）というフレームワークの違反を検出するため，それらフレームワークに脆弱性が存在すると修正を困難にする．一般的な SCAT は，False Positive と False Negative の問題があり，最終的に人の目が必要である．DSL は，Device Driver のセキュリティに対する究極のアプローチであるが設計／運用／保守のコストが高くなるという欠点があるため普及していない．さらに，汎用言語から DSL を呼ぶためのインタフェースに脆弱性があると，DSL で書かれたコードが安全であってもシステムは脆弱なままである．また，ソースコードの検証には人の目が必要である．

テスト工程における動的解析は，静的解析で見つけられないバグを発見できることができる．しかし全てのテスト項目を網羅することは難しく，取りこぼしが起きる．そのため，Device Driver 署名を得るための

検査や通常のソフトウェアテストも脆弱性を発見できないことがある．

実装／テスト工程で発見できない脆弱性からシステムを保護するのが運用時のセキュリティ対策である．特権モードの局所化は，特権モードで動作する Device Driver が残っており，Device Driver を保護する機能がない．次に特権モードで動作する Device Driver の排除では，パフォーマンスの観点から純粋な micro-kernel を採用するコモディティな OS（Linux や Windows 等）は無く，OS がクラッシュすることはないが，Device Driver に対する攻撃からシステムを保護する機能を持たない．Xen のような VMM に Device Driver を持たせない方法は，VMM を安全にするが，dom 0 の Device Driver を用いているため，攻撃者が Device Driver の脆弱性を突き，dom 0 のコントロールができるようになると，Xen のコントロールが可能となるという問題がある．L4VM の Reuse する Device Driver が安全であるとは証明できず，また，Device Domain は任意の VM または，VMM であり，Xen 同様にコントロール VM の問題を残す．隔離ができる Nooks は，Device Driver の故障からシステムを保護するために開発されたものであり攻撃からシステムを保護することができない．また，Device Driver は特権で動作しているままであり隔離も弱い．

特にコモディティな OS であることは重要であり，セキュア VM[17] では，その点を考慮して OS でなく VMM によって OS のセキュリティの担保しようとしている．VMM を用いることで OS に比べ，多くの特権レベルを使えるためセキュリティレベルを高められる．

このように，設計，ソフトウェアテストで脆弱性を完全に排除するには限界があり，既存の運用時に保護する仕組みは，結果的に特権モードで動作するか，一般的なアプローチでないため，根本的な解決は難しい．以上の理由から現在まで存在している 4 つの手法では VMM を保護できないと考えた．そこで本稿では，Nooks が Device Driver の故障から動的にシステム保護したように，Device Driver の脆弱性から動的にシステムを保護するアプローチをとる．このアプローチは，Secure OS の考え方と同様で脆弱性を持っていることを前提にしている．Secure OS の場合は，最小特権の原則（PLP: Principle of least privilege）によって，root 権限を分割し，強制アクセス制御（MAC: Mandatory access control）によって被害を拡大することを防ぐ．Device Driver に対してリファレンスモニタを導入し，I/O アクセスに対して MAC を執行することで，攻撃をアクセス拒否し，Device Driver の脆弱性からシステムを保護する．さらに Xen 上で実装することで Device Driver を特権モードで動作させる必要がないため，さらに VMM を安全にすることができる．

5. 提案アーキテクチャ

第 1，2 節で VMM が安全であるという前提に基づいて VMM における研究がされていること，Device Driver の問題が VMM の脆弱性となりえることを述べた．さらに第 3 節にて，行われている取り組み，第 4 節にて提案方式にいたった理由を述べた．本節では提案アーキテクチャについて述べる．

5.1 提案アーキテクチャの概要

我々は，Device Driver に対してリファレンスモニタを導入し，I/O アクセスに対して MAC を執行することで Device Driver の脆弱性からシステムを保護することを述べた．しかし，リファレンスモニタを採用した Secure OS や VMM の中には I/O 制限が可能なものがある．NSA が中心になって開発した Security-Enhanced Linux（SELinux）[18] や sHype などである．SELinux は，LSM(Linux Security Module) を採用し，PLP と MAC を Linux に提供することで Linux を Secure OS とする．しかしながら，SELinux は，アプリケーション空間でのみ有効である．そのため，Device Driver を LKM（Loadable Kernel Module）として実装する Linux の場合，Device Driver の脆弱性そのものがカーネルの脆弱性となるため SELinux では防ぐことはできない．sHype は，リファレンスモニタを VMM に内蔵し，VM 間の MAC を実現するが，攻撃からシステムの保護が目的で無いため，Device Driver の脆弱性に対する攻撃からシステムを保護できない．

そこで，本稿では，Device Driver に対してリファレンスモニタを提案する．提案アーキテクチャを Fig.2 に示す．提案アーキテクチャは Flask アーキテクチャ[19] のようにセキュリティポリシーによる MAC の執行を行うサブシステムとセキュリティポリシーに従ってアクセス可否を決定するサブシステムに分けて，より安全で柔軟なポリシーを扱えるリファレンスモニタを構築する．

提案アーキテクチャは，Device Driver に対してサンドイッチ型に Last Driver（以降:LD）と First Driver（以降:FD）というセキュリティポリシーによる MAC の執行を行うサブシステムと Device Driver Auditor（以降:DDA）というキュリティポリシーに従ってアク

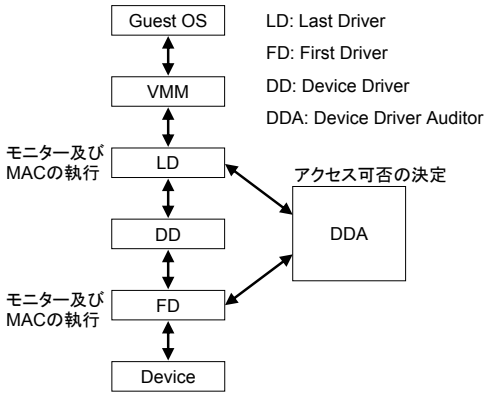

図 2 提案アーキテクチャ
Fig. 2 Proposed Architecture

セス可否を決定するサブシステムからなる．

提案アーキテクチャの特徴は，Device Driver をサンドイッチのように挟み込むように LD と FD が存在することである．サンドイッチ型にすることにより NIC がパケットを受信して，ハードウェア割り込みが発生するような，ネットワークドライバの非同期アクセスに対しても MAC の執行が可能となる．

Device Driver と関連する問題として不正な Device を接続し，DMA を用いて不正なメモリアクセスを行う攻撃があるが，本研究では，その攻撃をハードウェアで保護する Intel の VT-d[20] や AMD の AMD-V[21] が存在することやソフトウェアで保護する機能が Xen や L4VM で実現されていることから DMA を用いた攻撃からシステム保護することは対象としない．

5.2 動　　作

提案アーキテクチャはリファレンスモニタである以上，リファレンスモニタの必要条件を満たす必要がある．提案方式における DDA の役割は，アクセス可否の決定だけでなく，Device Driver，FD，LD の完全性を保証する機能を持つため DDA 自体の完全性を保証しなければならない．そのため，VMM の起動時に，何らかのプログラムによって DDA のメモリ空間を，書き換えられることを想定し，起動後に，再起動を行う．以下に起動手順を示す．

提案アーキテクチャの起動:

Step1: 電源 ON，BIOS，ブートローダ，VMM が起動する

Step2: DDA が VMM によって呼び出される

Step3: DDA 自らを再起動する

Step4: DDA は再起動時に自分自身が正しいことを確認する

Step5: DDA は FD，LD が正しいことを確認する．

Step6: DDA は Device Driver のハッシュ値を求め，格納する

Step7: DDA はポリシーを読み込み，提案アーキテクチャの動作準備が完了する

提案アーキテクチャは I/O アクセスに対しての MAC を執行するが場合によって FD と LD の動作が異なる．本稿では VMM から Device に通知する場合を Inside，Device から VMM に通知される場合を Outside と定義する．FD と LD は対の関係になっており，Inside からのアクセスと Outside からのアクセスで逆の動作を行う．それぞれの動作について Fig.3 と Fig.4 に示した．以下に Inside と Outside についての動作を示す．

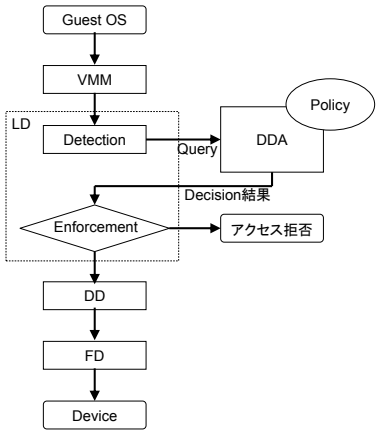

図 3 提案アーキテクチャの動作（Inside）
Fig. 3 Running of Proposed Architecture(Inside)

Inside の場合:

Step1: あるプログラムからハードウェアへのアクセス要求が発生し，Systemcall が発生する

Step2: LD は Systemcall をトラップし，メッセージを生成して Query を DDA に送信する

Step3: DDA は LD から送信された Query を受信しセキュリティポリシーと比較，アクセス可否の決定を下す（Decision）

Step4: DDA は Decision 結果を LD に送信する

Step5: LD は DDA から送信された Decision に基づいて MAC の執行を行う

Outside の場合:

Step1: 割り込み線に信号が入りハードウェア割り込みが生じ，割り込みハンドラが実行される．

Step2: FD は割り込みハンドラをトラップし，メッ

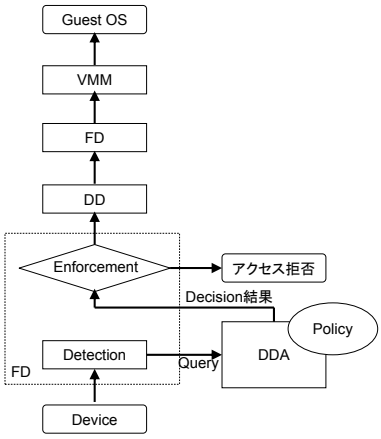

図 4 提案アーキテクチャの動作（outside）
Fig. 4 Running of Proposed Architecture(Outside)

セージを作成してする．DDA に Query を送信する．

Step3: DDA は FD から送信された Query を受信しセキュリティポリシーと比較，アクセス可否の決定を下す（Decision）

Step4: DDA は Decision を FD に送信する

Step5: FD は DDA から送信された Decision を基づいてセキュリティポリシーの執行を行う

5.3 Fast Driver と Last Driver

FD と LD の役割を確認し脆弱性によって，どのように検出するかを述べる．提案アーキテクチャは，LD,FD で検出した後に，DDA がアクセス可否の決定を行い，LD または FD が MAC の執行を行う．これらの執行は Inside，Outside によって異なるが脆弱性によっても異なる．脆弱性を case-1〜case-4 として検出元，検出方法，執行先と含めて Table1 に示す．

Device Driver の脆弱性は，何らかのアクションがなければ問題が起きないもの，Device が動作していれば起きるものの 2 通りである．アクションによって起きるものとして，システムを使うユーザが原因で起きる内部要因（case-1）とシステム外部からの外部要因と内部要因から起きる（case-2〜case-4）があり，Device を動作させているだけで起きる問題が（case-2）ある．例えば，バッファオーバフロー対策として，OSS であることを前提として，ソースコードからバッファサイズをシグネチャとして，登録しておき，バッファサイズを監視することが考えられる．しかし，これらの検出方法は，検出が可能であるという前提に基づいており，今後，検証する必要がある．

また，割り込みハンドラの実行による Device Driver へのアクセスに対応するために FD と LD はサンドイッチ型になっている．

次に，DDA の役割を確認し，4 つの case に対して DDA がどのように Decision を行うのかを述べる．

表 1 ポリシーと FD，LD の動作
Table 1 Policy and Running of FD, LD

	検出元	検出方法	執行先
case-1	LD	Systemcall の監視	LD
case-2	FD ／ LD	入力値の監視	FD ／ LD
case-3	FD ／ LD	メモリの監視	FD ／ LD
case-4	FD ／ LD	バッファの監視	FD ／ LD

case-1 : 権限の取り扱い（Inside）
case-2 : 入力値の取り扱い（Inside と Outside）
case-3 : メモリの取り扱い（Inside と Outside）
case-4 : バッファの取り扱い（Inside と Outside）

5.4 Device Driver Auditor

DDA は，起動時から一貫して完全性を維持しなければならない．また，ポリシーによる Decision について行う重要なサブシステムである．そのため，自らを再起動し，自分自身の完全性をチェックする．その後，Device Driver，FD，LD の完全性をチェックする．これら完全性のチェックについては SHA-1 または，MD5 を用いる．Device Driver の完全性チェックを定期的に行うことで FD，LD をバイパスされてしまうことや，Device Driver を改変してしまうような rootkit を検出が可能になると考えられる．

また，DDA はリファレンスモニタであるため柔軟なポリシー設定が求められる．それは，現在のポリシーを破棄し，新しいポリシーを読み込み，執行を行うことである．

そして，DDA はアクセス可否の決定を行うサブシステムであり，Table1 と同様に脆弱性を case-1〜case-4 として DDA における Decision について Table2 に示す．DoS 攻撃以外の場合は，Device の停止をせずに動作させるようにしている．これによって，Device Driver が脆弱性を持っていても安全に VMM を動作させることが可能となる．また，Decision には優先順位が存在する．最も優先度が高いのが権限の取り扱いで，次がバッファの取り扱い，入力値，メモリの取り扱いとなる．次節で実装について述べる．

6. 提案アーキテクチャの実装と評価

本節では，提案アーキテクチャの実装と評価方法について述べる．本研究は，リファレンスモニタである故に満たさなければならない要件が存在する．

表 2 脆弱性と Decision
Table 2 Vulnerability and Decision

	Decision
case-1	管理者の設定に従ってアクセス制御
case-2	入力値のサニタイズ
case-3	定期的にメモリの初期化と開放
case-4	アクセス拒否
	Device の停止（対 DoS）

case-1：権限の取り扱い（Inside）
case-2：入力値の取り扱い（Inside と Outside）
case-3：メモリの取り扱い（Inside と Outside）
case-4：バッファの取り扱い（Inside と Outside）

6.1 リファレンスモニタの要件

実装は，以下に示すリファレンスモニタの用件を満たす必要がある．

- リファレンスモニタは全てのアクセス要求に対して必ず呼び出されること
- リファレンスモニタのメカニズムは破壊や改ざんを受けないこと
- リファレンスモニタが正しく動作することが保証されていること

これらのリファレンスモニタの要件はリファレンスモニタの概念が初めて登場したアンダーソンレポート[22] にて，リファレンスモニタの要件として書かれたものである．リファレンスモニタの要件のうち，最初の要件は Device Driver をサンドイッチ型にしたことで必ずリファレンスモニタがアクセス要求に対して呼び出される．リファレンスモニタが改ざんされていないことは，DDA の完全性を保証することであり Xen に実装することで，DDA を Ring 0 で動作させ CPU のプロテクション機能で守ることができる．最後の項目であるリファレンスモニタが正しく動作することを保証するためにはコードの量が，出来る限り小さいことであり実装による．

6.2 提案アーキテクチャの実装

第 5 節にて提案アーキテクチャの概要について述べたが本節では実装にあたり，設計について述べる．提案のリファレンスモニタはサンドイッチ型で Device Driver をリファレンスモニタで挟むことで全ての I/O アクセスに対して対応できる．現状ではプロトタイプとして 1 つの Device に対してのリファレンスモニタとして実装を行う．リモート攻撃で VMM の特権を奪取できる可能性があるのが，NIC の Device Driver に対する攻撃あることを第 2 節で述べた．そこで提案アーキテクチャを FD，LD 単体として NIC に対して実装を行う．また，VMM としては Xen3 系を用いて dom 0，dom U には 2.6 系の Linux Kernel を用いる．Xen は dom 0 の Device Driver を用いるため Linux の Device Driver に対して提案アーキテクチャを実装することになる．Xen を用いる利点は OSS であること，また，Xen のアーキテクチャによって DDA を保護できることが挙げられる．DDA を特権モードで動作する Xen 自体（Ring 0）に持たせることで，CPU のプロテクション機能が利用できる．それによって DDA 保証することが可能となる．Xen に対する提案アーキテクチャの実装を Fig.5 に示す．

一般的に Linux の Device Driver は，LKM として作成されるため，Device Driver に対して

```
function_fd_call();
function_ld_call();
```

というような関数を加えることで実装が可能であり，特に FD と LD は対になっているため共通するコードが使えるためリファレンスモニタを小さくすることが可能だと考えられる．しかしながら，この実装の仕方では個々の Device Driver に対して FD と LD を作らなければならない．

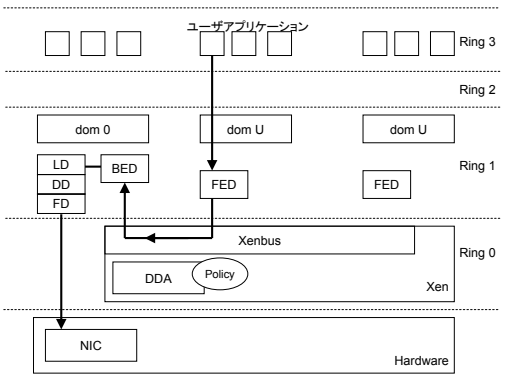

図 5 提案アーキテクチャの Xen への実装
Fig. 5 Implementation of proposed architecture by Xen

6.3 FD，LD，DDA の通信におけるデータ構造

FD，LD と DDA は通信を行う．提案アーキテクチャは単純な動作をするため最低限必要なデータ構造は Device のアイデンティティとハッシュ値だけでよい．提案アーキテクチャは I/O 処理に対して MAC を執行するためパフォーマンスの低下は明らかであり，データ構造に別の項目を加えてパフォーマンスの低下を軽減できる可能性がある．

6.4 評 価 方 法

提案アーキテクチャは，実装してみなければパフォーマンスの低下にどれほど影響を与えるのか不明である．そのため，全ての機能を有効にして，もっとも負荷を

与える状況での評価を行う．また，脆弱性を持った Device Driver を作成し，実際に攻撃を行い，有効性を証明する．

7. まとめと今後の課題

本稿では VMM における問題点を指摘し，その対策として Device Driver に対するリファレンスモニタを提案し，アーキテクチャを示した．しかしながら，DDA に関して抽象的な部分が多い．今後の課題としては，Device Driver に対するリファレンスモニタによって攻撃検知が可能であることを示すとともに，パフォーマンスへの影響を調べ，対策の検討を行う．また，ポリシーを検討する必要がある．

参 考 文 献

1) Xensource, “XenSource: Open Source Virtualization”. http://www.xensource.com/
2) kvm-wikiKernel, “Based Virtual Machine”. http://kvm.qumranet.com/kvmwiki
3) IBM Research, “sHype: Secure Hypervisor”. http://www.research.ibm.com /secure_systems_department/projectsHypervisor /index.html, 2005.
4) 安藤類央, 門林雄基, “仮想マシンモニタ（XEN）上のデバッグ機能の改良によるフォレンジックシステムの構築”, Oct. CSS2006.
5) Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, Dan Boneh, “Terra: A Virtual Machine-Based Platform for Trusted Computing”, ACM SIGOPS Operating Systems Review, Vol.37, No.5, pp.193-206, 2003.
6) Microsoft, “Next-Generation Secure Computing Base” http://www.microsoft.com/resources /ngscb/default.mspx
7) 藤澤 一樹, 橋本 正樹, 宮本 久仁男, 金 美羅, 辻 秀典, 田中 英彦, “仮想マシンモニタにおけるデバイスドライバ安全性向上に関する提案”, FIT2007 第 6 回情報科学技術フォーラム Sep. 2007.
8) Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, Benjamin Chelf, “Bugs as deviant behavior: a general approach to inferring errors in systems code”, In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, 2001.
9) Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, Abdullah Ustuner, “Thorough static analysis of device drivers”, In Proceedings of the 2006 Eurosys Conference.
10) Christopher L. Conway, Stephen A. Edwards, “NDL: A Domain-Specific Language for Device Drivers”, In Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES), June, 2004.
11) Microsoft, “ロゴ及び WHQL テスト”. http://www.microsoft.com/japan/whdc /GetStart/default.mspx
12) Microsoft “Windows Driver Foundation”. http://www.microsoft.com/japan/whdc /driver/wdf/default.mspx
13) Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, “Safe Hardware Access with the Xen Virtual Machine Monitor”, OASIS ASPLOS 2004 workshop.
14) Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, Stefan Götz, “Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines”, Proceedings of the 6th Symposium on Operating Systems Design and Implementation, Dec, 2004.
15) Swift, M. M., Bershad, B. N., and Levy, H. M., “Improving the reliability of commodity operating systems”, In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP ’03.
16) Michael Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy, “Recovering Device Drivers” ACM Trans. on Computer Systems 24(4), November 2006.
17) 加藤和彦, “セキュア VM プロジェクトの概要”, 第 1 回セキュア VM シンポジウム, Mar. 2007.
18) Stephen D. Smalley, “NSA SecurityEnhanced Linux”, Ottawa Linux Symposium BOF 2006.
19) Ray Spencer, Peter Loscocco, Stephen Smalley, Mike Hibler, David Anderson, and Jay Lepreau, “The Flask Security Architecture: System Support for Diverse Security Policies”, In Proceedings of the 8th conference on USENIX Security Symposium, pp.123-139, August 1999.
20) Intel, “Intel® Virtualization Technology for Directed I/O Architecture Specification”, 2006.
21) AMD, “AMD64 Technology AMD64 Architecture Programmer ’s Manual Volume 2: System Programming”, 2006.
22) James P. Anderson, “COMPUTER SECURITY TECHNOLOGY PLANNING STUDY”, 1972.