修士論文

動的更新を可能にする
ハイパーバイザ分割手法

Hypervisor Modularization Enabling Dynamic Update

平成29年2月3日提出

指導教員
坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-146418　齋藤　晃介

概要

近年，クラウド事業者がユーザーに対してコンピュータリソースを貸し出すクラウドコンピューティングサービスが普及している．Infrastructure as a Service (IaaS) はコンピュータを仮想化し，ユーザーに仮想マシンを貸し出すサービスであり，ハイパーバイザは仮想マシンを作成・管理するためのソフトウェアである．

現在一般的に用いられているハイパーバイザは動的更新を行う機能を持っておらず，更新の際に再起動を必要としている．クラウド事業者は仮想マシンを終了させるハイパーバイザの更新をあまり行いたがらないため，サービスを中断することなくシステムを更新する手法はクラウドコンピューティングのセキュリティを向上させることになる．

本研究では，動的更新を可能にする方法の 1 つとして，ハイパーバイザの機能を分割しモジュール化する手法である Modular Hypervisor を提案する．モジュール設計は動的更新に適しており，実際に OS に動的更新を適用するために OS をモジュール化する研究は多く行われている．Modular Hypervisor はほとんどの場合仮想マシンの実行を中断することなく更新でき，例外的な場合にも既存手法に比べて極わずかな時間で更新が終了することが見込め，動的更新の点では優位であることを示した．

目 次

図 目 次

表 目 次

第1章　はじめに

近年，企業が自社サーバーの代わりに外部のクラウドコンピューティングサービスを利用しているケースが多くなっている [3]．クラウドコンピューティングサービスはクラウド事業者がユーザーに対してコンピュータリソースを貸し出すサービスであり，その中でも Amazon EC2 [4] などの Infrastructure as a Service (IaaS) はコンピュータを仮想化し，ユーザーに仮想マシンを貸し出すサービスを行っている．ハイパーバイザは仮想マシンを作成・管理するためのソフトウェアであり，IaaS では図 1.1 のようにハイパーバイザを利用して仮想化環境を構築している．

また，2014 年の情報処理推進機構 (IPA) の調査では，自社サーバーにセキュリティパッチを適用していないと回答している企業が多く存在し，その理由の 1 つとして「パッチの評価や適用に莫大なコストがかかる」というものが挙げられている [5]．セキュリティパッチの適用にはシステムの再起動が必要になることが多く，これがセキュリティパッチを適用しない一因となっている．ゆえに，システムを再起動することなくセキュリティパッチを適用することができる動的更新が可能となれば，セキュリティパッチを適用しないサーバーが減少し，セキュリティが向上する．

現在一般的に用いられているハイパーバイザ（Xen [6]，Microsoft Hyper-V [7]，VMWare ESXi [8] など）は動的更新を行う機能を持っておらず，更新の際に再起動を必要としている．クラウド事業者はユーザーの仮想マシンの実行を中断する時間を最小限にしなければならない．例えば，Amazon Web Services では，サービス利用者に対して月間使用可能時間割合を 99.95 %以上にする契約をしている [9]．したがって，クラウド事業者は仮想マシンを中断する必要があるハイパーバイザの更新をあまり行いたがらないため，サービスを中断することなくシステムを更新する手法はクラウドコンピューティングのセキュリティを向上させることになる．現在，ライブマイグレーションを用いることにより疑似的に動的更新を行うという方法がある．ライブマイグレーションでは仮想マシンの動作を停止せずに実行している物理マシンから別の物理マシンに転送することができ，全ての仮想マシンを転送した後で元の物理マシンを再起動するという方法である．しかし，この方法には高いネットワークの負荷と時間がかかり，現実的に実行するには非効率である．

既存手法である Swap and Play では Xen を数十ミリ秒で動的更新している．一見すると十分に早いように思えるが，クラウドサービスを利用したオンラインゲームや株取引など，数ミリ秒の単位で時間を競う用途にとっては非常に遅い．人が

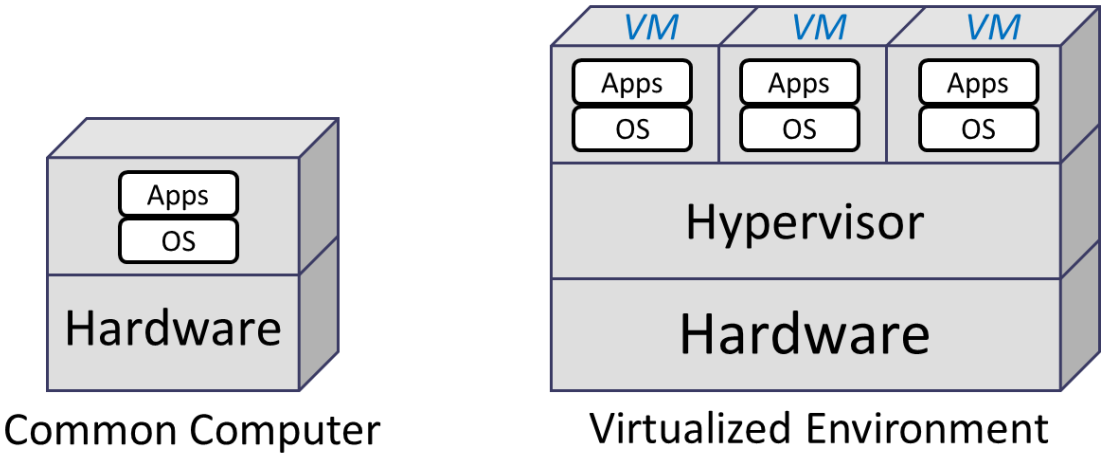


図 1.1: 一般的なコンピュータと仮想化環境

いない時間帯を狙って更新をする方法では動的更新のメリットが薄れる上，クラウドサービスでは世界中からのアクセスがあり，一日中仮想マシンを動かしていることも多いためそもそも人がいない時間帯に更新をすることはできない．

本研究では，動的更新を可能にする方法の 1 つとして，ハイパーバイザの機能を分割しモジュール化する手法である Modular Hypervisor を提案する．モジュール設計は動的更新に適しており，実際に OS に動的更新を適用するために OS をモジュール化する研究は多く行われている [10] [11]．Modular Hypervisor は基本的に仮想マシンの実行を全く中断せず，例外的な場合にもマイクロ秒のオーダーで動的更新を行うことができることが見込める．

本論文では，まず第 2 章で動的更新について説明し，第 3 章でモジュール化について説明する．第 4 章で提案手法であるハイパーバイザのモジュール化について説明する．

第2章　動的更新

動的更新システムの研究は古くは1970年代より存在している [12]．OSや一般的なアプリケーションにおける動的更新については多くの研究が存在するが，ハイパーバイザの動的更新については現在2014年に発表されたFerdinand Brasserらの Swap and Play [1] の一件しか存在していない．

一般的に，動的更新の手法は，モジュール化設計と計装ベース設計の2つに分けられる．モジュール化設計については第3章で解説し，本章では計装ベース設計とその他の手法である Swap and Play について解説する．

2.1　計装ベース設計

計装ベース設計とは図2.1のように，機能を更新する必要が生じた時に，メモリに新しいバージョンの関数を読み込み，更新前の関数から新しい関数へ分岐するように制御フローを変更する手法である．これは通常更新前の関数の先頭に新しい関数へのジャンプ命令を挿入することによって実装されており，この分岐の追加によりパフォーマンスが低下するのが計装ベース設計の欠点である．

Ksplice [13] はLinuxカーネルの動的更新システムであり，更新された新しいバージョンの関数をカーネルモジュールとして読み込み，更新前のバージョンから分岐することによってカーネルを更新する．Kspliceは2011年にOracleによって買収され，以降Oracle Linuxでのみサポートされているが，2014年にRed HatとSUSEがそれぞれ独自の動的更新システムである kpatch [14] と kGraft [15] を開発している．

また，2015年4月にリリースされたLinuxカーネル4.0ではlive patchという動的更新機能が組み込まれている [16]．その後1年半ほどは商用版のRed HatとSUSEでサポートされるのみだったが，2016年10月にCanonicalは商用版Canonicalおよびコミュニティ版Ubuntuユーザーに対してCanonical Livepatch Serviceの提供を開始した [17]．

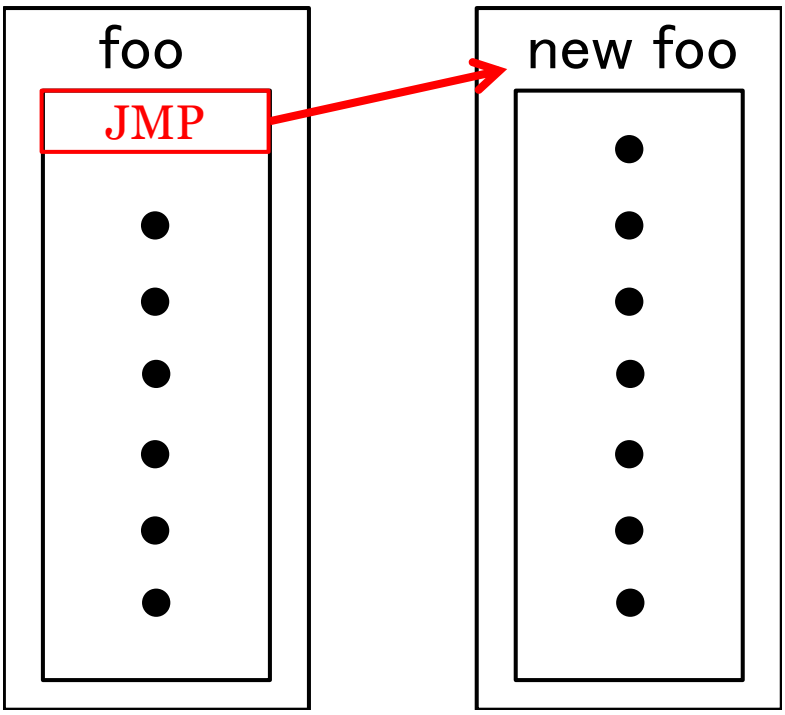


図 2.1: 計装ベース設計

2.2 Swap and Play

Swap and Play は Ferdinand Brasser らによって提案された，仮想マシンを再起動せずにハイパーバイザの動的更新を行うための手法である．Swap and Play では実行中の仮想マシンを終了させず，わずかな時間だけ実行を中断させている間にハイパーバイザ全体を置き換えて更新する．そのため，計装ベース設計のようにパフォーマンスが低下することがなく，また一般的なハイパーバイザに少しの改変を加えるだけで実装することが可能である．

更新の手順は図 2.2 のようになっている．まず，新しいバージョンのハイパーバイザをメモリ上に展開し，実行中のハイパーバイザが持つ内部状態（変数やポインタなど）を新しいハイパーバイザにコピーする．このままでは新しいハイパーバイザの持つポインタは更新前のハイパーバイザのアドレスを参照したままであるため，これを新しいアドレスを参照するように更新する必要がある．次に，ホストマシンが持つ情報（CPU レジスタ，割り込みコントローラなど）を更新前のハイパーバイザから新しいハイパーバイザに更新する．最後に，実行制御を新しいハイパーバイザに移し，更新前のハイパーバイザのメモリ領域を解放する．以上の手順によって，仮想マシンを終了させることなくハイパーバイザの更新が可能になっている．

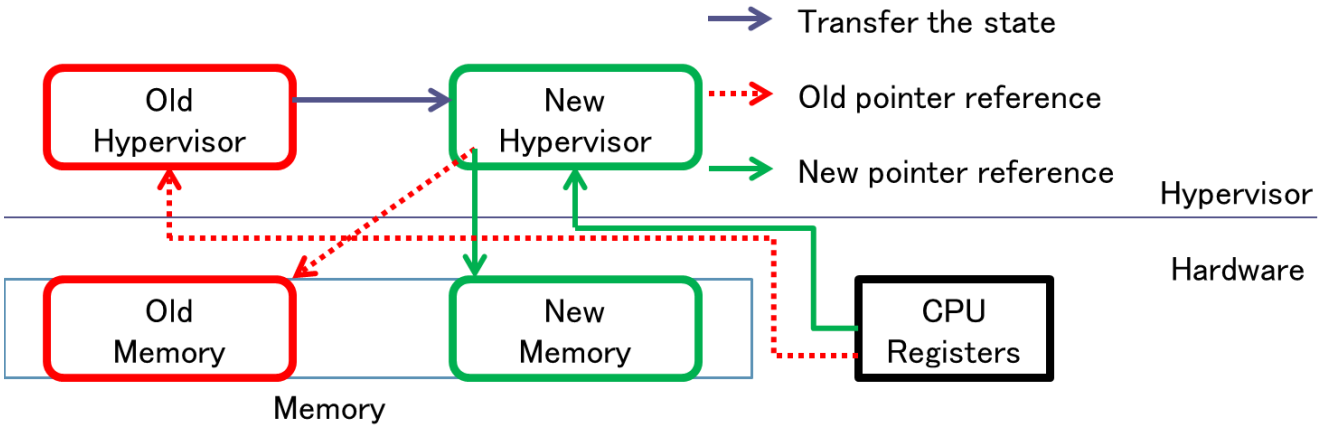


図 2.2: Swap and Play における動的更新

この手法はハイパーバイザの更新中のわずかな時間のみ仮想マシンの実行を中断するが，ほんのわずかな修正であってもハイパーバイザ全体を更新しなければならないという問題がある．

表 2.1: Xen の更新によるデータ構造の変化数 [1]

	4.2.1	4.2.2	4.2.3	4.2.4
4.2.0	0	2	4	5
4.2.1		2	4	5
4.2.2			2	3
4.2.3				1

2.3 SwapVisor

Swap and Play を今日クラウドサービスにおいて広く用いられているオープンソースのハイパーバイザである Xen に適用したものが SwapVisor である．SwapVisor における仮想メモリの構造は図 2.3 に示すようになっている．

SwapVisor は，Xen アーキテクチャに新しいハイパーコールを導入しており，ハイパーコールはドメインからハイパーバイザへのトラップになっている（アプリケーションからカーネルまでのシステムコールと類似）．ハイパーコールは，ページテーブルの更新などの特権操作を要求するためにドメインによって使用される．SwapVisor はこのハイパーコールを用いて，図 2.4 のように Xen を更新する．しかし，これではまだしか更新されておらず，図 2.5 の 1〜3 に示されるようにメモリのスタック領域と CPU のスタックポインタも更新する必要がある．

1. ベースポインタが保存されているアドレスを更新する．
2. リターンアドレスを更新する．
3. 更新するべきアドレスを更新し，スタックポインタを更新する．

このようにして SwapVisor では Xen を更新する．

実験では Xen バージョン 4.2.0 からバージョン 4.2.1 へのアップデートは 3.2GHz の CPU において約 45 ミリ秒以内に実行され，図 2.6 にあるように性能低下もかなり少ないことが示されている．また Xen の更新によりデータ構造に変化があった場合，その分は手動でパッチを当てなければならないのだが，表 2.1 にあるように更新によるデータ構造の変化は少なく，手動でも十分に対応が可能な範囲となっている．以上のことから，SwapVisor による動的更新は一般の用途では十分に有用であることが分かる．しかし，Xen4.2.1 のアップデートは小規模なものであったことを考えると，アップデート内容によっては仮想マシンの中断時間がさらに増大してしまう恐れがある．また 45 ミリ秒という時間仮想マシンの実行が中断されるのも，オンラインゲームの大会や株取引の自動プログラムなどの用途を考えると非常に長い時間であり十分とは言えない．

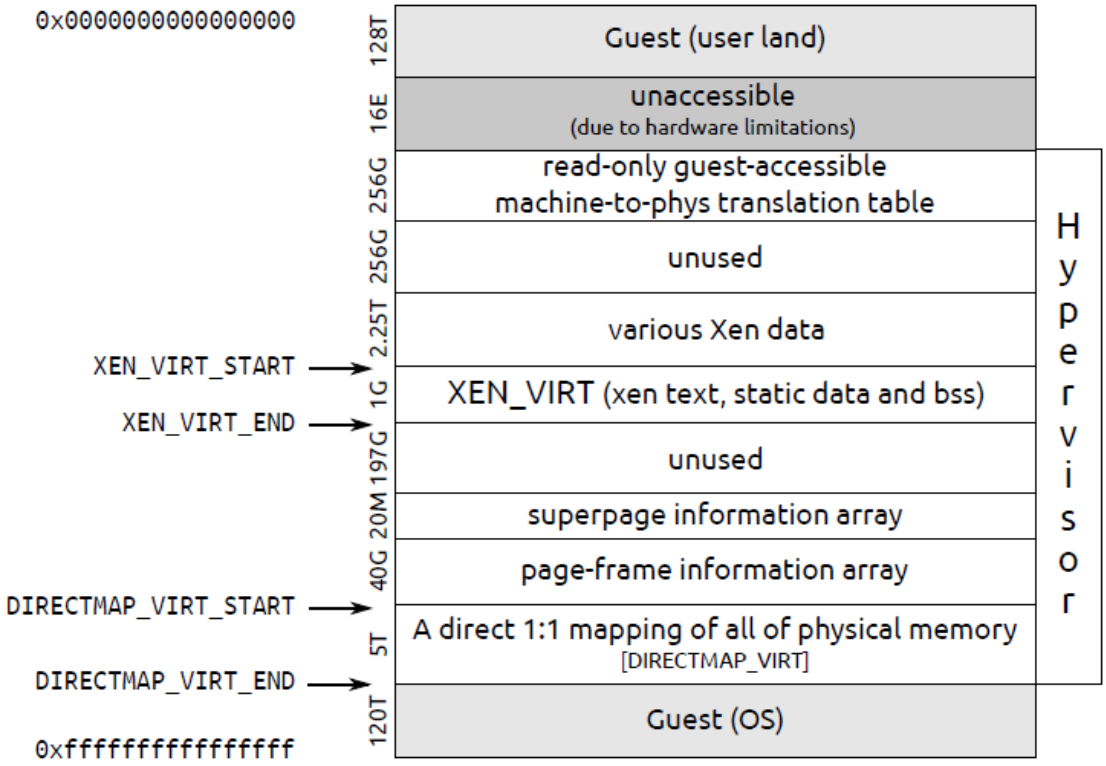


図 2.3: SwapVisor における仮想メモリ [1]

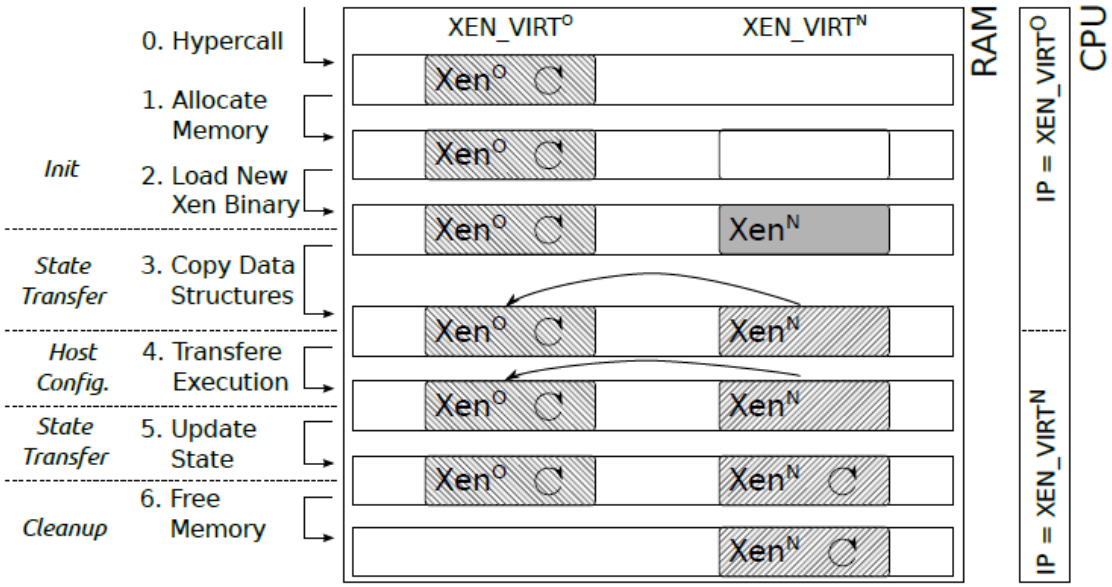


図 2.4: SwapVisor における Xen の動的更新 [1]

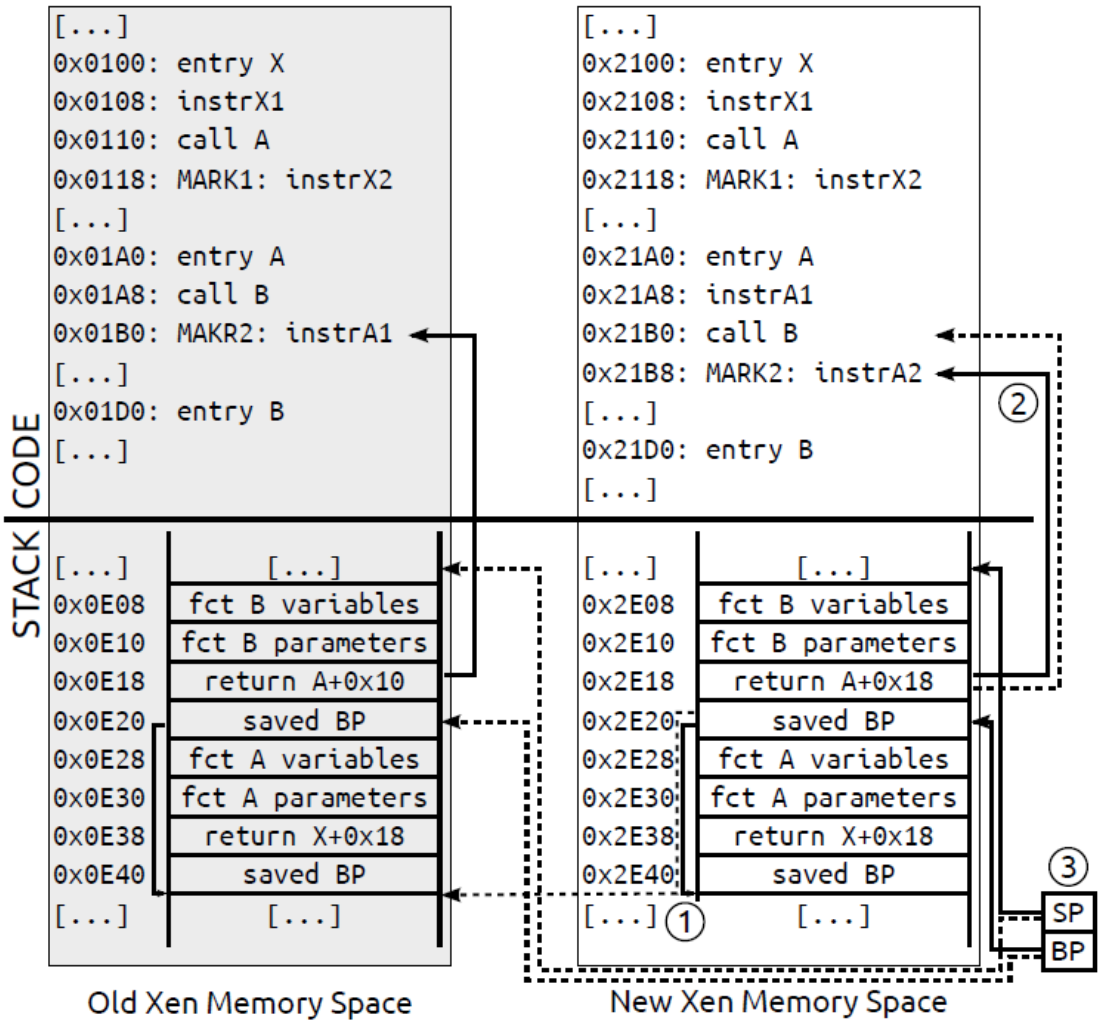
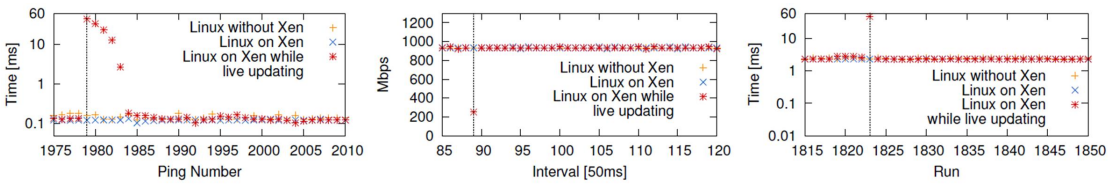


図 2.5: SwapVisor におけるスタックの更新 [1]



(a) Network Latency (b) TCP Throughput (c) CPU Throughput

図 2.6: SwapVisor の性能評価 [1]

第3章　モジュール化

第 2 章でも触れたように，モジュール化は動的更新に適した設計である．また同様に，OS 等のモジュール化については K42 [10] や Baumann ら [11] の研究など多くの研究がなされているが，ハイパーバイザのモジュール化についての研究はあまりなされていない．しかし，OS とハイパーバイザには多くの共通点があるため，これらについて知ることも重要である．

本章では，まず OS のカーネル機能をモジュール化したマイクロカーネルについて説明する．次に動的更新の機能は持っていないが，モジュール設計のハイパーバイザである eXtensible and Modular Hypervisor Framework (XMHF) [2] を紹介する．

3.1　マイクロカーネル

図 3.1 のように，全ての機能を 1 つのカーネルに内包するモノリシックカーネルに対し，カーネルには最小限の機能のみを持たせ，それ以外の機能をサーバ群と呼ばれるモジュールに分割する設計をマイクロカーネルという [18][19]．特権モードが必要になる IPC（プロセス間通信），基本スケジューラなどの機能だけがカーネル空間に置かれ，スケジューラ本体，メモリ管理，ファイルシステムなどの大部分の機能はモジュール化しユーザ空間で動作する．

基本的にマイクロカーネルはモノリシックカーネルに比べて次のような長所・短所を持つ．

長所：

- 保守が容易である．
- パッチの開発が容易で，更新に再起動を必要としない（動的更新が可能）．
- バグや障害に対する耐性が高い．

短所：

- モジュールと合わせてメモリを多く消費する．
- カーネルとモジュール間のメッセージングのバグが検出困難．
- プロセスの管理が複雑になる．

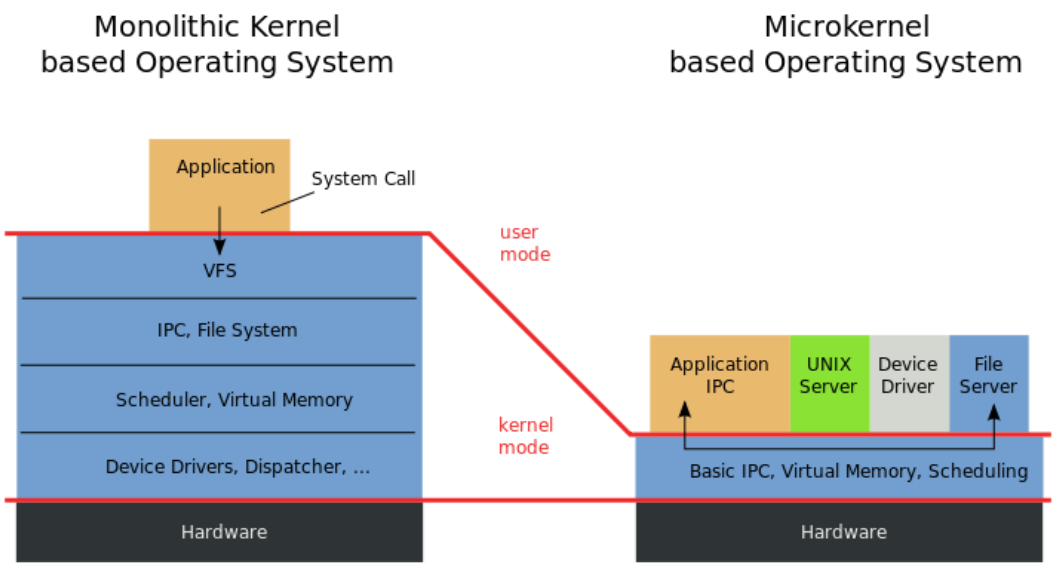


図 3.1: モノリシックカーネルとマイクロカーネル

- オーバーヘッドが生じ，性能が低下する．

このような長所と短所があるため，モノリシックカーネルとマイクロカーネルにはトレードオフの関係がある．マイクロカーネルのプロセス管理の複雑さやオーバーヘッドによる性能低下からか，現在 OS として主流であるのはモノリシックカーネルであることが多い．

また，モノリシックカーネルとマイクロカーネルのどちらの特徴も併せ持つカーネルをハイブリッドカーネルと呼び，ほとんどのカーネルは明確にどちらかに決まっているわけではないことが多い．例えば Linux などはモノリシックカーネルであるがモジュールをロードする機能を持っており，必要に応じて機能を拡張することができる．そのため Linux はハイブリッドカーネルでもあるといわれることがある．

3.2 XMHF

XMHF はシングルゲスト型のハイパーバイザであり，ハイパーバイザ上で 1 つの仮想マシンのみが実行されている．Xen などのマルチテナント型のハイパーバイザと違い複数の仮想マシンを実行することはできないが，その分物理リソースを動的に割り当てる必要がなく比較的簡単な設計になっている．

XMHF はモジュール設計がされており，図 3.2 のようにハイパーバイザとしての各機能を“hypapp”と呼ばれるアプリケーションに似たモジュールに分割し，それがハイパーバイザのコア部分によって管理されている．この研究はメモリの完全性

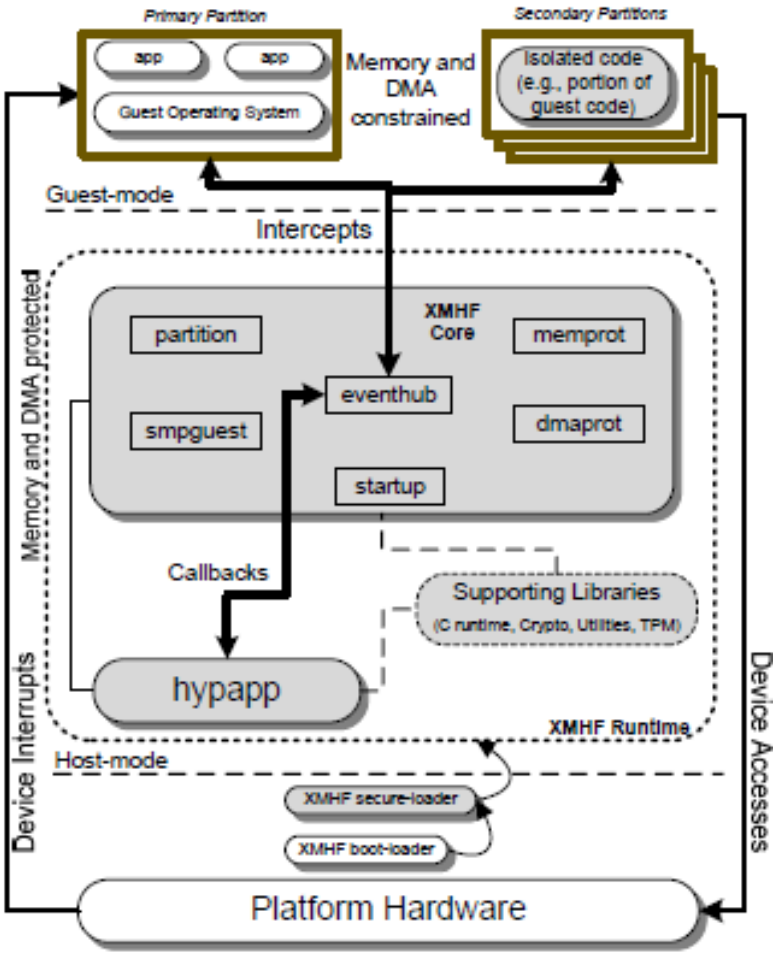


図 3.2: XMHF の構成 [2]

に重点を置いており，ハイパーバイザのコアがメモリへのアクセスを一括して行うことで，メモリの完全性を意識することなくモジュールを作成することができるように設計されている．SourceForge.net の Web サイト [20] では既に TrustVisor [21] や Lockdown [22] などの機能がモジュール化され公開されている．

第4章　Modular Hypervisor

動的更新に関する既存手法である Swap and Play では，数十ミリ秒もの時間仮想マシンの実行が中断されてしまうという問題がある．そこで，ハイパーバイザ全体を一度に更新するのではなく，機能を分割して個別に更新することで仮想マシンを実質的に中断せずに更新できる手法である Modular Hypervisor を提案する．Modular Hypervisor では基本的にモジュールの関数が終了した時点で関数のアドレスを書き換えるだけで更新が終わるので，仮想マシンを中断せずに更新を行うことができる．例外的に中断する場合にも，既存手法と同様に 3.2GHz の CPU で考えると中断時間はマイクロ秒のオーダーで更新できることが見込める．

本章では，ハイパーバイザの持つ機能を分割し，モジュール化する提案手法である Modular Hypervisor について説明する．Modular Hypervisor はセキュリティを向上させるために動的更新を可能にすることを目的としている．ハイパーバイザには OS 上で実行するホスト型ハイパーバイザと，ハードウェア上で直接実行するベアメタル型ハイパーバイザの 2 種類が存在するが，Modular Hypervisor では後者のベアメタル型ハイパーバイザに適用することを想定している．

第 3 章でも説明したように，OS とハイパーバイザには多くの共通点があるため，マイクロカーネルの考え方をハイパーバイザのモジュール化に役立てることができる．また，この研究の主目的はセキュリティの向上であるため，ハイパーバイザが改竄されないようにするための認証方法も検討する．

4.1 節ではハイパーバイザをどのように分割するかを説明し，4.2 節ではこのハイパーバイザを認証する方法を説明する．また，4.3 節ではモジュールの更新方法を具体的に説明する．

4.1　ハイパーバイザの分割

Modular Hypervisor では図 4.1 のように，ハイパーバイザをコアとモジュールに分割する．ハイパーバイザとしての機能はほぼ全てモジュールに分割し，コアの部分は最小限の機能のみを持たせるように設計する．そうすることによってコアを更新する必要がなくなり，モジュールのみを各々更新すればよいことになるため仮想マシンをほぼ中断せずに動的更新を行うことができるようになる．

そのためにハイパーバイザに追加しなければいけない機能として，モジュールを認証する機能とモジュールを管理する機能がある．この 2 つの機能を持ったモ

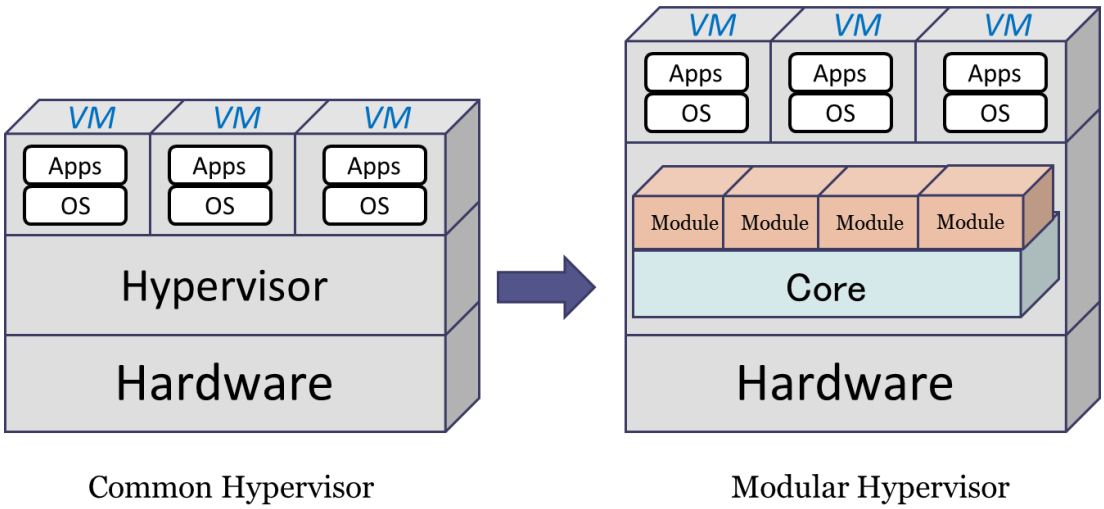


図 4.1: ハイパーバイザのモジュール化

ジュールについては他のモジュールとは異なる扱いをしなければならない．

4.1.1 コア

コアによって全てのモジュールが管理されるようにすると，コアを更新している間は仮想マシンの実行を中断しなければならなくなる．加えて，モジュールの管理や更新，認証，メモリ管理機能などの脆弱性の温床になりやすい機能は頻繁に更新したいため，コアの機能には含むべきではない．

そこで，コアには更新が不要な必要最小限の機能のみを持たせるように設計する．その必要な機能の 1 つとしてブートローダ機能がある．ブートローダは二次記憶からメインメモリ上にプログラムをロードするための小さなプログラムであり，主にシステムを起動する際にのみ使用される．システムを起動した後にはメモリ上から破棄され再度使用されることがないため，実行中に更新が必要になることがない．

コアにはブートローダ機能の他，更新が不要な機能のみを持たせ，その他の機能を全てモジュールに分割してしまうことにより，コアの更新は不要となり各モジュールを個別に更新すればよいことになる．

また，マイクロカーネルに関する議論では，一般にモジュール化はシステムの性能低下を引き起こすと言われている．その原因としては，モジュール間のコンテキストスイッチによるオーバーヘッドが主要因として挙げられる．このコンテキストスイッチは各モジュールが個別のプロセスにて動作していることによって発生するため，モジュールを同一アドレス空間上で動作させることによってオーバーヘッドを抑制することができる．

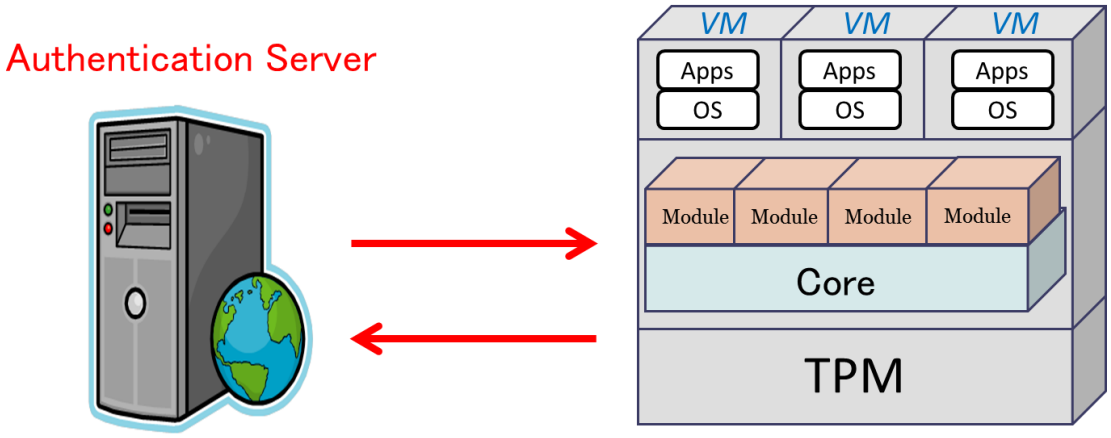


図 4.2: 認証サーバーと TPM による認証

4.1.2 モジュール

コアのブートローダ機能以外の全てのハイパーバイザとしての機能はモジュールとして分割する．このモジュール化に際して，次の 2 つの機能をモジュールとして新たに追加する必要がある．

1 つ目の機能はモジュールを認証する機能である（図 4.2）．この認証モジュールによって新しく読み込む全てのモジュールを認証する．詳しくは 4.2 節で解説するが，認証モジュールを認証するためには Trusted Platform Module (TPM) [23] を利用する．TPM はチップ内で RSA 暗号演算や鍵の生成・ハッシュ値計算を行うことができるハードウェア耐タンパー性を持つセキュリティチップであり，一般的にマザーボード上に設置されていることが多い．TPM は物理的に安全に暗号計算を行うための耐タンパーセキュリティチップであるため，性能自体は低くなっている．そのため CPU と接続する LPC バスや I2C バスなどが非常に遅くなっており，サイズの大きいプログラムを暗号化するには時間がかかる．提案手法では，TPM では認証と認証サーバーとの通信に必要な一部のモジュールのハッシュ値のみを計算するので，ハイパーバイザ全体を計算するよりも早く処理を行うことができる．

2 つ目の機能はモジュールを管理する機能である（図 4.3）．これは特権を持つ管理仮想マシンがモジュールの追加・削除・更新を行えるようにするための機能であり，この機能を持つモジュールを管理モジュールと呼ぶ．管理モジュールは全てのモジュールを管理するが，そのために必要なメモリ管理機能とシステム起動時に認証モジュールを認証する機能も管理モジュールが持つようにする．また管理モジュールが管理モジュール自身の更新を行う際には，他のモジュールとは違う特別な手順で更新を行う．

各モジュールは管理モジュールによって動的更新が可能となるように設計する．そうすることによって，Swap and Play と同様に新しいモジュールをメモリに展

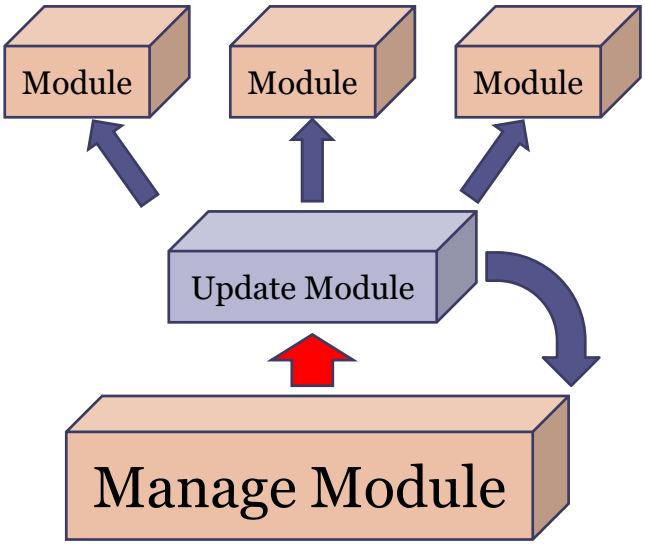


図 4.3: 管理モジュールと更新モジュール

開して動的更新を行うことができる．この際，更新の対象となるモジュール以外のモジュールは停止する必要がないため，更新の対象となるモジュールを使用している（あるいは使用しようとしている）仮想マシン以外の実行を中断する必要がなくなり，Swap and Play よりも仮想マシンの停止時間が大幅に短くなるという点で優れている．

また認証モジュールを認証することによって，不正なモジュールを検出することができ，セキュリティを向上させることができる．さらに図 4.4 のように，各ユーザーが使用したいモジュールを選択することができたり，逆に管理者が使用可能なモジュールを制限したりすることもできるようになり，新たなサービスを提供することが可能になる．

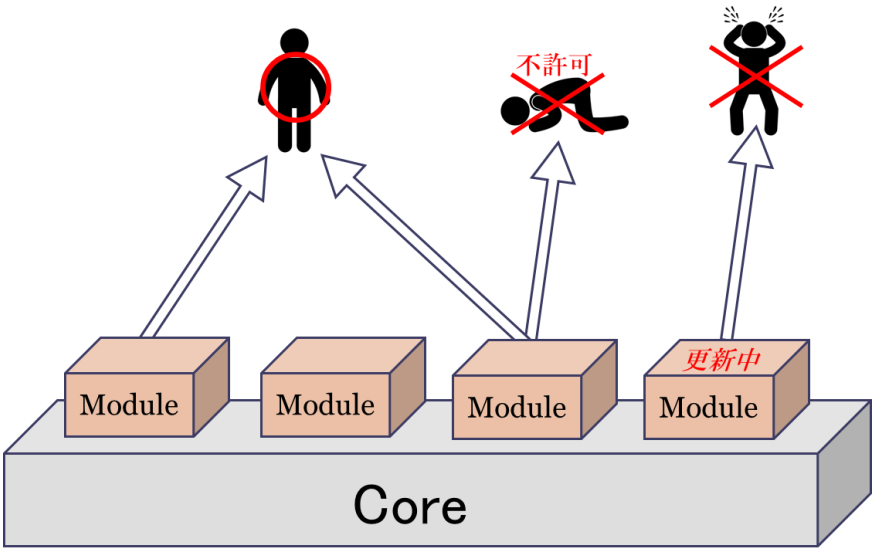


図 4.4: ユーザーとモジュール

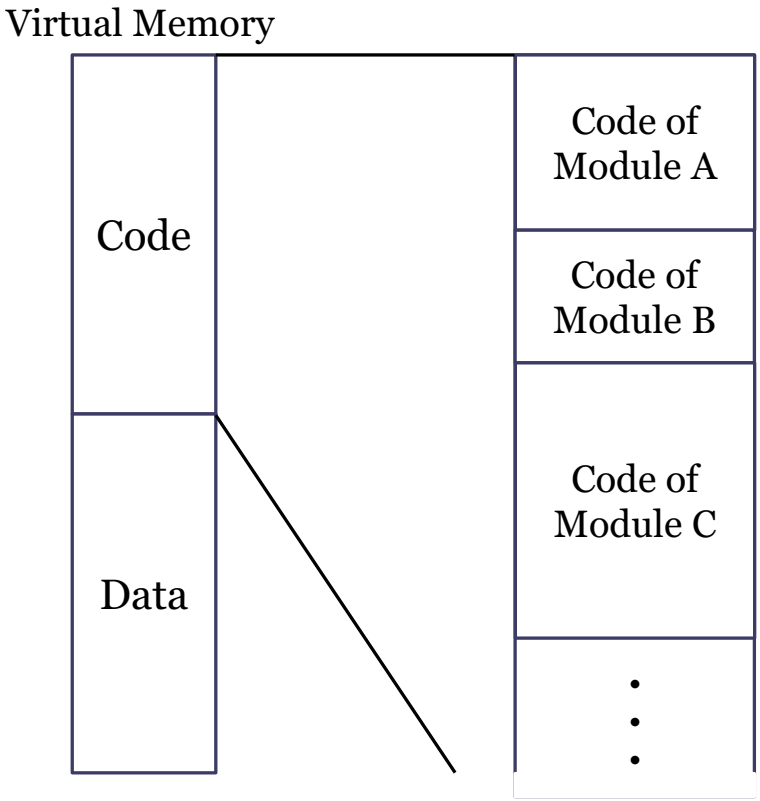


図 4.5: Modular Hypervisor の仮想アドレス空間

4.1.3 モジュールの設計

モジュールはハイパーバイザの機能をいくつかの関数単位に分割することによって実現する．同一のアドレス空間上に関数を配置することにより，分割する前のハイパーバイザと同じ動作をすることができる．また同一アドレス空間上にて動作するように設計することで，マイクロカーネルにおけるモジュール化設計において懸念されるプロセス間通信による性能低下はほぼ存在しなくなることが見込める．

出来る限り細かく分割することで更新するモジュールを小さくしたいが，1 つのモジュールに 1 つの関数を担当させるとモジュールの数が膨大になり管理が困難になる．関連性の高い関数は 1 つのソースコードにまとめて記述してあることが多いため，ソースコードのファイルを単位としてモジュール化するのが基本となるが，実際には関連性のあるいくつかのソースコードファイルをまとめて 1 つのモジュールとするのが保守の観点から見ても適当である．

図 4.5 のように，Modular Hypervisor は全てのモジュールを同一の仮想アドレス空間上に配置し，モジュールのコード領域とデータ領域を分割して配置する．ここで，モジュールのデータ領域は原則として更新してもアドレスが変化しないよ

うに設計する．そうすることによって，モジュールのコード領域を更新して関数のアドレスが変化しても，データ構成が変わらなければデータ領域を更新することなくデータの引継ぎを行うことができる．

Swap and Play ではハイパーバイザのコード領域とデータ領域全てをコピーした上でアドレスの更新を行っていたため短い時間ではあるが仮想マシンの実行を中断しなければならなかった．Modular Hypervisor では基本的にモジュールの関数が終了した時点で関数のアドレスを書き換えるだけで更新が終わるので，仮想マシンの実行を中断せずに更新を行うことができる．

モジュールを管理するのは管理モジュールの役割であるため，モジュール内の関数やデータ領域のアドレスを管理モジュールに知らせるように設計する．管理モジュールは各関数を同一の仮想アドレス空間上に配置する．このようにモジュールを同一の仮想アドレス空間上に配置することによって，外見上分割する前のハイパーバイザと同じ動作をするように見せかけることができる．

しかし，これでは元々のハイパーバイザの機能を分割することはできても，既存のモジュールに対して新たな機能を提供するモジュールを追加することができない．そのために，フックを行うための登録用関数を用意することで解決する．フックとは，プログラム中の特定の部分に独自の処理を追加できるようにすることであり，フックを行う部分はあらかじめ開発者によって定められることになる．モジュールの設計者は関数内にフックを行うための処理を記述しておき，登録用関数を用意しておくことによって，新たなモジュールの開発者が既存のモジュールの特定の部分に独自の関数を追加することができるようになる．

また，複数のモジュール間において共有される関数や変数の扱いについては複雑なものとなるため，管理モジュールによって管理されることになる．具体的には 4.3 節で解説するが，その関数や変数はモジュールごとではなく管理モジュールが管理し，その関数や変数を使用するモジュールにアドレスを知らせるように設計する必要がある．

4.1.4 管理モジュールの設計

管理モジュールは全てのモジュールを管理するためのモジュールである．管理モジュール自体もモジュールであるため管理モジュールによって管理されるが，その扱いは他のモジュールとは異なる特別な扱いをしなければならない．

モジュールを管理するために，管理モジュールは以下の条件を満たすように設計する．

- ハイパーバイザの仮想アドレス空間を管理する
  ハイパーバイザ全体で同一の仮想アドレス空間を構成することでメモリ管理は非常に簡潔になる．仮想アドレス空間を利用することによって，物理アドレス空間上では連続で配置することができないデータを仮想的に連続で配

置されているように見せかけることができ，また実際の物理アドレス空間の大きさよりも大きな仮想アドレス空間を確保することができる．また管理モジュールは仮想アドレス空間を管理するためにページテーブルを作成し，それを管理する．ページテーブルを使用することで，仮想記憶をサポートするハードウェアではTLBを利用して仮想アドレス空間から物理アドレス空間への変換を高速で行うことができるようになる．

- 仮想アドレス空間上のコード領域とデータ領域を分ける
  ハイパーバイザ全体のメモリ領域を，各モジュールのテキスト領域（プログラム部分）のみをまとめた領域と，データ領域のみをまとめた領域の2つの領域に分割する．なぜなら，データ領域はモジュールの更新によってアドレスが変更されないように設計したいので，モジュールごとにテキスト領域とデータ領域をまとめてしまうとモジュールの更新の際にデータ領域のアドレスを変更しなければならないからである．ただし，分割すると言っても同一の仮想アドレス空間上に配置されているため，メモリ管理プログラムの記述が簡単になる以外の効果は特に存在しない．データ領域のアドレスが変更されないという条件さえ守られていれば，仮想アドレス空間上でデータが点在していても問題はない．

- グローバルな関数と変数の名前とアドレスをグローバルリストに保存する
  モジュールを更新する際，そのモジュールが持つ関数や変数のアドレスが変更されることがある．すると外部のモジュールがアドレスを変更されたグローバルな関数や変数を呼び出すことができなくなってしまう．そこで管理モジュールは全てのモジュールのグローバルな関数と変数の名前とアドレスをリストに保存しておく．

  この名前とアドレスを保存するリストをここでは**グローバルリスト**と称することにする．グローバルリストの要素は関数や変数の名前を表す文字列と，そのアドレスを表すポインタ変数の2つを持つ構造体である．グローバルリストから関数や変数を取り出すときは，その名前で構造体の文字列を探索し，一致する構造体のポインタ変数を取り出すことになる．管理モジュールはモジュールの追加・更新の際に外部から参照される関数と変数をグローバルリストに保存し，外部のグローバルな関数や変数を呼び出す部分をポインタ変数で書き換えてしまう．それによりモジュールは外部のグローバルな関数や変数のアドレスを知ることができ，また更新によりアドレスが変更になってしまっても変わらずアクセスすることが可能となる（図4.6)．このグローバルリスト自身も変数であるため，データ領域に保存されることになる．ただし，もともと直接関数や変数を呼び出していた部分をポインタ変数を経由して呼び出すようにしているため，外部のグローバルな関数や変数を呼び出すのはモジュール内の関数や変数を呼び出すよりも時間がかかってしまう．

Global List

Pointer	Name	Pointer Value
*Foo’	Foo	0x0FE730C2
・・・	・・・	・・・
*Hoge’	Hoge	0xEA460D19
・・・	・・・	・・・

図 4.6: グローバルリスト

グローバルリストには外部のグローバルな関数や変数に依存する多数のモジュールが必要とするが，グローバルリストを書き換えてよいのは管理モジュールだけである．そのため他のモジュールによってグローバルリストが書き換えられてしまうと，グローバルリストが破壊されてしまうという問題が発生する．各モジュールにグローバルリストのポインタ変数を渡すのは管理モジュールの役割なので，アドレスを渡す際にグローバルリストを書き換えようとしているかどうか判断する．グローバルリストを書き換えようとしていた場合，そのモジュールの追加または更新にエラーを発生させることでグローバルリストの破壊を防ぐことができる．

- モジュールを追加・削除・更新する
  モジュールは管理モジュールによってメモリ領域を割り当てられている．そのため，管理モジュールはモジュールのアドレスや空いているメモリ領域の情報を全て持っている．モジュールを追加する場合には，空いているメモリ領域に新たなモジュールを配置する．モジュールを削除する場合には，そのモジュールに割り当てたメモリ領域を解放する．モジュールを更新する場合には，4.3 節にて解説する更新処理を行う．簡潔にいえば，モジュールの追加と削除を同時に行い，古いモジュールのアドレスに新しいモジュールを仮

想的に置き換えてしまえばよい．

以上のように設計することで管理モジュールはモジュールを管理することができる．管理モジュールは Modular Hypervisor の根幹となる機能であるため，更新する際には 4.3 節にて解説する他のモジュールとは異なる特別な処理を行う必要がある．

4.2 認証

ハイパーバイザが改竄されていると，マルウェアやバックドアなどによりクラウドサービスの利用者が意図しない挙動を行い仮想マシンの情報が流出するなどの危険性がある．ハイパーバイザが改竄されていないことを利用者に証明するためには，TPM を用いたハイパーバイザの完全性検証手法 [24] を用いる．ただし前提条件として，信頼できる第三者の認証サーバーが存在することを仮定している．この手法によって，Modular Hypervisor が改竄されていないことが利用者に証明される．

4.2.1 認証モジュールの認証

まず TPM を用いてプラットフォームの完全性を検証する Trusted Boot[25] を行う．Trusted Boot ではプラットフォームのシステムの起動時に，現在実行しているプログラムが TPM を用いて次に実行するプログラムのハッシュ値を計測し，TPM 内部の Platform Configuration Register (PCR) に保存される．この計測を BIOS 内部にある書き換え不可能な Core Root of Trust for Measurement (CRTM) というプログラムから開始し，BIOS，ブートローダ，プラットフォームという順番に PCR に積み重ねられる．PCR に積み重ねられたハッシュ値は TPM の秘密鍵により暗号化されるのだが，ハイパーバイザの完全性検証手法ではこの TPM の鍵の証明を行っている．それにより，認証サーバーでハッシュ値が正しいことが確認され，現在起動しているプラットフォームの完全性が証明される．

Modular Hypervisor における Trusted Boot では，システムの起動後 BIOS・コア・管理モジュール・認証モジュール・ネットワークモジュールの順に実行し PCR にハッシュ値を積み重ねる．認証サーバーは BIOS・コア・管理モジュール・認証モジュール・ネットワークモジュールが正しく起動しているときのハッシュ値をあらかじめ知っているものとし，認証モジュールはこれらを実際に起動した時の PCR のハッシュ値を認証サーバーに送信する．認証サーバーがこれらのハッシュ値が一致することを確認することによって，認証モジュールの完全性が証明される．

4.2.2 モジュールの認証

これ以降読み込まれる全てのモジュール（更新により読み込まれる新しいバージョンのモジュールも含む）は，認証モジュールにより以下のプロトコルに従って認証される．認証は以下の 1〜4 のプロトコルで行われる．（図 4.7）．

1. 認証モジュールはランダムな数値を用いて秘密鍵 S_{HV} と公開鍵 P_{HV} を作成する．また，認証モジュールはあらかじめ認証サーバーの公開鍵 P_{AS} を取得しておく．

2. 認証したいモジュールのハッシュ値 $Hash(Module)$ と P_{HV} を P_{AS} を用いて暗号化し，認証サーバーに送信する．

3. 認証サーバーは受信した暗号を秘密鍵 S_{AS} で復号し，ハッシュ値 H（Module）が正しいか間違っているかの結果を S_{AS} で暗号化し認証モジュールに送信する．

4. 認証モジュールは受信した暗号を P_{AS} で復号し，結果が正しかったモジュールのみ実行を許可することによってモジュールの認証が完了する．

以上の手順によって全てのモジュールが認証モジュールに認証される．認証モジュールは既に認証サーバーによって認証されているため，Modular Hypervisor の完全性が証明される．

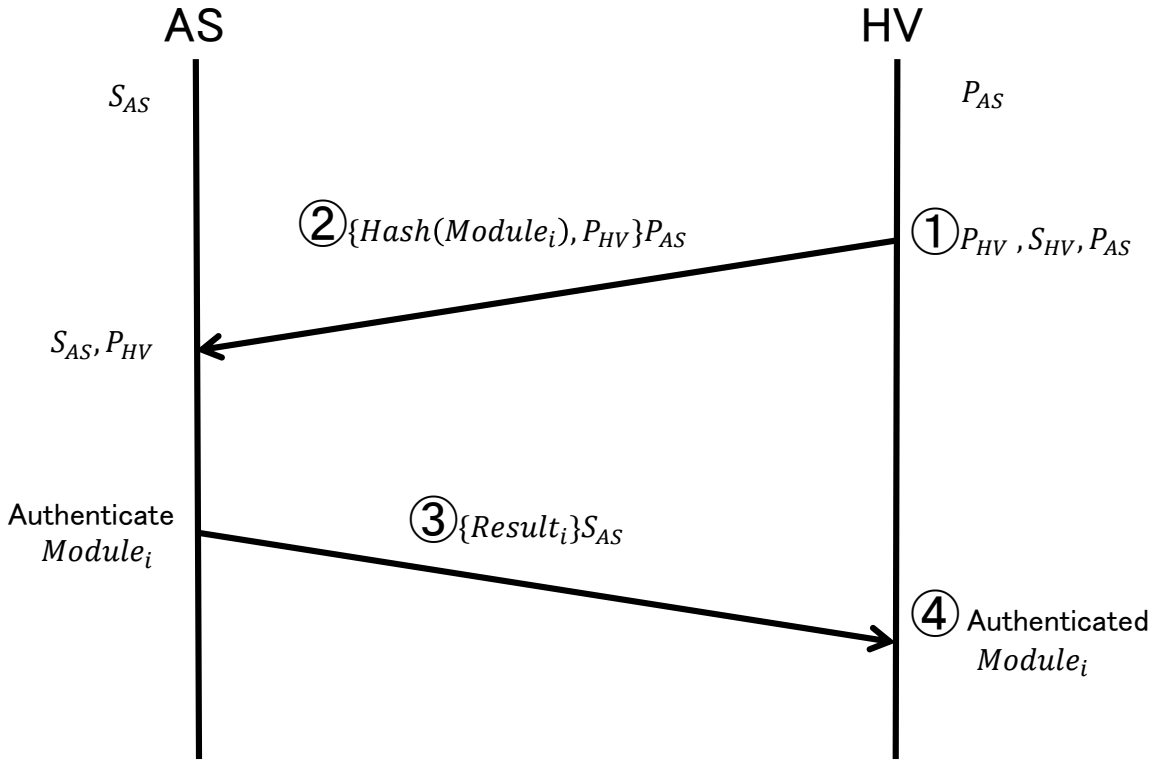


図 4.7: 認証モジュールの認証プロトコル

表 4.1: 図 4.7 の用語説明

symbol	***meaning***
AS	Authentication Server
HV	Hypervisor
P_{HV}	Public Key of Hypervisor
S_{HV}	Secret Key of Hypervisor
P_{AS}	Public Key of Authentication Server
S_{AS}	Secret Key of Authentication Server
Hash(X)	Hash value for X
{X}Y	Encrypted X by Y

4.2.3 利用者への証明

Modular Hypervisor の完全性は証明されたが，この時点ではまだそれを利用者が検証できていない．ハイパーバイザの完全性検証手法ではハイパーバイザの完全性が証明されたならば，ハイパーバイザと認証サーバーは共に共通鍵 C_{OT} を持ち，TPM の鍵を利用することでハイパーバイザが完全性を保ち続けていることを保証する．

利用者は以下の 1〜5 のプロトコルでハイパーバイザが改竄されていないことを検証する（図 4.8）．

1. 利用者は仮想マシンのディスクイメージ VM とそのハッシュ値 $Hash(VM)$ を共通鍵 C_{VM} で暗号化し，C_{VM} とランダムな数値 $Random$ を認証サーバーの公開鍵 P_{AS} で暗号化しハイパーバイザへ送信する．

2. ハイパーバイザは仮想マシンのディスクイメージを復号するための C_{VM} が P_{AS} で暗号化されているため，仮想マシンを起動できない．そのため，ハイパーバイザは暗号化された C_{VM} を認証サーバーに送信する．

3. 認証サーバーは，C_{VM} を秘密鍵 S_{AS} で復号し，C_{OT} で暗号化してハイパーバイザに送信する．

4. ハイパーバイザは C_{VM} を C_{OT} で復号し，VM のハッシュ値と 1 で送られた $Hash(VM)$ が一致することで VM が改竄されていないことを確認して仮想マシンを起動することができる．また $Random$ も復元し，起動した仮想マシンへの接続情報と $Random$ を C_{VM} で暗号化し利用者に送信する．

5. 利用者はハイパーバイザが $Random$ を持っていることを確認でき，仮想マシンへ接続することでハイパーバイザの完全性が認証サーバーにより証明されていることを確認できる．

以上により，ハイパーバイザが改竄されていないことを利用者に証明することができる．Modular Hypervisor も 4.2.2 節で完全性が証明されているため，Modular Hypervisor が改竄されていないことが利用者に証明される．

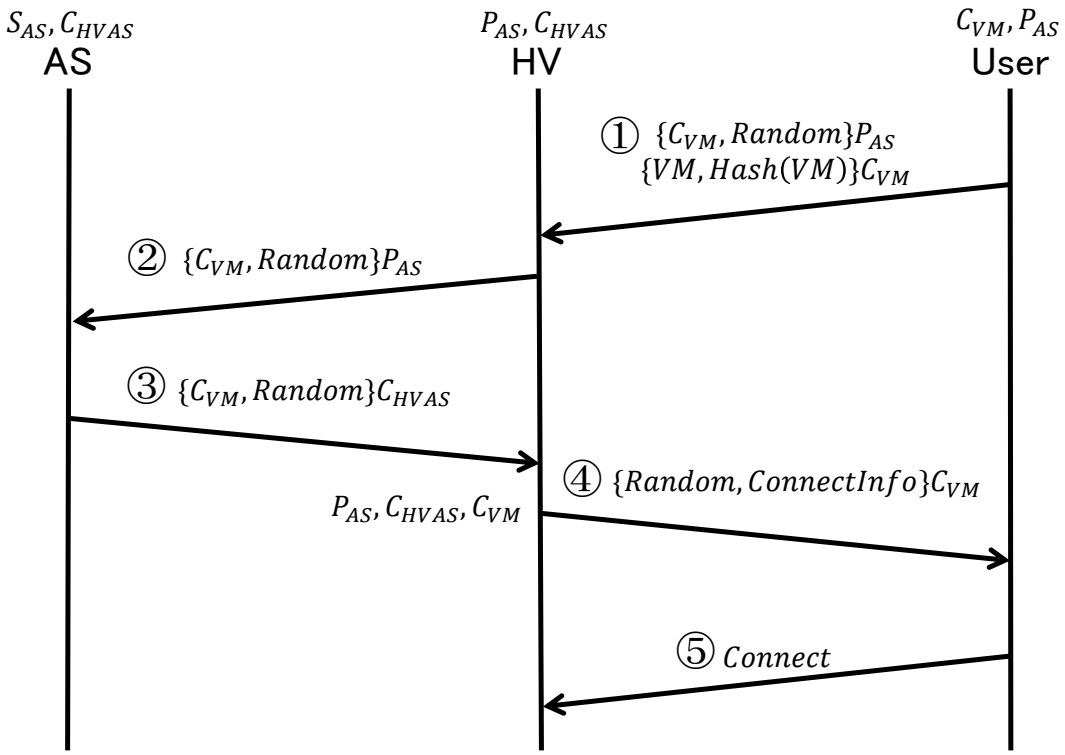


図 4.8: 利用者への証明プロトコル

表 4.2: 図 4.8 の用語説明

symbol	***meaning***
AS	Authentication Server
HV	Hypervisor
User	Service User
VM	Virtual Machine
P_{AS}	Public Key of Authentication Server
S_{AS}	Secret Key of Authentication Server
C_{VM}	Common Key of User
C_{HVAS}	Common Key of HV and AS
Hash(X)	Hash value for X
{X}Y	Encrypted X by Y

4.3 モジュールの更新

4.3.1 更新されるモジュール

モジュールを更新するにあたって，更新される対象モジュールは以下の条件を満たすように設計する．

- 更新時にのみ実行される更新関数を用意する
  更新関数は基本的に空の関数となる．更新関数が空とならないのは，更新によってデータ構造が変化した場合や管理モジュールを更新する場合のみである．更新関数が実行されている間は対象モジュールを使用しようとする仮想マシンの実行が中断してしまうため，モジュールは原則としてデータ構造を変化させないように設計するべきである．

- 管理モジュールに対象モジュール内のグローバルな関数と変数を知らせる
  管理モジュールはグローバルな関数や変数をグローバルリストに保存する必要がある．そのためモジュールを更新する際，対象モジュール内のグローバルな関数と変数の一覧を管理モジュールに知らせるように設計する．

- 管理モジュールに対象モジュールが呼び出す外部のグローバルな関数や変数を知らせる
  外部のグローバルな関数や変数を呼び出す場合には，管理モジュールからそのアドレスをグローバルリストのポインタ変数で書き換えてもらう必要がある．管理モジュールは要求された関数や変数の名前でグローバルリストを探索し，そのポインタ変数で対象モジュールの該当部分を書き換えることになる．

4.3.2 モジュールの更新方法

管理モジュール自身以外の全てのモジュールは管理モジュールによって更新が行われる．モジュールを更新するにあたって，管理モジュールは以下の条件を満たすように設計されている．

- グローバルリストを管理する
  管理モジュールは全てのモジュールのグローバルな関数と変数の名前とアドレスをグローバルリストとして保存している．モジュールは新しくメモリに展開された段階では外部のグローバルな関数や変数のアドレスを知らないため，そのままでは呼び出すことができない．外部のグローバルな関数や変数を呼び出す場合，グローバルリストのポインタ変数を経由することでグローバルな関数や変数を呼び出すことができる．

- 対象モジュールの内部状態を受け取り，新しいモジュールを初期化する
  データ構造に新たなデータが追加されていた場合には，あらかじめ十分に確保しておいたメモリ領域に追加することで解決する．しかし，既存のデータの変更や削除は互換性が取れなくなる恐れが非常に高いため，そのような設計はするべきではない．どうしてもデータの変更や削除を行わなければならないときには，以前のデータ構造と整合性が取れるように新しいデータを対応する古いデータで初期化し，不要になったデータを削除する．

このような機能を持つことによって管理モジュールはモジュールを更新することができる．更新は以下の 1〜10 の手順で行われる．

1. 競合の確認を行う
   グローバルな関数や変数は他のモジュールによって使用されている可能性があるため，名前などの構成の変更や削除があった場合には他のモジュールとの競合が発生するかどうか確認する必要がある．競合が発生していた場合，エラーとなり更新を中断しなければならない．その場合には競合が発生しないように更新を変更するか，競合するモジュールを同時に更新することで競合が発生しないようにする．

2. 新しいモジュールをメモリ領域に展開する
   新しいモジュールが持つ関数をメモリ領域に展開する．展開するのはコード領域だけで，データ領域を展開する必要はない．メモリ管理を行っている管理モジュールを更新する場合でも，この段階ではまだモジュールの実行は可能なため問題なく新しいモジュールを展開することができる．

3. 新しいモジュールを認証する
   メモリに展開した新しいモジュールを認証モジュールにより認証する．4.2 節にて説明したプロトコルに従って新しいモジュールが正しいことを認証する．

4. 新しいモジュールを書き換える
   この段階ではモジュールは外部のグローバルな関数や変数のアドレスが分かっていない．そのため管理モジュールは対象モジュールが必要とするグローバルな関数や変数の呼び出しアドレス部分をグローバルリストのポインタ変数で書き換える．これによりモジュールは外部のグローバルな関数や変数を呼び出す際にグローバルリストのポインタ変数を呼び出すことになる．管理モジュールから受け取るのはアドレスを持つポインタ変数であるため，他のモジュールの更新によって関数や変数のアドレスが変更された場合でも，変数が持つアドレスを変更するだけでモジュールの中身を変更する必要がなくなる．またモジュールのデータ領域は管理モジュールによって割り当てられているため，更新前のモジュールが使用していたデータ領域のアドレスも受け取る必要がある．

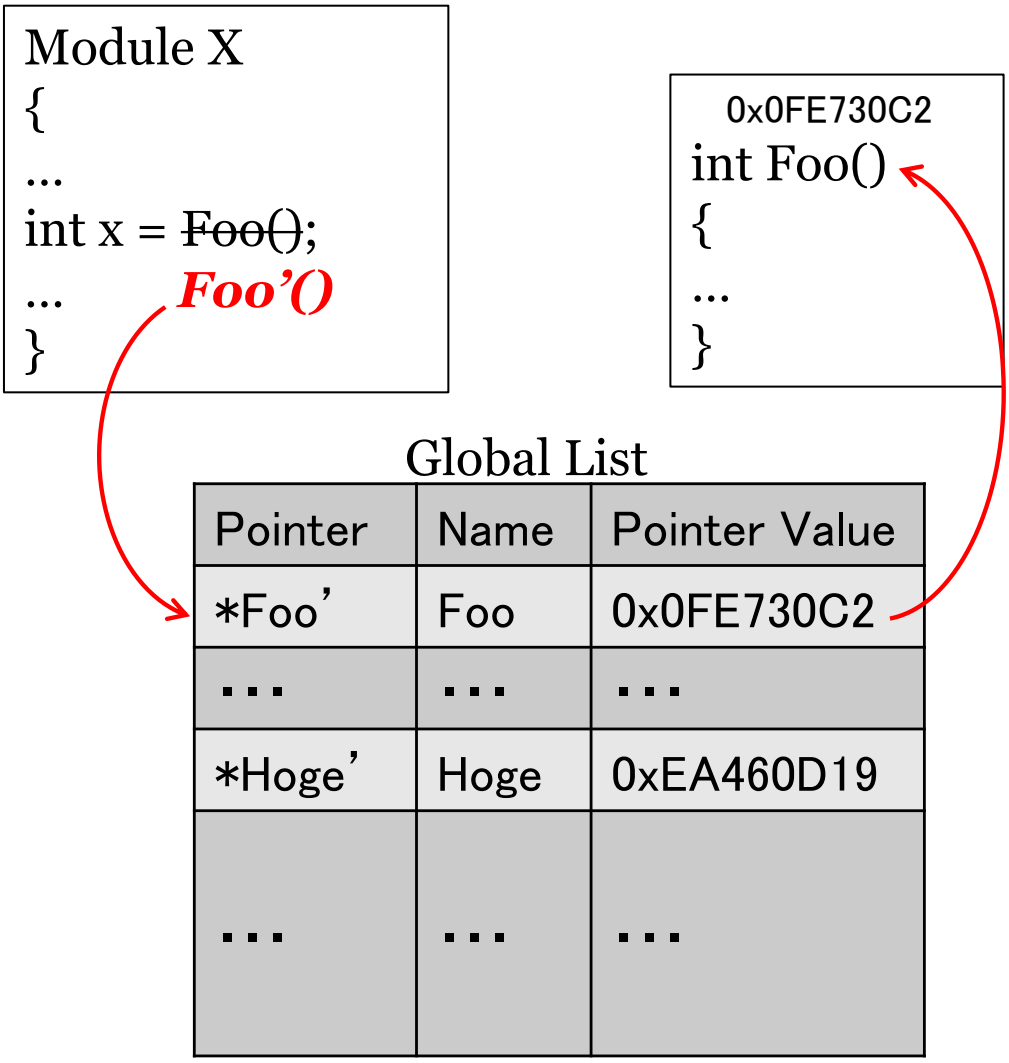


図 4.9: グローバルリストによる書き換え

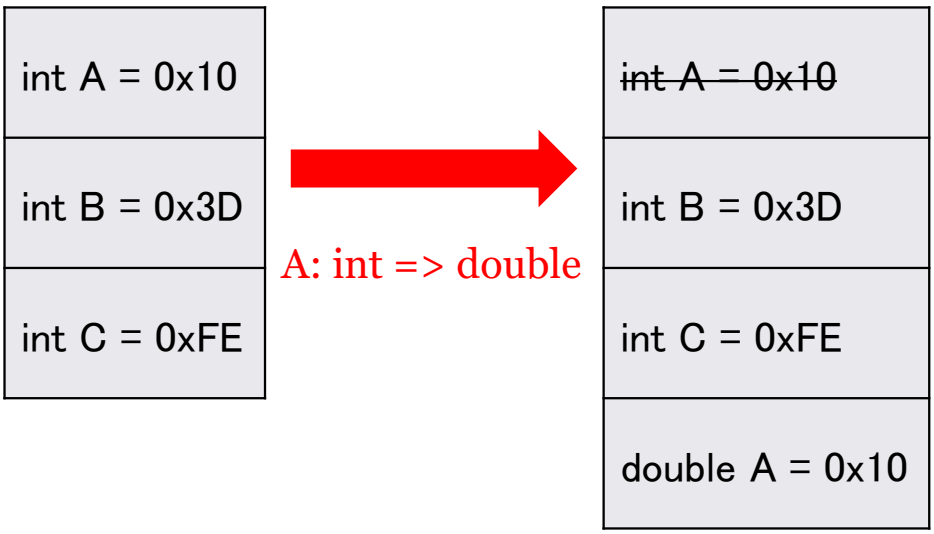


図 4.10: データ構造の変更

5. 更新関数を実行する

更新関数を実行している間，排他制御によって対象モジュールの実行を一時的に中断する必要がある．このとき，対象モジュールを使用したい仮想マシンの実行が中断されてしまうが，それ以外の仮想マシンの実行は中断されない．また基本的には更新関数は空の関数であるため，仮想マシンの実行が中断するのは「使用したい対象モジュールが更新中であり，かつ更新関数が空の関数でない」という非常に限られた条件を満たす場合のみである．そのような事態が基本的に起こらないようにモジュールを設計すればよいため，仮想マシンの実行を中断せずに更新を行うことができる．更新関数の中身については次の 6 や 7 などがある．

6. データ構造の変更への対応

Swap and Play ではハイパーバイザの内部状態を更新後のバージョンにコピーしていたが，Modular Hypervisor ではモジュールのデータ領域は移動しないためコピーをする必要はない．ただし，モジュールのデータ構造が変化している場合にはデータ領域を編集する必要がある．データ構造に新たなデータが追加されていた場合には，あらかじめ十分に確保しておいたメモリ領域に追加するだけで解決する．しかし，既存のデータの変更や削除は互換性が取れなくなる恐れが非常に高いため，そのような設計はするべきではない．どうしてもデータの変更や削除を行わなければならないときには，以前のデータ構造と整合性が取れるように新しいデータを古いデータによって初期化し，不要になったデータを削除する．例えば，図 4.10 のように整数 A の型を不動小数点型に変更するとき，データ領域の後ろに新しい A を追加し，

古い A の値を代入した後，古い A のメモリを解放する．削除による競合が起こる可能性については 3 で確認をとっているためここでは考える必要はない．また新しく追加したデータがグローバル変数であった場合，グローバルリストに登録する．

7. スタック領域を更新する
メモリのスタック領域にはローカル変数のアドレスやリターンアドレス，ベースポインタのアドレスなどが保存されている．Swap and Play による更新ではスタック領域ごと新しいメモリ領域にコピーされるためにスタックポインタやベースポインタのアドレスも更新する必要があった．しかし，Modular Hypervisor では対象モジュールの関数が終了した段階で更新関数を開始するため，スタック領域には更新前のモジュールのアドレスは基本的に存在しない．例外は更新関数を呼び出している管理モジュールを更新するときのみである．管理モジュールの更新の際にはスタック領域に更新前のモジュールを指すアドレスが存在しているため，特殊な処理が必要になる．

8. 管理モジュールのグローバルリストを更新する
管理モジュールのグローバルリストはこの段階では更新前のアドレスを指し示したままになっている．そのため，グローバルリストを新しいモジュールを指すように更新する．また，新しく追加されたグローバル関数があればグローバルリストに保存する．

9. 対象モジュールの実行が再開する
この段階で対象モジュールの更新は完了している．管理モジュールのグローバルリストは新しいモジュールを指しており，対象モジュールの関数呼び出しを再開することができる．つまり，中断していた仮想マシンの実行を再開することができるということである．

10. 不要なメモリ領域を解放する
更新前のモジュールが確保していたメモリ領域は既に不要になっているため，解放することができる．対象モジュールの実行は再開されているので，管理モジュールの更新の際にもメモリ領域を解放することが可能となっている．

以上の手順によってモジュールの更新が行われる．更新関数の実行中は対象モジュールを呼び出そうとする仮想マシンの実行を一時的に中断しなければならない．しかし，基本的には更新関数は空の関数であるため，仮想マシンの実行が中断するのは「使用したい対象モジュールが更新中であり，かつ更新関数が空の関数でない」という非常に限られた条件を満たす場合のみであり，さらにその時間も数マイクロ秒程度の仮想マシンの実行を妨げない程度の極短い時間であることが見込める．そもそも対象モジュールを呼び出さない仮想マシンの実行は全く妨げないため，実質的に仮想マシンの実行を中断せずに動的更新を実現することができていると言うことができる．

4.3.3　管理モジュールの更新方法

管理モジュールにはモジュールを更新する機能，メモリを管理する機能がある．そのため，通常のモジュールを更新するのと同じ手順で更新することはできない．管理モジュール自体を更新する際には更新関数に特殊な処理を追加する必要がある．

その更新は以下の 1〜10 の手順で行われる．

1. 競合の確認を行う
   ここでは通常のモジュール更新と同様に，競合が発生しないようにしなければならない．

2. 新しい管理モジュールをメモリ領域に展開する
   ここも通常通り新しい管理モジュールのコード領域をメモリ領域に展開する．ここで更新前の管理モジュールは，新しい管理モジュールの「更新関数を呼び出す命令の直後のアドレス」とベースポインタが指すアドレスを記憶しておく．

3. 新しいモジュールを認証する
   通常通り新しいモジュールをメモリに展開した後，認証モジュールを呼び出し認証を行う．

4. 新しい管理モジュールを書き換える
   グローバルリストは管理モジュールのデータ領域に存在するため，特別にコピーなどをする必要はない．通常の更新の通り，グローバルリストから外部のグローバルな関数や変数のポインタ変数を受け取り，新しい管理モジュールの該当部分を書き換える．また更新前の管理モジュールからデータ領域のアドレスを受け取る．

5. 更新関数を実行する
   更新関数は新しい管理モジュールの一部であり，更新前の管理モジュールから呼び出される．更新関数の実行中は管理モジュールが動作することはないが，この段階では更新関数の終了後のリターンアドレスは更新前の管理モジュールを指してしまっている．更新関数の終了後に新しい管理モジュールに実行を移すため，更新関数はスタックを改変する必要がある．

6. データ構造の変更への対応
   管理モジュールのデータ構造が変更されるようなことはないように設計するべきではあるが，変更が発生してしまったとしても通常の更新のようにデータ領域に新しいデータを追加すれば解決できる．

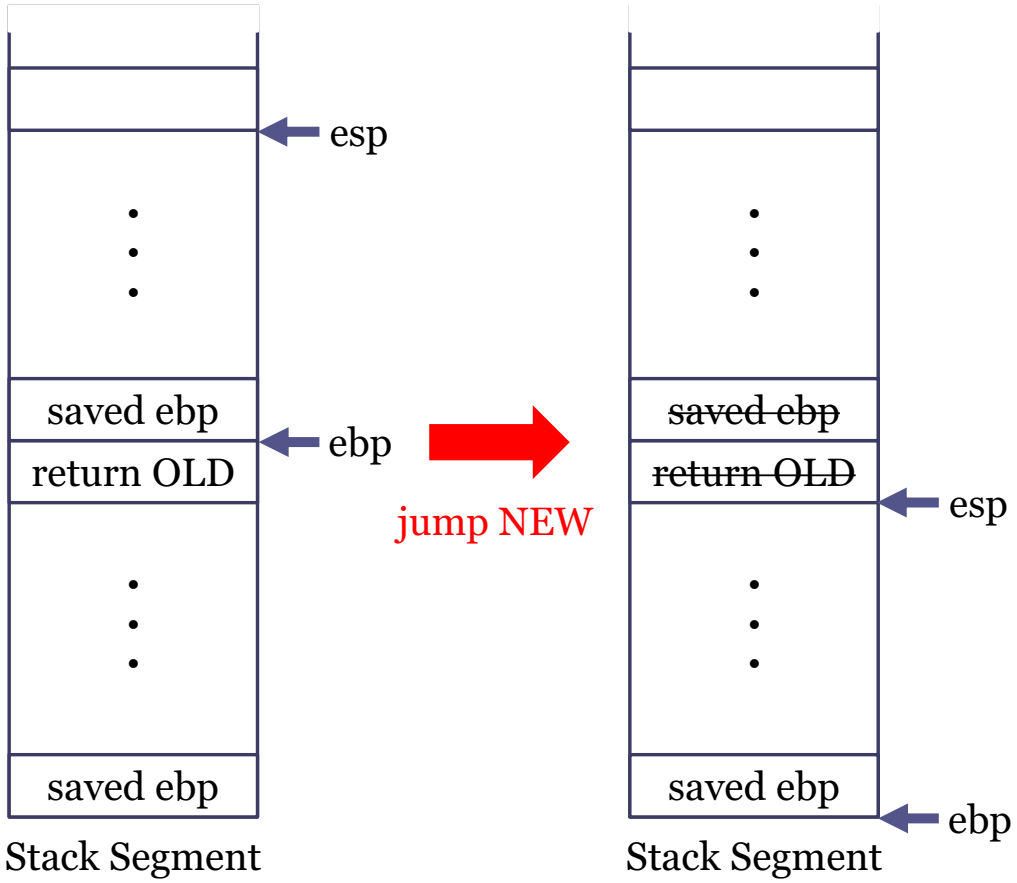


図 4.11: スタック領域の更新

7. スタック領域とスタックポインタのアドレスを更新する

この時点ではスタック領域にある更新関数からのリターンアドレスは更新前の管理モジュールを指している．更新関数の終了後に新しい管理モジュールに実行を移すためには 2 通りの方法がある．1 つ目の方法は，スタック領域のリターンアドレスを書き換える方法である．この場合には，リターンアドレスを 5 で記憶したアドレスに書き換え，更新関数が通常通り終了した後書き換えられたリターンアドレスにしたがって新しい管理モジュールに実行が移る．2 つ目の方法は，ジャンプ命令で直接ジャンプする方法である（図 4.11)．この場合には，スタックポインタとベースポインタを更新関数が終了した後の時点まで移動させる．その後リターン命令を無視しジャンプ命令によって 5 で記憶したアドレスまでジャンプする．これによってあたかも更新関数が正常に終了し新しい管理モジュールに実行が移るように見える．前者に対して後者の方が短いコードで実行が可能であるため，ここでは後者を採用する．

8. 管理モジュールのグローバルリストを更新する
   グローバルリストには管理モジュールのグローバルな関数や変数も含まれている．そのため更新関数からジャンプ命令で新しい管理モジュールに実行が移った後，当然管理モジュールのグローバルな関数や変数のアドレスもグローバルリストから更新する必要がある．

9. 管理モジュールの更新が完了する
   グローバルリストを更新した段階で管理モジュールの更新は完了しており，実行は新しい管理モジュールに移っている．

10. 不要なメモリを解放する
    更新前の管理モジュールは既に不要となっているため，通常通り更新前の管理モジュールが確保するメモリ領域を解放することができる．

以上の手順により管理モジュールの更新を行うことができる．管理モジュールの更新は更新関数の中でスタック領域を書き換えなければならないために他のモジュールと比べて少し手間がかかるが，その作業にかかる時間は結局マイクロ秒のオーダーであることが見込める．管理モジュールと他のモジュールのどちらにしろ Swap and Play のようにハイパーバイザ全体をコピーすることで更新するよりも遥かに早く，また仮想マシンをほぼ中断せずに更新することができるために動的更新の手法として優れている．

第5章　実現方法

本章では提案手法である Modular Hypervisor を実現する方法について検討する．Modular Hypervisor を適用するためのハイパーバイザとして，シングルゲスト型ハイパーバイザである BitVisor[26] を検討する．BitVisor を利用する理由としては以下の理由が挙げられる．

- シングルゲスト型ハイパーバイザである BitVisor は設計が簡潔であり，実装が容易であること．
- 機能ごとにソースファイルが細かく分割されており，モジュール化が容易であること．

本研究では BitVisor の Ver1.4，環境は Ubuntu 14.04 32bit，コンパイラは gcc version 4.8.2 を使用して BitVisor のモジュール化の検討を行った．

5.1　BitVisor

BitVisor はセキュリティの向上を目的として開発されたハイパーバイザである．ハードウェア上で直接稼動するベアメタル型ハイパーバイザであり，ゲスト OS に修正を必要としない完全仮想化されたハイパーバイザでもある．セキュア VM プロジェクトの中で開発されており，ソースファイルは BSD ライセンスで公開されている．

BitVisor では，準パススルー型というアーキテクチャを実装している．準パススルー型アーキテクチャとは，ハイパーバイザにおいてセキュリティ機能を実現するための最低限必要となるデバイスへのアクセスだけを監視・仮想化し，その他のゲスト OS のデバイスへのアクセスは可能な限りパススルーさせる方式である．ハイパーバイザで監視する必要のあるアクセスとしては，主に制御 I/O とデータ I/O の 2 種類がある．デバイスによるデータ転送を制御するための制御 I/O を監視することによりゲスト OS がハイパーバイザのメモリ領域やハードディスク領域に不正にアクセスすることを防ぎ，データ転送を行うデータ I/O を監視することによりゲスト OS を修正せずにデータの暗号化・復号を行うことができる．

前述の通り BitVisor のソースファイルは細かく分割されており，そのディレクトリ構成は表 5.1 のようになっている．BitVisor は簡潔で軽量なハイパーバイザであるため，サイズの大きい OpenSSL 関連ファイルを含む crypto ディレクトリを

表 5.1: BitVisor のソースファイル構成

ディレクトリ	内容
boot	ブートローダ
core	BitVisorコアファイル
crypto	暗号化ファイル
drivers	デバイスドライバファイル
edk	UEFIヘッダファイル
idman	ICカードドライバファイル
include	ヘッダファイル
process	保護ドメイン用ライブラリファイル
storage	ストレージ暗号化ファイル
tools	各種ツール
vpn	VPN接続ファイル

除いた全てのソースファイルの合計サイズは 5MB にも満たない．シングルゲスト型ハイパーバイザであるためにクラウドサービスにおける評価はできないが，実装の容易さからモジュール化の検討を行うには適している．特に，BitVisor は準パススルー型アーキテクチャという特徴を持っているため，デバイスへのアクセスはほとんどをゲスト OS にそのまま行わせるように設計されている．そのため，デバイスドライバファイルの大きさが一般のハイパーバイザと比べてかなり小さくなっており，デバイスドライバのモジュール化を容易に行うことができるようになっている．

5.2 BitVisor のモジュール作成

BitVisor のモジュールを作成する手順を具体例を挙げながら説明する．ここでは IEEE1394 ドライバを構成するソースファイルである driver/ieee1394.c から IEEE1394 モジュールを作成する．

ieee1394.c に記述される各関数および変数を以下で簡単に説明する．

- ieee1394_driver
  driver/pci.h ファイルで定義される pci_driver 構造体のインスタンス変数．

ieee1394_init 関数の中で pci_register_driver 関数に参照渡しされることで PCI ドライバのリストに登録される．

- ieee1394_new
- ieee1394_config_read
- ieee1394_config_write
  ieee1394_driver のメンバ関数であり，IEEE1394 デバイスが存在するときに呼び出される．
- ieee1394_init_boot
- ieee1394_init
  ドライバの初期化時に一度だけ呼ばれる関数．PCI_DRIVER_INIT および INITFUNC マクロによって initfunc_data 構造体のインスタンスのメンバに登録され，システム起動時に呼び出される．
- PCI_DRIVER_INIT(ieee1394_init_boot)
- INITFUNC(ieee1394_init)
  マクロによって定義された initfunc_data 構造体のインスタンス変数 __initfunc_ieee1394_init_boot，__initfunc_ieee1394_init を宣言する．それぞれ ieee1394_init_boot および ieee1394_init をメンバ変数に持ち，initfunc リストに登録される．その後システム起動時に initfunc に登録されたドライバの初期化を行う．

IEEE1394 モジュールの作成は以下の 1〜3 の手順で行う．

1．外部から呼び出される関数と変数の一覧を作成する
グローバルリストを作成するために管理モジュールに渡すための外部から呼び出す関数と変数の一覧を作成する．C 言語では static 修飾詞がついた関数や変数は外部から参照されないため，IEEE1394 にて外部から参照されうるのは 55 行目「ieee1394_init」関数および 69 行目「PCI_DRIVER_INIT」70 行目「INITFUNC」マクロで宣言されたインスタンス変数のみである．また今回は更新ではないので関係ないが，更新用のモジュールを作成する場合にはさらに更新関数を追加する必要がある．この更新関数は外部から参照されるため，更新関数のアドレスを管理モジュールに渡さなければならない．なお，構造体の構成や定数については，コンパイル時に依存関係がなくなるため書き換える必要はない．

2．外部に依存する関数と変数の一覧を作成する
今度は逆に外部から呼び出す関数と変数のアドレスを管理モジュールに書き換

えてもらうために，外部に依存する関数と変数の一覧を作成する．IEEE1394にて外部に依存するのは11行目「printf」関数，25行目「memcpy」関数，58行目「config」変数の3つである.「memcpy」は実態はマクロであるが，マクロにて関数が定義されているため実質関数と同じ扱いをすることになる．この3つの関数が既にメモリ上に展開されており，グローバルリストに保存されているという前提でモジュールを作成する．

3. ソースコードをコンパイルする
   ソースコードをコンパイルしてライブラリファイルを作成するのだが，管理モジュールが「ieee1394_init」関数および「PCI_DRIVER_INIT」「INITFUNC」マクロで宣言されたインスタンス変数をグローバルリストに登録し,「printf」「memcpy」「config」のアドレスをグローバルリストのポインタ変数に書き換えることができるようにするために該当部分に目印を付けておくようにコンパイラを改造しておく．管理モジュールがこのライブラリファイルを読み込むことでメモリにモジュールが配置され，目印を付けた部分が書き換えられる．

このように単純な作業によりIEEE1394モジュールを作成することができる．

さらに，C言語では実はGCCにてコンパイルしたバイナリファイルに外部に依存する関数と変数の一覧とそのアドレスが既に記述されている．そのため一覧を作成する作業は必要ない．しかし，C++ではバイナリファイルに記述される関数や変数の名前がソースコードにおける名前と異なっているため，その差分を修正する作業が必要になる．

5.3 モジュールの分割単位の検討

モジュールの分割単位は，理論的には全ての関数単位で分割するのが理想となる．なぜなら，Modular Hypervisorの目的は更新する単位を小さくするために機能を分割することだからである．しかし，そうするとモジュールの数が膨大となりモジュール間の依存関係が増えてしまう．グローバルリストのポインタ変数を経由して外部のグローバルな関数や変数にアクセスするのはモジュール内部の関数や変数に直接アクセスするよりも時間がかかるため，モジュール間の依存関係が増えると性能が低下する．また，保守の観点から見ても適当ではない．

実際に適用するハイパーバイザの設計にもよるが，基本的にはソースファイルごとに分割するのが適当である．一般にソースコードというのは関連性のある関数をまとめて記述しておくことが多く，ソースファイルごとに分割することでモジュール間の依存関係は減少することが見込める．

BitVisorにおいては，基本的にはソースファイル（.cファイル）もしくはアセンブラファイル（.sファイル）をそのまま1つの単位として分割するとよい．ただ

し，モジュール間の依存関係による性能低下を防ぐために，依存度が高くファイルについては複数のファイルをまとめて 1 つのモジュールにするべきである．例としては，core ディレクトリの VT 関連の 18 ファイルについては，1 つ 1 つのファイルの容量が小さいものが多く（表 5.2)，仮想化に関して関連性の高い関数が多く記述されている．また drivers/usb ディレクトリの USB デバイスドライバ関連の 14 ファイルなどにも各々のファイルに関連性の高い関数が多く記述されている．このように依存度の高い関数は 1 つのモジュールにまとめた方が性能の向上が見込める．

表 5.2: VT 関連のファイルサイズ

File	Size
vt.c	6.2KB
vt.h	2.8KB
vt_addip.h	1.9KB
vt_ept.c	5.2KB
vt_ept.h	2.0KB
vt_exitreason.c	4.3KB
vt_exitreason.h	1.7KB
vt_init.c	17KB
vt_init.h	1.8KB
vt_io.c	4.9KB
vt_io.h	1.9KB
vt_main.c	30KB
vt_main.h	1.8KB
vt_msr.c	11KB
vt_msr.h	2.2KB
vt_paging.c	8.2KB
vt_paging.h	2.4KB
vt_panic.c	3.3KB
vt_panic.h	1.7KB
vt_regs.c	24KB
vt_regs.h	3.0KB
vt_vmcs.h	3.8KB

第6章　おわりに

本論文では，ハイパーバイザをモジュール化し動的更新を可能にする Modular Hypervisor を提案した．ハイパーバイザの動的更新は近年クラウドコンピューティングサービスの利用が進んできている社会にとってセキュリティを守るための非常に有益な技術であり，既存手法では Xen4.2.0 から 4.2.1 への更新を約 45 ミリ秒で行っている．提案手法による動的更新は基本的に仮想マシンの実行を中断せず，例外的に中断する場合も中断時間はマイクロ秒のオーダーで更新できることが見込め，既存手法よりも優位であることを示した．さらにモジュールの作成方法を検討するために，BitVisor のドライバファイルのモジュール化をテストし，モジュールの分割単位の検討を行った．その結果，一般のモジュールの作成は単純な作業で可能であり，関数間の依存度が高いファイルは 1 つのモジュールにまとめた方が性能の向上が見込めることが分かった．

今後の課題としては，関数の依存関係を考慮してモジュール分割単位を最適化する手法を考案すること，実際にコアや管理モジュールを作成して BitVisor に適用しモジュール間の依存関係による性能低下を評価すること，規模の大きいマルチテナント型ハイパーバイザへ適用しクラウドサービスにおける性能評価を行うことなどが考えられる．

参考文献

[1] Ahmad-Reza Sadeghi Franz Ferdinand Brasser, Mihai Bucicoiu. Swap and play: Live updating hypervisors and its application to xen. In *CCSW 2014: The ACM Cloud Computing Security Workshop Proceedings*, 2014.

[2] A. Vasudevan, S. Chaki, Limin Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 430–444, 2013.

[3] Amazon Web Services. 国内のお客様の導入事例 Powered by AWS クラウド. https://aws.amazon.com/jp/solutions/case-studies-jp/.

[4] Amazon ec2. http://aws.amazon.com/jp/ec2/.

[5] 「2014 年度情報セキュリティ事象被害状況調査」報告書について. http://www.ipa.go.jp/security/fy26/reports/isec-survey/.

[6] The Xen Project, the powerful open source industry standard for virtualization. http://www.xenproject.org/.

[7] Hyper-V | マイクロソフト サーバー & クラウド プラットフォーム. http://www.microsoft.com/ja-jp/server-cloud/windows-server/hyper-v.aspx.

[8] Free VMware vSphere Hypervisor, Free Virtualization (ESXi). http://www.vmware.com/products/vsphere-hypervisor/.

[9] Amazon EC2 Service Level Agreement. http://aws.amazon.com/ec2/sla/.

[10] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pp. 133–145, New York, NY, USA, 2006.

[11] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pp. 32–32, Berkeley, CA, USA, 2005.

[12] M.E. Segal and O. Frieder. On-the-fly program modification: systems for dynamic updating. *Software, IEEE*, Vol. 10, No. 2, pp. 53–65, 1993.

[13] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pp. 187–198, New York, NY, USA, 2009.

[14] Introducing kpatch : Dynamic Kernel Patching. http://rhelblog.redhat.com/2014/02/26/kpatch/.

[15] kGraft | SUSE. https://www.suse.com/promo/kgraft.html.

[16] The Linux Kernel Archives. https://www.kernel.org/.

[17] Hotfix Your Ubuntu Kernels with the Canonical Livepatch Service! . http://blog.dustinkirkland.com/2016/10/canonical-livepatch.html.

[18] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, Vol. 39, No. 9, pp. 70–77, 1996.

[19] Christoph M. Kirsch, Marco A. A. Sanvido, and Thomas A. Henzinger. A programmable microkernel for real-time systems. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pp. 35–45, New York, NY, USA, 2005.

[20] eXtensible Modular Hypervisor Framework | SourceForge.net. http://sourceforge.net/projects/xmhf/.

[21] J.M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 143–158, 2010.

[22] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pp. 34–54, Berlin, Heidelberg, 2012.

[23] Trusted Computing Group. Trusted Platform Module (TPM) Summary. https://www.trustedcomputinggroup.org/files/resource_files/4B55C6B9-1D09-3519-AD916F3031BCB586/TrustedPlatformModuleSummary_04292008.pdf.

[24] 岡本拓也, 山口利恵, 五島正裕, 坂井修一. Tpm を用いたハイパーバイザの完全性検証手法の実装. 情報科学技術フォーラム講演論文集, Vol. 13, No. 4, pp. 141–148, 2014.

[25] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. *A Practical Guide to Trusted Computing*. First edition, 2007.

[26] 保理江高志, 榮樂英樹, 品川高廣, 加藤和彦. 仮想マシンモニタ bitvisor と i/o 仮想化技術. 情報処理学会研究報告システムソフトウェアとオペレーティング・システム（OS）, Vol. 2009, No. 6, pp. 35–41, 2009.

謝辞

本研究を進めるにあたって，坂井修一教授には全般的なご指導をいただくとともに，的確なアドバイスを数多くいただきました．

入江英嗣准教授には研究内容の詳細に渡るアドバイスをいただきました．また，大学院博士課程の宮永瑞紀さんには，実際のハイパーバイザの構成法や研究方針の相談など，研究現場に近い場面で多くの貴重なご意見をいただきました．秘書の八木原晴水さん，赤羽彩子さんには事務手続きや機材の利用など様々なことで助けていただきました．

また研究室の皆さんにも多くのアドバイスやサポートをしていただき，研究の励みになりました．心より感謝申し上げます．

本研究の一部は，公益財団法人セコム科学技術振興財団の助成を受けています．