修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　総合情報学専攻　博士前期課程		
氏　　名	河﨑 雄大	学籍番号	1230025
論 文 題 目	BitVisor のための OS の状態復元機能		

要　　旨

マルウェアによる脅威が多く発見されており，それに対する対抗策として，アンチウイルスソフトウェア等のセキュリティシステムによって OS のセキュリティを高める手法が一般的となっている．しかし，セキュリティシステム自体を無効化するマルウェアも存在し，OS 上での対策は限界がある．この問題を解決方法として，仮想マシンモニタ (VMM) を用いてセキュリティ処理を施す方法が存在する．

VMM を用いてセキュリティ処理を施す方法では，仮想マシン (VM) 上で OS を動かし，VMM 層で OS の挙動を解析して OS のセキュリティを高める．この方法を用いたシステムは，元々は OS のセキュリティを高めるために作られた一般ユーザが対象のシステムであったが，VM 上でマルウェアと思われるプログラムを実際に動かし，その挙動を監視する VM 上でのマルウェアの動的解析に使用するというマルウェア解析者が対象のシステムにも応用でき，研究されている．

マルウェアの動的解析に利用する場合，実環境とはかけ離れた環境の場合に動作を止めるマルウェアも存在するため，マルウェアの動的解析に用いる VMM はより実環境に近い環境であることが望ましい．より実環境に近い環境を提供する VMMとしては，BitVisor がある．BitVisor は実環境に近い環境を提供しつつ，デバイスへの I/O を監視できる機能をもつ．しかし，BitVisor はマルウェアの動的解析をするのに相応しい環境を提供しているが，マルウェアによって壊された環境を元の状態に戻す機能は提供していない．

そこで本研究では，BitVisor に対して OS の状態をチェックポイントとして保存，復元できる機能を提案する．チェックポイントとして保存，復元するものはディスク内にあるデータであるディスクデータとメモリ上にあるデータであるメモリデータである．また，OS の状態の保存，復元の適切なタイミングは，使用するユーザが一番知っていると考え，OS の状態の保存，復元のトリガーは，任意のタイミングで OS のユーザレベルから引くことができるといった手法を用いる．

我々はこの機能を BitVisor に実装してベンチマークによる実行時間のオーバヘッドを測定，評価し，提案システム導入後のオーバヘッドが実用に耐えられるレベルであることを確認した．

平成 25 年度修士論文

BitVisor のための OS の状態復元機能

電気通信大学大学院 情報理工学研究科
総合情報学専攻 セキュリティ情報学コース

学籍番号　　: 1230025
氏名　　: 河崎 雄大
主任指導教員 : 大山 恵弘 准教授　印
指導教員　　: 高田 哲司 准教授　印
提出日　　: 平成 26 年 1 月 30 日

要旨

マルウェアによる脅威が多く発見されており，それに対する対抗策として，アンチウイルスソフトウェア等のセキュリティシステムによって OS のセキュリティを高める手法が一般的となっている．しかし，セキュリティシステム自体を無効化するマルウェアも存在し，OS 上での対策は限界がある．この問題を解決方法として，仮想マシンモニタ (VMM) を用いてセキュリティ処理を施す方法が存在する．

VMM を用いてセキュリティ処理を施す方法では，仮想マシン (VM) 上で OS を動かし，VMM 層で OS の挙動を解析して OS のセキュリティを高める．この方法を用いたシステムは，元々は OS のセキュリティを高めるために作られた一般ユーザが対象のシステムであったが，VM 上でマルウェアと思われるプログラムを実際に動かし，その挙動を監視する VM 上でのマルウェアの動的解析に使用するというマルウェア解析者が対象のシステムにも応用でき，研究されている．

マルウェアの動的解析に利用する場合，実環境とはかけ離れた環境の場合に動作を止めるマルウェアも存在するため，マルウェアの動的解析に用いる VMM はより実環境に近い環境であることが望ましい．より実環境に近い環境を提供する VMM としては，BitVisor がある．BitVisor は実環境に近い環境を提供しつつ，デバイスへの I/O を監視できる機能をもつ．しかし，BitVisor はマルウェアの動的解析をするのに相応しい環境を提供しているが，マルウェアによって壊された環境を元の状態に戻す機能は提供していない．

そこで本研究では，BitVisor に対して OS の状態をチェックポイントとして保存，復元できる機能を提案する．チェックポイントとして保存，復元するものはディスク内にあるデータであるディスクデータとメモリ上にあるデータであるメモリデータである．また，OS の状態の保存，復元の適切なタイミングは，使用するユーザが一番知っていると考え，OS の状態の保存，復元のトリガーは，任意のタイミングで OS のユーザレベルから引くことができるといった手法を用いる．

我々はこの機能を BitVisor に実装してベンチマークによる実行時間のオーバヘッドを測定，評価し，提案システム導入後のオーバヘッドが実用に耐えられるレベルであることを確認した．

目次

1 はじめに

1.1 背景

マルウェアなど，多くのセキュリティ脅威が発見されている．セキュリティ脅威の多くはユーザの不注意をついたり，プログラムの脆弱性をついたりしてシステムを攻撃し，多くの被害が確認されている．このようなセキュリティ脅威による被害を防ぐ方法としては，OS 上のアンチウイルスソフトウェアなどのセキュリティシステムを用いて対策を施すのが一般的な方法となっている．

しかし，この方法を用いているとセキュリティシステムが動いている OS が攻撃され，乗っ取られてしまうとセキュリティシステム自体が無効化されてしまうといった問題点が存在する．このような，セキュリティシステム自体を無効化するマルウェアは古くから存在しており，CONFICKER [1] や BKDR_SRAOW.A [2] と呼ばれるものなどが存在する．この問題を解決方法として，仮想マシンモニタ (VMM) を用いてセキュリティ処理を施す方法や TPM [3] のようなハードウェアを用いてセキュリティ処理を施す方法が存在する．TPM を用いる方法では，専用のハードウェアが必要となってくるのでユーザの敷居が高い．故に，本研究では，仮想マシンモニタ (VMM) を用いる方法に注目した．

仮想マシンモニタ (VMM) とは，物理ハードウェアなどの計算機資源を仮想化し，管理，実行することで仮想マシン (VM) を提供するソフトウェアである．提供された VM 上では，オペレーティングシステム (OS) などのプログラムを実行することができる．

VMM には複数の種類があるが，種類問わず共通して，VMM は VM よりも高い権限で動作していることが多い．つまり，VMM は VM からは干渉ができない隔離された環境であり，OS が攻撃され，VM が乗っ取られてしまっても VMM まで乗っ取ることは難しい．したがって，VMM でセキュリティ処理を施すのは有効な対策となりうる．

VMM でセキュリティ処理を施す方法では，仮想マシン (VM) 上で OS を動かし，VMM 内でデバイスへの I/O を監視したりマルウェアなどの攻撃から防御したりして VM 上の OS のセキュリティを高める．VM 上の OS が乗っ取られてしまっても VMM が乗っ取られてしまう可能性は低いため，デバイスへの I/O の監視などの処理は継続してできるといった利点がある．この方法に基づくシステムとしては，Livewire [4] や VMwatcher [5]，BVMD [6] がある．

Livewire は，攻撃やマルウェアを VMM 層で検知するシステムであり，OS 上のデータの中から特定の文字列を検出する機能も有している．Livewire はホスト OS 上で動く

VMM である VMware Workstation [7] を改造して実装されている．

VMwatcher は，VM 上で動作する OS のためのアンチマルウェア処理を VM の外で施したシステムであり，ホスト OS や管理 VM など，対象 VM の外で動くソフトウェアによって実現されている．VMwatcher は VMware Server [7]，QEMU [8]，Xen [9]，User-Mode Linux [10] の 4 種類の VMM をサポートして実装されている．

BVMD は，BitVisor [11] に対して機能拡張を行い，読み書きされるストレージデータに対してマルウェアの検出を行う機能を備えたシステムである．ゲスト OS とディスクの間を流れるディスク I/O を捕捉し，シグネチャマッチングを行ってマルウェアを検出している．BVMD は BitVisor の準パススルードライバを拡張して実装されている．

このように，VMM でセキュリティ処理を施すシステムは多く存在する．しかし，VMM でセキュリティ処理を施すシステム以外にも，VMM でマルウェアの動的解析を行うシステムというものも存在する．マルウェアの動的解析とは，マルウェアの挙動を解析するために実際にマルウェアと思われるプログラムを実際に動かし，その挙動を VMM で監視するものである．このマルウェアの動的解析では，セキュリティ的に隔離された環境が望ましく前節で述べたような VMM でセキュリティ処理を施すシステムが重宝されている．

しかし，マルウェアの技術はどんどん進歩しており，セキュリティシステムからの検知を逃れるために様々な工夫を凝らしたものも発見されている．具体例として，AGOBOT [12] や SDBOT [13] のような，実環境とはかけ離れた環境で動いていることを検知すると自分自身の動作を止め，セキュリティシステムからの検知を防ぐマルウェアも発見されている．それ故，VMM を用いたマルウェアの動的解析ではより実環境に近い環境が求められる．より実環境に近い環境を提供する VMM としては BitVisor というものがある．

1.2 目的

BitVisor は実環境により近い環境，かつ，セキュリティ的に隔離された環境を提供することができる．また，BitVisor はデバイスへの I/O を監視できる機能ももつ．このように，BitVisor はマルウェアの動的解析をするのに相応しい環境を提供しているが，マルウェアによって壊された環境を元の状態に戻す機能は提供していない．

そこで，我々の研究では BitVisor のための OS の状態復元機能を提案する．保存対象は，ディスク内のデータであるディスクデータとメモリ上のデータであるメモリデータとする．以降，本論文では保存した OS の状態のことをチェックポイントと呼ぶ．また，保存，復元の適切なタイミングは，使用するユーザが一番知っていること考え，保存，復元はユーザ任意のタイミングでできるものとする．また，提案システムでは保存できる

チェックポイントは 1 つのみとし，チェックポイントは VMM が管理するものとする．

提案システムでは，BitVisor はゲスト OS よりも高い権限で動いるため，BitVisor 自身がマルウェアによって攻撃を受けることはないと考えている．したがって，BitVisor が提案システムの Root of Trust である．

1.3 論文の構成

本論文の構成は以下の通りである．2 章では仮想マシンモニタの詳細について述べ，3 章では提案システムの概要について述べる．4 章ではシステムの設計および実装について述べ，5 章ではシステム導入によるオーバヘッドの実験および評価について述べる．6 章では，提案システムに対しての議論を述べ，7 章では関連研究と本研究との比較について述べる．8 章ではまとめと今後の課題を述べる．

2　仮想マシンモニタ (VMM)

2.1　仮想マシンモニタ (VMM) の概要

前章でも軽く述べたが，仮想マシンモニタ (VMM) とは，物理ハードウェアなどの計算機資源を仮想化し，管理，実行することで仮想マシン (VM) を提供するソフトウェアである．VMM の中には 1 つの VMM 上で複数の VM を同時に動作させることが可能なものも存在し，VMM を用いれば 1 つの物理ハードウェア上に複数の OS を同時に動作させることも可能である．

VMM の分類を図 1 に示す．VMM は動作形態により 3 種類に分けられ，それぞれ Type-I，Type-II，Hybrid と呼ばれる．以下，本論文では，VM 上で動作している OS をゲスト OS，VMM が実行される OS をホスト OS と呼ぶ．

Type-I VMM は，ハードウェア上で直接 VMM を動作する VMM である．利点としては，ホスト OS を経由する必要なくハードウェアを直接操作するため，パフォーマンスの低下を抑え，より実環境に近い環境を提供することができる点がある．しかし，ホスト OS が存在しないため，各デバイスの管理やファイルシステムの利用といったホスト OS の機能を利用することができない．そのため，実装が複雑，困難になるといった欠点も存在する．Type-I VMM に分類される代表的な VMM には，BitVisor や Xen などがある．

Type-II VMM は，ハードウェア上でホスト OS が動作し，ホスト OS 上でユーザプロセスとして動作する VMM である．利点としては，ホスト OS が存在するため，スケ

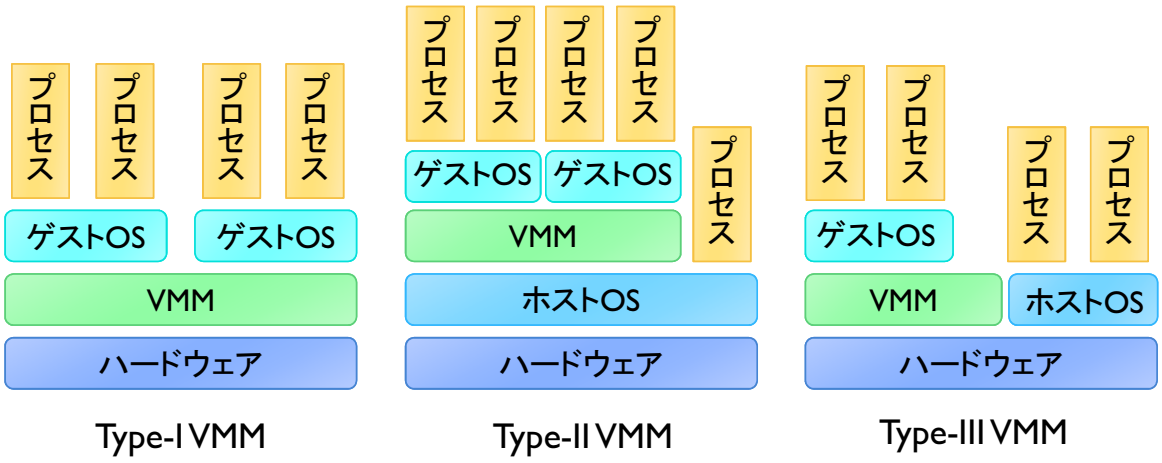

図 1　VMM の分類

ジューリングや資源管理に関して，ホスト OS の機能を利用することができる点がある．しかし，VMM がホスト OS 上のプロセスとして動いており，各 VM の資源が他のプロセスから参照されたり，改変されたりする可能性がある．そのため，セキュリティの面では Type-I VMM に劣るといった欠点も存在する．Type-II VMM に分類される代表的な VMM には，User-Mode Linux などがある．

Hybrid VMM は，Type-I VMM と Type-II VMM の特徴を併せ持った VMM であり，ホスト OS と VMM がハードウェア上で並列に動作する VMM である．利点としては，ホスト OS の機能を利用しつつ，VMM はカーネルモードで実行されるため，オーバヘッドは Type-II よりは軽減できる点がある．しかし，並行して動作するホスト OS も CPU サイクルなどの資源を必要とするので，Type-I と比べるとオーバヘッドは大きくなるといった欠点も存在する．Hybrid VMM に分類される代表的な VMM には，KVM [14] や VMware Workstation などがある．

2.2 BitVisor

BitVisor は，筑波大学などによって開発された，ハードウェア上で直接動作するセキュリティ向上のための Type-I VMM である．BitVisor はセキュリティ処理を行う上で最低限な部分のみを捕捉する準パススルーアーキテクチャを用いて作られた VMM であり，実行時間のオーバヘッドや Trusted Computing Base (TCB) を小さく抑えた設計となっている．最新のバージョンは 1.3 である．

準パススルーアーキテクチャとは，ゲスト OS からハードウェアへのアクセスを可能な限りパススルー (通過) させつつ，セキュリティ機能の実現のために最低限必要なアクセスのみを VMM が捕捉する方式である．BitVisor は，一部のアクセスのみを捕捉することによって，ストレージやネットワークの暗号化などのセキュリティ機能を実現している．また，BitVisor は Intel Virtualization Technology (Intel VT) [15] や AMD Virtualization (AMD-V) 等の CPU の仮想化支援機構を用いて実装されている．

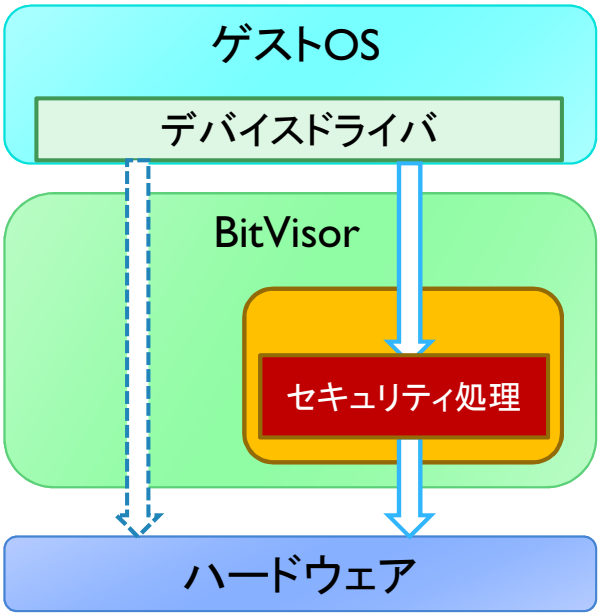

図 2　BitVisor の構成

BitVisor の構成を図 2 に示す．図中の実線矢印は BitVisor に捕捉される I/O 処理を表し，点線矢印は捕捉されない I/O 処理を表している．BitVisor では，準パススルードライバでデバイスの制御やゲスト OS が発行する I/O を捕捉をし，データを取得，更新することによって，データの暗号化などのセキュリティを高める処理を実現している．

このような設計から，BitVisor はマルウェアが VMM 上で動作していると判断できないくらい実環境に近い環境は提供しつつ，I/O を捕捉しているので，VMM を検知するマルウェアの動的解析を可能にしつつ，セキュリティ的に隔離された環境を提供することができる．

3 提案システム

3.1 BitVisor を用いる理由

本研究では目的を実現するため，VMM の中でも BitVisor に注目したが，注目した理由としては，マルウェアの動的解析の際はより精度の高い解析ができるという利点が存在するからである．すでに述べたように，BitVisor はオーバヘッドをできるだけ小さく抑える設計となっている．また，デバイスは極力仮想化しない設計となっており，デバイスの多くは直接ゲスト OS によって処理される．それ故に，BitVisor 上では実環境により近い環境でマルウェアを動かすことができるので，マルウェアの動的解析の際はより精度の高い解析ができる．この利点はマルウェアの動的解析に BitVisor を使う大きな利点である．

3.2 状態復元機能の利用例

ここでは，本研究で提案する状態復元機能をマルウェアの動的解析に用いる例について述べる．本研究で提案する OS の状態保存，復元機能のトリガーは全てユーザによって引かれる．したがって，今回の例の場合では，マルウェアを実際に実行させて挙動を観察する直前に保存機能のトリガーを引き，マルウェアの挙動の観察が終了した直後に復元機能のトリガーを引くことを想定している．

本研究で提案する状態復元機能をマルウェアの動的解析に用いる例を図 3 に示す．提案システムを用いてマルウェアの動的解析を行う場合は，ユーザは，まず，VM 上でゲスト OS を起動する．次に，現在の OS の状態を表すチェックポイントを作成するため，ユーザはゲスト OS 上で VMM を呼び出す特別なプログラムを実行する．このプログラムが実行されると，OS の状態保存機能のトリガーが引かれ，VMM によって現在の OS の状態がチェックポイントとして保存される．次に，VMM によるチェックポイントの作成が終わった後，ユーザはゲスト OS 上で実際にマルウェアと思われるプログラムを実行し，その挙動を観察，解析する．ここで，マルウェアと思われるプログラムを実行すると OS の状態は壊される．また，挙動の観察，解析は複数回に渡ることが多いので，挙動の観察，解析が終わった後の素早い OS の状態の復元が望ましい．そこで，挙動の観察，解析が終わった後，素早く OS の状態を復元するため，ユーザはチェックポイント作成時と同様にゲスト OS 上で VMM を呼び出す特別なプログラムを実行する．すると，今度は OS の状態復元機能のトリガーが引かれ，VMM によって OS の状態がチェックポイントの状態へと復元される．OS の状態が復元された後，ユーザは新たにチェックポイントを作成し，

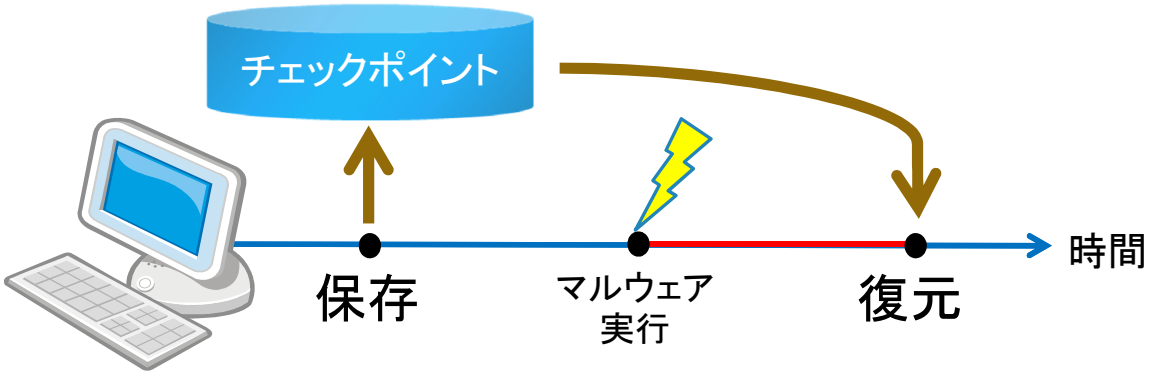

図 3　状態復元機能をマルウェアの動的解析に用いる例

再びマルウェアと思われるプログラムの挙動の観察，解析を続けていく．

4　システムの設計・実装

4.1　システムの概要

提案システムでは，チェックポイントとして保存，復元するデータはメモリデータとディスクデータであり，保存できるチェックポイントは 1 つのみとする．また，提案システムはゲスト OS の CPU はシングルコアを想定している．

ここで，BitVisor ではデバイスは極力仮想化しない設計となっているため，ディスクは仮想化されていない．したがって，BitVisor でこれらの保存，復元機能を実装するにはデータを保存する場所がない，実デバイスを扱わないといけないなどの問題点が存在し，BitVisor でこれらの機能を実現するのは単純なことではないということがわかる．そこで，提案システムでは次のような方法でこれらの機能を実現していく．

4.2　メモリデータの保存，復元方法

まず，本研究で提案するメモリデータの保存，復元方法を述べる前に物理メモリのマップの情報を取得する手法を述べる．物理メモリの中には，BIOS によって予約済みにされ，使えない領域が存在し，その領域はマシン毎に異なる．したがって，メモリデータを保存する際は，その領域を特定する必要があり，その領域を特定するために物理メモリのマップの情報を取得し，使用する．物理メモリのマップの情報を取得するため，提案システムでは，BIOS 命令である `e820` 命令を使用する．`e820` 命令で得られた物理メモリのマップ情報の例を図 4 に示す．

しかし，BIOS 命令は最近の OS の多くが起動後に用いているモードであるプロテクトモードなどでは使用することができない．したがって，提案システムでは，BitVisor が起動時に発行した `e820` 命令で得られたメモリマップを用いて，物理メモリのマップ情報を得ている．また，`e820` 命令で得られた物理メモリのマップ情報において，物理メモリのタイプは `usable`，`reserved`，`ACPI data`，`ACPI NVS` の 4 つが存在するが，提案システムでは，該当する範囲の物理メモリの内，`reserved` でない部分のメモリをメモリデータとして保存する．

次に，本研究で提案するメモリデータの保存方法を図 5 に示す．提案システムでは，メモリデータの保存は，ゲスト OS を実行している VM の仮想物理メモリの先頭から末尾のことを指すゲスト OS の状態を構成するデータと保存機能実行時のゲスト OS 用 CPU のレジスタの値をチェックポイントとして保存することで実現する．ここで，前に述べた

```
0-9d7ff,           usable     da6de000-dadcefff,   usable
9d800-9ffff,       reserved   dadcf000-dafdcfff,   reserved
e0000-fffff,       reserved   dafdd000-daffffff,   usable
100000-1ffffffff,  usable     100000000-21e5fffff, usable
20000000-201fffff, reserved   db800000-df9fffff,   reserved
20200000-3fffffff, usable     f8000000-fbffffff,   reserved
40000000-401fffff, reserved   fec00000-fec00fff,   reserved
40200000-d9cf7fff, usable     fed00000-fed03fff,   reserved
d9cf8000-da415fff, reserved   fed1c000-fed1ffff,   reserved
da416000-da695fff, ACPI NVS   fee00000-fee00fff,   reserved
da696000-da69afff, ACPI data  ff000000-ffffffff,   reserved
da69b000-da6ddfff, ACPI NVS
```

図 4　e820 命令で得られた物理メモリのマップ情報の例

通り BitVisor はディスクの仮想化をしていなかったり，ホスト OS が存在せず，ファイルシステムの利用といったホスト OS の機能を利用することができなかったりする．したがって，BitVisor ではチェックポイントをファイルの形式で保存することができないという問題点が存在する．

そこで，提案システムでは，メモリ上のゲスト OS の状態を構成するデータとゲスト OS のレジスタの値をチェックポイントとしてメモリ上の空いている領域に保存する．この空いている領域とは VMM もゲスト OS も使っていない部分である．また，ゲスト OS のレジスタの値とは，Intel VT-x での VMCS 内の GUEST-STATE AREA に格納されている値と RBP レジスタのことを指している．

最後に，本研究で提案するメモリデータの復元方法を図 6 に示す．メモリデータの復元は，ゲスト OS の状態を構成するデータと保存機能実行時のゲスト OS 用 CPU のレジスタの値が格納されているデータ構造体をメモリ上の保護された領域に保存されているチェックポイントを用いて元の状態に戻すことによって実現する．ここで，ゲスト OS の状態を構成するデータの復元に関しては，単にチェックポイントとして保存されているデータで上書きすることで実現し，保存機能実行時のゲスト OS 用 CPU のレジスタの値が格納されているデータ構造体の復元に関しては，`VMWRITE` のような CPU 命令を用いて

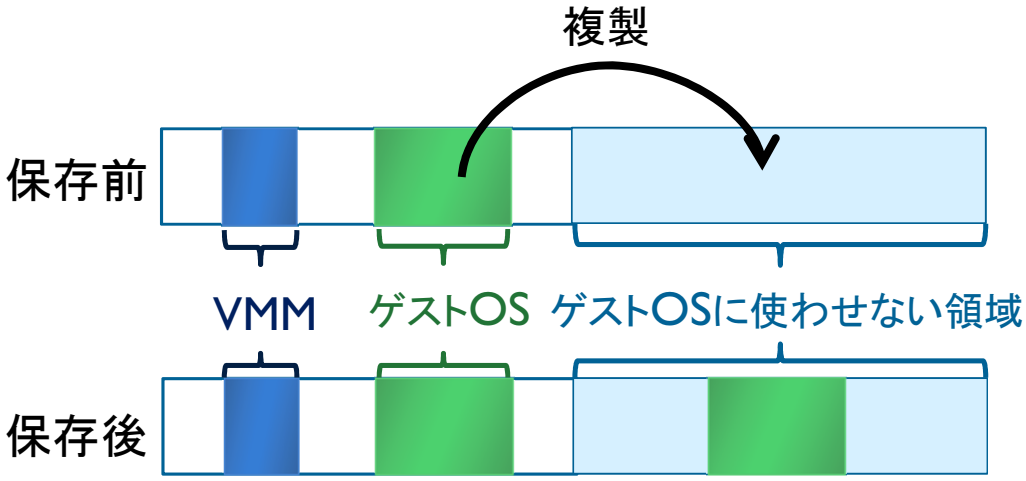

図 5　メモリデータの保存方法

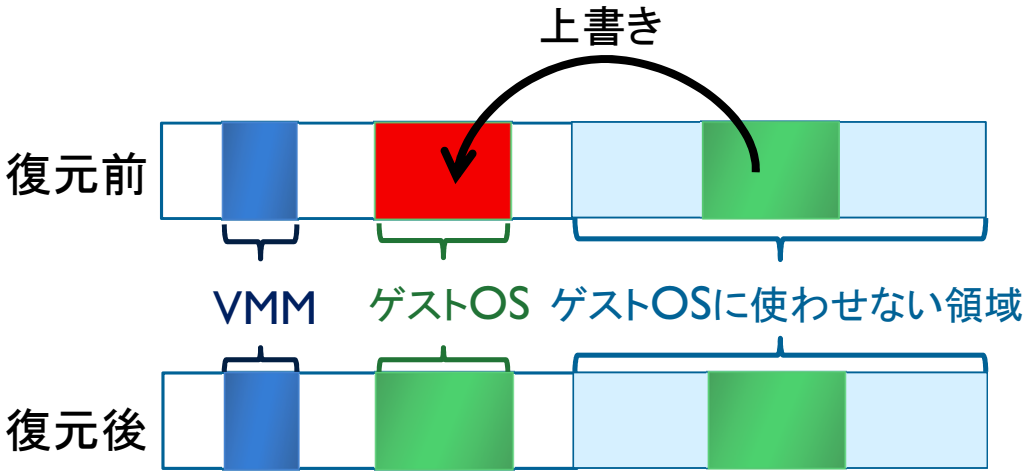

図 6　メモリデータの復元方法

1 つずつ書き換えることで実現する.

4.3 ディスクデータの保存，復元方法

前に述べた通り BitVisor ではチェックポイントをファイルの形式で保存することができないという問題点が存在する．そこで，提案システムでは，ディスクデータの保存，復元に関しては，ディスク内にメインパーティションと差分パーティションの 2 つのパーティションを用い，アクセス先のパーティションを変えることで実現する．ディスクデータを保存，復元する部分のシステムの概要を図 7 に示す．ここで，メインパーティションとは OS がインストールされた一般的なストレージのことであり，差分パーティションとはディスクデータの保存後にディスクに書き込まれるデータを一時的に保存するためのストレージである．

ディスクデータの保存，復元時の処理の詳細について述べる．ディスクデータの保存時の処理に関しては，ディスクへのアクセス先のパーティションをメインパーティションから差分パーティションに変えることで実現する．ディスクデータの復元時の処理に関しては，ディスクへのアクセス先のパーティションを差分パーティションからメインパーティ

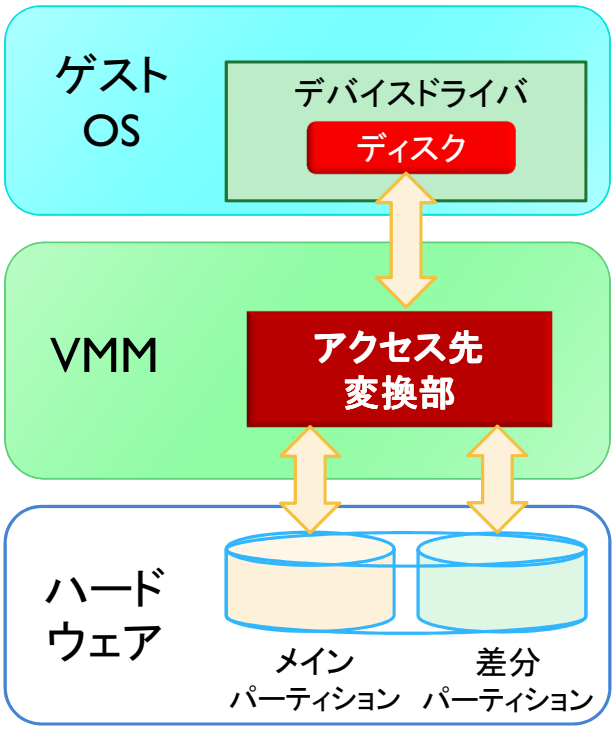

図 7　ディスクデータを保存，復元する部分のシステムの概要

ションに戻し，差分パーティションに書きこまれた書き込みを全て破棄することで実現する．なお，ディスクへのアクセス先の変更は次に述べるアクセス先変換部を用いて実現する．

4.3.1 アクセス先変換部

アクセス先変換部とはディスクアクセス際のアクセス先をバックグラウンドで変更する機構である．この機能を使い，差分パーティションとメインパーティションの切り替えを行う．提案システムでは，ゲスト OS とディスクデバイスの間を流れるディスク I/O を捕捉，変更することでアクセス先変換部を実現する．

アクセス先変換部の実装例を図 8 に示す．提案システムでは，ディスクへは LBA (Logical Block Addressing) でアクセスする環境を想定しており，アクセス先変換部ではディスク I/O の LBA 番号を変更することでアクセス先を変更する．また，提案システムでは，差分パーティションはメインパーティションと同等のサイズのパーティションを想

```
void change_dst(struct ahci_port *port, int cmdhdr_index, int rw) {
  union cmdfis *cfis;
  if(ret_another_dst_flag()) {
    u64 lba = port->my[cmdhdr_index].dmabuf_lba;
    u64 count = port->my[cmdhdr_index].dmabuf_nsec;
    lba = rw ? pre_write(lba, count) : pre_read(lba, count);
    port->my[cmdhdr_index].dmabuf_lba = lba;
    cfis = &port->my[cmdhdr_index].cmdtbl->cfis;
    cfis->fis_0x27.sector_number = (lba >> 0) & 0xff;
    cfis->fis_0x27.cyl_low = (lba >> 8) & 0xff;
    cfis->fis_0x27.cyl_high = (lba >> 16) & 0xff;
    cfis->fis_0x27.sector_number_exp = (lba >> 24) & 0xff;
    cfis->fis_0x27.cyl_low_exp = (lba >> 32) & 0xff;
    cfis->fis_0x27.cyl_high_exp = (lba >> 40) & 0xff;
  }
}
```

図 8　アクセス先変換部の実装例

LAB 番号	ダーティフラグ
0x120000	0
⋮	
0x12ffff	0
0x130000	1
⋮	
0x13ffff	1
0x140000	0
⋮	
0x14ffff	0

図 9　アクセス先管理テーブルの例

定しており，差分パーティションの LBA 番号はメインパーティションの LBA 番号に定数 A を足したものと定義する．ここで，この定数 A は差分パーティションの先頭の LBA 番号からメインパーティションの先頭の LBA 番号を引いたものと定義する．

各状態のディスクアクセスの詳細は先で述べるが，ディスクデータの保存後から復元までの間のディスクアクセスを管理するため，アクセス先変換部ではアクセス先管理テーブルというテーブルを用いる．

4.3.2　アクセス先管理テーブル

アクセス先管理テーブルとはディスクデータの保存後から復元までの間のディスクアクセスの整合性を保つための管理テーブルである．アクセス先管理テーブルは LBA 番号とダーティフラグの 2 つの要素から成り立つ．また，アクセス先管理テーブルでは LBA 番号はセクタサイズ毎に存在する．アクセス先管理テーブルの例を図 9 に示す．提案システムでは，差分パーティションへの書き込みを捕捉，監視し，書き込みが起きた LBA 番号のダーティフラグを立てることでアクセス先管理テーブルを実現する．

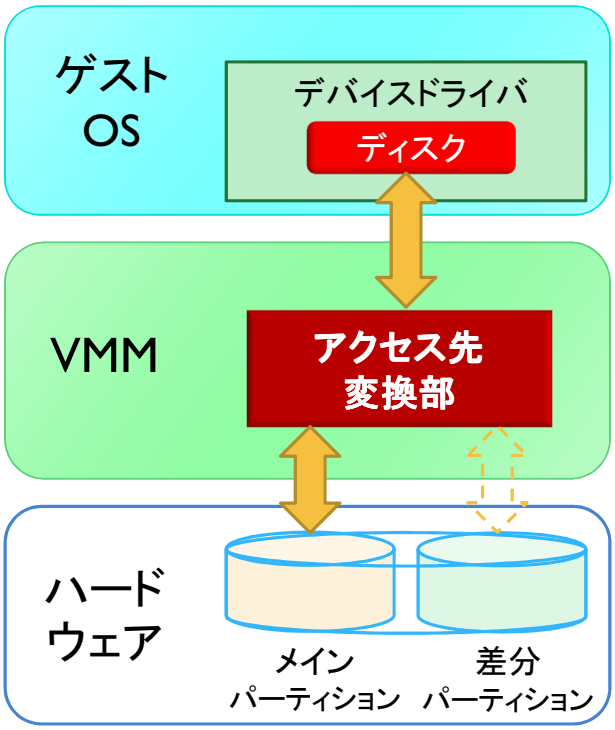

図 10　ディスクデータの保存前と復元後のディスクアクセス

4.3.3　ディスクデータの保存前と復元後のディスク I/O

ディスクデータの保存前と復元後とは，通常時の状態のことを示している．ディスクデータの保存前と復元後のディスクアクセスの様子を図 10 に示す．ディスクデータの保存前と復元後のディスク I/O では，ディスクへ書き込みも読み込みも全てメインパーティションに対して実施される．したがって，ディスクデータの保存前と復元後のディスク I/O では差分パーティションは使用されない．

4.3.4　ディスクデータの保存後から復元までの間のディスク I/O

ディスクデータの保存後から復元までの間とは，マルウェアの動的解析をするような特別な状態のことを示している．ディスクデータの保存後から復元までの間のディスクアクセスの様子を図 11 に示す．ディスクデータの保存後から復元までの間では，ディスクへの書き込みは全て差分パーティションに対して行われるが，ディスクの読み込みはアクセス先管理テーブルのダーティフラグの値によって異なる．ここからは，ディスクデータの保存後から復元までの間のディスク I/O についての詳細について述べていく．

まず，ディスクデータの保存後から復元までの間に起きるディスクへの書き込みに対する処理方法について述べる．ディスクデータの保存後から復元までの間にディスクへの書き込みが起きた際は，まず，アクセス先変換部がディスク I/O の LBA 番号を全て差分パーティションの LBA 番号に変更する．そして，アクセス先管理テーブル内の該当する部分のダーティフラグを立て，LBA 番号を差分パーティションに変更した I/O をディスクへと渡す．提案システムではこのような手法を用いて，ディスクデータの保存後から復元までの間のディスクへの書き込みを実現する．

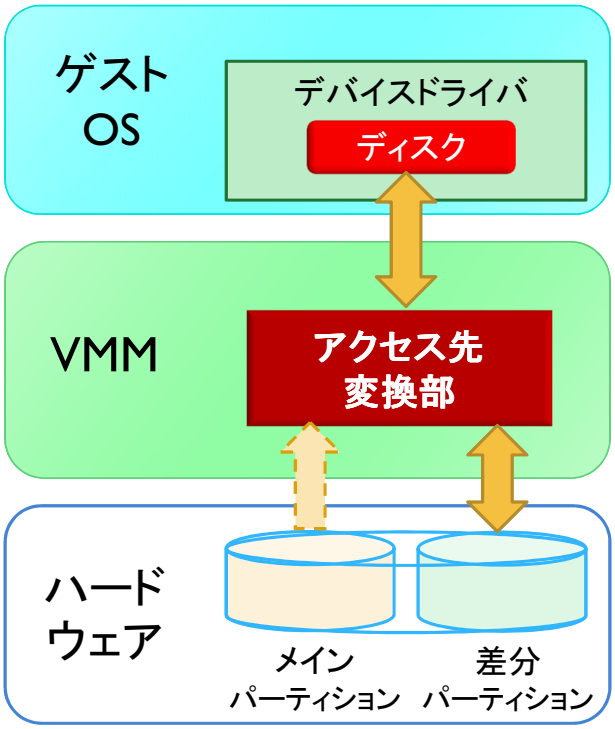

図 11　ディスクデータの保存後から復元までの間のディスクアクセス

次に，ディスクデータの保存後から復元までの間に起きるディスクの読み込みに対する処理方法について述べる．ディスクデータの保存後から復元までの間にディスクの読み込みが起きた際は，まず，アクセス先変換部がディスク I/O の LBA 番号を全て差分パーティションの LBA 番号に変更し，変更した I/O をディスクへと渡す．すると，差分パーティションから該当する部分が読み込まれ，ディスクの読み込みが完了したことを通知する I/O がディスクから発行される．通常のディスクの読み込みでは，ディスクから発行される完了したことを通知する I/O をそのままゲスト OS に渡すことで終了する．しかし，このままディスクの読み込み処理を終了すると，差分パーティションに対して書き込みがされていない部分も同時に読み込まれデータの整合性がとれなくなってしまう．

そこで，提案システムではアクセス先管理テーブルテーブルを用いる．ディスクの読み込みの該当部分でダーティフラグが立っていない部分に対しては，アクセス先変換部が新たにディスク I/O を発行し，メインパーティションから新たにデータを読み込む．そして，メインパーティションから読み込んだデータを先に差分パーティションから読み込んだデータの該当部分に上書きする．全てのデータの上書き処理が終わった後，捕捉しているディスクの読み込みが完了したことを通知する I/O をゲスト OS へと最後に渡す．提案システムではこのような手法を用いて，ディスクデータの保存後から復元までの間のディスクの読み込みをデータの整合性を保ちつつ実現する．

4.3.5 ディスクデータの保存後から復元までの間のディスク I/O の具体例

ディスクデータの保存後から復元までの間のディスク I/O は少し複雑なので，ここでは具体例とともに説明する．具体例ではメインパーティションの先頭の LBA 番号が 0x100000 で，差分パーティションの先頭の LBA 番号が 0x6500000 であるとする．したがって，差分パーティションとメインパーティションの LBA 番号の差である定数 A は 0x6400000 となる．

まず、LBA 番号が 0x130000 から 0x13ffff へデータを書き込む場合について見ていく．0x130000 から 0x13ffff へデータを書き込もうとするとアクセス先変換部によってディスク I/O の LBA 番号が変更され，差分パーティションの 0x6530000 から 0x653ffff に対してデータが書き込まれる．そして，図 9 のように，アクセス管理テーブルの 0x130000 から 0x13ffff の部分にダーティフラグを立てられる．

次に，LBA 番号が 0x120000 から 0x14ffff のデータを読み込む場合について見ていく．この場合のディスクの読み込みの様子を図 12 に示す．0x120000 から 0x14ffff のデータを読み込もうとするとまず，アクセス先変換部によってディスク I/O の LBA 番号が変更され，差分パーティションの 0x6520000 から 0x654ffff のデータがメモリに読み込まれる．

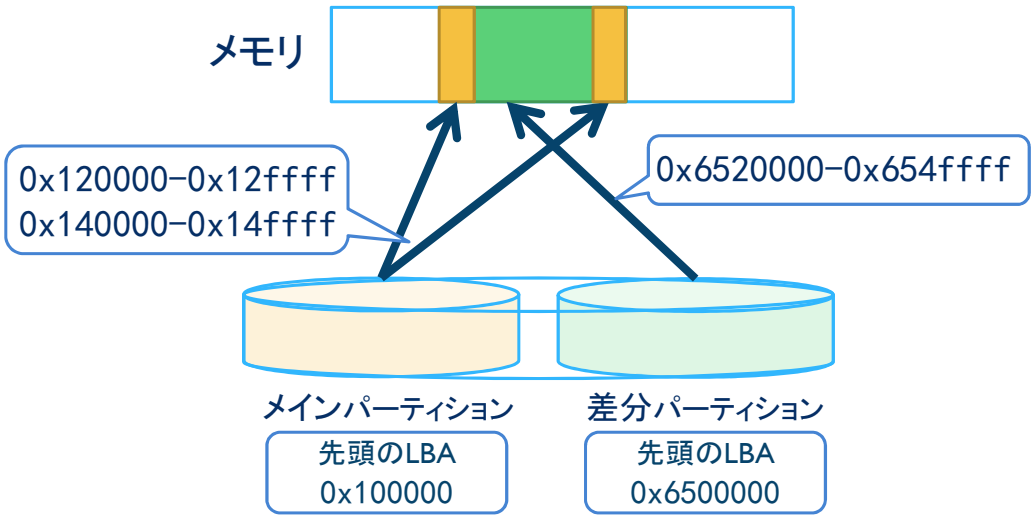

図 12　ディスクデータの保存後から復元までの間に起きるディスクの読み込みの具体例

次に，読み込みが完了するとディスクから完了したことを通知する I/O が発行されるので，アクセス先変換部が I/O を捕捉し，アクセス先管理テーブルを参照して新たなディスクの読み込みの I/O を発行する．今回の場合，図 9 のように 0x130000 から 0x13ffff 以外の部分のフラグが立っていないので，0x120000 から 0x12ffff と 0x140000 から 0x14ffff の部分がメインパーティションから読み込まれ，先ほど読み込んだデータの該当する部分に上書きされる．最後に，2 つのメインパーティションからの読み込みとデータの上書きが終わった後，捕捉したディスクの読み込みが完了したことを通知する I/O をゲスト OS へ渡す．

5　実験・評価

5.1　動作確認の実験

提案システムによって，OS の状態が期待通り復元できるかを確認するために，Ubuntu 12.04 (32 bit), Fedora 20 (32 bit)，Windows 7 (32 bit) を用いて OS の状態の保存，復元の実験を行った．本実験では，まず，OS 上でターミナルとインターネットブラウザを起動させ，提案システムの保存機能を用いてチェックポイントを取った．チェックポイントを取った後，ターミナルとインターネットブラウザを閉じ，提案システムの復元機能を用いてターミナルやウインドウの状態が元に戻るかを確かめた．動作確認の実験の様子を図 13 に示す．

Ubuntu 12.04，Fedora 20 の環境では，提案システムの復元機能のみでウインドウが表示されている場所や表示内容，ターミナルやインターネットブラウザの履歴などの状態を保存時の状態に戻すことができた．また，OS の状態を戻した後も，継続して通常通りの処理を行うことができた．

Windows 7 の環境では，画面の再描画は必要となったが，ウインドウが表示されてい

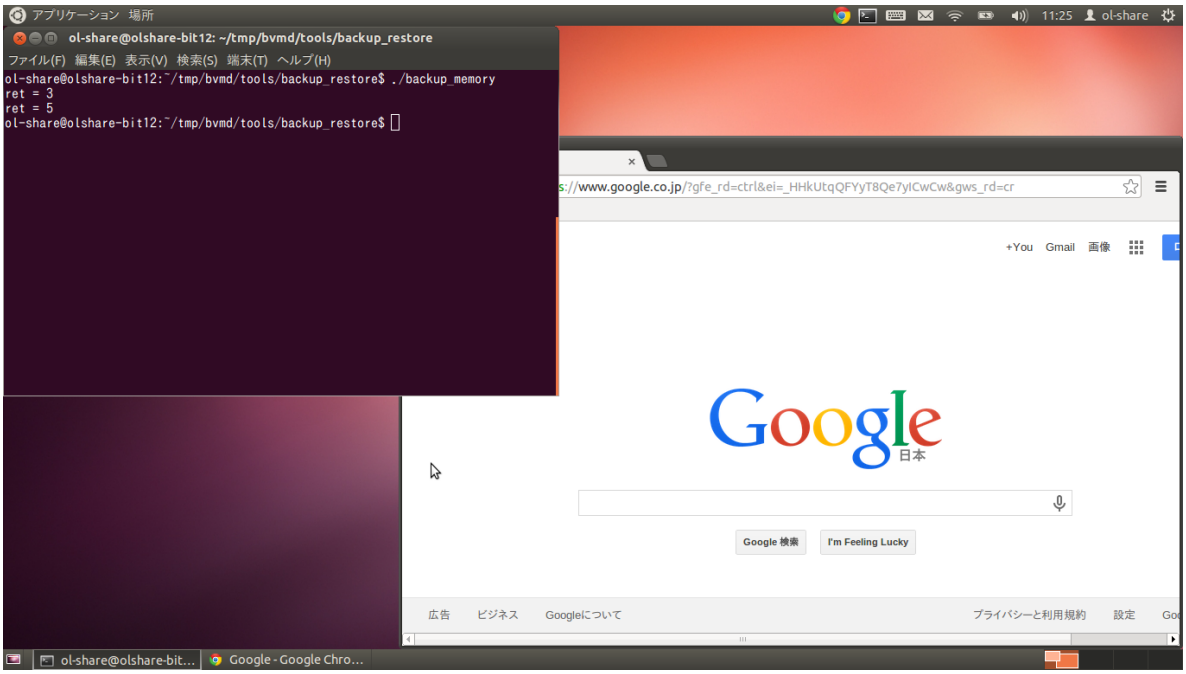

図 13　動作確認の実験の様子

る場所や表示内容，ターミナルやインターネットブラウザの履歴などの状態を保存時の状態に戻すことができた．また，OS の状態を戻した後も，継続して通常通りの処理を行うことができた．

次に， 提案システムの保存機能を用いてチェックポイントを取った後，デスクトップ上のディレクトリやファイルを全て削除した．ディレクトリやファイルの削除が完了した後，提案システムの復元機能を用いてファイルの状態が元に戻るかを確かめた．

この実験では，Ubuntu 12.04，Fedora 20，Windows 7 の環境ともに，提案システムの復元機能のみでディレクトリやファイルの属性やサイズ，格納されているデータの内容の状態を保存時の状態に戻すことができた．また，OS の状態を戻した後も，通常通り，ディレクトリやファイルの読み書きなどの処理を行うことができた．

最後に， 提案システムの保存機能を用いてチェックポイントを取った後，インターネットブラウズである Opera [16] を OS 上にインストールした．Opera のインストールが終わった後，提案システムの復元機能を用いて Opera がインストールされる前の状態に戻るかを確かめた．

この実験では，Ubuntu 12.04，Fedora 20，Windows 7 の環境ともに，インストールされたアプリケーションの一覧を確認したところ，Opera を見つけることが出来ず，Opera がインストールされる前の状態に戻すことができた．

5.2 オーバヘッド測定の実験

提案システムの有効性を評価するために，提案システム導入による実行時間のオーバヘッドの実験を行った．実験環境を以下に示す．なお，提案システムで OS の状態の保存，復元にかかる処理時間は数秒程度と短いものであったので，今回の実験では対象から外した．

- CPU: Intel Core i3-21200 3.3 GHz
- Memory: 8 GB
- HDD: Seagate ST500DM002-1BD14 500 GB
- Chipset: Intel 7 Series/C210 Series Chipset
- VMM: BitVisor 1.3
- ゲスト OS: Ubuntu 13.04(32 bit), 3.8.0-32-generic

本実験の比較対象は，実環境と提案システム導入前の BitVisor が入った環境，提案システムが入った環境の 3 つであり，以降，実環境を **Native**，提案システム導入前の BitVisor が入った環境を **BitVisor**，提案システム導入後の環境を **Our System** と呼ぶ．ただし，提案システムでは，OS の状態の保存前と復元後では I/O の捕捉，監視は BitVisor が自身が実施するもの以外は特にしていない．したがって，提案システムが入った環境の内，オーバヘッドが大きくなると推測できる OS の状態の保存後から復元までの間の状態となった環境を Our System として実験した．また，BitVisor，Our System ともに，BitVisor のコンパイルオプションとして，Shadow Page Table 等のデフォルトの機能は ON にし，暗号化や VPN の機能は OFF にした．

提案システム導入による実行時間のオーバヘッドの測定として，本実験では，Bonnie++ [17] による I/O ベンチマークと UnixBench [18] によるシステム・ベンチマーク，BitVisor 1.3 のビルド時間を測定する BitVisor 1.3 のビルド・ベンチマークの 3 つのベンチマークを用いて実験した．各ベンチマークのバージョンを以下に示す．

- Bonnie++ による I/O ベンチマーク: Bonnie++ version 1.97
- Unixbench によるシステム・ベンチマーク: Unixbench version 5.1.3
- BitVisor 1.3 のビルド・ベンチマーク: gcc version 4.7.3

5.3 Bonnie++ による I/O ベンチマーク

本実験ではまず，I/O ベンチマークである Bonnie++ を用いて提案システム導入による I/O にかかる実行時間のオーバヘッドを測定した．実験の際は，ページキャッシュを全てクリアしてから実行時間を測定した．Bonnie++ による I/O ベンチマークの結果を図 14 に示す．図 14 では，各要素は以下のことを表している．また，各要素の値は Native を 100% とした時の相対スループットであり，値が大きいほど性能がよいことを示している．

- Random Seeks: ディスクに対してランダムアクセスにかかった時間
- Sequential Input Block: ディスクの連続した領域をブロック単位で読み込むのにかかった時間
- Sequential Input Per Chr: ディスクの連続した領域をキャラクタ単位で読み込むのにかかった時間
- Sequential Output Rewrite: ディスクの連続した領域を読み込んだ後，書き込むのにかかった時間
- Sequential Output Block: ディスクの連続した領域をブロック単位で書き込むのにかかった時間
- Sequential Output Per Chr: ディスクの連続した領域をキャラクタ単位で書き込むのにかかった時間

結果を見ていくと，ディスクの読み込みも書き込みも，ブロック単位のアクセスよりキャラクタ単位のアクセスで大きいオーバヘッドが Our System に発生している．これは，ディスクへのアクセスの大きさが小さいほどディスク I/O の数が増え，捕捉，監視等の処理が増えるからだと推測できる．しかし，Native と比較すると Our System に大きいオーバヘッドが出ていても BitVisor と比較するとオーバヘッドは小さい．したがって，ここで発生しているオーバヘッドの大部分は BitVisor によるオーバヘッドであると推測できる．

また，Sequential Output Rewrite では 53% の性能低下という大きいオーバヘッドが Our System に発生している．これは，連続した領域の読み込み，書き込みが続くとダーティフラグが立っている部分と立っていない部分が混ざった領域の読み込みが多発し，メインパーティションからの読み込みが増えたことが原因だと推測できる．実際，実験時にシステムが出したログを見てみるとメインパーティションの読み込みが多発していた．しかし，これは提案システムで発生する最悪なケースなので，最悪ケースでも最大 53% の

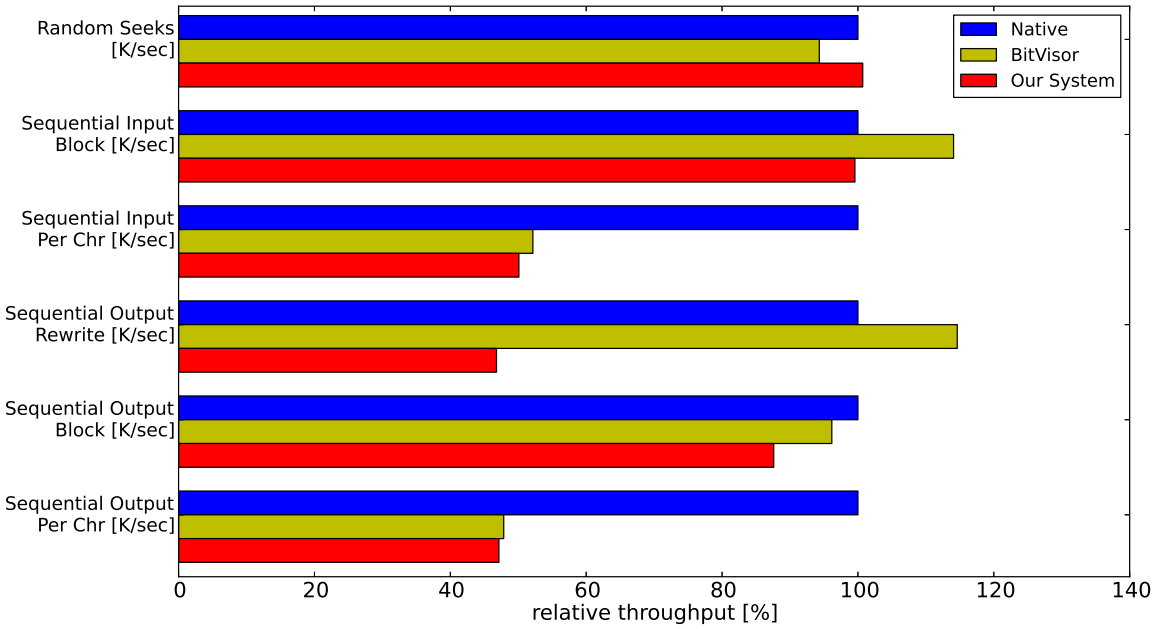

図 14　Bonnie++ による I/O ベンチマーク

性能低下で済むということがわかった.

5.4　UnixBench によるシステム・ベンチマーク

次に，システム・ベンチマークである UnixBench を用いて提案システム導入によるシステム全体に与えるオーバヘッドを測定した．UnixBench によるシステム・ベンチマークにおいて総合的なベンチマークの値である System Benchmarks Index Score を表 1 に，各要素の結果を図 15 に示す．図 15 では，各要素は以下のことを表している．また，各要素の値はベンチマークの結果として出力される性能の指標である Index であり，値が大きいほど性能がよいことを示している.

- System Call Overhead: システムコールの性能
- Shell Scripts (8 concurrent): シェルスクリプト実行の性能 (8 個並列)
- Shell Scripts (1 concurrent): シェルスクリプト実行の性能 (並列なし)
- Process Creation: プロセス生成の性能
- Pipe-based Context Switching: pipe ベースでのコンテキストスイッチングの性能
- Pipe Throughput: pipe 処理の性能
- File Copy 4096 bufsize 8000 maxblocks: ファイルコピーの性能 (バッファサイズ

表 1　System Benchmarks Index Score の比較

	System Benchmarks Index Score
Native	3088.1
BitVisor	1924.8
Our System	2010.0

4096 バイト)

- File Copy 256 bufsize 500 maxblocks: ファイルコピーの性能 (バッファサイズ 256 バイト)
- File Copy 1024 bufsize 2000 maxblocks: ファイルコピーの性能 (バッファサイズ 1024 バイト)
- Execl Throughput: execl 関数実行処理の性能
- Double-Precision Whetstone: 倍精度浮動小数点演算の性能
- Dhrystone 2 using register variables: 2 つのレジスタを使用した整数・文字列演算の性能

まず，System Benchmarks Index Score の値から見ていくと，Our System の性能は BitVisor の性能より高いことがわかる．しかし，Our System と BitVisor の差は Our System と Native の差に比べると小さく，Our System と BitVisor の性能差は誤差であるといえる．したがって，システム全体みると提案システムで発生するオーバヘッドの大部分は BitVisor によるオーバヘッドであるといえる．

次に，各要素の結果を見ていくと，Pipe Throughput で 64% の性能低下という大きいオーバヘッドが Our System に発生している他，ファイルコピーなどで大きいオーバヘッドが Our System に発生している．しかし，Our System と BitVisor だけを比較すると，Our System で発生したオーバヘッドは小さく，むしろ性能が向上した要素もいくつかある．

このことから，UnixBench のようなシステム・ベンチマークでは，提案システムを BitVisor に導入した際に発生する新たなオーバヘッドはほとんど現れないということがわかった．したがって，ユーザが普通に使う分には，提案システムは BitVisor と同様の性能で使用できるということが推測できる．

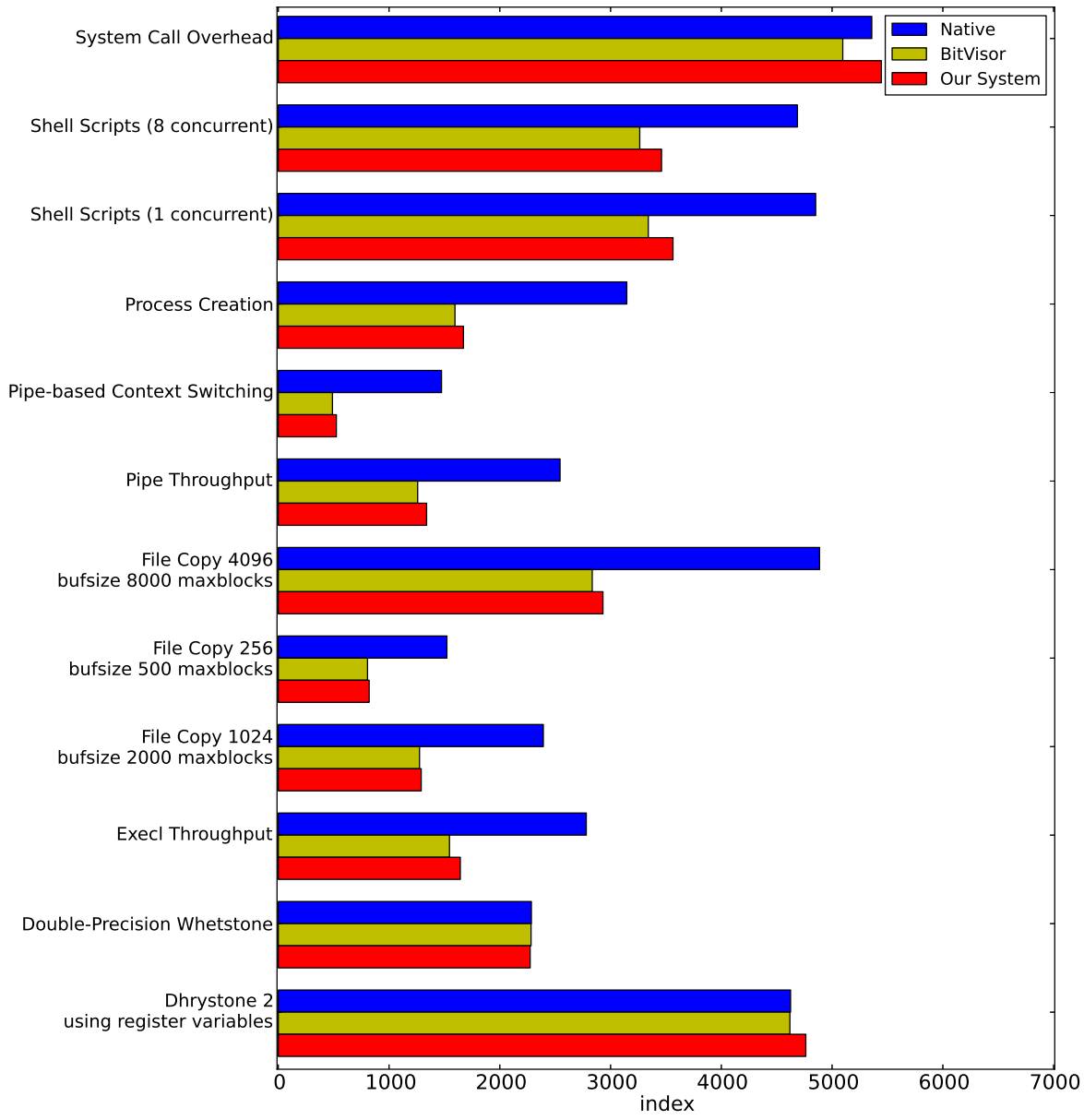

図 15　UnixBench によるシステム・ベンチマーク

5.5 BitVisor 1.3 のビルド・ベンチマーク

最後に，BitVisor 1.3 のビルドにかかる時間を測定する BitVisor 1.3 のビルド・ベンチマークを用いて，ユーザが実際に提案システム上で体感するであろうオーバヘッドを測定した．並列コンパイル機能を ON にした際の BitVisor 1.3 のビルド・ベンチマークの結果を図 16 に，並列コンパイル機能を OFF にした際の BitVisor 1.3 のビルド・ベンチマークの結果を図 17 に示す．図 16，17 は BitVisor 1.3 のビルドにかかった時間を表しており，実験回数は各環境とも 5 回ずつである．また，各環境で生じた平均ビルド時間を表 2 に示す．

まず，図 16，17 の結果を見ていくと，各環境とも各回のビルド時間に多少のばらつきがあるが一定の範囲に収束している．そこで次に，各環境での平均ビルド時間である表 2

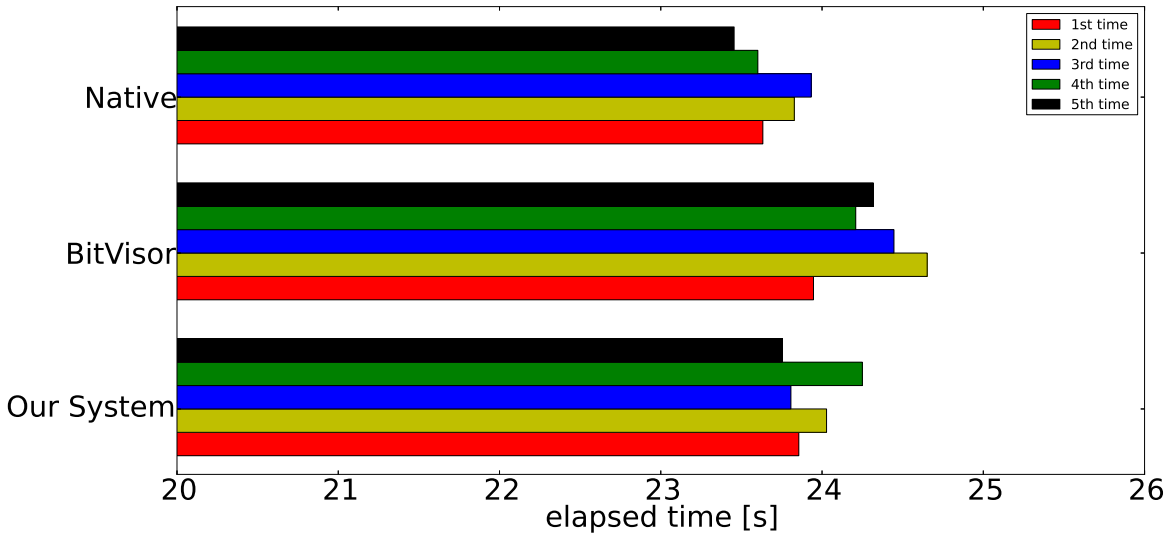

図 16　並列コンパイル機能を ON にした際の BitVisor 1.3 のビルド・ベンチマーク

表 2　各環境での平均ビルド時間 [sec]

	並列コンパイル機能を ON 時	並列コンパイル機能を OFF 時
Native	23.69	63.69
BitVisor	24.51	64.98
Our System	23.94	64.84

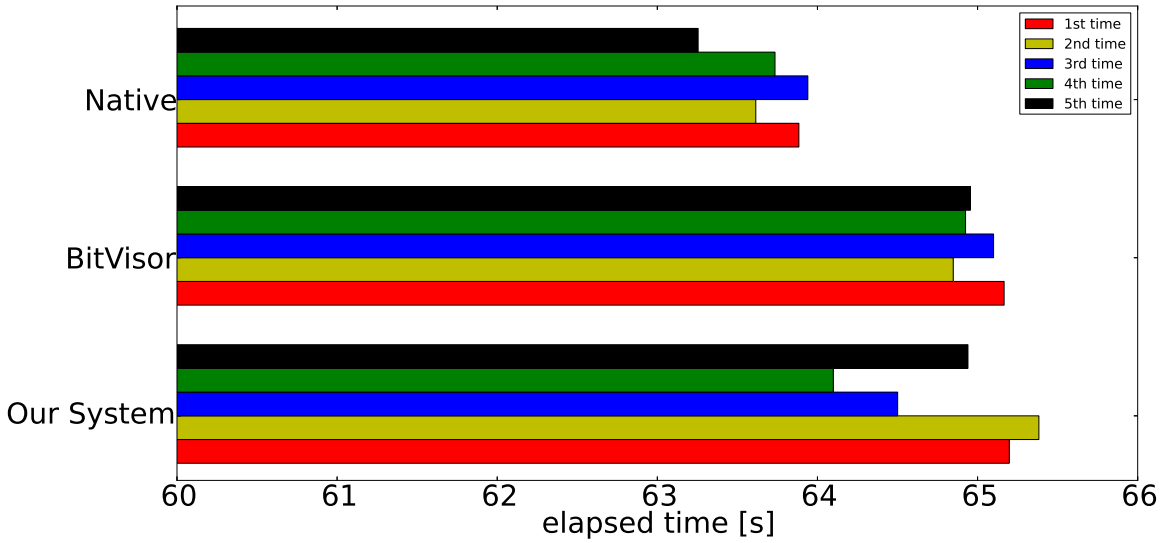

図 17　並列コンパイル機能を OFF にした際の BitVisor 1.3 のビルド・ベンチマーク

を見ていく．平均ビルド時間を見た結果，並列コンパイル機能を ON にした際，OFF にした際を通じて，Our System のオーバヘッドは Native と比べて最大でも 2 %しか発生していない．

このことから，ユーザが実際に提案システム上で体感するであろう提案システムのオーバヘッドは実環境と比べても僅かである．したがって，提案システムを導入しても，ユーザがシステムを普通に使う分にはオーバヘッドはほとんどなく，提案システムは有効であると推測できる．

6 議論

6.1 デバイスの状態の復元について

各デバイスは ACPI で定義されている Device Power States のような状態を持つ．提案システムでは，このデバイスの状態の復元までは考慮していない．しかし，OS の状態の保存時と復元時でデバイスの状態が異なる可能性も存在する．

この問題に対しては，ゲスト OS の機能を借りることになるが，ゲスト OS をサスペンドした後，ゲスト OS を再開させることで各デバイスがリフレッシュされ，デバイスの状態が復元できるのではないかと考えている．このような手法を用いれば，デバイスの状態の復元はできるが，デバイスの状態が OS の状態の保存時と復元時で異なるデバイスは多くはないと考えている．例えば，マルウェアの動的解析の場合のように OS の状態の保存と復元の間隔が短い場合では，OS の状態の保存時と復元時でデバイスの状態が異なるデバイスは多くはない．したがって，提案システムでは，デバイスの状態の復元は未対応となっている．

6.2 データの保存先について

提案システムでは，OS の状態の保存，復元時間の低減のため，メモリデータの保存はメモリ上のゲスト OS の領域をメモリ上の空いている領域にチェックポイントとして保存することで実現している．故に，提案システムを再起動すると消えてしまう問題点やゲスト OS に割りてることができるメモリサイズは減少する問題点が存在する．提案システムを再起動すると消えてしまう問題点に関しては，提案システムが再起動するとメモリ上のゲスト OS の状態を構成するデータもリフレッシュされ，OS の状態の復元が必要なくなるのではないかと考えている．ゲスト OS に割りてることができるメモリサイズは減少する問題点は提案システムに改善の余地があると考えており，適切なメモリデータの保存場所は他にも存在すると考えている．

また，提案システムでは，提案システム導入後のオーバヘッドの抑制のため，ディスクデータの保存は，同一ディスク上に別パーティションとして差分パーティションを作り，ディスクアクセスの際に LBA 番号に定数を足し，アクセス先を変えることで実現している．故に，ゲスト OS がインストールされているメインパーティションと同等サイズの差分パーティションが必要となり，メインパーティションとして使えるディスクサイズが減少する問題点が存在する．この問題点に関しては，差分パーティションに書かれるデータ

の配置方法の変更やデータの圧縮など改善の余地が十分にあり，今後の課題としたい．

6.3 保存できるチェックポイントの数について

提案システムでは，保存できるチェックポイントは 1 つのみとなっている．保存できるチェックポイントは 1 つのみというのは，本研究で想定しているようなマルウェアの動的解析に用いる例の場合では，最低限の要求を満たしているが，サーバの定期的なバックアップに使用するなど，他の用途に応用するには厳しい．

提案システムに対して，複数のチェックポイントを保存できるように拡張するのは，メモリデータの保存，複数に関しては多少の設計の変更で済むが，ディスクデータの保存，複数に関しては大幅な設計の変更が必要となってくる．例えば，ディスクデータの保存に使用する差分パーティションの数を単に増やしたとしても，ディスクの読み込みの際やディスクデータの復元の際に，どのパーティションのデータが有効であるかの管理が新たに必要となる．したがって，提案システムに対して保存できるチェックポイントの数を増やすのは単純なことではなく，設計の変更など議論の余地が十分にあり，今後の課題としたい．

7 関連研究

7.1 状態の復元機能を有したシステム

マルウェアの動的解析を行うために OS の状態機能を有した VMM としては，BareBox [19] がある．BareBox は，提案システムと同様に，メモリデータやディスクデータをゲスト OS のリブートなしで復元することができる．しかし，BareBox は Xen を拡張して作られており，復元対象のゲスト OS が動いている VM と異なる VM で動いている管理 OS というものが存在する．したがって，チェックポイントの管理などの処理で管理 OS の機能を利用している点が本研究とは異なる．提案システムでは，管理 OS の機能を利用しない分，実環境に近い環境を提供できるので，マルウェアの動的解析において，BareBox よりも精度の高い解析ができる．

マルウェアの動的解析のために，元のシステムの状態を保存しつつ，マルウェアを実際に動かすことができる環境を提供するシステムとしては Alcatraz [20] がある．Alcatraz は，本研究と同様にディスクの書き込み先を変え，書き込みを差分として保存しておくことで元のシステムの状態を維持し，差分を捨てることでシステムの状態を元に戻す．Alcatraz では，システムコール引数にあるファイルパスの書き換え，ファイルシステムを仮想化することでディスクのアクセス先を自在に変えている．したがって，Alcatraz では，ディスクの書き込み先の変更をファイルパスの書き換えで実現している点が，LBA 番号の書き換えで実現している本研究と異なる．提案システムでは，LBA 番号というファイルパスより低レベルな部分の変換をしている分，実環境により近い環境を提供できるので，マルウェアの動的解析において，Alcatraz よりも精度の高い解析ができる．

マルウェアの動的解析の用途に限らず，サーバのバックアップなどの用途に用いるため，OS の状態復元機能を有した VMM としては，VMware Workstation や KVM，Hyper-V [21] がある．これらの VMM は，提案システムと同様に任意のタイミングで OS の状態を保存，復元することができる．しかし，これらの VMM では，ホスト OS が存在したり，ディスクなどのデバイスを仮想化していたりする．したがって，チェックポイントの管理などの処理でホスト OS の機能を利用している点や，デバイスを仮想的に管理している点が本研究とは異なる．提案システムでは，ホスト OS の機能の利用やデバイスの仮想化をしていない分，実環境に近い環境を提供できるので，マルウェアの動的解析において，これらの VMM よりも精度の高い解析ができる．

BitVisor を拡張して，VM 上の OS の状態を他のマシンにマイグレーションする研究

としては，深井ら [22] の研究がある．深井らの研究は，異なる物理マシン間のライブマイグレーションに注目をしているが，マイグレーションの対象を同一マシンの未来の時間に置き換えれば，本研究と同様に，BitVisor を拡張した OS の状態復元機能となる．深井らの研究では，メモリデータやゲスト OS のレジスタの値は保存，復元できるが，ディスクデータの保存，復元はできない点が本研究とは異なる．提案システムでは，ディスクデータの保存もできるので，深井らの研究では解析しにくいディスクデータを壊すようなマルウェアもより容易に解析できる．

状態の復元機能であるスナップショット機能を持ったシステムとしては，ZFS [23], Btrfs[24] がある．ZFS，Btrfs はスナップショット機能を持ったファイルシステムであり，コピーオンライトと呼ばれる方式を用いてファイルシステム自身がファイルシステムを複製し，スナップショットをとっている．ZFS，Btrfs では，用意にディスクデータのスナップショットをとることができるが，使用することができる OS が限定されている．一方提案システムでは，使用することができる OS が限定されていない BitVisor を使用している点が異なる．したがって，提案システムでは ZFS，Btrfs を使用したシステムよりも多くの環境でマルウェアの動的解析ができる．

7.2 マルウェア解析用の VMM

マルウェアを解析するため VMM としては，Ether [25] や MAVMM [26] がある．Ether は自分自身の存在をできるだけ隠しつつ，マルウェアを解析するためのシステムであり，Xen を拡張して作られている．Xen を拡張して作らているため，Ether は VMM と監視対象のゲスト OS が動いている VM と異なる VM で動いている管理 OS が協力し，システムコールやメモリ書き込みなどのイベントを監視している．また，これらの機能を用いた様々なマルウェアの解析の実証実験も行われており，有効性も示されている．Ether はマルウェアの解析に必要な機能を多く持っているが，本研究で提案する OS の状態復元機能のような状態を戻す機能は持っていない点と管理 OS の機能を利用することができる点が本研究とは異なる．提案システムでは，OS の状態復元機能をもっている分，解析環境の素早い復元ができるため，単時間あたりより多くの解析ができる．

MAVMM はマルウェアの動的解析に特化した軽量な VMM であり，提案システムが用いている BitVisor と同様に，極力デバイスを仮想化しない方針で，セキュア，かつ，実環境に近い環境を提供している．MAVMM ではメモリダンプやディスクアクセス，プロセストレースをログとして記録する機能も持っている．また，これらの機能を用いた様々なマルウェアの解析の実証実験も行われており，有効性も示されている．MAVMM はマル

ウェアの解析に必要な機能を多く持っているが，本研究で提案する OS の状態復元機能のような状態を戻す機能は持っておらず，この点が本研究とは異なる．提案システムでは，OS の状態復元機能をもっている分，解析環境の素早い復元ができるため，単時間あたりより多くの解析ができる．

8 おわりに

8.1 まとめ

本研究では，BitVisor のための OS の状態復元機能を提案した．また，3 種類のベンチマークを用いた比較実験を行い，システム導入後のオーバヘッドを測定，評価した．評価の結果，I/O に負荷がかかる処理では提案システムに最大 53% の性能低下が見られたが，システム全体のベンチマークやビルド・ベンチマークでは，提案システム導入前の BitVisor と同等の性能が出ていることがわかり，提案システム導入後のオーバヘッドが実用に耐えられるレベルであることを確認した．

8.2 今後の課題

今後の課題としては，次の 3 つが挙げられる．1 つ目は，デバイスの状態の復元への対応である．議論の章でも述べたが，OS のサスペンド機能などを使ってこの課題を課題を取組んでいきたい．2 つ目は，チェックポイントを保存する場所の改善である．議論の章でも述べたが，既存の手法ではいくつか問題点が存在するので，最適な保存場所を探していきたい．3 つ目は，チェックポイントを複数取れるように拡張することである．議論の章でも述べた通り，提案システムで複数のチェックポイントを取れるように拡張することは容易ではなく，今後の課題としてさらなる調査や考察をしてきたい．

謝辞

本研究にあたり，多大なご指導を賜わりました大山恵弘准教授に深く感謝いたします．また，貴重で有益な助言を頂きました高橋一志特任助教，佐々木慎君，ともに研究に励み，心の支えとなった大山研究室の皆様，そして，2 年間の学生生活を暖かく支えてくださった家族に深く感謝の意を表します．

参考文献

[1] CONFICKER, http://about-threats.trendmicro.com/us/malware/conficker.

[2] BKDR_SRAOW.A, http://about-threats.trendmicro.com/us/malware/bkdr_sraow.a.

[3] TPM, http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[4] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *In Proceedings of Network and Distributed Systems Security Symposium (NDSS 2003)* (2003).

[5] Jiang, X., Wang, X. and Xu, D.: Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction, *In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)* (2007).

[6] Oyama, Y., Giang, T. T. D., Chubachi, Y., Shinagawa, T. and Kato, K.: Detecting Malware Signatures in a Thin Hypervisor, *In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012)*, pp. 13–20 (2012).

[7] VWware, http://www.vmware.com/.

[8] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *In Proceedings of the USENIX Annual Technical Conference, FREENIX*, p. 41 (2005).

[9] 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, *In Proceedings of the 19th ACM symposium on Operating Systems Principles (SOSP19)*, pp. 164–177 (2003).

[10] User-Mode Linux, http://user-mode-linux.sourceforge.net/.

[11] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y. and Kato, K.: BitVisor: a thin hypervisor for enforcing I/O device security, *In Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)* (2009).

[12] WORM_AGOBOT.GEN, http://about-threats.trendmicro.com/us/malware/worm_agobot.gen.

[13] SDBOT, http://about-threats.trendmicro.com/us/malware/sdbot.

[14] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux

virtual machine monitor, *In Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)* (2007).

[15] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H. and Smith, L.: Intel Virtualization Technology, *Computer*, Vol. 38, No. 5, pp. 48–56 (2005).

[16] Opera, http://www.opera.com/.

[17] Bonnie++, http://www.coker.com.au/bonnie++/.

[18] UnixBench, https://code.google.com/p/byte-unixbench/.

[19] Kirat, D., Vigna, G. and Kruegel, C.: BareBox: efficient malware analysis on bare-metal, *In Proceedings of the 27th Annual Computer Security Applications Conference* (2011).

[20] Liang, Z., Sun, W., Venkatakrishnan, V. N. and Sekar, R.: Alcatraz: An Isolated Environment for Experimenting with Untrusted Software, *ACM Trans. Inf. Syst. Secur.*, Vol. 12, No. 3, pp. 14:1–14:37 (2009).

[21] Hyper-V, http://www.microsoft.com/hyper-v-server/.

[22] 深井貴明, 表祐志, 品川高廣, 加藤和彦：物理マシン間のライブマイグレーション手法の提案, 情報処理学会研究報告, Vol. 2013-OS-127, No. 13 (2013).

[23] Rodeh, O. and Teperman, A.: zFS - A Scalable Distributed File System Using Object Disks, *In Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS 2003)*, pp. 207– (2003).

[24] Btrfs, https://btrfs.wiki.kernel.org/index.php/Main_Page.

[25] Dinaburg, A., Royal, P., Sharif, M. and Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions, *In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pp. 51–62 (2008).

[26] Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T. and Nguyen, H. D.: MAVMM: Lightweight and Purpose Built VMM for Malware Analysis, *In Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC 2009)*, pp. 441–450 (2009).