情報処理学会論文誌 Vol.61 No.9 1495–1506 (Sep. 2020)

推薦論文

重要サービスの動作不可視化手法における システムコール代理実行処理の効率化

奥田 勇喜[1] 佐藤 将也[1,a)] 谷口 秀夫[1]

受付日 2019年11月25日, 採録日 2020年6月1日

概要：計算機上で動作するセキュリティソフトウェアやログ収集ソフトウェアなどの重要サービスへの攻撃を回避するために，重要サービスの動作を攻撃者から不可視化する手法が提案されている．この手法は，保護対象仮想計算機（VM）上で動作する重要サービスが発行したシステムコールを仮想計算機モニタ（VMM）により捕捉し，別 VM 上の代理プロセスにより代理実行する．本稿では，代理実行処理において保護対象 VM や重要サービスの性能を著しく低下させる 2 つの要因を示し，これらに対処することにより代理実行処理を効率化する手法を述べる．1 つ目の対処は，代理実行する間における保護対象 VM の停止回避であり，他のプロセスに CPU 使用権を譲るシステムコールを重要サービスに実行させることにより，代理実行する間に保護対象 VM を動作させる．2 つ目の対処は，代理プロセスによる代理実行の依頼取得における遅延削減であり，VMM のイベント通知機構を利用して依頼取得における遅延を削減することにより，代理プロセスの応答時間を短縮する．

キーワード：重要サービス，仮想計算機，処理効率化

Efficiency Enhancement in Proxy Execution of System Call on Behavior-hiding Method for Essential Services

Yuuki Okuda[1] Masaya Sato[1,a)] Hideo Taniguchi[1]

Received: November 25, 2019, Accepted: June 1, 2020

Abstract: To avoid attacks on essential services such as security software or logging programs, a method to make the behavior of the essential services invisible to attackers was proposed. The method detects a system call invoked by an essential service on a protection target virtual machine (VM) by using a VM monitor (VMM) and proxies it by the proxy process on another VM. In this paper, we point out two factors that significantly degrade the performance of protection target VMs or essential processes. Then, we present how to make proxy execution more efficient by countermeasures. The first countermeasure is to avoid pausing the protection target VM during proxy execution. To operate the protection target VM during proxy execution, the VMM intervenes the essential service to execute an alternative system call that yields the processor to other processes. The second countermeasure is to reduce the delay in request acquisition for proxy execution by proxy processes. To reduce the delay, we utilized the event-notification mechanism of the VMM. This reduction resulted in the short response time of proxy execution.

Keywords: essential service, virtual machine, efficiency enhancement

1. はじめに

計算機上で提供されるサービスには，攻撃の防止や攻撃による被害を軽減するためのものがある．たとえば，セキュリティソフトウェアは，悪意のあるソフトウェア（以降，マルウェア）からの攻撃を防止する．また，ログ収集

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

本稿の内容は 2019 年 10 月のコンピュータセキュリティシンポジウム 2019（CSS2019）で報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

ソフトウェアは，攻撃の検知や被害の分析のために有用となるプログラムの動作ログを収集する．以降，これらのサービスを重要サービスと呼ぶ．重要サービスは，攻撃者にとって不都合な存在であるため，攻撃の対象となり，無効化される可能性がある．文献 [1] では，いくつかのセキュリティソフトウェアの脆弱性を利用した攻撃手法が報告されている．また，ルートキットの一種である Adore-ng [2] は，マルウェアの侵入ログを隠蔽するために，ログ収集ソフトウェアである syslog におけるログ受信を妨害する．重要サービスが無効化されると，攻撃による被害の軽減や原因の調査が困難になるため，重要サービスを保護することは重要である．

重要サービスを保護する手法として以下がある．ANSS [3] は，攻撃者によるセキュリティソフトウェアの終了を防止するために，仮想計算機モニタ（Virtual Machine Monitor，以降，VMM）を用いて仮想計算機（Virtual Machine，以降，VM）を監視し，セキュリティソフトウェアを終了させるようなシステムコールが発行されたときに，これを無効化する．また，Process Out-grafting [4] は，脆弱性のある VM 上で動作するセキュリティソフトウェアを信頼できる VM に移動させ，離れた VM からのマルウェア解析を可能にする．上記に対し，重要サービスの存在の特定を困難化することができれば，重要サービスへの攻撃をより困難にでき，攻撃を回避できる可能性を高くできる．

我々は，攻撃者による重要サービスの特定を困難化することに着目し，重要サービスの動作を不可視化する手法を提案した [5], [6]．この手法は，保護対象 VM で発行されたシステムコールを VMM により捕捉し，重要サービスを提供するプロセス（以降，重要プロセス）が発行したファイル操作や通信操作に関するシステムコールを代理 VM 上に用意した代理プロセスにより代理実行する．これにより，当該システムコールは保護対象 VM 上で実行されないため，保護対象 VM 内では不可視となる．このため，ファイル操作内容や通信内容をもとにした重要サービスの存在の特定を困難化できる．しかし，この手法には，保護対象 VM や重要プロセスの性能を著しく低下させる以下の 2 つの要因がある．

(1) 代理実行する間における保護対象 VM の停止
VMM は，保護対象 VM の CPU を占有して代理実行の結果返却を待機する．このため，代理実行する間，保護対象 VM が停止する．

(2) 代理実行の依頼取得における遅延
代理プロセスが代理実行の依頼の有無を VMM に対してポーリングすることにより，代理実行の依頼取得が遅延し，代理プロセスの応答時間が大きくなる．

本稿では，重要サービスの動作を不可視化する手法において，システムコールの代理実行を効率化する 2 つの対処を述べる．1 つは代理実行する間における保護対象 VM の停止回避であり，他のプロセスに CPU 使用権を譲るシステムコールを重要プロセスに実行させることにより，代理実行する間に保護対象 VM を動作させ，保護対象 VM の性能低下を抑制する．もう 1 つは代理プロセスによる代理実行の依頼取得における遅延削減であり，VMM のイベント通知機構を利用することにより，依頼取得における遅延を削減する．これにより，代理プロセスの応答時間を短縮し，重要プロセスの性能低下を抑制する．また，評価により，これらの対処の効果を示す．

2. 重要サービスの動作不可視化手法

2.1 基本構造

我々は，重要プロセスが行うファイル操作や通信操作に関する情報をもとにした重要サービスの存在の特定を困難化するために，重要サービスの動作不可視化手法（以降，従来手法）を提案した．従来手法の基本構造を**図 1** に示す．従来手法は，保護対象 VM で発行されたシステムコールを VMM により捕捉し，重要プロセスが発行したファイル操作や通信操作に関するシステムコールを代理 VM 上に用意した代理プロセスにより代理実行する．従来手法では，VMM として Xen [7] を使用し，VM 上で動作するオペレーティングシステム（Operating System，以降，OS）として Linux を用いる．VMM 上では，重要サービスが動作する保護対象 VM と代理プロセスが動作する代理 VM が動作する．保護対象 VM は，Intel VT-x を用いて完全仮想化され，`syscall` 命令によりシステムコールが発行される環境を想定する．

以下に，従来手法における代理実行の処理流れを示す．

(1) 重要プロセスがファイル操作または通信操作に関するシステムコールを発行する．
(2) VMM は，ハードウェアブレークポイントを利用して保護対象 VM 上でのシステムコール発行を捕捉する．
(3) 捕捉したシステムコールが重要プロセスによるファイル操作または通信操作である場合，VMM は代理実行の依頼として自身のメモリ領域にシステムコール情報を作成する．そうでない場合は，保護対象 VM に制御を戻す．システムコール情報は，代理プロセスが代理実行するために必要な情報であり，システムコー

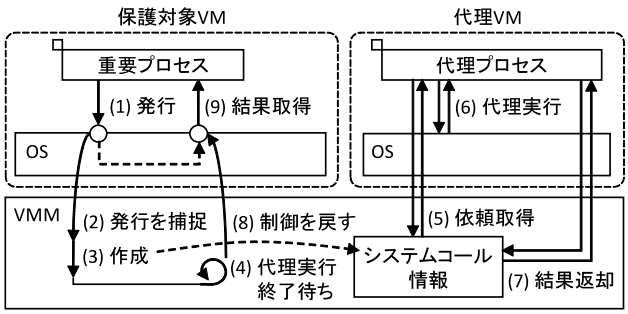

図 1 従来手法の基本構造
Fig. 1 Design of the conventional method.

ルの番号，引数，およびバッファからなる．システムコールの番号と引数は，保護対象 VM のレジスタから取得できる．なお，可変長のデータを引数として渡す `write()` や `sendto()` を捕捉した場合は，引数のポインタが指すメモリ領域の内容を保護対象 VM のメモリからシステムコール情報のバッファにコピーする．

(4) VMM は，代理プロセスによる代理実行の終了を待つ．このとき，CPU はビジーループして代理実行の結果返却の有無を確認する．

(5) 代理プロセスは，代理実行の依頼の有無を VMM に対してポーリングすることにより，システムコール情報を VMM から取得する．

(6) 代理プロセスは，取得したシステムコール情報をもとに，システムコールを代理実行する．

(7) 代理実行後，代理プロセスは，代理実行の結果を VMM に返却する．このとき，VMM は，(4) のビジーループを抜ける．

(8) VMM は，代理実行の結果を保護対象 VM のレジスタに格納する．なお，可変長のデータを受け取る `read()` や `recvfrom()` を代理実行した場合は，代理プロセスから受け取ったバッファの内容を引数のポインタが指すメモリ領域に書き込む．その後，命令ポインタを操作してシステムコール終了処理に制御を戻す．

(9) 保護対象 VM では，システムコール終了処理が実行される．これにより，重要プロセスは，代理実行されたシステムコールの結果を受け取る．

上記の処理流れにより，重要プロセスが発行したシステムコールを代理 VM 上で代理実行することで，重要サービスが行うファイル操作や通信操作を保護対象 VM 上の攻撃者から不可視化できる．

従来手法は，OS やアプリケーションを改変しないことを要件とし，VMM と代理プロセスにより実現されている．従来手法を VMM により実現することで，手法自体が攻撃の被害を受けにくいという利点がある．また，OS やアプリケーションを改変しないことにより，複数の VM に適用する際に OS ごとに修正する必要がない．

2.2 問題点

従来手法には，保護対象 VM や重要プロセスの性能を著しく低下させる 2 つの要因がある．これを**図 2** に示し，以下で説明する．

(A) 代理実行する間における保護対象 VM の停止

VMM は，重要プロセスが発行したシステムコールを捕捉 (2) してから，代理実行の結果を重要プロセスに返却する (8) まで，保護対象 VM の CPU を使用して動作する．つまり，保護対象 VM は，(2) から (8) までの間，この CPU を使用できない．特に，保護対象 VM に割り当てられている仮想 CPU の数が 1 つの場合，保護対象 VM は停止する．なお，停止する時間はシステムコールの処理時間を含むため，処理時間が長いシステムコールを代理実行する際には，保護対象 VM の性能低下が大きくなる．

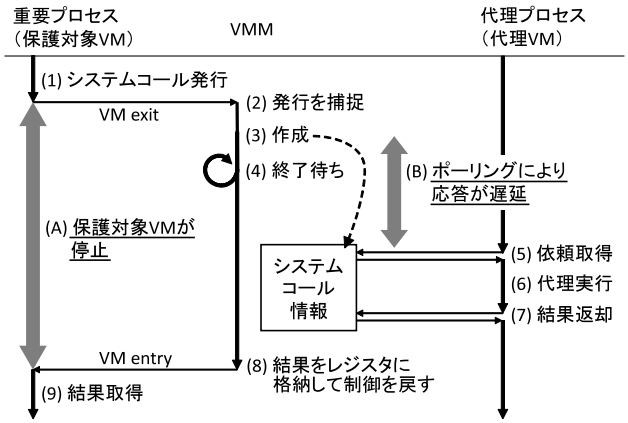

図 2 従来手法における性能低下の要因 (A)，(B)

Fig. 2 The factors (A) and (B) that cause performance degradation in the conventional method.

(B) 代理実行の依頼取得における遅延

(5) において，代理プロセスは，代理実行の依頼の有無を VMM に対してポーリングし，依頼があるとき，システムコール情報を取得して代理実行する．つまり，代理プロセスによる依頼取得は，ポーリング周期に従って遅延する．なお，ポーリング周期は，代理プロセスが複数走行する場合における代理実行の平等性の観点から，1 ms 以上にする必要がある [6]．ポーリング周期が 1 ms である場合，代理実行の依頼取得が平均で約 0.5 ms 遅延する．依頼取得が遅延すると，代理プロセスの応答時間が増加し，重要プロセスのシステムコールオーバヘッドが増加する．

本稿では，従来手法における上記の問題に対して，システムコールの代理実行を効率化する 2 つの対処を述べる．1 つは，代理実行する間における保護対象 VM の停止回避（問題点 (A) の対処）であり，もう 1 つは代理プロセスによる代理実行の依頼取得における遅延削減（問題点 (B) の対処）である．問題点 (A) の対処の内容は 3 章で述べ，問題点 (B) の対処の内容は 4 章で述べる．なお，この 2 つの対処は，従来手法における OS やアプリケーションを改変しないという要件を満たす方法で実現する．

3. 保護対象 VM の停止回避

3.1 要件

保護対象 VM を効率的に動作させるためには，代理実行する間に保護対象 VM に CPU を割り当てて，保護対象 VM の停止を回避する必要がある．以下に，保護対象 VM の停止を回避するための要件を示す．

（要件 1）保護対象 VM に CPU を割り当てて，重要プロセ

ス以外のプロセス（以降，他プロセス）に動作する機会を与えること

（要件 2）重要プロセスが代理実行の結果を受け取れること

保護対象 VM の停止を回避するためには，保護対象 VM に CPU を割り当てて，他プロセスに動作する機会を与える（要件 1）必要がある．また，保護対象 VM に CPU を割り当てると，VMM は代理実行の終了を待機できないため，代理実行の結果が代理プロセスから VMM に返却された際，代理実行の結果を重要プロセスに返却できない．このため，重要プロセスが代理実行の結果を受け取れるようにする（要件 2）必要がある．

3.2 対処

（要件 1）の対処として，保護対象 VM に制御を戻し，`sched_yield()` システムコールが実行されるようにする．`sched_yield()` は，他のプロセスに CPU 使用権を譲る機能を持つ．これにより，保護対象 VM に制御を戻したときに，重要プロセスが `sched_yield()` を発行したことになるため，他プロセスに動作する機会を与えることができる．このとき，保護対象 VM 上では，OS が持つスケジュール機能の呼び出しにより，通常のスケジューリング時と同様に再スケジュールが行われるため，他プロセスが動作しても問題ない．保護対象 VM の重要プロセスに `sched_yield()` を実行させるためには，保護対象 VM に制御を戻す前に，システムコール番号が格納されている RAX レジスタの内容を `sched_yield()` のシステムコール番号（Linux 3.2.0 では 24）に変更する．なお，`sched_yield()` は引数を持たないため，保護対象 VM の状態の変更はシステムコール番号のみとなる．

（要件 2）の対処として，`sched_yield()` の終了時に代理実行の結果の有無を確認する処理を追加する．具体的には，`sched_yield()` の終了処理を VMM により捕捉し，代理実行の結果が代理プロセスから VMM に返却されている場合は，結果を保護対象 VM のレジスタやメモリに格納し，保護対象 VM に制御を戻す．代理実行の結果が返却されていない場合は，保護対象 VM で再度 `sched_yield()` を実行させる．これにより，重要プロセスに代理実行の結果を返却できる．なお，`sched_yield()` の捕捉には，ハードウェアブレークポイントを用いる．OS は，システムコール処理実行の前後に，システムコール共通前処理とシステムコール共通後処理を実行する．このため，システムコール共通後処理のアドレスにハードウェアブレークポイントを設定することで，VMM は，`sched_yield()` の終了処理を捕捉できる．また，命令ポインタにシステムコール共通前処理のアドレスを設定し，システムコール番号を `sched_yield()` の番号に変更して保護対処 VM に制御を戻すことで，重要プロセスに `sched_yield()` を再度実行させることができる．

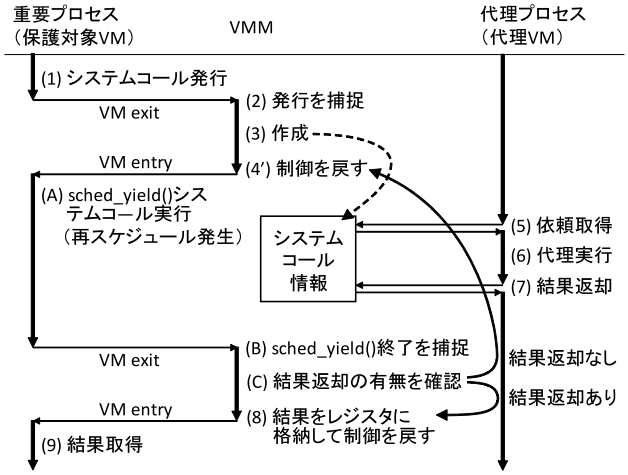

図 3 `sched_yield()` を用いた保護対象 VM の停止回避
Fig. 3 Avoidance of the VM pausing by using `sched_yield()`.

3.3 `sched_yield()` を用いた保護対象 VM の停止回避

図 3 に `sched_yield()` を用いた保護対象 VM の停止回避について示し，以下で説明する．従来手法の処理 (4) を変更して (4′) とし，処理 (A)，(B)，および (C) を追加した．

(4′) VMM は，システムコール番号が格納されている RAX レジスタの値を `sched_yield()` のシステムコール番号に置き換えた後，保護対象 VM に制御を戻す．

(A) 保護対象 VM では，重要プロセスが `sched_yield()` を発行したことになり，再スケジュールが行われる．これにより，代理プロセスが代理実行する間，他プロセスに動作する機会を与えることができる．

(B) (4′) により重要プロセスに発行させた `sched_yield()` の終了処理を VMM により捕捉する．

(C) 代理プロセスによる結果返却の有無を確認する．すでに結果が返却されている場合は，(8) に移行する．まだ返却されていない場合は，命令ポインタに OS 内のシステムコール共通前処理のアドレスを設定した後，(4′) に遷移して重要プロセスに再度 `sched_yield()` を実行させる．

上記の処理により，代理実行する間，保護対象 VM に CPU を割り当てて他プロセスに動作する機会を与えることができる．また，代理プロセスによる代理実行の結果を重要プロセスに返却できる．

1 つの保護対象 VM 上で重要プロセスが複数動作する場合，VMM がある重要プロセスのシステムコール情報を作成した後，保護対象 VM に制御を戻したとき（図 3 (A)）に，別の重要プロセスが代理実行する対象のシステムコールを発行する可能性がある．この場合，2 つ目のシステムコール情報を作成して代理実行する必要がある．しかし，これを実現するためには，1 つの保護対象 VM から得られた複数のシステムコール情報を管理する方法や，複数の代理実行を効率的に行う仕組みが必要となる．このため，こ

れらの対処は今後の課題とする．

4. 代理実行の依頼取得における遅延削減

4.1 依頼取得における遅延

代理プロセスは，代理実行の依頼の有無を VMM に対してポーリングする．つまり，代理プロセスによる依頼取得は，最大でポーリング周期分の遅延が発生する．この遅延により，代理プロセスの応答時間が増大し，重要プロセスのシステムコールオーバヘッドが増加する．その結果，重要プロセスの性能が低下するため，依頼取得による遅延を削減する必要がある．

対処として，次の手法を採用した．

(1) 依頼取得のためのポーリングの廃止
あらかじめ代理プロセスを休眠させておき，VMM によるシステムコール情報作成を契機に代理プロセスを起床させ，依頼取得させる．

この対処により，依頼取得による遅延を削減し，システムコールオーバヘッドを削減できる．代理プロセスの起床には，イベントチャネルを用いる．

4.2 イベントチャネル

イベントチャネル [8] とは，Xen と VM 間でのイベント通知を実現する機構である．Xen では，物理 IRQ，仮想 IRQ，および IPI は “イベント” として抽象化され，イベントは，Xen–VM 間または VM–VM 間で確立したイベントチャネルを介して配信される．イベントが配信されたとき，VM で動作する Linux がイベントを受信し，イベントを待機しているプロセスを起床させる．

この仕組みを利用して VMM から代理プロセスへ代理実行の依頼を通知する．具体的には，代理実行の依頼を示すイベントを 1 つ新たに定義し，このイベントを受信するためのイベントチャネルを確立した代理プロセスをあらかじめ休眠させておく．VMM はシステムコール情報を作成した後，代理プロセスにイベントを通知して起床させる．これにより，代理実行の依頼取得における遅延を削減できる．

4.3 イベントチャネルを用いた依頼取得

図 4 にイベントチャネルを用いた依頼取得について示し，以下で説明する．従来手法の処理流れに処理 (A)，(B)，(C)，および (D) を追加した．

(A) 代理プロセスは，代理実行の依頼を示すイベントを受信するためのイベントチャネルを確立し，通知待ちのための WAIT 状態になる．

(B) VMM は，システムコール情報を作成した後，代理プロセスに代理実行の依頼を通知するためにイベントを配信する．

(C) 代理プロセスは，イベント配信を契機に起床し，RUN 状態になる．その後，代理実行処理 (5)，(6)，(7) を行う．

(D) 代理実行後，代理プロセスは通知待ちのため再度 WAIT 状態になる．

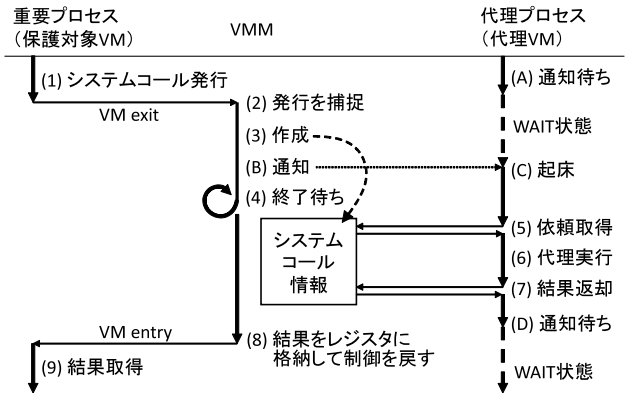

図 4 イベントチャネルを用いた依頼取得
Fig. 4 Acquisition of the request by using the event-channel mechanism.

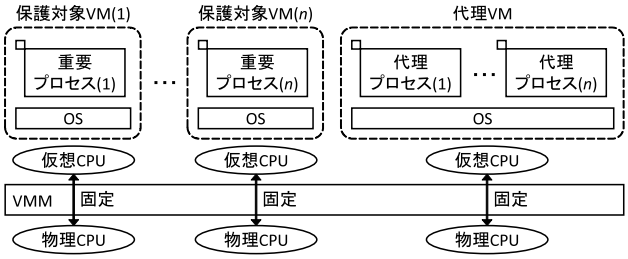

図 5 評価モデル
Fig. 5 Model of the evaluation.

上記の処理により，代理プロセスは，イベント通知により起床して代理実行の依頼を取得できるため，依頼取得における遅延を削減できる．なお，保護対象 VM が複数ある場合は，保護対象 VM ごとに代理プロセスを用意し，それぞれに異なるイベントチャネルを確立させる．これにより，各代理プロセスはそれぞれに対応する通知を受信できる．

5. 評価

5.1 内容と環境

提案した 2 つの対処の有効性を示すために以下を評価した．

保護対象 VM の停止回避の効果

(1) 保護対象 VM の停止時間

(2) 他プロセスの処理性能

(3) `sched_yield()` のオーバヘッド

代理実行の依頼取得における遅延削減の効果

(4) 代理プロセスによる依頼取得の応答時間

(5) 複数の代理プロセスによる代理実行の平等性

2 つの対処を適用した際の効果

(6) ベンチマークを用いた性能評価

評価モデルを図 5 に示す．本評価では，保護対象 VM を n 台動作させ，各保護対象 VM 上で動作する重要プロセ

表 1 評価環境
Table 1 Environment of the evaluation.

計算機 A		
CPU	Intel Core i7-2600（3.4 GHz，4 コア）	
メモリ	8 GB	
VMM	Xen 4.2.3	
	保護対象 VM（$\times n$）	代理 VM
仮想 CPU	1 コア	1 コア
メモリ	1 GB	$8-n$ GB
OS	Debian 7.3（Linux 3.2.0，64 bit）	
計算機 B		
CPU	Intel Core i7-4790（3.6 GHz，4 コア）	
メモリ	8 GB	
OS	Debian 9.0（Linux 4.9.0，64 bit）	

スの数を 1 とする．代理 VM 上では重要プロセスと同数，つまり，n 個の代理プロセスを走行させる．なお，重要プロセス i（$1 \leq i \leq n$）の代理実行は，代理プロセス i が行う．また，各 VM が持つ仮想 CPU の数を 1 とし，それぞれに異なる 1 つの物理 CPU コアを固定で割り当てる．これは，他の VM の影響を排除するためである．

評価環境を**表 1** に示す．計算機 A では，図 5 に示した構成で保護対象 VM と代理 VM が動作する．計算機 B では，保護対象 VM 上の重要プロセスが VM 外のプロセスと通信する際の通信相手となるプロセスが動作する．

重要プロセスが通信する際，通信を実行するスレッドは 1 つとする．従来手法と提案手法は，重要プロセスが複数スレッド（各スレッドが通信）で構成する場合に対応していないためである．この対応は，今後の課題とする．

5.2 保護対象 VM の停止回避の効果

5.2.1 保護対象 VM の停止時間

保護対象 VM の停止回避により削減できる保護対象 VM の停止時間（以降，VM 停止時間）を測定し，保護対象 VM の停止回避の効果を示す．以下に VM 停止時間の測定方法を示す．

(1) 評価モデル（図 5）における $n = 1, 2, 3$ の環境を使用する．
(2) 重要プロセスは，`sendto()` システムコールを 1,000 回連続で発行する．このとき，代理実行する間に VMM が保護対象 VM の CPU を使用して動作した時間を測定し，これを VM 停止時間とする．なお，`sendto()` の送信データのサイズを 1 KB，2 KB，3 KB，および 4 KB とした．
(3) 計算機 B では，`recvfrom()` システムコールを繰り返し発行する受信プロセスが n 個動作する．
(4) 代理プロセスが代理実行の依頼の有無をポーリングすること（図 1 (5)）による VM 停止時間への影響を取り除くために，ポーリング周期を 0 秒とする．

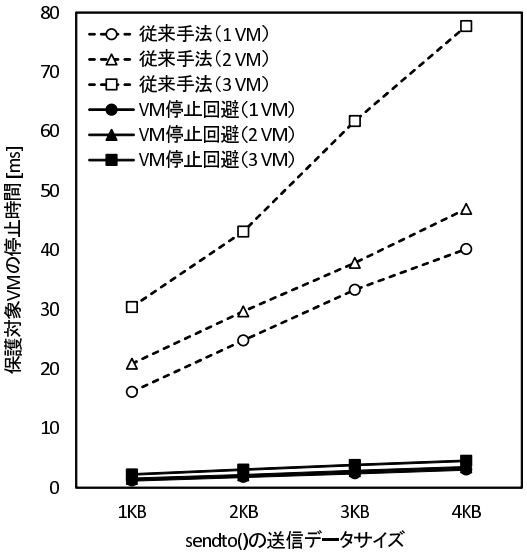

図 6 `sendto()` を 1,000 回代理実行した際の保護対象 VM の停止時間
Fig. 6 Downtime of the protection target VM for proxying `sendto()` 1,000 times.

測定結果を**図 6** に示し，以下で説明する．

(1) 測定したすべての送信データサイズと VM 数において，保護対象 VM の停止回避により VM 停止時間を削減できている．これは，従来手法では VMM が代理実行の終了待ちをする（図 2 (4)）一方で，保護対象 VM の停止回避では VMM から保護対象 VM に制御を戻し（図 3 (4′)），結果返却の確認時のみ VMM が動作するためである．
(2) 測定したすべての VM 数において，送信データサイズが大きいほど，保護対象 VM の停止回避により削減した VM 停止時間は大きい．これは，保護対象 VM の停止回避により削減できる VM 停止時間が代理プロセスの動作する時間に依存し，代理プロセスによる `sendto()` の実行時間が送信データサイズに依存するためである．なお，保護対象 VM の停止回避により削減した VM 停止時間の割合は，どの送信データサイズにおいても約 90%である．また，VM 停止回避は従来手法より送信データサイズが増加したときの VM 停止時間の増加の傾きが小さい．これは，送信データの複写回数の影響である．従来手法では，送信データの複写（システムコール情報作成時，および代理実行時）が VM 停止時間に含まれる．一方，VM 停止回避では，システムコール情報作成時の複写のみが VM 停止時間に含まれる．このため，VM 停止回避は従来手法より傾きが小さい．
(3) 測定したすべての送信データサイズにおいて，動作する保護対象 VM の数の増加にともない，保護対象 VM の停止回避により削減できる VM 停止時間は増加する．これは，代理 VM 上で動作する代理プロセスの数

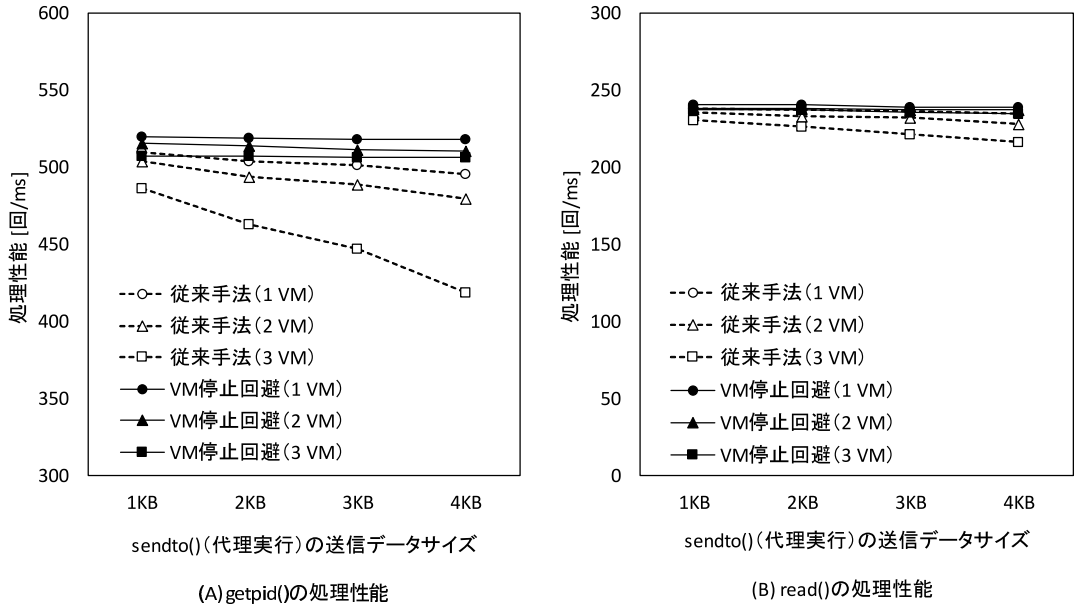

図 7 他プロセスが発行するシステムコールの処理性能
Fig. 7 Performance of system calls invoked by the other process.

が増加したことにより，各代理プロセスに待ち時間が発生し，各代理プロセスの依頼取得が遅延したためである．また，VM 停止回避を適用した場合において，保護対象 VM の数の増加にともなって VM 停止時間がわずかに増加する．これは，代理プロセスの実行時間が増加したことで，重要プロセスによる `sched_yield()` の実行回数が増加したためである．

5.2.2 他プロセスの処理性能

保護対象 VM の停止回避により VM 停止時間を削減した場合における他プロセスの処理性能への影響を示す．測定する項目は以下の 2 つである．

(1) `getpid()` の処理性能
(2) `read()` の処理性能

これら 2 つの処理性能を調べる理由を以下に示す．完全仮想化された VM 上で `read()` が発行され，外部記憶装置へのアクセスが発生すると，Domain-0 と呼ばれる VM で，外部記憶装置へのアクセスをエミュレートする処理が実行される．また，従来手法では，代理実行のオーバヘッドを小さくするために，Domain-0 を代理 VM として使用している．このため，保護対象 VM や代理 VM の動作が複雑化し，`read()` の処理性能の評価だけでは，保護対象 VM の停止回避による効果の分析が難しい．以上のことから，処理内容が単純で保護対象 VM 内で処理が完結する `getpid()` の処理性能を示した後に `read()` の処理性能を示す．

測定方法を以下に示す．

(1) 評価モデル（図 5）における $n = 1, 2, 3$ の環境を使用する．
(2) `getpid()` や `read()` の性能を測定するプロセス（以降，測定プロセス）は，保護対象 VM (1) 上で測定対象のシステムコールを 10,000 回発行し，1 ms あたりのシステムコール発行回数を測定する．重要プロセスは，測定プロセスより前に起動し，測定プロセスが動作する間，`sendto()` の発行を繰り返す．
(3) `read()` の測定では，ディスク（RAW デバイス）上の 4 KB 境界のランダムな位置から 4 KB のデータを読み込む．
(4) 代理プロセスのポーリング周期は 0 ms とする．

測定結果を図 7 に示し，以下で説明する．

(1) VM 停止回避は，従来手法に比べ処理性能が高い．これは，保護対象 VM の停止回避により，測定プロセスの実行時間が長くなったことによる．
(2) いずれの場合も，`sendto()` の送信データサイズを大きくすると，処理性能は低下する．これは，送信データの複写（システムコール情報作成時，および代理実行時）によるものである．従来手法では，このすべての複写が `sendto()` 処理時間の長大化を招くため，処理性能の低下が大きい．一方，VM 停止回避では，このすべての複写が `sendto()` 処理時間の長大化を招くものの，保護対象 VM の停止回避によりシステムコール情報作成時の複写のみが測定プロセスの実行時間に影響を与えるため，処理性能の低下は少ない．
(3) いずれの場合も，保護対象 VM 数が多くなると，処理性能は低下する．これは，代理 VM 上の代理プロセス数の増加により，`sendto()` の処理時間が長くなるためである．従来手法では，sendto() の処理時間の長大化により，処理性能の低下が大きい．一方，VM 停止回避では，sendto() の処理時間が長大化しても，保護対象 VM の停止回避により測定プロセスの実行時間への影響が少ないため，処理性能の低下は少ない．
(4) `read()` は，`getpid()` に比べ OS 処理量が多いため，

表 2 sched_yield() のオーバヘッド（μs）
Table 2 Overhead of sched_yield() (μs).

	適用前	適用後	オーバヘッド
実行時間	0.15	4.40	4.25

処理性能は高くない．しかし，上記の 3 つの事項については同様な特徴を持つ．

5.2.3 sched_yield() のオーバヘッド

保護対象 VM の停止回避における対処により，sched_yield() の終了時に代理実行の結果返却の有無を確認する処理を追加したため，保護対象 VM 内で実行される sched_yield() にオーバヘッドが生じる．**表 2** に sched_yield() のオーバヘッドを示す．表 2 より，保護対象 VM の停止回避を適用した際，sched_yield() に 4.25 μs のオーバヘッドが生じることが分かる．オーバヘッドのほとんどは，保護対象 VM と VMM 間のモード遷移である．これは，sched_yield() の終了処理を捕捉した際に VMM が行う処理が，重要プロセスか否かを判定する処理，結果返却の有無を確認する処理，および結果を保護対象 VM のレジスタやメモリに格納する処理であり，他プロセスが sched_yield() を発行する際は，VM と VMM 間のモード遷移と VMM による重要プロセスか否かを判定する処理が実行されるためである．なお，この sched_yield() のオーバヘッドは，このシステムコールを発行するアプリケーションの性能を低下させる．しかし，Linux 上の多くの応用プログラムは，sched_yield() をほとんど発行しない．たとえば，Web サーバ（Apache2）は，sched_yield() を発行しない．また，sched_yield() を発行する \bin 以下のコマンドは，140 個中 1 個だけであった．このため，sched_yield() のオーバヘッドが保護対象 VM へ与える影響は小さいといえる．

5.3 代理実行の依頼取得における遅延削減の効果

5.3.1 代理プロセスによる依頼取得の応答時間

従来手法の依頼取得（ポーリング）とイベントチャネルを用いた依頼取得におけるシステムコールのオーバヘッドを評価することにより，依頼取得における応答時間の短縮効果を示す．以下にオーバヘッドの測定方法を示す．

(1) 評価モデル（図 5）における $n = 1$ の環境を使用する．
(2) オーバヘッド測定の対象とするシステムコールとして，socket()，close()，および sendto() を用いる．なお，sendto() の送信データのサイズを 1 KB とした．
(3) 代理プロセスは，代理実行の依頼取得と結果返却を行い，代理実行としてのシステムコールは発行しない．これにより，重要プロセスにおいてシステムコールの発行前後で取得したタイムスタンプの差から代理実行によるオーバヘッドを算出できる．
(4) 従来手法におけるポーリング周期を 1 ms とする．これは，代理プロセスが複数走行する場合における代理実行の平等性の観点から，ポーリング周期を 1 ms 以上にする必要があるためである．
(5) 重要プロセスは，ランダムな時間（5–10 ms）の sleep() と測定対象のシステムコールの発行を交互に 1,000 回ずつ行う．これは，オーバヘッドが代理プロセスによる依頼取得周期に依存しないようにするためである．
(6) 保護対象 VM の停止回避は適用しない．これは，保護対象 VM の停止回避の適用による影響を取り除くことで，イベントチャネルを用いた依頼取得の効果をより明確にするためである．

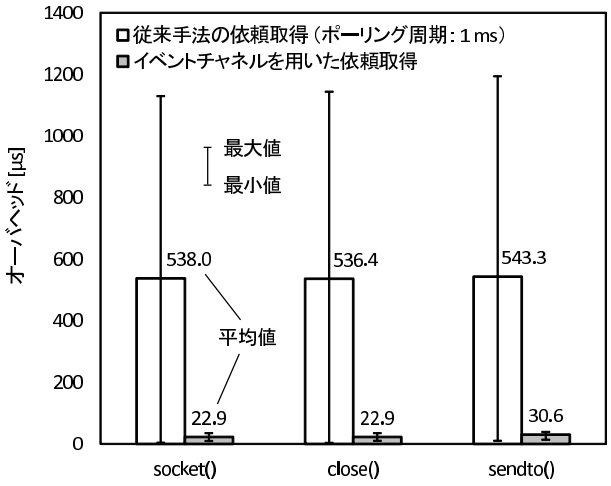

図 8 依頼取得におけるシステムコールオーバヘッド
Fig. 8 System call overhead in each method for acquiring the request.

測定結果を**図 8** に示し，以下で説明する．棒の高さは平均値を示し，棒に重なる縦線の上端と下端は，それぞれ最大値と最小値を示す．

(1) 測定対象のシステムコールにおいて，イベントチャネルを用いた依頼取得におけるオーバヘッドの平均値は，従来手法における依頼取得より小さい．これは，従来手法の場合，ポーリングによる sleep() が終了するまで代理プロセスが依頼取得できないことに対し，イベントチャネルを用いた場合，VMM からのイベント通知により即座に代理プロセスが起床して依頼取得するためである．平均値に着目すると，イベントチャネルを用いた依頼取得を従来手法に適用することにより，依頼取得の応答時間を約 95%短縮できることが分かる．

5.3.2 複数の代理プロセスによる代理実行の平等性

代理 VM に複数の代理プロセスが存在する場合において，各代理プロセスの動作を分析することにより，複数の代理プロセスによる代理実行の平等性を明らかにする．従来手法における代理実行の平等性は，文献 [6] で評価済である．本評価では，イベントチャネルを用いた依頼取得をする場合における代理実行の平等性を示し，従来手法との

違いを明らかにする．以下に測定方法を示す．

(1) 評価モデル（図 5）における $n = 3$ の環境を使用する．
(2) 重要プロセスは，`socket()` の発行と `close()` の発行を交互に 1,000 回ずつ行う．
(3) 代理プロセスは，代理実行としてのシステムコールを発行しない代わりに T_e だけのビジーループを行う疑似的なシステムコールを発行する．また，従来手法におけるポーリング周期を T_s とする．これは，システムコールの実行時間（T_e）とポーリング周期（T_s）が代理実行の平等性に与える影響を分析するためである．なお，T_e と T_s がともに 1 ms の場合は，文献 [6] での評価結果より平等に代理実行できることは明らかであるため除外する．また，イベントチャネルを用いた依頼取得では，ポーリングしないため，T_s は考慮しない．
(4) 代理プロセスは，代理実行の依頼取得の直後にタイムスタンプを取得する．3 つの代理プロセスから得られたタイムスタンプを時系列にプロットすることで，各代理プロセスの動作を示す．

測定結果を**図 9** に示す．図 9 (A) は従来手法の依頼取得（ポーリング）の場合，図 9 (B) はイベントチャネルを用いた依頼取得の場合の各代理プロセスの動作を示す．なお，各代理プロセスで取得したタイムスタンプは + マーカで表

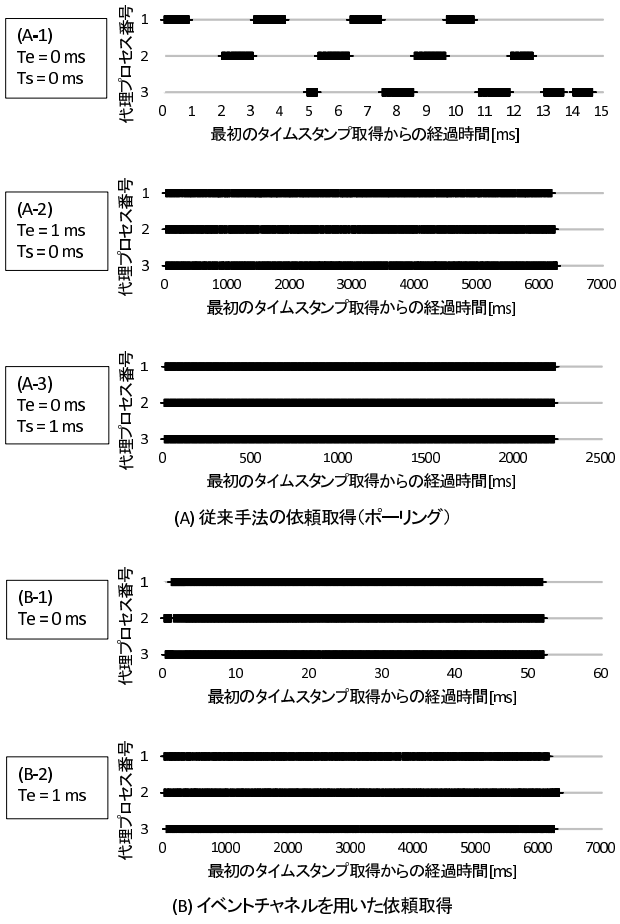

図 9 代理プロセスによる代理実行の平等性
Fig. 9 Equality of the proxy execution by proxy processes.

示している．図 9 より，以下のことが分かる．

(1) (A-1) では，各代理プロセスの代理実行処理を平等に実行できていない．グラフ上に短い棒が複数あるように見えるのは，部分的に集中して代理実行が行われているためである．また，代理実行が約 1 ms 行われた後に他の代理プロセスに処理が切り替わっていることから，各代理プロセスはタイムスライスに従って OS のスケジューラにより切り替えられたと考える．つまり，$T_s = 0$ ms かつ代理実行処理が短時間（< 1 ms）で完了する場合，他の保護対象 VM の代理実行処理に影響を与えることが分かる．
(2) (A-2)，(A-3)，(B-1)，および (B-2) では，各代理プロセスの代理実行処理を平等に実行できている．各グラフ上に 3 本の棒があるように見えるのは，交互に代理プロセスのタイムスタンプが並んでいるためである．従来手法（T_e または T_s が 1 ms）とイベントチャネルによる依頼取得でのみ平等に代理実行できるのは，代理実行処理や待ちによりプロセス切替えが起こるためである．

上記より，ポーリングによる依頼取得では各代理プロセスの代理実行処理を平等に実行できない条件下においても，イベントチャネルを用いた依頼取得では平等に代理実行できる．

5.4 ベンチマークを用いた性能評価

ベンチマークとして `iperf` [9]（無改変）を用いて重要プロセスのネットワーク性能（スループット）を示す．測定は次の 4 つの条件で実施する．(A) 提案手法と (B) 従来手法では，それぞれ `iperf` を重要サービスとして指定し，システムコールを代理実行させた際の性能を測定する．提案手法とは，“保護対象 VM の停止回避” と “依頼取得における遅延削減” の対処を従来手法に適用した手法である．(C) 代理実行なしでは，提案手法において，`iperf` を重要サービスとして指定せずに測定する．これは，重要サービスではない場合のオーバヘッドを測定するためである．代理実行なしにおいて，システムコールは，VMM が捕捉するものの，`iperf` は重要サービスでないため，保護対象 VM で実行される．(D) Vanilla 環境では，無改変の Xen で VM を動作させ，`iperf` の性能を測定する．これは，提案手法による動作がない場合の性能をベースラインとするためである．

以下に測定方法を示す．

(1) 評価モデル（図 5）における $n = 1, 2, 3$ の環境を使用する．
(2) `iperf` は，保護対象 VM 上でクライアントモード実行し，計算機 B 上でサーバモード実行する．また，バッファサイズは 4 KB とし，通信プロトコルは TCP/IP とする．

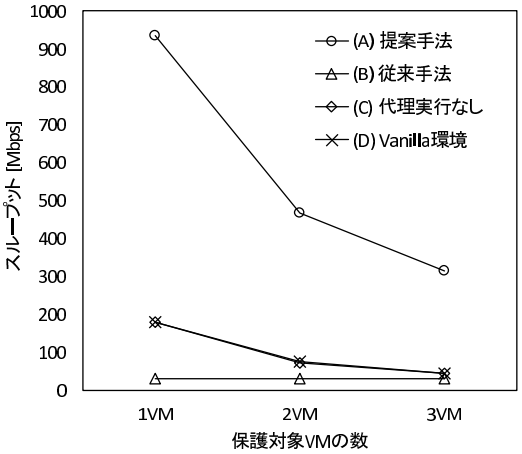

図 10 `iperf` ベンチマークの性能（スループット）
Fig. 10 Performance of `iperf` benchmark (throughput).

(3) 従来手法における代理プロセスのポーリング周期は 1 ms とする．

測定結果を**図 10** に示し，以下で説明する．

(1) 提案手法のスループットは最も高い．従来手法より高い理由は，依頼取得における遅延削減により，ポーリングする際の 1 ms の WAIT がなくなったためである．代理実行なしより高い理由は，代理実行なしでは入出力を Domain-0 によりエミュレートする一方で，提案手法では代理実行することによりエミュレートの必要がないためである．Vanilla 環境より高い理由も同様である．

(2) 提案手法，代理実行なし，および Vanilla 環境では，保護対象 VM の数の増加にともない，スループットは低下する．これは，代理プロセスの数が増加したことにより，代理 VM の負荷が増加し，処理ネック（PU ネックおよび NIC ネック）が発生したためである．また，従来手法では，保護対象 VM の数にかかわらず，スループットは一定である．これは，ポーリングする際の 1 ms の WAIT により，代理プロセスの数が増えても処理ネックが発生しにくいためである．なお，5.2.2 項の評価では，重要プロセスが `sendto()` を連続発行するため，VM 数が多いと処理ネックが発生している．一方，`iperf` は，重要プロセスに比べデータ送信間隔が大きいため，処理ネックが発生しにくい．

(3) 代理実行なしと Vanilla 環境のスループットはほぼ同じである．これは，`iperf` によるシステムコール実行時間に比べ，代理実行なしの場合に VMM が介入する時間が非常に小さいためである．このことから，提案手法において，重要サービスではない場合のオーバヘッドは小さいといえる．

6. 関連研究

システムコールを代理実行する手法として，ProxyOS [10] や Shadow Context [11] がある．ProxyOS は，アプリケーションと OS のインタフェースであるシステムコールの信頼性をアプリケーション側で設定できる OS である．アプリケーションが信頼できないと設定したシステムコールは，アプリケーションが動作する VM とは別に用意した信頼できる VM 上で実行される．ProxyOS は，システムコールを代理実行するために，OS のソースコードを変更している．Shadow Context は，信頼できる VM 上で動作する `ps` や `lsmod` などの調査プログラムが発行したシステムコールを VMM により捕捉し，信頼できない VM 上の任意のプロセスにシステムコールを実行させる．これにより，信頼できない VM とは別の信頼できる VM 上で調査プログラムを実行させながら信頼できない VM を調査できるため，調査プログラムを信頼できない VM 上で実行することを回避できる．Shadow Context は，信頼できない VM を調査するために，調査プログラムが動作する OS のソースコードを変更している．これらの手法とは異なり，従来手法は，攻撃者から従来手法の存在を検知されることを避けるために，OS や重要プロセスのソースコードを変更しておらず，VMM と代理プロセスにより実現している．また，保護対象 VM の停止回避と依頼取得の遅延削減は，従来手法の改良版であり，従来手法と同様に OS やアプリケーションを改変せず，VMM や代理プロセスの改良により実現している．

OS の性能低下を回避する手法として，PicoDriver [12] がある．PicoDriver は，最小限のデバイスドライバ機能を軽量カーネルに実装し，残りのデバイスドライバ機能を Linux カーネルから利用するデバイスドライバである．これにより，軽量カーネルの性能低下を回避しつつ，軽量カーネルにデバイスドライバを適用することができる．PicoDriver は，デバイスドライバを分割することにより軽量カーネルの性能低下を回避している一方で，保護対象 VM の停止回避では，保護対象 VM に制御を戻してプロセスが動作する機会を与えることにより，保護対象 VM 上の OS の性能低下を回避している．

VM 内のプログラムコードを利用する手法として，Virtuoso [13] がある．Virtuoso は，VM 内のプログラムの実行をトレースしてプログラムコードを収集し，VMM 内で動作する VM 監視ツールを自動生成する．これにより，OS の内部動作に関する知識なしに VM 監視ツールを生成できる．Virtuoso は，VM 内のプログラムコードをもとに VM 監視ツールを自動生成するという形で VM 内のプログラムコードを利用する一方で，保護対象 VM の停止回避では，重要プロセスが `sched_yield()` システムコールを発行したかのように保護対象 VM を動作させるという形で VM 内のプログラムコードを利用する．

7. おわりに

重要サービスの動作を不可視化する手法について，システムコールの代理実行を効率化する 2 つの対処を述べた．1 つは保護対象 VM の停止回避であり，システムコールを代理実行する間，あたかも重要サービスが CPU 使用権を譲るシステムコールを発行したかのように保護対象 VM を動作させる．これにより，重要サービス以外のプロセスに動作する機会を与える．もう 1 つは代理実行の依頼取得における遅延削減であり，VMM のイベント通知機構を利用して代理実行の依頼取得を行う．これにより，代理実行の依頼取得における遅延を削減し，代理プロセスの応答時間を短縮する．

評価では，保護対象 VM の停止回避により，重要プロセスが発行したシステムコールを代理実行する間に保護対象 VM が停止する時間を従来手法に比べて約 90%削減できることを示した．また，これにより，保護対象 VM 上で動作する他プロセスの処理性能が向上することを示した．さらに，代理実行の依頼取得における遅延削減により，代理プロセスによる依頼取得の応答時間を従来手法に比べて約 95%短縮できることを示した．また，これにより，代理プロセスが複数動作する場合の平等実行を可能にすることを示した．

謝辞 本研究の一部は，JSPS 科研費 18K18051 の助成を受けたものです．

推薦文

システムコールの代理実行において保護対象 VM が停止してしまう問題を解決するために，重要プロセスの代理実行時に保護対象 VM を止めない新しい方法を提案している．保護対象 VM 停止を回避するための手法とその有効性をしっかりと述べている．論文として良くまとまっており，設定した問題を解決していることが示されている．また，性能評価が詳細に行われていることも高く評価できる．以上のことから，推薦論文に推薦いたします．

（コンピュータセキュリティシンポジウム 2019 プログラム委員長 國廣 昇）

参考文献

[1] Min, B. and Varadharajan, V.: A Novel Malware for Subversion of Self-protection in Anti-virus, *Softw. Pract. Exper.*, Vol.46, No.3, pp.361–379 (2016).
[2] Stealth: A new Adore root kit, available from ⟨http://lwn.net/Articles/75990/⟩ (accessed 2018-01-19).
[3] Hsu, F.-H., Wu, M.-H., Tso, C.-K., Hsu, C.-H. and Chen, C.-W.: Antivirus Software Shield Against Antivirus Terminators, *IEEE Trans. Information Forensics and Security*, Vol.7, No.5, pp.1439–1447 (2012).
[4] Srinivasan, D., Wang, Z., Jiang, X. and Xu, D.: Process Out-grafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring, *Proc. 18th ACM Conference on Computer and Communications Security*, pp.363–374 (2011).
[5] Sato, M., Taniguchi, H. and Yamauchi, T.: Design and Implementation of Hiding Method for File Manipulation of Essential Services by System Call Proxy using Virtual Machine Monitor, *International Journal of Space-Based and Situated Computing*, Vol.9, No.1, pp.1–10 (2019).
[6] Okuda, Y., Sato, M. and Taniguchi, H.: Implementation and Evaluation of Communication-Hiding Method by System Call Proxy, *International Journal of Networking and Computing*, Vol.9, No.2, pp.217–238 (2019).
[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles*, pp.164–177 (2003).
[8] Xen: Event Channel Internals, available from ⟨https://wiki.xen.org/wiki/Event_Channel_Internals⟩ (accessed 2019-10-10).
[9] SourceForge.net: iperf2, available from ⟨https://sourceforge.net/projects/iperf2/⟩ (accessed 2020-03-03).
[10] Ta-Min, R., Litty, L. and Lie, D.: Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable, *Proc. 7th Symposium on Operating Systems Design and Implementation*, pp.279–292 (2006).
[11] Wu, R., Chen, P., Liu, P. and Mao, B.: System Call Redirection: A Practical Approach to Meeting Real-World Virtual Machine Introspection Needs, *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp.574–585 (2014).
[12] Gerofi, B., Santogidis, A., Martinet, D. and Ishikawa, Y.: PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems, *Proc. 27th International Symposium on High-Performance Parallel and Distributed Computing*, pp.2–13 (2018).
[13] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J. and Lee, W.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection, *2011 IEEE Symposium on Security and Privacy*, pp.297–312 (2011).

奥田 勇喜 （学生会員）

2018 年岡山大学工学部情報系学科卒業．同年同大学大学院自然科学研究科博士前期課程入学．オペレーティングシステム，仮想化技術に興味を持つ．CSS2019 学生論文賞受賞．

佐藤 将也 （正会員）

2010 年岡山大学工学部情報工学科卒業．2012 年同大学大学院自然科学研究科博士前期課程修了．2014 年同大学大学院自然科学研究科博士後期課程修了．2013 年日本学術振興会特別研究員（DC2）．現在，岡山大学大学院自然科学研究科助教．博士（工学）．コンピュータセキュリティ，仮想化技術に興味を持つ．2012 年度情報処理学会論文賞受賞．電子情報通信学会会員．

谷口 秀夫 （正会員）

1978 年九州大学工学部電子工学科卒業．1980 年同大学大学院修士課程修了．同年日本電信電話公社電気通信研究所入所．1987 年同所主任研究員．1988 年 NTT データ通信株式会社開発本部移籍．1992 年同本部主幹技師．1993 年九州大学工学部助教授．2003 年岡山大学工学部教授．2010 年岡山大学工学部長．2014 年岡山大学理事・副学長．博士（工学）．オペレーティングシステム，実時間処理，分散処理に興味を持つ．著書『並列分散処理』（コロナ社）等．電子情報通信学会，ACM 各会員．本会フェロー．