Vol.2018-OS-143 No.2
2018/5/21

VMMを用いて重要サービスの通信操作を不可視化する通信処理制御法

奥田 勇喜[1]　佐藤 将也[1]　谷口 秀夫[1]

概要：多くの計算機では，システムの管理や保護のために，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が動作している．重要サービスは，攻撃者にとって不都合な場合があることから，攻撃の対象となり，無効化される可能性がある．攻撃への対策として，重要サービスの動きを攻撃者に見せない，つまり不可視化する技術がある．この技術により，攻撃者による重要サービスの特定を困難化でき，攻撃を回避できる．本稿では，重要サービスの通信処理に着目し，VMM を用いて重要サービスの通信操作を不可視化する通信処理制御法を述べる．

1. はじめに

攻撃防止や攻撃の被害を抑制するためにセキュリティソフトウェアが研究開発されている．また，攻撃の検知や証拠保全のためにプログラムの動作を記録するログ収集ソフトウェアが利用されている．ここでは，システムの保護や管理を行うこれらのソフトウェアを重要サービスと呼ぶ．

重要サービスは，攻撃のために作成され利用者や計算機に不正または有害な動作を行うソフトウェア（以降，マルウェア）にとって不都合な場合があることから，攻撃対象となり，無効化される可能性がある [1][2][3]．セキュリティソフトウェアが無効化されると，マルウェアへ対策できず，被害を受ける可能性がある．また，ログ収集ソフトウェアが無効化されると，マルウェアのインストールなどのログが収集不可能となり，被害が拡大する可能性がある．

重要サービスへの攻撃の対策として，重要サービスの動きを攻撃者に見せない（不可視化する）技術がある．文献 [4] では，重要サービスのプロセス情報を偽の情報に置換することで，重要サービスのプロセス情報を攻撃者から不可視化する技術を提案している．また，文献 [5] では，重要サービスの関連ファイルを計算機の外部に配置し，関連ファイルへのファイルアクセスを重要サービスからのみ可能にすることで，重要サービスの関連ファイルを攻撃者から不可視化する技術を提案している．このように，重要サービスの動きを不可視化することで，攻撃者による重要サービスの特定を困難化し，重要サービスへの攻撃を回避できる．不可視化が必要な重要サービスの動きとして，重要サービスのプロセス情報とファイルアクセスに加え，重要サービスの通信操作がある．例えば，OS 処理のパケットキャプチャや監視により，重要サービスの通信を攻撃者に監視されると，通信内容をもとに重要サービスの存在を特定され，無効化される可能性がある．

本稿では，重要サービスの通信操作を不可視化する通信処理制御法を述べる．具体的には，重要サービスが発行した通信に関するシステムコールを捕捉し，代理プロセスにシステムコールを代理実行させる．これにより，重要サービスの通信操作を不可視化し，通信内容をもとにした重要サービスの特定を困難にする．

2. 通信処理制御法

2.1 要求

重要サービスの通信操作を不可視化する通信処理制御法に対して，次の要求がある．

（要求 1）重要サービスの通信操作を不可視化すること
（要求 2）重要サービスのプログラムを変更せずに通信処理制御法を実現すること

重要サービスの通信操作を攻撃者に監視されると，通信内容をもとにして重要サービスの存在が特定され，無効化される可能性がある．このため，重要サービスの通信操作を不可視化する（要求 1）必要がある．また，攻撃を回避するために重要サービスのプログラムを変更すると，重要サービスを攻撃内容に合わせて個別に変更しなくてはならない．この変更の工数は大きいため，重要サービスのプログラムを変更せずに通信処理制御法を実現する（要求 2）必要がある．

[1] 岡山大学 大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University

2.2 課題

（要求 1）と（要求 2）を満足するために，仮想化技術を用いる．具体的には，仮想計算機 (Virtual Machine，以降 VM) 上でオペレーティングシステム（Operating System，以降，OS）や重要サービスを動作させ，仮想計算機モニタ（VM Monitor，以降 VMM）のプログラムを変更することにより通信処理制御法を実現する．VMM は，VM の動作を監視するために有用であり，侵入検知やファイル保護にも利用されている [6][7]．この環境において，重要サービスによる通信操作を VMM により捕捉し，重要サービスが走行する VM とは異なる VM 上で走行する代理プロセスに通信操作を代理実行させる．これにより，重要サービスのプログラムを変更せずに重要サービスの通信操作を不可視化できる．

以上より，通信処理制御法の確立には，以下の課題に対処する必要がある．

（課題 1）重要サービスによる OS への通信要求を捕捉すること
（課題 2）捕捉した通信要求の内容を取得すること
（課題 3）取得した内容に基づき，通信操作を代理実行すること
（課題 4）代理実行の結果を重要サービスに返却すること

重要サービスによる通信操作を代理実行するためには，重要サービスによる OS への通信要求を捕捉する（課題 1）必要がある．次に，捕捉した通信要求の内容を取得（課題 2）し，取得した内容に基づき，代理プロセスにより通信操作を代理実行する（課題 3）必要がある．代理実行後は，代理実行の結果を代理プロセスから重要サービスへ返却する（課題 4）必要がある．

2.3 対処

2.3.1 想定する環境

本制御法では，VMM として Xen[8] を用い，VM 上で動作する OS として Linux を用いる場合を想定する．VMM 上では，VMM や他の VM を管理する VM（以降，管理 VM）が 1 つと，重要サービスが走行する VM（以降，保護対象 VM）が 1 つ以上動作する．また，保護対象 VM は Intel VT-x を用いて完全仮想化され，x86_64 命令セットを使用し，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．

2.3.2 基本構造

通信処理制御法の基本構造を**図 1** に示す．VMM 上に保護対象 VM と管理 VM を用意する．保護対象 VM の OS 上では，重要サービスを提供するプロセス（以降，重要プロセス）が走行し，管理 VM の OS 上では，システムコールを代理実行するプロセス（以降，代理プロセス）が走行する．この環境において，重要プロセスが発行するシステムコールを VMM により捕捉する [9][10]．VMM は，捕捉したシステムコールの引数をもとに，重要サービスによる OS への通信要求の内容を取得する．この内容を管理 VM 上の代理プロセスに転送し，システムコールの代理実行を依頼する．代理プロセスは，代理実行の依頼を受けてシステムコールを代理実行する．代理実行後，VMM を介して，代理実行の結果を重要プロセスに返却する．

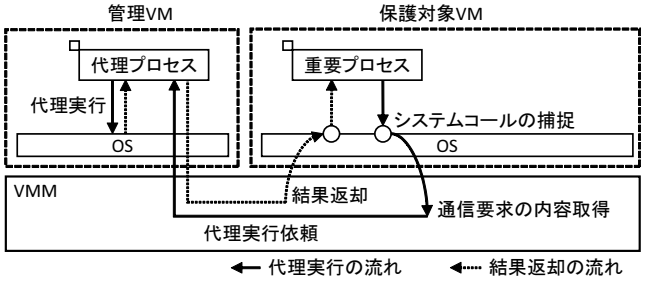

図 1 基本構造

2.3.3 対処内容

（課題 2）と（課題 3）については，図 1 の基本構造にすることにより，必要となる対処を VMM と管理 VM 上の代理プロセスで実現する．なお，本制御法は VMM を利用するため，本制御法を実現する上で以下の制約がある．

（制約）VM 上で動作する OS が制御しているプロセスについて，当該プロセスが発行するシステムコールを VMM により捕捉した場合，VMM 内で当該プロセスを「事象待ち」にできない．

この制約のため，VMM と代理プロセスの処理を以下のようにしている．

(1) VMM は，重要プロセスが発行したシステムコールを捕捉した際，システムコールの代理実行の終了をポーリングして確認する．このため，代理実行の終了まで継続して PU 処理を行う．
(2) 代理プロセスは，代理実行の依頼の取得を一定間隔（T_s）で行い，依頼内容を取得する．これにより，複数の代理プロセスを同時走行させ，システムコールの代理実行処理を平等に実行することができる．

（課題 1）への対処を以下に述べる．VMM により保護対象 VM におけるシステムコールを捕捉するために，システムコール開始処理の先頭アドレスにハードウェアブレークポイントを設定する．VMM は，保護対象 VM のデバッグレジスタを操作し，保護対象 VM 上で当該アドレスの命令が実行される直前にデバッグ例外が発生するように設定する．さらに，保護対象 VM の実行制御フィールドを操作し，デバッグ例外の発生により VM exit が発生するように設定する．これにより，保護対象 VM におけるシステムコールの発行を契機に VMM に処理が遷移し，システムコールを捕捉できる．

（課題 4）への対処を以下に述べる．代理プロセスは，システムコールの代理実行後，代理実行の結果を VMM に返

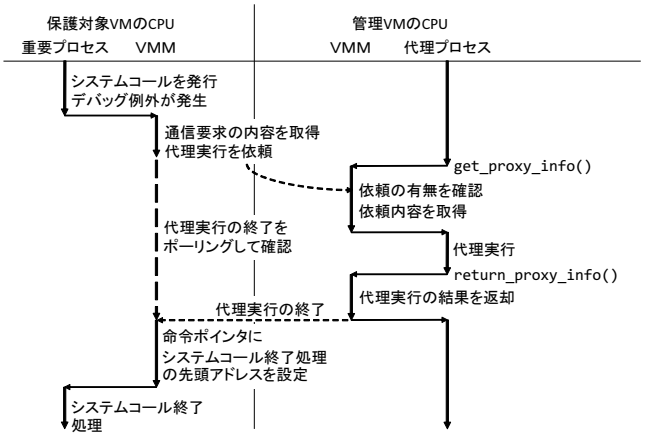

図 2　通信処理制御法の処理流れ

却する．VMM は，代理実行の返り値を保護対象 VM のレジスタに格納する．データを受信する read() や recv() などのシステムコールを代理実行した場合は，受信データを重要プロセスがシステムコールの引数で指定した領域に格納する．この後，命令ポインタにシステムコール終了処理の先頭アドレスを設定し，保護対象 VM に処理を返却する．これにより，保護対象 VM 上の OS 内でシステムコールを実行させずに，システムコール終了処理を実行できる．

2.3.4　通信処理制御法の処理流れ

通信処理制御法の処理流れを図 **2** に示す．通信処理制御法は，VMM と管理 VM 上の代理プロセスにより実現する．保護対象 VM と管理 VM は，それぞれ異なる CPU 上で動作し，VMM への遷移の契機はそれぞれ異なる．保護対象 VM では，デバッグ例外が発生したことによる VM exit により VMM に遷移する．管理 VM では，ハイパーコールの発行により VMM に遷移する．ハイパーコールとは，Xen における VM と VMM 間のインタフェースである．なお，本制御法を実現するために，新たに 2 つのハイパーコールを追加した．1 つは，代理実行の依頼の有無を確認する `get_proxy_info()` ハイパーコールで，もう 1 つは，代理実行の結果返却を行う `return_proxy_info()` ハイパーコールである．これらを用いることにより，VMM 間で代理実行の依頼や代理実行の終了を通知できる．VMM の処理流れと代理プロセスの処理流れは，それぞれ 2.3.5 項と 2.3.6 項で述べる．

2.3.5　VMM の処理流れ

VMM の処理流れを図 **3** に示し．以下に説明する．

(1) 保護対象 VM におけるシステムコールの発行により VM exit が発生し，VMM に処理が遷移する．VMM に処理が遷移した際に，VM exit の発生原因がデバッグ例外であり，かつ例外の発生したアドレスがシステムコール開始処理の先頭アドレスである場合，通信処理制御法の導入により発生したデバッグ例外として，システムコールの発行を検知する．

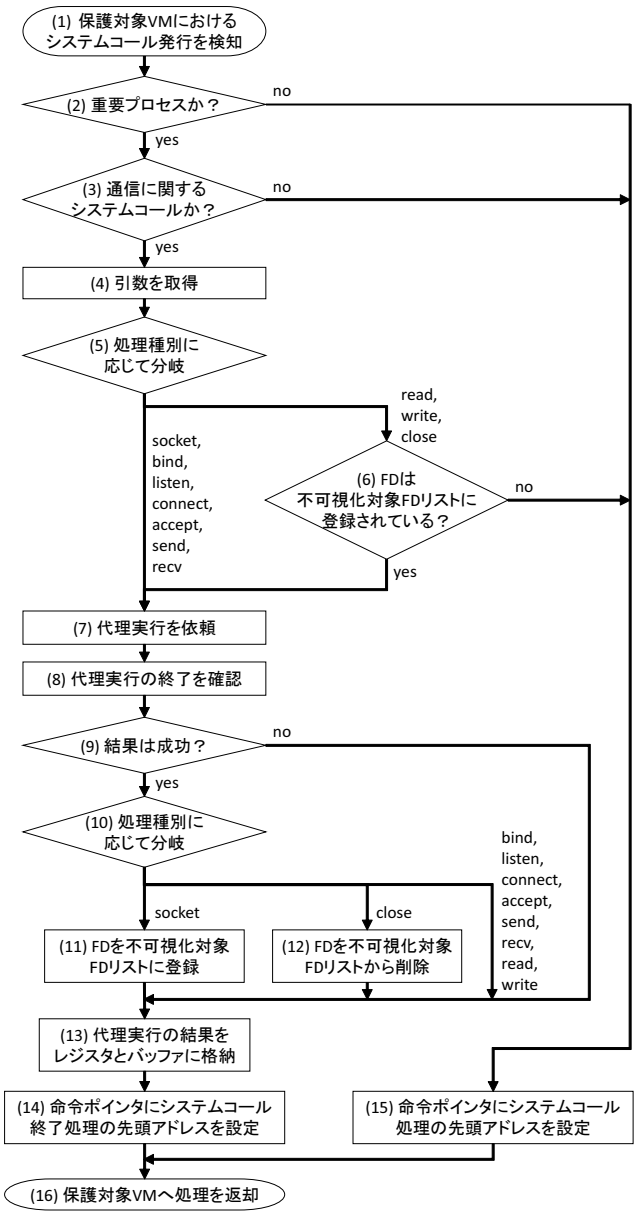

図 3　VMM の処理流れ

(2) システムコールの発行元のプロセスが重要プロセスであるか否かを判定し，重要プロセスである場合は (3) に，そうでない場合は (15) に移行する．これは，デバッグ例外を用いてシステムコールを捕捉すると，保護対象 VM 上で発行される全てのシステムコールが VMM により捕捉されるためである．

(3) システムコールの種類に応じて分岐する．本制御法では，通信に着目しているため，通信に関するシステムコール（2.3.8 項にて後述）である場合は (4) に，そうでない場合は (15) に移行する．

(4) 保護対象 VM のレジスタ値をもとシステムコールの引数を取得する．データを送信する send() や write() などのシステムコールを代理実行する場合は，送信データを引数で指定した領域から VMM へコピーする．本制御法では，VMM 内にシステムコールの番号や引数

などの代理実行に必要な情報（以降，代理実行情報）を格納する領域を保護対象 VM の数だけ用意する．これにより，代理実行情報を保護対象 VM ごとに保持できる．

(5) 通信に関するシステムコールのうち，read()，write()，および close() の場合は (6) に，それ以外の場合は (7) に移行する．

(6) 第 1 引数で与えられた FD の値が不可視化対象 FD リストに登録されている場合は (7) に，登録されていない場合は (15) に移行する．

(7) 代理プロセスに代理実行を依頼する．

(8) VMM は，代理実行の終了，つまり代理実行の結果の返却をポーリングして確認し，終了を確認すると (9) に移行する．

(9) 代理実行の結果が成功の場合は (10) に，失敗の場合は (13) に移行する．

(10) システムコールの種類別に分岐する．socket() の場合は (11) に，close() の場合は (12) に．それ以外の場合は (13) に移行する．

(11) socket() の返り値である FD を不可視化対象 FD リストに登録する．これは，通信に利用される FD を VMM により判別可能にするためである．

(12) close() の第 1 引数である FD を不可視化対象 FD リストから削除する．これは，(11) と同様に，通信に利用される FD を VMM により判別可能にするためである．

(13) 代理実行したシステムコールの返り値を保護対象 VM のレジスタに格納する．データを受信する read() や recv() などのシステムコールを代理実行した場合は，受信データを VMM から引数で指定した領域へコピーする．

(14) 命令ポインタにシステムコール終了処理の先頭アドレスを設定する．これは，保護対象 VM において，OS 内でシステムコール処理が実行されないようにするためである．これにより，保護対象 VM に処理が返却されたとき，システムコール終了処理が実行される．

(15) 保護対象 VM の命令ポインタにシステムコール処理の先頭アドレスを設定する．これにより，代理実行しない場合，保護対象 VM 上でシステムコールを実行できる．

(16) 保護対象 VM に処理を返却する．

2.3.6 代理プロセスの処理流れ

代理プロセスの処理流れを**図 4** に示す．代理プロセスの起動は，保護対象 VM の VMID （2.3.7 項にて後述）を指定して行う．これにより，複数の保護対象 VM が動作する場合，それぞれの保護対象 VM に対応する代理プロセスにより，保護対象 VM ごとに独立して代理実行できる．

(1) 代理実行の依頼の有無を確認する．この確認には，`get_proxy_info()` ハイパーコールを用いる．`get_proxy_info()` ハイパーコールは，代理実行の依頼がある場合は，代理実行情報を VMM から代理プロセスへコピーする（このとき，返り値は 0）．代理実行の依頼がない場合は，何も行わずに代理プロセスに処理を返す（このとき，返り値は −1）．

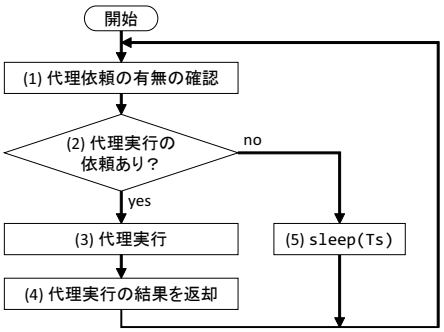

図 4 代理プロセスの処理流れ

(2) 代理実行の依頼があるか否かを判定する．この判定には，`get_proxy_info()` ハイパーコールの返り値を用いる．依頼がある場合は (3) に，依頼がない場合は (5) に移行する．

(3) `get_proxy_info()` ハイパーコールにより取得したシステムコールの番号や引数をもとに，システムコールを代理実行する．

(4) 代理実行の結果を VMM へ返却する．この返却には，`return_proxy_info()` ハイパーコールを用いる．`return_proxy_info()` ハイパーコールは，代理実行の結果を代理プロセスから VMM へコピーし，代理実行の完了を VMM に通知する．この後，(1) に移行する．

(5) 時間 T_s の WAIT 状態に入った後，(1) に移行する．これにより，一定間隔（T_s）で代理実行の依頼を取得できる．

2.3.7 代理実行情報

VMM は，保護対象 VM におけるシステムコールの発行を検知した際，代理実行情報を保護対象 VM から取得する．本制御法では，文献 [5] で述べられている代理実行情報から重要プロセスのカレントディレクトリ（通番 3）を除いた**表 1** に示す情報を使用する．

(1) VMM 上で複数の VM が動作する環境を想定するため，システムコールが発行された VM が保護対象 VM か否かを判定する必要がある．VMID は，VM を管理するために VM ごとに一意に割り当てられている識別子である．このため，保護対象 VM の識別に VMID を用いる．VMID は，VMM が保持している仮想 CPU に関するデータ構造から取得できる．

(2) VM 上で複数のプロセスが動作する環境を想定するため，システムコールを発行したプロセスが重要プロセ

表 1　代理実行情報

通番	名前	説明
(1)	VMID	保護対象 VM を識別するために用いる．
(2)	ページディレクトリ	重要プロセスを識別するために用いる．
(3)	システムコール番号	通信に関するシステムコールを識別するために用いる．
(4)	システムコールの引数	システムコールに渡す引数として用いる．

表 2　通信に関するシステムコール

通番	システムコール	説明
(1)	socket()	新規ソケットを作成する．
(2)	bind()	自ソケットに名前をつける．
(3)	listen()	自ソケットへの接続を待つ．
(4)	connect()	通信先ソケットへの接続を行う．
(5)	accept()	自ソケットへの接続を受け付ける．
(6)	sendto()	通信先ソケットへメッセージを送信する．
(7)	recvfrom()	通信先ソケットからメッセージを受け取る．
(8)	read()	ファイルディスクリプタから読み込む．
(9)	write()	ファイルディスクリプタに書き込む．
(10)	close()	ファイルディスクリプタを削除する．

スか否かを判定する必要がある．ページディレクトリ（以降，PD）は，ページング機構に使用されるプロセス固有のアドレスである．このため，重要プロセスの識別に PD を用いる．PD は，保護対象 VM の CR3 レジスタから取得できる．プロセスの識別にプロセス ID（以降，PID）を用いず PD を用いる理由は，PID の取得によるオーバヘッドを抑制するためである．PID は，保護対象 VM 上のプロセス管理表をもとにメモリから取得する必要がある．一方，PD は，CR3 レジスタに格納されており，VM exit の発生により VMM に処理が遷移する際に退避されているため，メモリを参照する必要がない．このため，重要プロセスの識別に PID を用いず PD を用いる．

(3) 捕捉したシステムコールが通信に関するシステムコールか否かを判定する必要がある．このため，システムコールごとに指定されているシステムコール番号を用いる．システムコール番号は，保護対象 VM の rax レジスタから取得できる．

(4) システムコールの代理実行には，システムコールの引数が必要である．システムコールの引数は最大で 6 つあり，保護対象 VM の rdi，rsi，rdx，r10，r8，および r9 レジスタから取得できる．send() や write() などの場合は，引数で指定した領域から VMM へ送信データを取得する必要がある．

2.3.8　通信に関するシステムコール

本制御法では，保護対象 VM 上で動作する OS として，Linux 3.2.0 を用いる．このため，通信に関するシステムコールとして，Linux 3.2.0 において実装されている**表 2** のシステムコールを代理実行の対象とする．

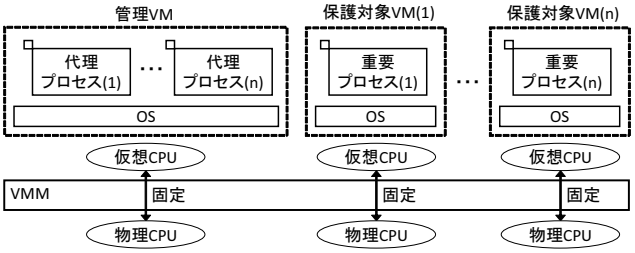

図 5　評価モデル

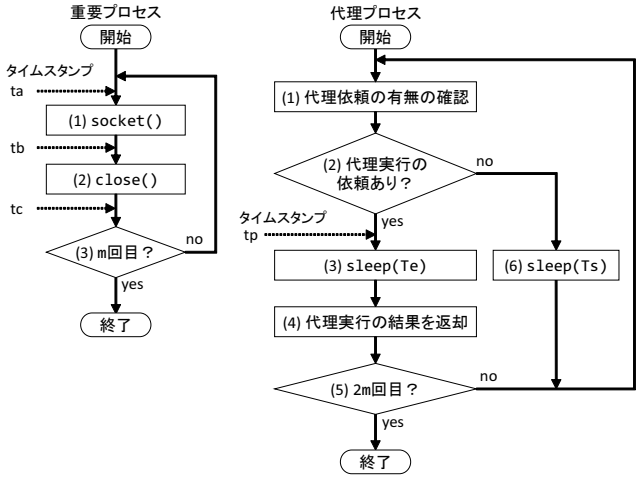

図 6　評価に用いる重要プロセスと代理プロセスの処理流れ

データの送受信を攻撃者に監視されると，送受信データをもとに重要サービスの存在が攻撃者に特定される可能性がある．このため，データの送受信を行うシステムコール sendto()，recvfrom()，read()，および write() を代理実行する．また，これらのシステムコールを代理実行するには，代理プロセスにおいて通信先のプロセスとのコネクションを事前に確立しておく必要がある．このため，socket()，bind()，listen()，connect()，accept()，および close() も代理実行する．なお，(8)，(9)，および (10) は，通信に利用されるファイルディスクリプタを対象とし，ファイル操作に用いられるファイルディスクリプタは対象としない．

3.　評価

3.1　評価内容

通信処理制御法を適用した環境において，以下の項目を評価した．

(1) 代理実行の依頼取得に関する動作とオーバヘッド時間の分析（3.3 節）

(2) 代理実行処理が「事象待ち」を含むかどうかによる動作分析（3.4 節）

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 VM 上の重要プロセスの数を 1 とする．管理 VM 上では重要プロセスと同数の 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

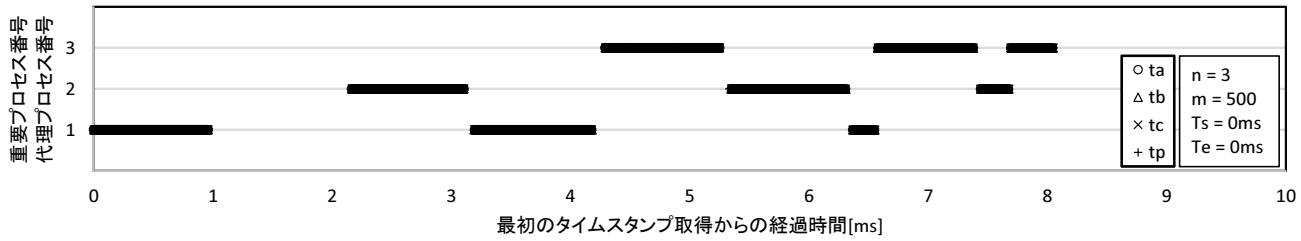

図 7 重要プロセスと代理プロセスの動作（$T_s = 0$ms，$T_e = 0$ms）

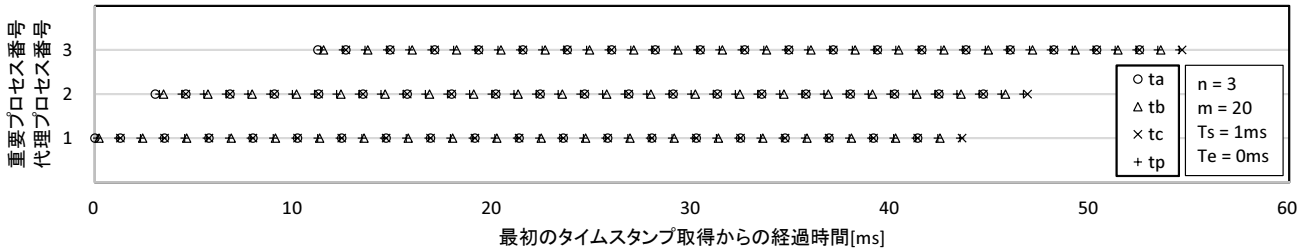

図 8 重要プロセスと代理プロセスの動作（$T_s = 1$ms，$T_e = 0$ms）

表 3 評価環境

ソフトウェア	
VMM	Xen 4.2.3
OS(管理 VM)	Debian 7.3(Linux 3.2.0 64bit)
OS(保護対象 VM)	Debian 7.3(Linux 3.2.0 64bit)
計算機環境	
CPU	Intel Core i7-2600(3.4GHz，4 コア)
仮想 CPU(管理 VM)	1 コア
仮想 CPU(保護対象 VM)	1 コア
メモリ (全体)	8GB
メモリ (管理 VM)	5GB
メモリ (保護対象 VM)	1GB

理実行は，代理プロセス i が行う．また，各 VM に 1 つの仮想 CPU を用意し，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，複数の仮想 CPU が同じ物理 CPU コアを共有することで，各仮想 CPU の物理 CPU コアへの割り当てが不安定になり，測定結果が不安定になることを避けるためである．

評価に用いる重要プロセスと代理プロセスの処理流れを**図 6** に示す．重要プロセスは，socket() システムコールと close() システムコールの発行を m 回繰り返す．また，タイムスタンプ ta，tb，および tc をそれぞれ socket() 発行前，socket() 発行後，および close() 発行後の 3 か所で取得する．代理プロセスは，代理実行の依頼の有無の確認を一定間隔（T_s）で行い，依頼がある場合は，代理実行を行う代わりに時間 T_e の sleep 処理を行う．また，タイムスタンプ tp を sleep 処理の前で取得し，対応する重要プロセスの代理実行が終了したとき，プログラムを終了する．

3.2 評価環境

評価環境を**表 3** に示す．評価は，VMM として Xen 4.2.3 を用い，管理 VM と保護対象 VM 上で動作する OS として Debian 7.3（Linux 3.2.0 64bit）を用いた．また，8GB のメモリと Intel Core i7-2600（3.4GHz，4 コア）の CPU を搭載した計算機を使用した．管理 VM には 1 コアの仮想 CPU，5GB のメモリを割り当てた．保護対象 VM には 1 コアの仮想 CPU，1GB のメモリを割り当てた．

3.3 代理実行の依頼取得に関する動作とオーバヘッド時間の分析

代理プロセスは，一定間隔（T_s）で代理実行の依頼取得を行う．この T_s を変化させた場合の重要プロセスと代理プロセスの動作とシステムコールのオーバヘッド時間について評価する．システムコールのオーバヘッド時間は，$T_e = 0$ms としたときのタイムスタンプ ta，tb，および tc について，それぞれの取得時刻の差を算出することで求められる．なお，重要プロセスと代理プロセスをそれぞれ 3 つ同時走行（$n = 3$）させ，$T_s = 0$ms の場合は，システムコールの繰り返し回数（m）を 500 回とし，$T_s = 1$ms の場合は，20 回とする．

$T_s = 0$ms の場合と $T_s = 1$ms の場合の重要プロセスと代理プロセスの動作をそれぞれ**図 7** と**図 8** に示す．この図は，重要プロセスで取得した最初の 2 つのタイムスタンプの中央で代理プロセスが最初のタイムスタンプを取得すると仮定してタイムスタンプを重ね合わせている．これは，代理プロセスと重要プロセスが異なる仮想 CPU 上で動作するため，それぞれの仮想 CPU から取得したタイムスタンプの間に「ずれ」が生じるためである．これに伴い，グラフの横軸を「最初のタイムスタンプ取得からの経過時間」としている．図 7 と図 8 から，以下のことがわかる．

(1) 図 7 では，約 1ms ごとに代理実行が行われていることがわかる．これは，タイマ割り込みによりプロセス切

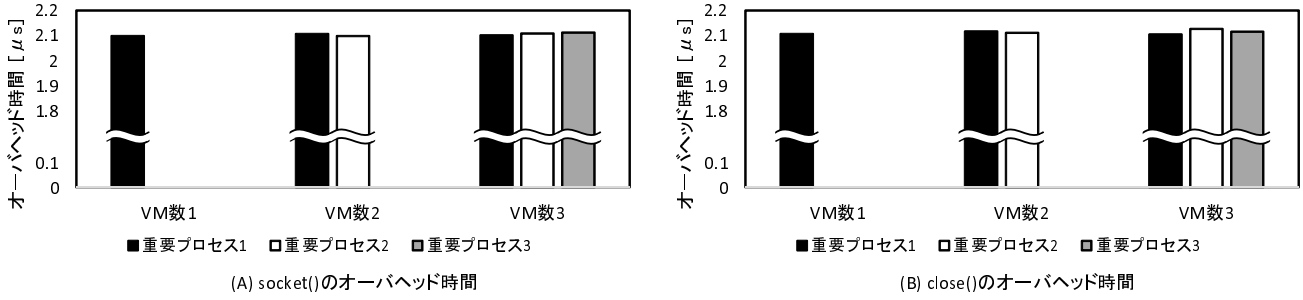

図 9　オーバヘッド時間（$T_s = 0$ms，$T_e = 0$ms）

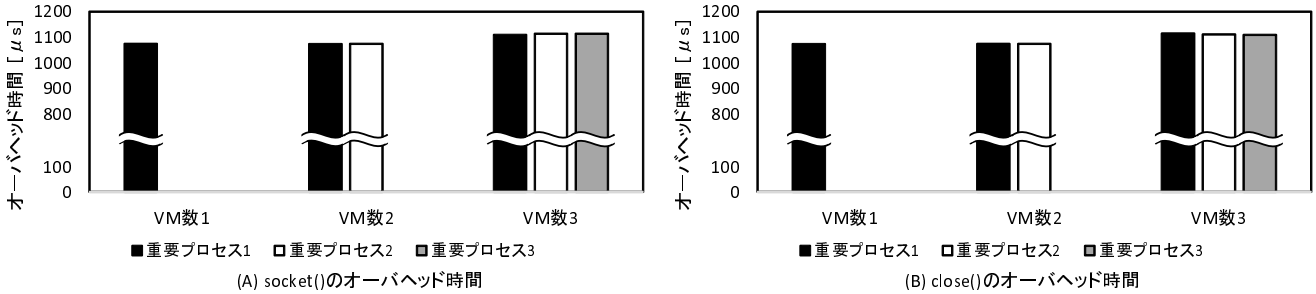

図 10　オーバヘッド時間（$T_s = 1$ms，$T_e = 0$ms）

り替えが行われているためであると考える．

(2) 図 8 では，交互に代理実行が行われていることがわかる．これは，代理実行の間隔（T_s）を 1ms にしたことにより，連続して代理実行する前にタイマ割り込みにより代理プロセスが切り替えられたためであると考える．

上記 (1)，(2) より，$T_e = 0$ms のとき，$T_s = 0$ms の場合は代理実行を平等に実行不可であり，$T_s = 1$ms の場合は代理実行を平等に実行可能であるといえる．

$T_s = 0$ms の場合の socket() と close() のオーバヘッド時間を**図 9** に示す．図 9 から，以下のことがわかる．

(3) $T_s = 0$ms の場合のオーバヘッド時間は，socket() と close() のどちらも約 2.1μs であることがわかる．これは，通信処理制御法を適用したことにより発生するシステムコールのオーバヘッドに等しい．

(4) 棒グラフの高さに変化がないことから，同時に動作する保護対象 VM が増加してもオーバヘッド時間の変化は小さいことがわかる．さらに，各重要プロセスのオーバヘッド時間の差も小さいことがわかる．

(5) 評価で用いる代理プロセスは，システムコールを代理実行しないため，socket() と close() の相違点としてシステムコールの処理内容は考慮しない．処理内容以外の相違点は引数の数であり，socket() の引数は 3 つ，close() の引数は 1 つである．この引数の数の違いによる影響は，システムコールの引数を保護対象 VM から VMM へ，VMM から管理 VM へコピーする際のサイズが異なることある．しかし，図 9 の (A) と (B) に大きな変化がないことから，この違いがオーバヘッド時間に与える影響は小さいといえる．

$T_s = 1$ms の場合の socket() と close() のオーバヘッド時間を**図 10** に示す．図 10 から，以下のことがわかる．

(6) 図 9 と同様のことがいえる．ただし，オーバヘッド時間が 1100μs 前後であることから，オーバヘッド時間は T_s の影響を受けることがわかる．オーバヘッド時間が 1100μs 前後である理由は，sleep 処理が「少なくとも引数で指定した時間の間，sleep 処理を行う」ものであることから，管理 VM のタイマの影響で sleep 処理時間にばらつきが発生するためであると考える．

上記 (1) から (6) より，代理実行の依頼を取得する間隔（T_s）が 0ms の場合，socket() と close() のオーバヘッド時間は約 2.1μs である．しかし，複数の代理実行の依頼があるとき，平等に代理実行できない．また，代理実行の依頼を取得する間隔（T_s）が 1ms の場合，socket() と close() のオーバヘッド時間は T_s の影響を受ける．一方で，複数の代理実行の依頼があるとき，平等に代理実行できる．

3.4 代理実行処理が「事象待ち」を含むかどうかによる動作分析

代理実行の依頼の取得の間隔（T_s）が 0ms であるとき，代理実行処理が「事象待ち」を含む場合と含まない場合の重要プロセスと代理プロセスの動作について評価する．この評価では，「事象待ち」を含まない場合を $T_e = 0$ms とし，「事象待ち」を含む場合を $T_e = 1$ms とする．このため，「事象待ち」を含まない場合の重要プロセスと代理プロセスの動作は，図 7 の結果を用いる．図 7 から，以下のこ

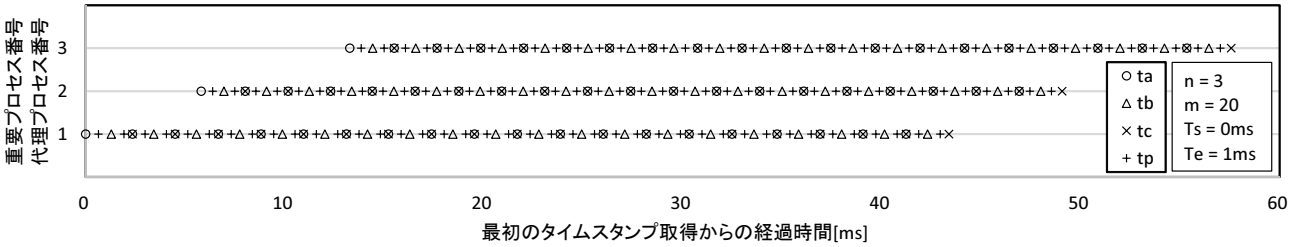

図 11 重要プロセスと代理プロセスの動作（$T_s = 0$ms，$T_e = 1$ms）

とがわかる．

(1) タイマ割り込みによるプロセス切り替えが発生するまで連続して代理実行を行うため，複数の代理実行の依頼があるとき，平等に代理実行できない．

「事象待ち」を含む場合，つまり T_e を 1ms とした場合の重要プロセスと代理プロセスの動作を**図 11** に示す．図 11 から，以下のことがわかる．

(2) 複数の代理実行の依頼がある場合でも，交互に代理実行処理が行われていることがわかる．これは，T_s を 1ms にしたことにより，「事象待ち」をしている間にタイマ割り込みにより代理プロセスが切り替えられるためであると考える．

上記 (1) と (2) より，代理実行の依頼を取得する間隔（T_s）が 0ms の場合においても，システムコールが「事象待ち」を含む場合は，複数の代理実行の依頼があるとき，平等に代理実行できる．しかし，実際は，代理実行処理には「事象待ち」の有無が混在するため，$T_s > 0$ が必須である．

4. おわりに

重要サービスの通信操作を不可視化する通信処理制御法について述べた．具体的には，保護対象 VM 上の重要プロセスが発行する通信に関するシステムコールを VMM により捕捉し，管理 VM 上の代理プロセスによりシステムコールを代理実行する．これにより，重要サービスの通信操作を不可視化し，通信内容をもとにした重要サービスの特定を困難にする．提案方式は，保護対象 VM 上のプログラムを変更せずに，VMM と代理プロセスのみにより通信処理制御法を実現する特徴を持つ．つまり，重要サービスのプログラムを変更することなく，本通信処理制御法を適用できる．

評価では，代理実行の依頼の取得間隔とオーバヘッド時間についての分析を行い，複数の代理実行の依頼を平等に処理するためには，代理実行の依頼の取得間隔は 0 より大きくする必要があることを示した．また，通信処理制御法を適用したことによるシステムコールのオーバヘッドは，socket() と close() を代理実行する場合においてどちらも約 2.1μs であることを示した．

謝辞 本研究の一部は，JSPS 科研費 18K18051，16H02829 の助成を受けたものです．

参考文献

[1] Hsu, F.-H., Wu, M.-H., Tso, C.-K., Hsu, C.-H. and Chen, C.-W.: Antivirus Software Shield Against Antivirus Terminators, *IEEE Trans. Inf. Forensic Secur.*, Vol. 7, No. 5, pp. 1439–1447 (2012).

[2] Min, B. and Varadharajan, V.: A novel malware for subversion of self-protection in anti-virus, *Software: Practice and Experience*, Vol. 46, No. 3, pp. 361–379 (2016).

[3] stealth: A new Adore root kit, available from ⟨ http://lwn.net/Articles/75990/⟩ (accessed 2018-1-19).

[4] Sato, M., Yamauchi, T., Taniguchi, H.: Process Hiding by Virtual Machine Monitor for Attack Avoidance, *Journal of Information Processing*, Vol.23, No.5, pp. 673–682 (2015).

[5] 佐藤 将也，山内 利宏，谷口 秀夫：仮想計算機を用いた重要ファイル保護手法，情報処理学会シンポジウムシリーズ コンピュータセキュリティシンポジウム 2017（CSS2017）論文集，Vol.2017，No.2，pp.1302–1308（2017）．

[6] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed Systems Security Symposium*, pp. 191–206 (2003).

[7] Wang, J., Yu, M., Li, B., Qi, Z. and Guan, H.: Hypervisor-based protection of sensitive files in a compromised system, *Proc. 27th Annual ACM Symposium on Applied Computing*, pp. 1765–1770 (2012).

[8] 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, *SIGOPS Oper. Syst. Rev.*, Vol. 37, No. 5, pp. 164–177 (2003).

[9] Jonas, P., Christian, S., Claudia, E.: Nitro: hardware-based system call tracing for virtual machines, *Proc. 6th International conference on Advances in information and computer security,* pp. 96–112 (2011).

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