Vol.2020-DPS-184 No.9
Vol.2020-EIP-89 No.9
2020/9/11

VMMによるデバッグレジスタの読み出しと書き込みの隠蔽手法の提案

佐藤 将也[1,a)]　谷口 秀夫[1]　仲村 亮祐[1]

概要：仮想計算機で動作するオペレーティングシステム（ゲスト OS）のセキュリティを向上させるために，仮想計算機モニタがハードウェアブレークポイントの設定によりゲスト OS の動作を監視する手法（ゲスト OS 動作監視手法）がある．しかし，デバッグレジスタはゲスト OS から読み書き可能である．このため，悪意のあるプログラムは，デバッグレジスタを読み書きすることで，ゲスト OS 動作監視手法を検知または無効化できる．そこで本稿では，ゲスト OS によるデバッグレジスタの読み出しと書き込みの隠蔽手法を述べる．提案手法は，ゲスト OS によるデバッグレジスタの読み出しと書き込みを仮想計算機モニタにより検知し，読み出しと書き込みが成功したようにゲスト OS に見せる．これにより，ゲスト OS 動作監視手法の検知や無効化を防止する．また，性能評価により提案手法がゲスト OS の性能へ与える影響を示す．

キーワード：仮想計算機モニタ，デバッグレジスタ，システムセキュリティ

1. はじめに

仮想化技術はクラウドコンピューティングなどの基盤技術として様々な場面で利用されている．例えば，従来の物理計算機で提供していたサービスを仮想計算機（Virtual Machine，以降，VM）に移行し，ハードウェアを集約することで，余剰の計算機資源を有効に活用できる．VM を提供する仮想計算機モニタ（VM Monitor，以降，VMM）は，VM の動作を制御しハードウェアへのアクセスを仲介する機能を持つことから，VM の実現だけでなく，VM 上で動作する OS（以降，ゲスト OS）やプロセスの動作監視にも利用できる．具体的には，悪性プログラム（以降，マルウェア）の解析 [1] に利用される．また，VM 上のプログラムを攻撃から保護することでセキュリティを向上する研究 [2], [3] にも利用される．

VMM によりマルウェア解析やセキュリティの向上を実現するために VM 上のプログラムの動作を監視する手法（以降，ゲスト OS 動作監視手法）がある．ゲスト OS 動作監視手法は，VM 上のプログラムが特定の処理を行ったことを VMM により検知する．ゲスト OS 動作監視手法により検知できる処理には，命令実行，メモリの読み書き，およびハードウェアへの処理依頼などがある．より細粒度に VM の動作を監視するためには，命令の実行を監視する必要がある．命令の実行を監視する手法の 1 つに，ブレークポイントを用いた手法がある．ブレークポイントを設定することで，特定の処理実行時に例外が発生し，デバッガはその例外を検知することでプログラムの監視を行う．ブレークポイントはデバッグのためだけでなく，VM 上のプログラムの動作監視にも用いられる．ブレークポイントには，ソフトウェアブレークポイントとハードウェアブレークポイントがある．ソフトウェアブレークポイントは，メモリの一部を `INT3` 命令に書き換え，`INT3` 命令の実行により発生するブレークポイント例外を用いる，VMM はブレークポイント例外を検知することで，VM 上のプログラム実行を監視できる．ハードウェアブレークポイントは，デバッグレジスタに監視対象のアドレスを指定することで，そのアドレスにおける命令実行または読み書きにより発生するデバッグ例外を用いる．VMM はデバッグ例外を検知することで，VM 上のプログラム実行を監視できる [2], [3].

ブレークポイントを用いたゲスト OS 動作監視手法は，命令単位での実行を監視できる利点があるものの，VM 上のプログラムからブレークポイントの検知や無効化ができる問題がある．ソフトウェアブレークポイントは，INT3 命令をメモリから検出することや完全性検査により検知できる．ハードウェアブレークポイントは，デバッグレジスタの読み込みにより検知できる [4]．ブレークポイントが検知または無効化されると，ゲスト OS 動作監視手法によ

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

る監視が困難になる．例えば，ブレークポイントを検知して動作を変更するマルウェアや，ブレークポイントを無効化するマルウェアにより，ゲスト OS 動作監視手法が困難になる可能性がある．また，ブレークポイントを用いたマルウェア [5] により，ゲスト OS 動作監視手法のために用いるデバッグレジスタの内容が上書きされることで，ゲスト OS 動作監視手法が無効化される可能性がある．これらの問題へ対処するため，ブレークポイントを VM 上のプログラムから隠蔽する必要がある．ソフトウェアブレークポイントを隠蔽する手法として SPIDER [6] がある．SPIDER は，監視対象のプログラムがマッピングされるメモリを Code View と Data View に分割し，Code View にソフトウェアブレークポイントを設定する．また，Intel の CPU が持つ Extended Page Table (EPT) を用い，命令取得時のみ Code View をマッピングし，メモリの読み書き時には Data View をマッピングする．これにより，VM 上のプログラムからは検知できないソフトウェアブレークポイントを実現している．一方，ハードウェアブレークポイントを隠蔽する具体的な手法は，著者らが調べた限り充分に検討されていない．デバッグレジスタの読み込みを行う関数が呼び出された際に，ハードウェアブレークポイントをリセットする手法がある [7]．しかし，他の関数によりデバッグレジスタの読み込みが可能な環境では対処できない．

本稿では，ハードウェアブレークポイントを用いたゲスト OS 動作監視手法を検知または無効化されること防止するために，デバッグレジスタの読み出しと書き込みの隠蔽手法（以降，隠蔽手法）を提案する．隠蔽手法は，デバッグレジスタの読み出しを VMM により監視し，ゲスト OS 動作監視手法によるデバッグレジスタの設定値をゲスト OS から隠蔽する．また，ゲスト OS によるデバッグレジスタへの書き込みが成功したように見せることで，ゲスト OS 動作監視手法の無効化を防止する．本稿では，隠蔽手法の基本方式および評価結果を報告する．

2. ハードウェアブレークポイントを用いたゲスト OS 動作監視手法

2.1 ゲスト OS 動作監視手法

ゲスト OS 動作監視手法について，ハードウェアブレークポイントを用いる手法 [2], [3] の処理を**図 1** に示し，以下に説明する．

(1) VMM は，デバッグレジスタを利用して，ゲスト OS 上の監視したい特定のアドレスをハードウェアブレークポイントとして設定する．

(2) ハードウェアブレークポイントとして設定されたアドレス上の命令をゲスト OS 上のプログラムが実行しようとしたとき，デバッグ例外が発生する．

(3) VMM は，発生したデバッグ例外を捕捉し，ゲスト OS 上のプログラムの情報を取得する．

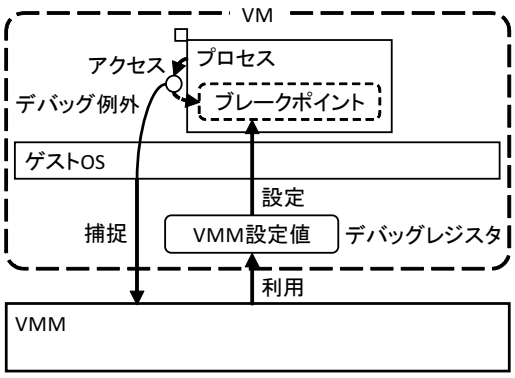

図 1 ハードウェアブレークポイントを用いたゲスト OS 動作監視手法

ハードウェアブレークポイントは，CPU が提供する機能であり，主にデバッグのために利用される．ハードウェアブレークポイントはデバッグレジスタに監視対象のアドレスを格納しておくことで利用でき，指定したアドレスにおいて命令が実行される直前にデバッグ例外を発生させることができる．デバッガはデバッグ例外を監視することでプログラムの動作を監視できる．Intel VT を用いた仮想化環境では，デバッグ例外発生時に VMM へ走行モードを遷移（VM exit）するよう設定できる．そこで，ゲスト OS 動作監視手法は，監視対象のプログラムが配置されているメモリ領域のアドレスをハードウェアブレークポイントに設定することで，ゲスト OS 上のプログラムの動作を VMM により監視する．

2.2 問題点

ゲスト OS 上のプログラムは，デバッグレジスタの値を読み書きできる．この機能はデバッガが利用するだけでなく，デバッガ回避 [4] やルートキットの挿入 [5] のために用いられる．このため，ゲスト OS 動作監視手法に基づく処理には，以下の問題がある．

(1) ゲスト OS 動作監視手法に基づく処理の検知
ゲスト OS 上のマルウェアは，デバッグレジスタの値を読み出すことで設定されているハードウェアブレークポイントを特定できる．これにより，ゲスト OS 動作監視手法に基づく処理の存在を検知されてしまう．

(2) ゲスト OS 動作監視手法に基づく処理の無効化
ゲスト OS 上のマルウェアは，デバッグレジスタの値を上書きしてハードウェアブレークポイントを変更できる．この際，設定されているハードウェアブレークポイントは削除されてしまう．また，ゲスト OS 動作監視手法を無効化するためにマルウェアがデバッグレジスタの値を書き換えることも考えられる．これらの原因により，ゲスト OS 動作監視手法に基づく処理が無効化されてしまう．

3. デバッグレジスタの読み出しと書き込みの隠蔽手法

3.1 想定する脅威

2.2 節の問題へ対処するために，VM 上のプログラムによるデバッグレジスタの読み書きによりゲスト OS 動作監視手法を検知または無効化する攻撃へ対処する．このため，VM の利用を前提とする．従って，VM で動作していることを検知することで解析回避を行う [8] など，VM 上でのハードウェアブレークポイントによるゲスト OS 動作監視手法以外の特徴を基に振舞いを変化させるマルウェアは対象としない．マルウェアはユーザモードとカーネルモードで攻撃コードを実行可能とする．つまり，マルウェアはデバッグレジスタの読み書きを行うコードを実行できるものとする．

また，隠蔽手法を無効化する攻撃へ対処する．ただし，隠蔽手法はマルウェアが攻撃コードを挿入するよりも前に有効化できるものとする．また，VMM を攻撃するようなマルウェアは想定しない．

3.2 基本方式

ゲスト OS 動作監視手法で利用しているデバッグレジスタの設定情報の読み出しや上書きを防止し，かつゲスト OS のプログラムからデバッグレジスタの読み出しと書き込みが可能なように見せる隠蔽手法を以下に述べる．

隠蔽手法の基本方式を**図 2** に示し，以下に説明する．

(1) VMM は，ゲスト OS のプログラムによるデバッグレジスタの読み出しまたは書き込みを捕捉し，デバッグレジスタの値をゲスト OS のプログラムが設定した値（以降，ゲスト OS 設定値）に置き換える．

(2) VMM は，ゲスト OS のプログラムによるデバッグレジスタの読み出しまたは書き込みの終了を捕捉し，デバッグレジスタの値をゲスト OS 動作監視手法のためのハードウェブレークポイント設定値（以降，VMM 設定値）に置き換える．

隠蔽手法は，VM が起動しゲスト OS のプログラムが動作を開始する前に有効化できる．この有効化の後は，ゲスト OS のプログラムによるデバッグレジスタの読み出しと書き込みからゲスト OS 動作監視手法によるハードウェアブレークポイントを隠蔽できる．

3.3 実現方式

3.3.1 前処理

隠蔽手法の前処理を**図 3** に示し，処理流れを以下に説明する．前処理はゲスト OS 動作監視手法の有効化後に行うものとする．

(1) デバッグレジスタの値をゲスト OS 設定値として保存する．

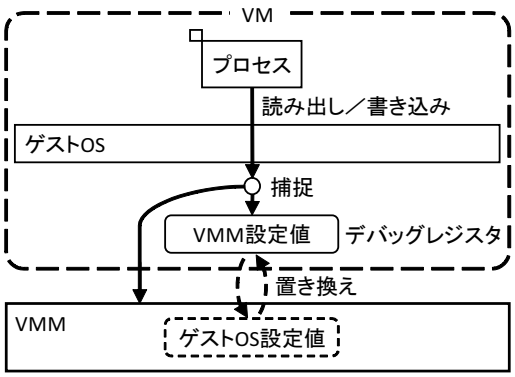

図 2 隠蔽手法

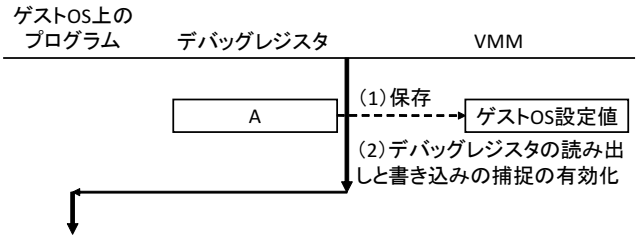

図 3 前処理の処理流れ

(2) デバッグレジスタの読み出しと書き込みの捕捉を有効にする．具体的には，デバッグレジスタの読み出しと書き込みの際に，VM exit 遷移が発生するようにする．

また，上記を実現するために，VM を起動する前にデバッグレジスタへの読み書きで VM exit が発生するように設定する．具体的には，VM-Execution Control と呼ばれる領域の MOV-DR exiting を有効にする [9]．VM-Execution Control は CPU が VM exit を起こすか否かを判定するために用いられるメモリ領域であり，VMM により操作できる．MOV-DR exiting フィールドを有効にしておくことで，ゲスト OS が動作する走行モード（VMX non-root モード）におけるデバッグレジスタの読み書きの直前に VM exit を起こさせることができる．これにより，VM 上のプログラムによるデバッグレジスタの読み書きを VMM から検知できる．

3.3.2 デバッグレジスタの読み出し時と書き込み時の処理

次に，隠蔽手法に基づく処理の流れとして，読み出し時を**図 4** に示し，書き込み時を**図 5** に示す．読み出し時の処理流れを以下に説明する．

(1) デバッグレジスタ読み出しを捕捉する．

(2) デバッグレジスタの値を退避する．

(3) デバッグレジスタの値をゲスト OS 設定値に書き換える．

(4) デバッグレジスタ読み出し終了の捕捉を有効化する．具体的には，シングルステップ実行によって VM exit が発生するように設定する．

(5) デバッグレジスタ読み出しの捕捉を無効化し，ゲスト

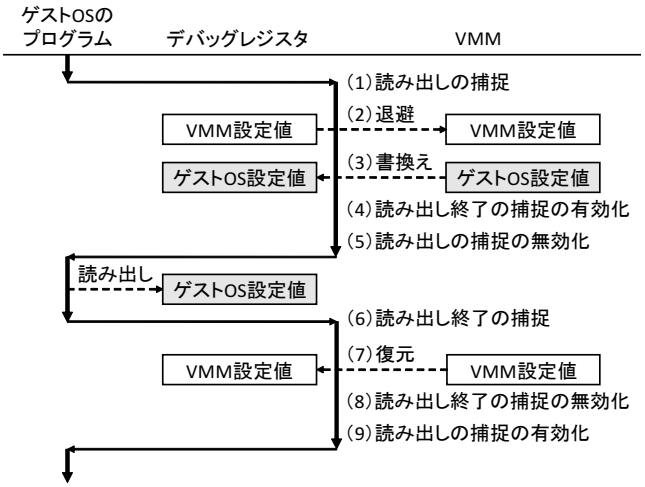

図 4　デバッグレジスタ読み出し時の処理流れ

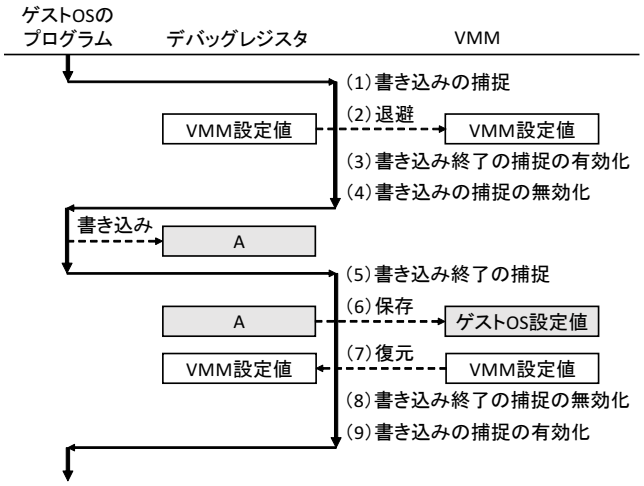

図 5　デバッグレジスタ書き込み時の処理流れ

OS のプログラムに処理を戻す．
（ゲスト OS のプログラムは読み出しを行う．）
(6) デバッグレジスタ読み出し終了を捕捉する．
(7) デバッグレジスタの値を VMM 設定値に復元する．
(8) デバッグレジスタ読み出し終了の捕捉を無効化する．
(9) デバッグレジスタ読み出しの捕捉を再び有効化し，ゲスト OS のプログラムに処理を戻す．

また，書き込み時の処理流れを以下に説明する．
(1) デバッグレジスタ書き込みを捕捉する．
(2) デバッグレジスタの値を退避する．
(3) デバッグレジスタ書き込み終了の捕捉を有効化する．具体的には，シングルステップ実行によって VM exit が発生するように設定する．
(4) デバッグレジスタ書き込みの捕捉を無効化し，ゲスト OS のプログラムに処理を戻す．
（ゲスト OS のプログラムは書き込みを行う．）
(5) デバッグレジスタ書き込み終了を捕捉する．
(6) デバッグレジスタに書き込まれた値（図 5 の A）をゲスト OS 設定値として保存する．
(7) デバッグレジスタの値を VMM 設定値に復元する．
(8) デバッグレジスタ書き込み終了の捕捉を無効化する．
(9) デバッグレジスタ書き込みの捕捉を再び有効化し，ゲスト OS のプログラムに処理を戻す．

以上より，VM からデバッグレジスタを参照する際に，実際とは異なる値を返却することで，ゲスト OS 動作監視手法が設定した値を参照不可にし，ゲスト OS 動作監視手法を攻撃者から検知しにくくする．また，VM からデバッグレジスタを書き換えた際に，書き換えが成功したように見せかけつつ，ゲスト OS 動作監視手法の無効化を防止できる．

3.4　期待される効果と制限

隠蔽手法は，マルウェアがデバッグレジスタの読み書きによるゲスト OS 動作監視手法の検知および無効化を防止できる．また，隠蔽手法の無効化は困難である．これは，デバッグレジスタへの読み書きの検知が CPU の機能を用いて実現されており，かつ，ゲスト OS のプログラムを改変していないためである．隠蔽手法はマルウェアが動作するより前に有効化済であり，かつ，デバッグレジスタの読み書きは VM-Execution Control により検知するため，VM 上のマルウェアが CPU による動作モード遷移を回避できない限り隠蔽手法を回避できない．また，隠蔽手法はゲスト OS を改変しない．このため，マルウェアがゲスト OS の利用するメモリを検査することで隠蔽手法を検知することはできない．さらに，SPIDER [6] と異なり，EPT を利用しない環境にも適用できる．

提案手法には以下の制限がある．
(1) 攻撃者がデバッグレジスタ操作の処理時間を基に提案手法を検知できる可能性がある．
(2) デバッグレジスタによるハードウェアブレークポイントの利用可否を確認するようなプログラムを実行することで提案手法を検知できる．

提案手法ではゲスト OS によるデバッグレジスタの読み出しや書き込み毎に VMM へ処理が遷移するためオーバヘッドが発生する．このため，攻撃者がデバッグレジスタの読み書きにおける処理時間を測定することで隠蔽手法の有無を確認できる．また，攻撃者は，ハードウェアブレークポイントの利用可否を利用することで隠蔽手法を検知できる．例えば，Self-Debugging という手法により，マルウェアが他プロセスからデバッグされるのを防止する手法が存在する．この手法により，攻撃者がマルウェア自身にハードウェアブレークポイントを設定し，マルウェアをそのプロセスにデバッガとしてアタッチした際にデバッグ例外を検知できない場合，隠蔽手法が検知される可能性がある．これらの隠蔽手法の検知への対処は今後の課題とする．しかし，隠蔽手法を検知できた場合でも，攻撃者によるデバッ

表 1　評価環境

CPU	Intel Core i7-3770（3.4GHz, 4 コア）	
仮想 CPU	Domain0	1 コア
	測定用 VM	1 コア
メモリ	全体	4 GB
	Domain0	3 GB
	測定用 VM	1 GB

グレジスタの書き換えは防止できるため，ゲスト OS 動作監視手法の無効化は有効であるといえる．

4. 評価

4.1 目的と環境

隠蔽手法の適用により，デバッグレジスタへの書き込みまたは読み出しによりオーバヘッドが発生する．本評価では，隠蔽手法の適用によるオーバヘッドを明らかにする．

評価環境を**表 1** に示す．評価では，各 VM に仮想 CPU を 1 コアずつ割当てた．また，各仮想 CPU は異なる物理 CPU コアに固定した．メモリは測定 VM に 1 GB を割り当て，残りを Domain0 に割り当てた．測定用 VM は Intel VT-x を用いた完全仮想化環境で動作させた．提案手法で利用した VMM は Xen 4.2.3 であり，Domain0 と測定用 VM では Debian 7.3 (Linux 3.2.0) を用いた．また，各 OS はシングルユーザモードで走行させた．

4.2 性能評価

隠蔽手法のオーバヘッドを測定した．具体的には，デバッグレジスタの読み出しと書き込みを行うシステムコールをそれぞれ作成し，各システムコールを 1,000 回ずつ実行したときの実行時間を隠蔽手法適用前と適用後で比較した．

実行時間の測定結果を**図 6** に示す．隠蔽手法の適用によるオーバヘッドは，読み出しで約 3.574μs，書き込みで約 3.521μs である．オーバヘッドの要因の 1 つは，デバッグレジスタからの読み出しまたは書き込みにおいて発生する VM exit である．適用前はデバッグレジスタの読み出しや書き込みによる VM exit は発生しない．一方適用後は，図 4 と図 5 で示したとおり，一度の読み書きにおいて 2 回の VM exit が発生する．

また，オーバヘッドは，読み出しが書き込みより大きい．これは，以下の理由による．隠蔽手法における読み出し時では，デバッグレジスタの書き込みを 2 回行い，書き込み時では 1 回行う．図 6 より，隠蔽手法適用前は，読み出しより書き込みの処理時間が長い．隠蔽手法適用後は，デバッグレジスタへの書き込みが 1 回多いことから，読み出し時の方がオーバヘッドが大きい．

なお，ハードウェアブレークポイントはプログラムのデバッグや解析の用途で用いられるため，通常はデバッグレジスタの読み書きが行われることは想定されない．また，

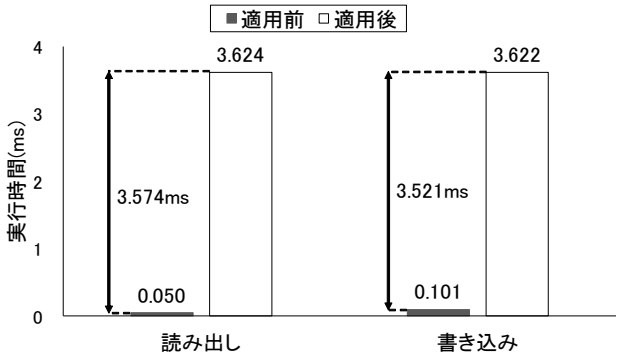

図 6　デバッグレジスタの読み出しと書き込み 1,000 回実行時の処理時間

隠蔽手法の適用によりデバッグレジスタの読み書きの処理時間が増加したとした場合に影響を受けるのはデバッグ処理であり，隠蔽手法によるオーバヘッドがゲスト OS で実行されるプログラムにおいて問題になることは少ない．

5. 関連研究

5.1 仮想化環境における解析検知の防止

Ether [1] はマルウェア解析の機能を VMM により実現することでマルウェアによる解析検知を防止している．具体的には，シングルステップ実行により解析対象の動作を命令実行毎に監視することで解析を行っている．このため，解析をゲスト OS から隠蔽するために，特定の命令実行時にエミュレーションを行っている．具体的には，ゲスト OS における `MOV` 命令などによる `EFLAGS` レジスタの操作はステップ実行が検知される可能性があるため，隠蔽処理を行う．レジスタの読み書きを検知する点で Ether と隠蔽手法は同様の対処を行っているが，隠蔽手法はデバッグレジスタの書き込みが成功しているように見せる点で対処方針が異なる．ただし，Ether における解析検知の防止は充分でないことが示されている [10]．

Proskurin らは，ARM 環境において Virtual Machine Introspection (VMI) フレームワークをマルウェアから隠蔽する手法を実現している [11]．この手法では，ゲスト OS に Secure Monitor Call (`SMC`) 命令を挿入することで，VMI による監視を隠蔽している．しかし，`SMC` 命令は x86 アーキテクチャの CPU では利用できない．また，`SMC` 命令を用いるのは，従来のハードウェアブレークポイントやシングルステップ実行などのハードウェア機能を用いた監視手法は VMM によるエミュレーションが不完全であるため隠蔽は困難であるためと述べられている．隠蔽手法は，この手法とは異なり，ハードウェア機能を用いたゲスト OS 動作監視手法のエミュレーションの精度を向上させることで，監視手法の検知を困難にすることが目的である．

Vogl らはパフォーマンスカウンタ (PMC) を用いることで，ゲスト OS から観測できないブレークポイントを提案した [12]．この手法では，PMC が特定のイベントごとに

カウントアップされることを利用し，イベント発生時にPMC がオーバフローするように設定しておくことでイベント発生時に Non-Maskable Interrupt (NMI) を起こさせる．仮想化環境では NMI の発生により VM exit を起こさせることができるため，NMI を利用して VMM からゲスト OS の特定イベントの発生を検知できる．この手法は，ブレークポイントをゲスト OS から観測されないように，隠蔽手法と同様に VM-Execution Control を操作することで PMC の読み書きを禁止し，ブレークポイントの検知や無効化を防止している．

Deng らは，ソフトウェアブレークポイントの隠蔽機能を備える SPIDER を提案した [6]．SPIDER は，監視対象のプログラムが配置されたページを Code View と Data View の 2 つに分け，EPT を利用することで読み出し時には Data View を見せる．Data View には本来のコードが格納されているが Code View はソフトウェアブレークポイントを格納しているため，ブレークポイントによる監視が可能である．SPIDER は EPT を前提としているが，隠蔽手法はデバッグレジスタを用いる環境であれば EPT の有無に関わらず適用できる．

5.2 仮想化を前提としない解析検知の防止

文献 [13] はマルウェアがデバッガを検知する方法を体系的に調査し，デバッガ検知を防止する手法を提案している．この手法はハードウェアブレークポイントへの対策は対象外としてる．また，仮想化環境を想定していない．

VM やエミュレータによるマルウェア解析はマルウェアから検知または妨害されることから，System Management Mode (SMM) を用いたマルウェア解析フレームワーク MALT が提案されている [14]．MALT は透過的な解析環境を実現するために SMM に標準的なデバッガの機能を実現している．MALT は従来のデバッガの機能を SMM に移植することを想定しているが，隠蔽手法は従来ゲスト OS 動作監視手法の隠蔽を目的としており，適用領域が異なる．

6. おわりに

仮想計算機におけるデバッグレジスタの読み出しと書き込みを隠蔽する手法を述べた．隠蔽手法は，デバッグレジスタの読み出しと書き込みが成功させているように見せかけることで，ハードウェアブレークポイントを用いたゲスト OS 動作監視手法の検知や無効化を防止する．具体的には，ゲスト OS によるデバッグレジスタの読み出しと書き込みを検知し，読み出し時にはゲスト OS が設定した値を返却し，書き込み時には書き込み後に動作監視手法が設定した値を復元する手法を述べた．性能評価では，ゲスト OS におけるデバッグレジスタの読み出しと書き込み時のオーバヘッドを明らかにした．

謝辞 本研究の一部は JSPS 科研費 18K18051 および 19H04109 の助成を受けたものです．

参考文献

[1] 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).
[2] 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).
[3] 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).
[4] Ferrie, P.: The “Ultimate” Anti-Debugging Reference, available from ⟨https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf⟩ (accessed 2020-08-15).
[5] Halfdead: Mystifying the debugger for ultimate stealthnes, *Phrack*, Vol.0x0c (2008).
[6] Deng, Z., Zhang, X. and Xu, D.: SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization, *Proc. 29th Annual Computer Security Applications Conference*, pp.289–298 (2013).
[7] Afianian, A., Niksefat, S., Sadeghiyan, B. and Baptiste, D.: Malware Dynamic Analysis Evasion Techniques: A Survey, *ACM Computing Surveys (CSUR)*, Vol.52, No.6, pp.1–28 (2019).
[8] Shi, H., Mirkovic, J. and Alwabel, A.: Handling Anti-Virtual Machine Techniques in Malicious Software, *ACM Transactions on Privacy and Security (TOPS)*, Vol.21, No.1, pp.1–31 (2017).
[9] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3C: System Programming Guide, Part 3, available from ⟨https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-volume-3c-system-programming-guide-part-3.html⟩ (accessed 2020-08-15).
[10] Pék, G., Bencsáth, B. and Buttyán, L.: nEther: In-guest Detection of Out-of-the-guest Malware Analyzers, *Proc. Fourth European Workshop on System Security (EuroSec)*, pp.1–6 (2011).
[11] Proskurin, S., Lengyel, T., Momeu, M., Eckert, C. and Zarras, A.: Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection, *Proc. 34th Annual Computer Security Applications Conference*, pp.407–417 (2018).
[12] Vogl, S. and Eckert, C.: Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture, *Proc. 2012 European Workshop on System Security (EuroSec)*, Vol.12 (2012).
[13] Shi, H. and Mirkovic, J.: Hiding Debuggers from Malware with Apate, *Proc. Symposium on Applied Computing*, pp.1703–1710 (2017).
[14] Zhang, F., Leach, K., Stavrou, A., Wang, H. and Sun, K.: Using Hardware Features for Increased Debugging Transparency, *2015 IEEE Symposium on Security and Privacy*, pp.55–69 (2015).