Vol.2017-OS-141 No.5
2017/7/26

OS クラッシュからのプロセスコンテキスト保護手法

寺田 献[1,a)]　山田 浩史[1,b)]

概要：オペレーティングシステム（OS）が障害を検知した際の一般的な解決方法は OS とアプリケーションプログラム（App）の再起動である．これまでに OS の不具合からプロセスコンテキストを復元したり保護したりする手法が提案されているが，これらを両立する手法は提案されていない．再起動にかかる時間はサービスを提供することができないサービスダウンタイムとなり，サービスの可用性を大きく損なう．既存の手法では OS の障害によって App の実行状態が破壊される可能性があり，OS の障害から App の保護を行うことはできていない．本論文で提案する ShadowBuddy は発生した障害から App の実行を守るために，OS が不正な操作によって App の実行状態を破壊することを防ぎ，OS から保護された領域に App の実行状態を保存する．OS による不正なページテーブル操作，OS による App のメモリ空間への不正な読み書きを防ぎ，OS 内に存在する App の実行状態を破壊したとしても再現することのできるチェックポイントを作成する．ShadowBuddy は最小で数%のオーバヘッドで，OS 透過に App のメモリ保護を行うことができる．

キーワード：オペレーティングシステム，信頼性

1. はじめに

OS には高い信頼性が求められる．しかしながら OS のバグは数多く報告されている．要因の一つとして，OS が提供する機能が多くなっており複雑化していることが挙げられる．Windows では 1000 行あたり 20 箇所程度のコード上の欠陥が存在すると推定されている [1]．特定の条件下でのみ発症するバグもあるため，すべての欠陥が障害を引き起こすわけではないが，欠陥が存在することで信頼性は潜在的に損なわれる．またオープンソースの OS で利用数も多い Linux では，数ヶ月ごとにマイナーバージョンアップが行われ，各マイナーバージョンに対して数千から 1 万個程度のパッチが作成される [2]．プログラムのバグを修正するためにはアップデートが行われるが，OS のアップデートが他のバグを含んでしまうという場合も確認されており，14.8%から 25.0%の修正には新たなバグを含んでいるという分析 [3] もある．OS で障害が発生すると，OS の処理性能が低下したり，OS の実行が継続できなくなったりすることがあり，可用性を低下させる．またメモリ上のデータオブジェクトが破壊されたり，それによって誤った挙動をしたりしてしまうこともある．中でも OS クラッシュは，OS が停止するほかデータが破壊された可能性がある物を指す．

OS の可用性が失われると，その上で動作する App が提供するサービスの可用性も失われてしまう．App は OS の機能を利用して動作するため，OS の動作が停止すれば App の動作も停止してしまう．そのため，OS の可用性は可能な限り高いことが望ましい．OS 内で障害が起こることは避けられないが，障害から速やかに復旧することで可用性の低下を抑制することができる．あるサービスの停止時間がサービス提供者にどれだけの損失を与えるかは，そのサービスによって異なる．たとえば，証券会社やクレジットカード会社では 1 時間あたり数百万ドル，インターネット通販サイトでは数十万ドルに上る [4]．

OS のバグによる障害を緩和するために，いくつかの手法が提案されている．しかしながら，既存の手法では OS の障害によって App の実行状態が破壊される可能性があり，OS の障害から App の保護を行うことはできていない．OS 障害の検知後に再起動した OS がメモリ上のデータオブジェクトを引き継ぐ手法 [5] では，OS 内のデータオブジェクトを直接コピーして使用するため，OS 内のデータオブジェクトが破壊される障害では誤った実行状態で App を再開させてしまう可能性がある．OS の障害が発生した後に App の実行を復元する手法 [6], [7], [8], [9] では，あらかじめ App の実行状態を定期的に保存しておき，障害発生時

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[a)] terada@asg.cs.tuat.ac.jp
[b)] hiroshiy@cc.tuat.ac.jp

に App の実行状態を復元する．これらの手法は OS 内の機能や実装に依存することが多く，OS 内の障害から App の実行状態を保護することはできない．OS 内の障害が App の実行状態を破壊することを防ぐ手法 [10], [11], [12] では，OS の挙動によって App の実行状態が不正な値で上書きされることが無いように，OS より特権の強い仮想マシンモニタ（VMM）やハードウェアの保護機能を用いて App のユーザ空間のメモリを保護する．これらの手法では OS 内のデータ構造を破壊してしまう整数型オーバーフローなどが引き起こす障害や，OS の実行が継続できなくなるような障害に対しては効果が無い．

本論文では App の実行状態を保護しつつ，障害発生時に App の実行を再開させる手法を提案する．我々の提案手法では OS 内のデータ破壊が起こったとしてもサービスを復旧することができる．App の実行状態は OS から保護され，OS の障害によって破壊されることはない．想定する障害は，OS と App の再起動によって復旧することができるものとする．

提案手法を評価するために，App の保護開始にかかる時間と，保護中に発生するオーバヘッドについて計測を行う．前者は App の初期化中に一度だけ発生するオーバヘッド，後者は App の実行中常に発生するオーバヘッドである．

2. 関連研究

障害が起こった場合に OS を高速に再起動して App の実行を再開する手法 [13], [14] では，OS の再起動にかかる時間を短縮することで App のサービスダウンタイムを短縮することができる．これらの手法では App の実行状態が失われることでサービスの利用が打ち切られるため，サービスの状態が巻き戻されたり初期化されたりする．また，メモリ上のキャッシュを失うことでサービスの再開後に App のスループットが低下する場合もある．たとえばインメモリ DB では，ディスクやネットワークを経由して得た DB の内容をメモリ上にのみキャッシュとして保持しており，このキャッシュを失った後は再びキャッシュを温め直さなければならない．キャッシュを温めるためには多くの I/O が発生し，温めている最中は I/O 待ちによって DB のスループットが低下する．

Otherworld [5] は OS の障害を App から隠蔽する．OS の障害が起こるとメモリの空き領域を使って OS を再起動させ，OS 内のデータオブジェクトを引き継いで App の実行状態を復元する．OS 障害によって App のユーザ空間のメモリが破壊されることを防ぐために，OS 実行中は App のユーザ空間のメモリに書き込み保護を行う．しかし OS 内の App 実行状態であるプロセスメタ情報は保護の対象となっておらず，OS の障害が発生した時にこれらの情報が破壊されれば，正しく App を引き継ぐことはできない．また OS のバグが障害の原因である場合，OS 内のデータオブジェクトを直接引き継ぐことで，再起動後も同じバグを実行して障害を引き起こす可能性がある．

再起動すること無く OS 内の障害を元に戻す手法 [15] では，あらかじめ OS の実行状態を更新する際にアンドゥログを作成しておき，偶発的に発生するような OS 内の障害を検知した時にログから OS の実行状態を巻き戻して再実行する．OS 再起動を伴わず App の実行が中断されないためサービスダウンタイムは短いが，復帰可能な障害は再実行すれば回避できるものに限定される．CPU 等の一時的な故障や，同時に実行していたプログラムと干渉したなどといった偶発的なエラーに対しては有効であるものの，たとえばカーネル内でメモリリークが発生した場合や，破壊されたカーネル内の変数を参照することで障害が起こる場合などには，この手法で復帰をしても同じ障害が発生する．FGFT [16]，Phoenix [17] はデバイスドライバのロールバックを可能とするものである．OS 内の変数のほか，デバイスの状態についても考慮して戻すことで，障害発生前の状態を再現する．これらは App の実行状態については考慮されていない．

App の Checkpoint/Restart によって，OS の障害が発生した後に，OS 障害の前に取得しておいた Checkpoint から App の実行状態を復元する事が可能である．Checkpoint は App が持つユーザ空間のメモリと，OS 内にあるプロセスメタ情報の集合である．BLCR [6] などの Kernel-level Checkpoint では，主に OS 内のプロセスメタ情報を直接保存して Checkpoint を作成する．OS 内の障害で破壊されたデータオブジェクトを保存してしまった場合，その Checkpoint を使って Restart をしても正しく App を再開させることができない．DMTCP [7]，CRIU などの User-level Checkpoint では，主にシステムコールを追跡して Checkpoint のプロセスメタ情報を作成する．User-level からは参照することができない OS 内のプロセスメタ情報が存在するため，OS 内の全実行状態を再現できるわけではない．いずれの手法でも Checkpoint はファイルに書き込まれるが，ファイルは OS の機能であるため Checkpoint は OS の障害から保護されていない．

Shielded Execution はセキュリティ上の観点から App を OS から保護する．OS は App の全資源にアクセスできるため，Rootkit が OS の脆弱性をつくと App のメモリ内容や通信内容などが不正に参照できるようになる．App の管理するメモリ内容には機密情報も含まれるため，メモリ内容が流出することで機密性が失われる．また改ざんされればサービス利用者に不利益を与える場合がある．OS よりも特権の強い VMM から App を保護する手法 [11], [18], [19] では，OS の挙動を監視して App の実行中は保護を外し，OS の実行中は保護をかける．App が持つユーザ空間のメモリを暗号化する手法 [11], [18] では，App の実行前に復号し，OS の実行前に暗号化する．メモリ空間を分割する

表 1　既存手法と提案手法 ShadowBuddy の比較

	OS 拡張なし	プロセスの保護	プロセスの実行継続
Phase-based Reboot [13], ShadowReboot [14]	✓	×	×
Otherworld [5]	×	✓	✓
RecoveryDomains [15], FGFT [16], Phoenix [17]	×	×	✓
BLCR [6], DMTCP [7], CRIU	✓	×	✓
Overshadow [11], InkTag [18]	✓	×	×
AppShield [19], Haven [12], SCONE [21]	✓	✓	×
ShadowBuddy	✓	✓	✓

手法 [19] では，App 用のメモリと OS 用のメモリとを区別し，保護対象の App が持つメモリをその App 以外のページテーブルにマッピングすることや，App 以外のメモリページを App の仮想アドレス空間にマッピングすることを禁止する．ハードウェアの支援機能である Intel SGX [20] を用いて App を保護する手法 [12], [21] では，App が自らのメモリに保護をかけ，ハードウェアが App に対する OS の不正な読み書きを遮断する．いずれの手法も OS の障害が発生して停止した場合には App の実行を継続することはできない．また OS 内にある App の実行状態は保護対象外であり，OS の障害によって OS 内にある App の実行状態は破壊される場合がある．

表 1 に既存手法の特徴をまとめる．プロセスコンテキストを復元したり保護したりする手法が提案されているが，これらを両立する手法は存在しない．

3. 提案

本論文では OS のクラッシュによって発生するメモリ破壊から App の実行状態を保護する ShadowBuddy を提案する．ShadowBuddy は既存手法の弱点を克服するために，OS クラッシュ時において，プロセスコンテキストの保護ならびに実行の継続を両立する．デザインゴールは以下の通りである．

- OS を変更しない
- App のプロセスコンテキストを保護する
- App の実行を再開する

図 1 のように App のユーザ空間のメモリを OS からの不正な書き込みから保護し，OS 内のプロセスメタ情報はシステムコールの履歴をチェックポイントとして保存しておくことで，OS クラッシュ後にメタ情報を再現する．OS クラッシュ発生時には OS 内のデータオブジェクトは全て信頼できないものとなるため，OS が主体となってデータオブジェクトを保存する手法は採用しない．

3.1 ユーザ空間のメモリ保護

OS は全メモリに対するアクセス権限を持っているため，多くの App のメモリ保護手法で用いられているページテーブルを使っても，OS からの不正なメモリアクセスを防ぐことはできない．そこで本研究では VMM で使われている MMU 仮想化技術の Extended Page Table（EPT）を用いる．

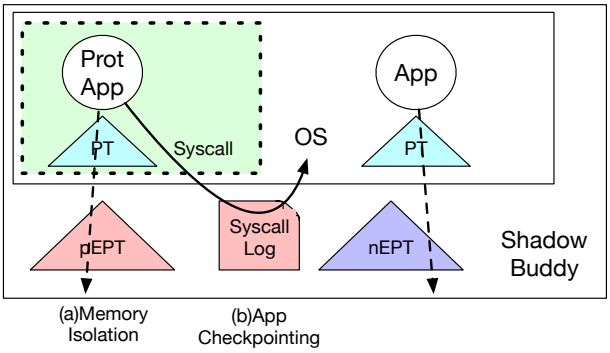

図 1　ShadowBuddy によるメモリ保護とチェックポインティング

3.1.1 Extended Page Table

ページテーブルは App が用いる仮想アドレスと，物理メモリページとの対応付けを行う．しかし単一マシン上で複数の OS を稼働させる仮想化環境上では，ゲストマシンが任意の物理アドレスにアクセスできてしまうと，ゲストマシン間の isoation を行うことができなくなる．そのため完全仮想化環境ではゲストマシンが認識するゲスト物理メモリと，実際の物理アドレスであるマシン物理メモリを用意する．OS はゲスト物理メモリが全物理メモリであると認識され，それ以外のメモリにアクセスすることはできない．

図 2 に仮想化環境でのアドレス変換処理を示す．区別のため，仮想化環境上の App の仮想アドレスをゲスト仮想アドレス（GVA），OS が認識する物理アドレスをゲスト物理アドレス（GPA），VMM が認識する物理アドレスをマシン物理アドレス（MPA）と呼ぶ．GVA と GPA の対応付は OS がページテーブルを用いて行い，GPA と MPA の対応付は VMM が EPT を用いて行う．OS は通常 1 App に対して 1 つのページテーブルを用意し，VMM は通常 1 OS に対して 1 つの EPT を用意する．

3.1.2 EPT Separation

OS からの不正な書き込みを防ぐために，**図 3** に示すように ShadowBuddy によって保護された App が動作するための EPT（pEPT）と，それ以外の App や OS が動作するための EPT（nEPT）の 2 つを用意する．OS からの不

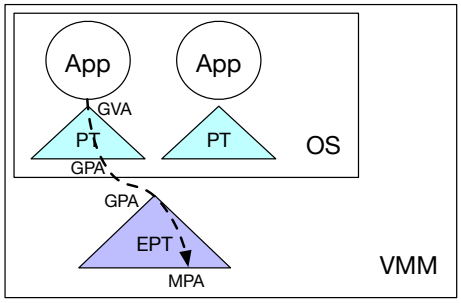

図 2　ページテーブルと EPT の 2 段階アドレス変換

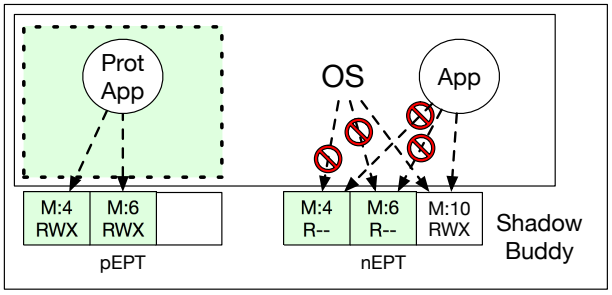

図 3　nEPT に対する書き込みと実行の禁止

正な書き込みを防ぐために，保護対象の App が使用するメモリは nEPT 上では全て書き込みを禁止する．OS から保護対象の App に実行が移った瞬間を補足するために，保護対象の App が持つ実行可能なメモリ領域は，nEPT 上では全て実行を禁止する．保護対象の App 実行中は pEPT で動作するため，OS が App のために作成したページテーブルは pEPT にも割り当てる．

3.1.3　EPT Switching

保護対象 App の実行中は pEPT，それ以外は nEPT を利用するために，App の挙動に合わせて EPT の切り替えを行う．しかし Linux のように単一仮想アドレス空間で OS と App が動作する場合には，VMM からは通常 App への切り替えを捕捉することはできない．そこで保護対象 App に実行が切り替わったことを捕捉するために，保護対象 App の実行可能メモリは nEPT 上では実行を禁止する．**図 4** に示すとおり，OS が保護対象 App に実行を戻した時点で EPT Violation が発生し，ShadowBuddy が捕捉して pEPT に切り替える．また OS の実行可能メモリは pEPT にはマッピングされていないため，システムコールや割込みなどによって OS に実行が移った時点で EPT Violation が発生し，ShadowBuddy が捕捉して nEPT に切り替える．

3.1.4　EPT Tracking

メモリ割り当ての更新があった場合に nEPT と pEPT で同期をとるために，ページテーブルの更新を捕捉する．App を実行している間には，App が確保要求を出したメモリ領域に OS がメモリを割り当てることがある．デマンドページングと呼ばれる手法では，mmap システムコールなどのメモリ確保要求のシステムコール時に物理メモリを割り当てるのではなく，App や OS が実際にページに読み書きを行った時点で物理メモリを確保し，ページテーブルに登録する．OS は nEPT 上で動作するため，新しい物理ページは nEPT 上では作成されるが，保護対象の App が動作する pEPT 上にはエントリが作成されない．そのため，OS から保護対象の App に実行を戻して pEPT に切り替えると，OS が割り当てた物理メモリを参照することができない．保護対象 App のために OS が作成したページテーブルは全て EPT によって書き込みを禁止する．OS がページテーブルを更新した際には ShadowBuddy が捕捉して，ページテーブルの更新が不正ではないことを確認した上でページテーブルの更新をエミュレートする．また pEPT にも対応するエントリを作成した上で，pEPT に切り替えて保護対象 App の実行を継続させる．

3.2　Syscall Logging

保護対象 App が発行したシステムコールは全て ShadowBuddy が捕捉する．EPT Switching によって保護対象 App と OS の遷移時を ShadowBuddy が捕捉することができるため，どのような原因で遷移が発生したのかを調べる．システムコールの場合には，システムコール番号と引数を保存し，ログに追加する．システムコールの呼び出し方にはいくつかの方法があり，int 命令によってソフトウェア割込みを用いて行うものと，SYSENTER 命令を用いて行うものと，SYSCALL 命令を用いて行うものがある．それぞれ OS のハンドラのジャンプ先は，IDT エントリ，MSR_IA32_SYSENTER_EIP，MSR_LSTAR に登録される．EPT Switching 発生時のジャンプ先がどれかを判定した上で，正しいシステムコールであれば記録を行う．

システムコール番号から引数の数と用途を調べて適切な処理を行う．システムコールは以下のように分類できる．

- (a) 無視するもの…getpid，write など
- (b) ログに追加するもの…mmap，open，dup2 など
- (c) 適切な処理を行ってからログに追加するもの…read など
- (d) ログを更新するもの…exit，nice など

(a) これらのシステムコールは OS 内のデータ構造から App がデータの読み込みを依頼するものであり，ShadowBuddy が何かをする必要も，ログに追加する必要もない．(b)OS にデータオブジェクトの確保を要求するシステムコールは，OS 再起動後にもう一度実行する必要がある．そのため引数とともにログに追加する．(c)OS と App が連携をするシステムコールのうち，OS から App のメモリに書き込むものなどが該当する．例えば read システムコール時には，OS がファイルなどから読み込んだ結果を App のメモリに書き込むが，ShadowBuddy は OS から App のメモリ領域に書き込むことを禁止してしまう．そのため，nEPT 中の書き込み先の GPA に対応するエントリには新しいバッファ用の物理メモリを割り当て，OS にはバッファ

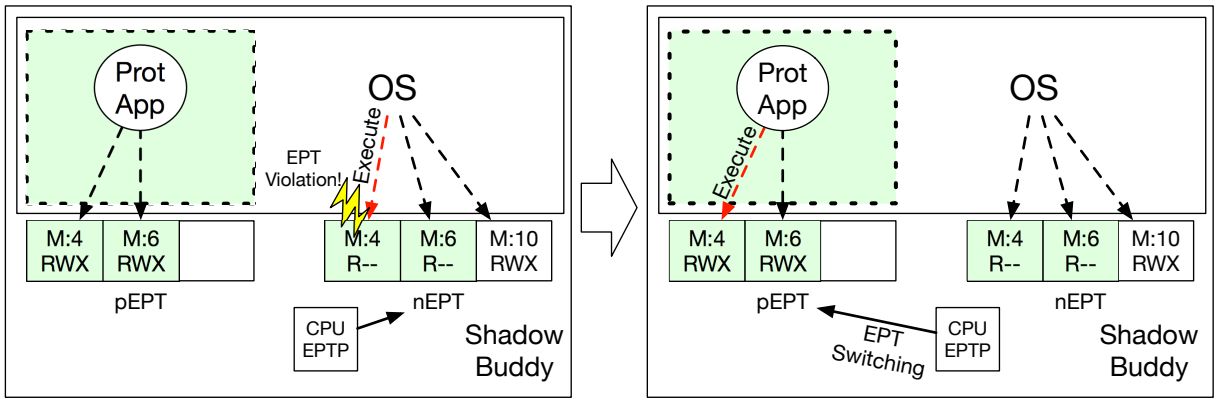

図 4　OS から App への実行切り替え時の EPT Switching

用のメモリに書き込ませてから，ShadowBuddy が実際に App のメモリ領域に書き込みを行う．(d) これらのシステムコールは新しいログを作成するのではなく，取得していたログを更新，削除していく．例えば exit 時には今まで取得していたログを削除する．

ログはシステムコール時に作成するが，OS からの返り値によっては破棄する．例えば nice 値の更新は，優先度を上げる場合には負の値を指定する．しかし優先度を上げることはスーパーユーザのみに許されている．ShadowBuddy は OS 内のデータ構造を直接参照することはないため，あるプロセスがスーパーユーザ作成したプロセスであるかを判定することはできない．そこで OS から App に戻る際のエラー番号を見て，成功していない場合には該当ログを破棄する．

4. 実装

ShadowBuddy は Xen 4.5.2 上に実装を行い，約 1900 行を追加した．OS には改変を加えていない．App からの VMCALL が発生してから保護を開始し，pEPT の作成を行って適宜 nEPT の書き込みや実行を禁止する．VMCALL 命令のハンドラには EPT Separation を実装，EPT Violation が発生した時に呼ばれるハンドラには EPT Switching，EPT Tracking，Syscall Logging を実装した．

5. 実験

ShadowBuddy によって発生するオーバヘッドを評価するために，nEPT と pEPT の切り替えや EPT の更新にかかる処理が，App の実行速度をどれだけ低下させるかを計測した．各プログラムはアセンブラで書かれた自作プログラムで，アセンブル時に必要最小限のライブラリと静的にリンクを行った．他の App とのメモリ共有は無く，ShadowBuddy が保護対象外の App から干渉を受けることはない条件で計測を行った．

実験は Xeon E3-1270 v2(8-core，3.5GHz)，16GB RAM，240GB SSD のマシンで行った．仮想マシンには 8vCPU，

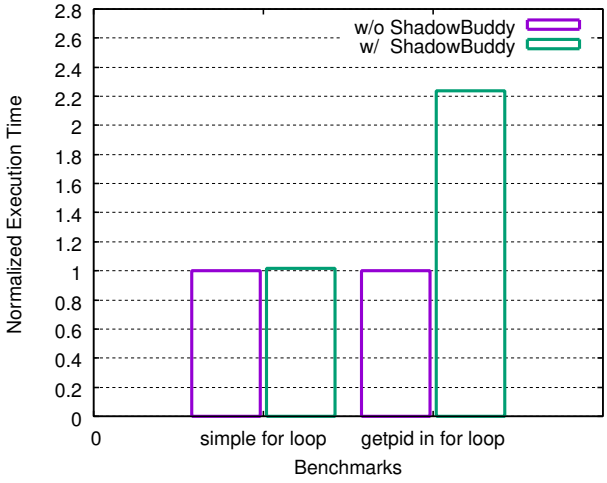

図 5　単純な繰り返しと getpid の繰り返しの実行時間

2GB RAM を割り当て，ゲスト OS は改変を行っていない Linux 4.10.12 を用いた．ただし実験で用いる int 0x80 によるシステムコールを可能とするオプションを付けている．

5.1 Protect Initialization

保護開始時に行う pEPT の作成と，nEPT の書き込み・実行の禁止にかかる時間を計測した．ページテーブルを辿って 585 ページ分の pEPT 作成するためにかかった時間は 2 ミリ秒であった．585 ページのうち 61 ページ分がページテーブル自体のページであり，524 ページ（ユーザ 353 ページ，カーネル 171 ページ）がプログラムが利用するものであった．保護開始に行う処理は，その時点でページテーブルに割り当てられているメモリサイズに比例する．

5.2 Overhead

ShadowBuddy を利用しないものと利用するものとで，ワークロードの実行時間の変化を計測した．**図 5** に結果を示す．

繰り返しによって 1 から 100,000,000 までカウントアップするプログラムを実行した．保護なしでは 174 ミリ秒，保護ありでは 177 ミリ秒と，オーバヘッドは 3 ミリ秒（2%）

であった．保護機構の初期化に 2 ミリ秒かかっていたため，計算処理が中心であればそれ以外のオーバヘッドはほとんど発生しない．

int 命令を用いて getpid システムコールを 100,000,000 回繰り返した．保護なしでは 20.7 秒，保護ありでは 46.3 秒と，オーバヘッドは 124%であった．システムコールが発生するたびに EPT の切り替えが 2 回行われ，切り替えのたびに TLB キャッシュと EPT キャッシュを毎回クリアしたことがオーバヘッドの原因となっている．VPID によって TLB フラッシュ範囲を抑制し，また挙動に合わせて不要なキャッシュクリアの削減を行うことで，このオーバヘッドは低減することができる．

6. 結論

OS クラッシュ発生時には App のユーザ空間メモリを保護し，再起動後に App の実行を保証する手法は存在しなかった．OS 内のプロセスメタ情報が破壊されたり，Checkpoint データが破壊される可能性があったり，OS 停止時には利用できなかったりしたためである．ShadowBuddy はユーザ空間のメモリ保護とシステムコールの監視によるチェックポイントの作成を行う．OS 再起動後の App 再実行を保証するため，OS クラッシュ時にも利用可能である．ShadowBuddy は最小で数%のオーバヘッドで，OS による App 保護機構と独立した保護機構を提供することができた．従来ハードウェア利用効率の向上を担ってきた VMM に対し，ShadowBuddy は App の可用性向上という新しい役割を与えることができ，信頼性の高い App 実行環境を提供することができる．

参考文献

[1] Dacey, R. F.: Effective Patch Management is Critical to Mitigating Software Vulnerabilities, Technical report, Government Accountability Office (2003).

[2] Corbet, J., Kroah-Hartman, G. and McPherson, A.: Linux Kernel Development – How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It (2010). `https://www.linuxfoundation.org/docs/lf_linux_kernel_development_2010.pdf`.

[3] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S. and Bairavasundaram, L.: How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems, *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pp. 26–36 (2011).

[4] Oppenheimer, D., Brown, A., Beck, J., Hettena, D., Kuroda, J., Treuhaft, N., Patterson, D. A. and Yelick, K.: ROC-1: Hardware Support for Recovery-Oriented Computing, *ACM Transactions on Computer Systems (TOCS)*, Vol. 52, No. 2, pp. 100–107 (2002).

[5] Depoutovitch, A. and Stumm, M.: Otherworld - Giving Applications a Change to Servive OS Kernel Crashes, *Proc. of the 5th ACM European Conference on Computer Systems (EuroSys '10)*, pp. 181–194 (2010).

[6] Duell, J., Hargrove, P. and Roman, E.: Requirements for Linux Checkpoint/Restart, Technical Report 8, Berkeley Lab Technical Report, LBNL-49659 (2002).

[7] Rieker, M., Ansel, J. and Cooperman, G.: Transparent User-Level Checkpointing for the Native POSIX Thread Library for Linux, *Proc. of 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '06)*, pp. 492–498 (2006).

[8] Baumann, A., Lee, D., Fonseca, P., Glendenning, L., Lorch, J. R., Bond, B., Olinsky, R. and Hunt, G. C.: Composing OS Extension Safely and Efficiently with Bascule, *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 239–252 (2013).

[9] Vogt, D., Giuffrida, C., Bos, H. and Tanenbaum, A. S.: Lightweight Memory Checkpointing, *Proc. of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pp. 474–484 (2015).

[10] David, F. M., Chan, E. M., Carlyle, J. C. and Campbell, R. H.: CuriOS: Improving Reliability through Operating System Structure, *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 59–72 (2008).

[11] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J. and Ports, D. R. K.: Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems, *Proc. of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)* (2008).

[12] Baumann, A., Peinado, M. and Hunt, G.: Shielding Applications from an Untrusted Cloud with Haven, *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 267–283 (2014).

[13] Yamakita, K., Yamada, H. and Kono, K.: Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery, *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pp. 169–180 (2011).

[14] Yamada, H. and Kono, K.: Traveling Forward in Time to Newer Operating Systems using ShadowReboot, *Proc. of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '11)* (2011).

[15] Lenharth, A., Adve, V. and King, S. T.: Recovery Domains: An Organizing Principle for Recoverable Operating Systems, *Proc. of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pp. 49–60 (2009).

[16] Kadav, A., Renzelmann, M. J. and Swift, M. M.: Fine-Grained Fault Tolerance using Device Checkpoints, *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 473–484 (2013).

[17] Smith, R. and Rixner, S.: Surviving Peripheral Failures in Embedded Systems, *Proc. of the 2015 USENIX Annual Technical Conference (ATC '15)*, pp. 125–137 (2015).

[18] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z. and Witchel, E.: InkTag: Secure Applications on an Untrusted Operating System, *Proc. of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 265–278 (2013).

[19] Cheng, Y., Ding, X. and Deng, R. H.: Efficient Virtualization-Based Application Protection Against Untrusted Operating System, *Proc. of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '15)*, pp. 345–356 (2015).

[20] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C., Shafi, H., Shanbhogue, V. and Savagaonkar, U.: Innovative Instructions and Software Model for Isolated Execution, *Proc. of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*, pp. 1–8 (2013).

[21] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., DanO' Keeffe, Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C. : SCONE: Secure Linux Containers with Intel SGX, *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 689–703 (2016).