Vol.2020-CSEC-91 No.1
Vol.2020-SPT-40 No.1
Vol.2020-EIP-90 No.1
2020/11/25

プロセス実行監視による CPU キャッシュを対象としたサイドチャネル攻撃に対する緩和機構

榎本 秀平[1]　葛野 弘樹[2]

概要：サイドチャネル攻撃はハードウェアのアーキテクチャ特性に基づいた攻撃手法であり，攻撃対象はクラウドにおけるサーバからモバイル端末まで多岐に渡る．近年，Meltdown / Spectre と呼ばれるサイドチャネル攻撃が深刻な問題とされており，従来，CPU ならびにオペレーティングシステムにより保護されていた機密情報の盗取が可能となる．防御のためにハードウェアに変更を加えることは非常に困難であり，ソフトウェアによる既存の防御手法は適用による性能の劣化を招く点が課題である．本研究では，Meltdown / Spectre の利用する FLUSH+RELOAD 攻撃に関する CPU 命令に着目し，攻撃緩和と検出を低オーバーヘッドで達成する新たな手法を提案する．提案手法では，FLUSH+RELOAD がリークデータの復元のために CPU キャッシュをフラッシュする点に着目し，プロセスのキャッシュフラッシュ命令実行をカーネルにて透過的に禁止することでサイドチャネル攻撃の緩和を実現する．提案手法を Linux にて実装し，評価実験の結果，攻撃プロセスによるサイドチャネル攻撃を緩和可能であること，ならびに既存の防御機構より 2.04 % から 19.05 % 低いオーバヘッドであることを確認した．

キーワード：サイドチャネル攻撃対策，オペレーティングシステム，システムセキュリティ

1. はじめに

コンピュータシステムに対する攻撃手法のひとつとしてサイドチャネル攻撃がある．サイドチャネル攻撃は CPU やメモリ等のシステムを構成するハードウェアの動作特性に着目し，動作を観察することでシステム内の機密データを不正取得を試みる．汎用的に使用されるハードウェアがサイドチャネル可能な特性や脆弱性を持つ場合，その影響は非常に広範囲に及ぶ．x86 アーキテクチャの場合，一般的なデスクトップ・ラップトップコンピュータに留まらず，Amazon Web Service（AWS） [1] をはじめとするクラウドシステムへの影響が考えられる．さらに，該当のサイドチャネル攻撃が ARM 等の他のアーキテクチャに転用可能な場合，Internet of Things（IoT） を構成する組込み機器やモバイル端末への影響も考えられる．

近年，深刻なサイドチャネル攻撃として Meltdown [2,3] や Spectre [4] が報告されている．Meltdown / Spectre は Intel CPU の投機的実行機能をサイドチャネル攻撃に利用する．Meltdown ではオペレーティングシステム（OS） 内においてカーネルや他のアプリケーションが持つ機密データのリークが可能となり，Spectre では攻撃対象となるアプリケーション内の機密データのリークが可能となる．

Meltdown / Spectre は OS の種類に依存しないため，Windows，macOS，ならびに Linux など幅広い OS に対する攻撃が考えられる．また，投機的実行は Intel CPU に限らず，ARM 等の CPU にも実装されており，広範囲への影響が考えられる．

Meltdown / Spectre の防御にあたり，投機的実行を禁止するように CPU の実装を変更することが最も有効であるが，一般的なユーザが実装を変更することは困難である．投機的実行は CPU のパフォーマンスを向上させるための機能であるため，ハードウェア実装で禁止することは大幅な性能劣化につながる．また，ソフトウェアベースの防御手法においても適用により性能劣化が指摘されており，従来手法には以下の課題が存在する．

- **ソフトウェアベースの防御手法における課題**
  Linux における Meltdown の防御手法である Kernel Page Table Isolation（KPTI） [5,6] の適用により，約 22 % 程度のオーバーヘッドが発生する．Spectre の緩和手法である Retpoline [7] は適用により，約 66 % 程度のオーバーヘッドが発生すると報告されている [8]．KPTI のオーバーヘッドを軽減する手法 [9] が提案さ

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[2] セコム株式会社 IS 研究所
Intelligent Systems Laboratory, SECOM Co., Ltd., Japan

Vol.2020-CSEC-91 No.1
Vol.2020-SPT-40 No.1
Vol.2020-EIP-90 No.1
2020/11/25

れているが，仮想化環境を前提としており，Meldown の発生しうる幅広い OS や CPU に広く適用することはできない．

本研究では，Meltdown / Spectre 等のサイドチャネル攻撃を低オーバーヘッドに緩和可能，かつ様々な環境に対して広く適用可能な新たな手法である FlushBlocker を提案する．FlushBlocker は CPU キャッシュを利用する攻撃手法のひとつである FLUSH+RELOAD に着目し，アプリケーションの CPU キャッシュフラッシュ命令のみを禁止することで低オーバーヘッドと正しい機密データの読み出し防止の両立を実現可能とする． また，FlushBlocker はカーネル内のコンポーネントとして動作するため，先行研究とは異なり，適用対象が仮想化環境のみに制限されることはない．本研究の貢献を以下に示す．

(1) FLUSH+RELOAD 攻撃の緩和機構 FlushBlocker を提案した．FlushBlocker によりサイドチャネル攻撃を緩和し，安全性の高いシステムを実現可能とした．
(2) キャッシュフラッシュ命令をトラップすることで透過的な実行の禁止と命令のモニタリングを実現する FlushBlocker の設計と実装を示した．
(3) Linux 5.7.15 に実装を行い，評価として，Meltdown / Spectre の PoC を防御可能であること，FlushBlocker の回避策対策が有効であることを検証し，提案手法のパフォーマンスを測定した．性能評価の結果より，既存の防御手法より 2.04 % から 19.05 % 低いオーバヘッドで実行可能であることを確認した．

2. 背景

2.1 FLUSH+RELOAD 攻撃

Meltdown / Spectre [2–4] は Intel CPU および一部の ARM CPU の脆弱性を利用したサイドチャネル攻撃である．CPU の備える投機的実行機能を利用することで，従来読込みにあたり特権レベルの必要なページや条件分岐により読込まれることがないページの読込みを可能にする．

投機的実行により，実行された命令の演算結果は CPU アーキテクチャ内の専用バッファに格納される．サイドチャネル攻撃にて投機的実行を利用した場合，レジスタや物理メモリに対しては直接的な影響を与えない．一方，演算結果は CPU キャッシュにストアされるため，CPU キャッシュに対しては直接的な影響を与える．サイドチャネル攻撃では，投機的実行内で不正にページの読み込みを行うことで内容を CPU キャッシュに残し，投機的実行終了後にキャッシュから内容を取り出すことで機密データのリークを可能とする．CPU キャッシュを利用したサイドチャネル攻撃のうち，キャッシュフラッシュ命令を使用するものは FLUSH+RELOAD [10] 攻撃と呼ばれる．FLUSH+RELOAD は **図 1** に示すように三段階のフェーズより構成される．例として，1 バイト分の機密データをリークさせる状況について以下に説明する．

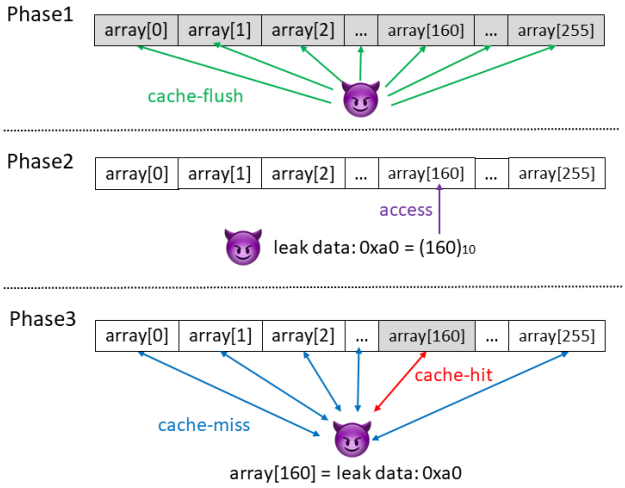

図 1 FLUSH+RELOAD を用いた機密データの格納と取得

フェーズ 1: キャッシュのフラッシュ

はじめに，攻撃者は不正なプロセスを OS 内にロードする．プロセスは $array[0]$ から $array[255]$ までの要素を持つ 1 バイト配列を用意する．なお各要素はページアライメントされており，1 要素につき 1 ページが割り当てられる．プロセスはキャッシュフラッシュ命令を用いて 256 個の全要素を CPU キャッシュから追い出す．

フェーズ 2: キャッシュへの機密データ格納

次にプロセスは Meltdown / Spectre などを用いて不正にページを読み込み，取得したデータをインデックスとする配列にアクセスする．例として，Meltdown を用いて 1 バイトのデータをカーネルページから読み込む場合， 0 から 255 までのいずれかの値が取得されるため，取得した値をインデックスとする $array[0]$ から $array[255]$ までの 1 バイト配列のいずれかにアクセスする．

フェーズ 3: キャッシュからの機密データ取得

最後にプロセスは $array[0]$ から $array[255]$ のうち，どの要素がキャッシュにストアされているか確認する．そのために要素へのアクセスの前後にタイムスタンプを取り，アクセス時間を計測する．CPU キャッシュにストアされている場合，アクセス時間は非常に小さくなり，ストアされていない場合，アクセス時間は大きくなる．アクセス時間の計測操作を全ての要素で行うことでキャッシュヒットする要素のインデックスが判明し，攻撃者に 1 バイト分の機密データがリークされる．

2.2 想定する脅威モデル

本研究の脅威モデルとして, 攻撃者は FLUSH+RELOAD ベースのサイドチャネル攻撃を用いて任意データのリークを行うことを想定し，ターゲットとなる OS を一般ユーザ権限で操作できるものとする．また，OS のセキュリティ機能として，Kernel Adress Space Layout Randomization

Vol.2020-CSEC-91 No.1
Vol.2020-SPT-40 No.1
Vol.2020-EIP-90 No.1
2020/11/25

（KASLR） [11] は無効とし，Data Execution Prevention（DEP） [12] は有効であるものとする．

3. 提案手法

本研究では FLUSH+RELOAD を用いたサイドチャネル攻撃を防御するための新たな手法 FlushBlocker を提案する．提案手法が満たす要件を以下に示す．

要件 1：低オーバーヘッドに攻撃緩和を実現する．
要件 2：幅広い環境に対し適用可能とする．

要件 1 を満たすため，FlushBlocker をカーネルへ適用可能とし，攻撃緩和機能による OS への負荷を最小限に留めるよう設計する．これにより，アプリケーションをネイティブ動作時に近い性能で実行可能とする．

要件 2 を満たすため，FlushBlocker は特定のハードウェア機能の利用を前提とせずに設計する．様々なプラットフォームへの移植性の考慮により，幅広い環境に適用可能とする．

3.1 提案手法のアプローチ

脅威モデルにおいて，攻撃者は任意のプロセスを生成可能である（2.2 節を参照）．攻撃者のプロセス（以下，攻撃プロセス）は FLUSH+RELOAD を用いて攻撃対象である OS 内の機密データのリークを試みる．FlushBlocker では，攻撃緩和として，FLUSH+RELOAD がキャッシュアクセス時間の計測操作時にキャッシュフラッシュ命令を発行する点に着目し，攻撃プロセスの発行するキャッシュフラッシュ命令を透過的に禁止する．

FlushBlocker を適用した OS において，攻撃プロセスはキャッシュフラッシュ命令を実行不可能であり，一部または全ての配列の要素がキャッシュにストアされた状態で攻撃が開始される．そのため，キャッシュから攻撃対象の機密データを取得するフェーズにおいて，複数の要素がキャッシュヒットし，サイドチャネルを行えないことから機密データの復元は極めて困難となる．

4. 設計

4.1 プロセスの実行監視

FlushBlocker は，透過的に FLUSH+RELOAD を捕捉し無効化するため，実行監視としてプロセスの発行するキャッシュフラッシュ命令をトラップし，該当命令の実行をスキップする．トラップには CPU アーキテクチャの持つハードウェアブレークポイント機能であるデバッグレジスタを使用する．プロセス開始時に，プロセスがトラップ対象のキャッシュフラッシュ命令を含むかのスキャン，およびトラップのためのデバッグレジスタの設定を行う．

FlushBlocker の全体の処理の流れを**図 2** に示し，各段階での処理内容を以下に説明する．

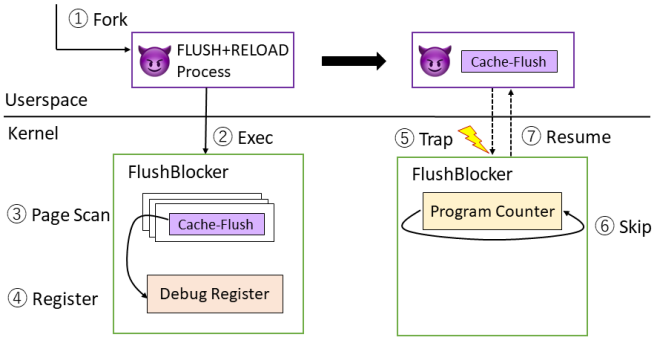

図 2 FlushBlocker によるキャッシュフラッシュ命令の禁止の流れ

(1) 攻撃者はシェルに攻撃プログラム名を入力する．シェルは clone システムコール により新規プロセスを生成する．
(2) 生成されたプロセスは exec システムコールにより攻撃プログラムをメモリにロードする．
(3) FlushBlocker は exec システムコール呼び出しを起点とし，プロセスの持つ全てのコードページをスキャンする．スキャンとして，対象のページにキャッシュフラッシュ命令が含まれるか否かを確認する．
(4) キャッシュフラッシュ命令がページに含まれる場合，該当する命令の仮想アドレスをデバッグレジスタに設定し，プロセスを監視対象としマーキングする．
(5) 攻撃プロセスは FLUSH+RELOAD を開始し，キャッシュフラッシュ命令を実行する．
(6) FlushBlocker はキャッシュフラッシュ命令実行をトラップし，トラップ対象プロセスが監視対象マーキング済みか否かを確認する．マーキング済みであればトラップ対象のプロセスへブレークポイントシグナルを送信せず，プロセスのプログラムカウンタを次の命令に進める．
(7) 攻撃プロセスはキャッシュフラッシュ命令の次の命令から実行を再開する．

FlushBlocker はプロセスによるキャッシュフラッシュ命令の実行を透過的にスキップするため，一般のアプリケーションにキャッシュフラッシュ命令が含まれている場合においても動作は継続される．また，スキャン対象はプロセスのページのみであるため，カーネルにおけるキャッシュフラッシュ命令には影響を及ぼさない．特定の命令を禁止することで一般アプリケーションへ及ぼす影響については 6 章にて述べる．

4.2 FlushBlocker の回避策とその対応

FlushBlocker が適用された OS に対して攻撃を試みる場合，以下の回避策が考えられる．

回避策 1：攻撃プロセスから新規プロセスやスレッドを生成し，新規プロセスやスレッドにてキャッシュフラッ

シュ命令を実行する場合

回避策 2：攻撃プロセスの実行中にキャッシュフラッシュ命令を含む実行可能ページを新規生成し，新規ページを用いて命令を実行する場合

回避策 1 として，既存の OS の実装において，デバッグレジスタはスレッド単位で設定されるため，デバッグレジスタを設定したプロセスから生成された新規プロセスやスレッド中でキャッシュフラッシュ命令が実行された場合，デバッグレジスタは無効であり，命令をトラップすることは不可能となる．回避策 1 への対応として，FlushBlocker は clone システムコールをトリガーにデバッグレジスタの設定を生成元プロセスから生成先のプロセスやスレッドにコピーし，生成先においてもキャッシュフラッシュ命令のトラップを可能にする．

回避策 2 として，プロセスにて exec システムコール以降に実行可能ページを新たにマップし，新規ページにキャッシュフラッシュ命令を含ませた場合， exec システムコール発行時のみのページスキャンでは新規ページにおけるキャッシュフラッシュ命令は捕捉できず，トラップは不可能となる．FlushBlocker における回避策 2 への対応の流れを示す．

(1) 攻撃プロセスは mmap システムコールにより実行可能ページを新規生成する．
(2) FlushBlocker は mmap システムコール呼び出しを起点とし，マップ先の仮想アドレスとサイズを記録する．
(3) 攻撃プロセスはキャッシュフラッシュ命令を含むペイロードをマップ先にコピーする．
(4) FlushBlocker はデマンドページングによるページフォールトをトリガーとして，ページフォールト時の仮想アドレスが事前に記録した範囲内か否かを確認する．範囲内の場合は該当仮想アドレスのページの実行権を無効にする．
(5) 攻撃プロセスはマップ先にコピーされたペイロード中のキャッシュフラッシュ命令を実行する．
(6) FlushBlocker はページの実行権無効によるページフォールトを起点とし，ページフォールト時の仮想アドレスが事前に実行権を無効にしたページか否かを確認する．該当ページの場合はページスキャンを行い，ページの実行権を有効にする．

以上により，回避策 2 への対応として，攻撃プロセスの実行中に新規実行可能ページをマップした場合においても，新規ページに含まれるキャッシュフラッシュ命令のトラップを可能にする．

5. 実装

本稿における FlushBlocker の実現環境は x86 アーキテクチャの Linux を想定する．本章では，FlushBlocker の実装内容について述べる．

5.1 ページスキャン

Flushblocker におけるページスキャンは， exec システムコールの do_exec 関数呼び出しをトリガーとし，対象プロセスの利用するページマップ後に FlushBlocker の持つ flushblocker_exec 関数を呼び出す．flushblocker_exec 関数では対象プロセスの mm_struct 構造体から vm_area_struct 構造体を辿り，プロセスのコード領域のページ，ならびに該当ページの仮想アドレスを取得する．次に，該当ページ内にキャッシュフラッシュ命令が含まれているか否かをチェックする．

x86 において，プロセスから非特権命令として発行可能なキャッシュフラッシュ命令は clflush，clflushopt の二種類である．Flushblocker でのページスキャンのチェック処理では，スキャン対象のページ内にこれら二種類のキャッシュフラッシュ命令パターンが含まれているか検索する．例として，clflush 命令は 2 バイトのオペコード 0x0F 0xAE と 1 バイトの Mod/RM より構成されるため，合わせた 3 バイトのパターンマッチングを行う．

5.2 デバッグレジスタの設定

Flushblocker におけるデバッグレジスタの設定では，Flushblocker の持つ register_user_hw_breakpoint 関数を用いて対象となるキャッシュフラッシュ命令の仮想アドレスをデバッグレジスタに登録する．Flushblocker では，監視対象プロセスのマーキングとして，予め task_struct 構造体に追加したフラグエントリを有効にする．register_user_hw_breakpoint 関数の呼出し時の引数に指定する perf_event_attr 構造体に，HW_BREAKPOINT_X を設定する．HW_BREAKPOINT_X の指定により，監視対象としたプロセスが対象の仮想アドレスの実行を試みた場合，ハードウェアデバッグ割り込みが発生する．

x86 はデバッグレジスタとして DR0 から DR7 までの 8 つのレジスタを持つ．DR0 から DR3 までの 4 つのレジスタにトラップ対象の仮想アドレスをセット可能である．デバッグレジスタへの仮想アドレスのセットやアンセットは特権命令であり，攻撃プロセスが不正にレジスタの操作を行うことは困難である．デバッグレジスタは ptrace システムコールを用いることで操作可能だが，ptrace システムコールの発行はスーパユーザ権限が必要となり，本稿での脅威モデル（2.2 節）の前提条件より攻撃プロセスが発行することは困難である．

5.3 キャッシュフラッシュ命令のトラップ

Flushblocker でのキャッシュフラッシュ命令のトラップでは， do_debug 関数呼び出しをトリガーとし，ハー

ドウェアデバッグ割り込み対象のプロセスが監視対象としてマーキングされている否かをチェックする．監視対象かのチェックにはデバッグレジスタ登録時に設定した task_struct 構造体内の監視対象かのフラグエントリを用いる．マーキングされている場合，監視対象プロセスによるキャッシュフラッシュ命令発行と判断し，プロセスの IP レジスタを次の命令にスキップする．マーキングされていない場合は通常用途によるデバッグ割り込みと判断し，プロセスに SIGTRAP シグナルを送信する．

5.4 FlushBlocker を前提とした攻撃への対応

5.4.1 clone システムコールによるトラップ回避策 1

FlushBlocker での回避策 1 への対策として，do_fork 関数呼び出しをトリガーとし，プロセスの監視対象有無のマーキングチェックを行う．マーキングされている場合，FlushBlocker の持つ flushblocker_clone 関数を呼び出す．flushblocker_clone 関数では生成元スレッドの持つデバッグレジスタの内容を生成先スレッドにコピーする．

5.4.2 mmap システムコールによるトラップ回避策 2

FlushBlocker での回避策 2 への対策では，ksys_mmap_pgoff 関数呼び出しをトリガーとし，mmap システムコール呼び出し時のフラグに PROT_EXEC が含まれているかチェックする．PROT_EXEC を含む場合，mmap により確保されたページの仮想アドレスとサイズを task_struct 構造体に追加した専用のエントリに記録する．次に do_page_fault 関数呼び出しをトリガーとし，ページフォールト対象の仮想アドレスをチェックする．記録した範囲の仮想アドレスであれば FlushBlocker の持つ flushblocker_demand_fault 関数を呼び出す．flushblocker_demand_fault 関数では，対象仮想アドレスの vm_area_struct 構造体から VM_EXEC フラグをクリアした vm_flags をページテーブルエントリに設定し，page 構造体に追加したフラグエントリを有効にする．

監視対象プロセスにより対象ページ内のコードが実行された場合，再度 do_page_fault 関数が呼び出される．その際に page 構造体のフラグエントリをチェックし，有効の場合は FlushBlocker の持つ flushblocker_nx_fault 関数を呼び出す．flushblocker_nx_fault 関数では 5.1 節と同様のページスキャン，5.2 節と同様のデバッグレジスタ設定を行い，VM_EXEC フラグをセットした vm_flags をページテーブルエントリに設定し，page 構造体のフラグエントリを無効にする．

6. 評価

提案手法に対する評価実験として，実際の PoC を用いたサイドチャネル攻撃を防御可能かの検証，回避策への対応検証，パフォーマンスに与える負荷の測定，ならびに通常のアプリケーションへのキャッシュフラッシュ命令の調査を行う．評価の目的と内容を以下に示す．

（評価 1）攻撃の検証
提案手法を適用した OS において，Meltdown および Spectre V1 PoC 実行時に攻撃対象のカーネル変数の仮想アドレスを指定し，FlushBlocker により攻撃を防御可能か評価した．

（評価 2）回避策対策の検証
提案手法を適用した OS において，clone および mmap システムコールを利用した回避策 1 ならびに回避策 2 のサンプルプログラムに対し，FlushBlocker により clflush 命令を実行防止可能か評価した．

（評価 3）パフォーマンスの測定
提案手法を適用した OS において，マイクロベンチマークならびに実アプリケーションを動作させ，オーバヘッドを測定した．

（評価 4）通常アプリケーションのキャッシュフラッシュ命令調査
通常利用するアプリケーションにおいて，キャッシュフラッシュ命令がどの程度含まれているか調査した．

評価実験環境として **表 1** に示す物理計算機を使用した．提案手法の実装を Linux のバージョン 5.7.15 に行い，398 行の追加で実現した．

表 1 実験環境

Machine	Thinkpad T440p
CPU	Intel(R) Core(TM) i7-4800MQ @ 2.70GHz
CPU core	8
Memory	16384 MiB
Kernel	Linux 5.7.15
Distribution	Ubuntu 18.04.2 LTS

6.1 攻撃の検証

はじめに FlushBlocker 適用，KPTI 未適用下において，既存の Meltdown PoC [13] を実行した場合に攻撃が成立するか否かを確認した．次に FlushBlocker 適用，Spectre V1 緩和機構未適用下において，既存の Spectre V1 PoC [14] を実行した場合に攻撃が成立するか否かを確認した．

各 PoC の実行結果を **表 2** に示す．既存のサイドチャネル対策手法ならびに FlushBlocker 未適用下では PoC による攻撃は成功する．一方，FlushBlocker 適用により，Meltdown と Spectre V1 ともにリークが防止されることが確認された．

6.2 回避策対策の検証

FlushBlocker の回避策 1 として clone システムコールを介して clflush を実行した場合，および回避策 2 として，mmap システムコールを介して clflush を実行した場合において，clflush 命令を透過的に防止可能か確認する．

回避策 1 および回避策 2 を利用した clflush 命令の実行

表 2　サイドチャネル攻撃結果の比較（✓ 攻撃防止成功; – 攻撃成功）

攻撃種別	概要	提案手法未適用	提案手法適用
Meltdown	KPTI 未適用下にて PoC [13] を実行	–	✓
Spectre V1	Spectre V1 緩和機構未適用下に PoC [14] を実行	–	✓

表 3　回避策対応の比較（✓ clflush 実行防止成功; – clflush 実行成功）

攻撃種別	概要	回避策対策無効	回避策対策有効
回避策 1	clone を利用した子プロセスにて clflush 実行	–	✓
回避策 2	mmap 実行後の作成ページ上にて clflush 実行	–	✓

結果を **表 3** に示す．いずれの場合でも，FlushBlocker の回避対策は動作し，clflush 命令の実行は防止されることを確認した．

6.3　パフォーマンスの測定

専用のプログラム，UnixBench [15] を用いたマイクロベンチマーク，および ApacheBench [16] を用いたベンチマークを以下の 3 種類の OS 環境で実行し，結果を比較した．

(1) FlushBlocker, KPTI, Spectre V1 緩和機構, Retpoline（Spectre V2 緩和機構）全て未適用下のネイティブ OS
(2) FlushBlocker のみ適用下の OS
(3) KPTI，Spectre V1 緩和機構，Retpoline 全て適用下の OS

6.3.1　マイクロベンチマークによる測定

マイクロベンチマークは fork および exec を発行する C プログラム，および UnixBench 5.1.3 の二種類について性能を測定した．fork / exec プログラムは親プロセスから子プロセスを生成し，子プロセス上で pwd コマンドを実行し，10,000 回の実行結果の中央値を計測する．

fork / exec プログラムの実行結果を **図 3** 左 に示す．既存の防御機構適用下における実行時間は約 12.96 % 程度のオーバーヘッドに対し，FlushBlocker 適用下における実行時間は約 10.92 % 程度のオーバーヘッドを示した．

UnixBench の実行結果を 図 3 中央 に示す．既存の防御機構適用下のパフォーマンススコアが約 20.77 % 程度の減少に対し，FlushBlocker 適用下は約 10.91 % 程度の減少を示した．

6.3.2　実アプリケーションによる測定

Apache httpd 2.4.46 を起動した上で同一マシン内から ApacheBench 2.3 を実行し，性能を測定した．ベンチマーカーは 100 クライアントが並列に 100 リクエストを送信し，1 秒あたりのリクエスト数を比較する．

ApacheBench の実行結果を 図 3 右に示す．既存の防御機構適用下のスループットは約 22.31 % 程度の減少に対し，FlushBlocker 適用下では 約 3.26 % 程度の減少を示した．以上の結果から，FlushBlocker は既存機構よりネイティブに近い性能を達成していることが確認された．

表 4　キャッシュフラッシュ命令を含むアプリケーションの解析結果

解析環境	解析数	clflush	clflushopt
Ubuntu Linux 20.04.1 LTS Desktop	3,479	1	0

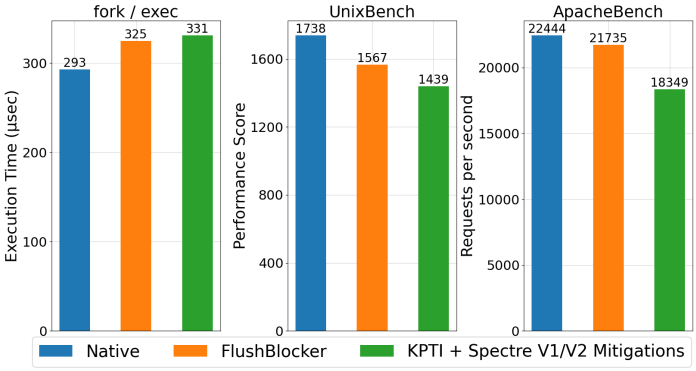

図 3　各ベンチマークプログラムの実行結果

6.4　通常アプリケーションのキャッシュフラッシュ命令調査

FlushBlocker はキャッシュフラッシュ命令の実行を透過的に防止する．通常のアプリケーション動作への影響が懸念されるため，通常アプリケーションにおいて，clflush および clflushopt 命令がどの程度含まれるか調査した．

調査結果を **表 4** に示す．調査環境は Ubuntu Linux 20.04.1 LTS Desktop とした．標準的にインストールされるアプリケーションの ELF バイナリ 3,479 個を静的解析し，キャッシュフラッシュ命令として，clflush 命令を含む ELF バイナリ 1 個のみ検出した．該当アプリケーションは gnome-control-center であり，GUI 環境の GNOME の設定パネルを描画し，管理に用いるプログラムである．

7.　考察

7.1　評価への考察

FlushBlocker は Meltdown / Spectre V1 PoC に対する FLUSH+RELOAD 攻撃を防止可能なこと，ならびに FlushBlocker 回避策 1 / 回避策 2 に対して，キャッシュフラッシュ命令の実行回避を防止可能なことを評価にて示した．性能評価では，従来の防御機構よりオーバヘッドを約 2.04 % から 19.05 % 程度軽減可能なことを示した．

FlushBlocker はプロセス実行前のページスキャンならびに FLUSH+RELOAD 攻撃発生時にのみデバッグレジス

タから呼び出される処理を必要とすることから，従来の防御手法よりも軽量であると考えられる．特に，プロセス起動後に mmap の発行を伴わない ApacheBench にて負荷が小さくなる傾向を示した．また，回避策 1 への対策はプロセス生成時のデバッグレジスタ処理の追加のみと負荷は少ない．一方，回避策 2 への対策はページフォルト処理を意図的に発生させることから，今後，意図的にページフォルトを発生させた場合の性能負荷の測定を行う予定である．

7.2 提案手法の拡張

攻撃プロセスが FlushBlocker を想定し，より高度な回避手法を持つ場合における対応を行う必要がある．回避事例として，Linux における FlushBlocker の実装では，PROT_EXEC フラグ付きの mmap システムコールによる回避は防御可能であるが, mprotect システムコールにより，ページ作成後に実行可能領域化するような回避策については防御不可能である．FlushBlocker における mprotect を用いた回避策への対策はプログラムの動作順序について考慮する必要がある．まず，mmap 呼び出し後に対象領域のページが生成されていない状態で mprotect が呼び出された場合には，mprotect 時に対象領域の先頭アドレスとサイズを記録することでページフォールトをトリガーとする命令のスキャンが可能となる．一方， mmap 呼び出し後に対象領域のページを生成した上で mprotect が呼び出された場合，ページフォールトをトリガーとすることが不可能であるため mprotect 時に命令スキャンを行う必要がある．そのため，mprotect 実行内で対象領域ページの存在の有無に応じて処理を切り替える必要がある．

7.3 実装の検討

x86 アーキテクチャにおける FlushBlocker は， 4 個のハードウェアデバッグレジスタを用いたトラップを行う．攻撃者が 5 個以上のキャッシュフラッシュ命令を配置している場合に全てをトラップすることができない．5 個以上のキャッシュフラッシュ命令のトラップに対応するため，5 個目以降のキャッシュフラッシュ命令を含むページの実行権限を落とす．次に，ページフォールト時にデバッグレジスタへ登録し，デバッグレジスタから追い出されたキャッシュフラッシュ命令を含むページの実行権限を同様に落とす．これにより，数に制約のあるデバッグレジスタでキャッシュフラッシュ命令の全てをトラップすることが可能となる．

7.4 移植可能性

Linux における FlushBlocker は exec システムコール呼出し時に監視対象プロセスのページ上のキャッシュフラッシュ命令有無のスキャンを行い，x86 のハードウェアデバッグレジスタを利用して実行を補足している．他 OS においてもプロセス実行前にページのスキャンが可能である場合，かつ x86 アーキテクチャ上では，Linux と同様の設計にて実現可能であると考えている．例として，Windows ではカーネルモードドライバ API の PsSetCreateProcessNotifyRoutine [17] 関数により，プロセス生成時のトラップが可能であり，PsGetContextThread / PsSetContextThread [18] 関数を用いることでアプリケーションスレッドにおけるデバッグレジスタの取得と設定が可能である．

他 CPU アーキテクチャにてデバッグレジスタを利用可能である場合，FlushBlocker は実現可能であるが，デバッグレジスタの利用が困難な場合は異なる実装を検討する必要がある．

表 5 先行研究との比較（✓ 対応済; △ 部分的に対応可能）

比較項目	KPTI [5]	EPTI [9]	提案手法
低オーバヘッド		✓	✓
IoT / モバイル機器対応	△		✓
FLUSH+RELOAD 攻撃対応	✓	✓	✓
その他サイドチャネル攻撃対応	✓	✓	△

8. 関連研究

8.1 攻撃に関連した先行研究

Meltdown の補助的な役割を果たすサイドチャネル手法として FLUSH+RELOAD [10] がある．FLUSH+RELOAD は CPU の L3 キャッシュをターゲットにプロセス間の共有メモリの内容をリークさせる手法であり，キャッシュフラッシュ後にアクセス時間を計測し，アクセス時間が小さい部分をリークを見なす方法は Meltdown がキャッシュから機密データを取得するフェーズにおいて活用可能である．

8.2 防御に関連した先行研究

KPTI のオーバーヘッドを軽減させることを目的とした先行研究として EPTI [9] がある．EPTI は仮想化環境において，ゲスト OS のページテーブル分離を Intel Extended Page Table（EPT） 上で行うことでゲスト OS に KPTI を適用することなく透過的に分離を実現している．また，EPT の切り替えには VMFUNC 命令を使用する．VMFUNC は，Extended Page Table Pointer（EPTP）を変更する命令であり，VMX Non-root mode の非特権命令として動作するため，高速な切り替えを実現可能である．

8.3 先行研究との比較

先行研究である KPTI [5] ならびに EPTI [9] と FlushBlocker の比較を**表 5** に示す．KPTI はページテーブルの切り替え処理が高コストであり，KPTI の適用により約 22 % 程度のオーバーヘッドが発生することが報告されている [8] ．KPTI による Meltdown の防御とパフォーマンスはトレードオフの関係といえる．EPTI は仮想化

環境を前提としており，IoT やモバイル機器等の仮想化を使用しない環境における防御は想定していない．また KPTI と EPTI は共に攻撃プロセスの検出は不可能であるため，例えば Meltdown の機能を含むマルウェア [19] を検出して停止させるようなことはできない．FlushBlocker は，FLUSH+RELOAD 攻撃の検出を行いサイドチャネル攻撃を防止する．現在の実装では攻撃プロセスの停止機能は存在しないが，プロセス毎にキャッシュフラッシュ命令発行数を監視することで FLUSH+RELOAD 実行の可能性があるプロセスを停止させることが可能である．しかしながら，ページテーブルの分離は伴わないため，キャッシュフラッシュ命令を用いないサイドチャネル攻撃には利用される命令毎への対応が必要である．

9. おわりに

Meltdown / Spectre 等のサイドチャネル攻撃への対策として，KPTI や Retpoline などの導入が行われた．ソフトウェアによる対応には性能低下の影響があり，先行研究では，仮想化機能を利用した高速化の可能性が示されたが幅広い環境での適用には課題が存在する．

本稿では，FLUSH+RELOAD を用いたサイドチャネル攻撃にキャッシュフラッシュ命令が伴うことに着目し，プロセスからのキャッシュフラッシュ命令を透過的に捕捉，無効化する機構である FlushBlocker を提案した． FlushBlocker はカーネル内のコンポーネントとして動作し，プロセスの利用するページのスキャンとハードウェアブレークポイント機能のデバッグレジスタを組み合わせることでキャッシュフラッシュ命令を透過的に禁止し，攻撃者が攻撃対象とする機密情報を CPU キャッシュから読み取ることを困難にする．

評価として， FlushBlocker を最新の Linux に実現し，Meltdown / Spectre による FLUSH+RELOAD 攻撃を行う PoC を防御可能であること，FlushBlocker を前提とした攻撃による回避策を検討し，実際に対応可能であることを確認した．FlushBlocker は OS への僅かな変更で適用できることから，評価にて従来の防御機構より負荷を 2.04 % から 19.05 % 軽減する結果となり，仮想化機能を利用せずに性能への影響を低減できることを示した．

参考文献

[1] Amazon Web Services Inc: Amazon Web Services (AWS) Cloud Computing Services, https://aws.amazon.com/. (accessed 2020-10-03).
[2] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M.: Meltdown: Reading Kernel Memory from User Space, *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, USENIX Association, pp. 973–990 (2018).
[3] Google: Reading privileged memory with a side-channel, https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. (accessed 2020-10-03).
[4] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution, *40th IEEE Symposium on Security and Privacy (S&P'19)*, pp. 1–19 (2019).
[5] Kernel.org: Page Table Isolation (PTI), https://www.kernel.org/doc/html/latest/x86/pti.html. (accessed 2020-10-03).
[6] jiiksteri: Kernel page-table isolation merged, https://lwn.net/Articles/742404/ (2017). (accessed 2020-10-03).
[7] Paul Turner: Retpoline: Binary mitigation for branch-target-injection (aka "Spectre"), https://lwn.net/Articles/742980/ (2018). (accessed 2020-10-03).
[8] Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M. and Yuan, D.: An Analysis of Performance Evolution of Linux's Core Operations, *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, New York, NY, USA, Association for Computing Machinery, pp. 554–569 (2019).
[9] Hua, Z., Du, D., Xia, Y., Chen, H. and Zang, B.: EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs, *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, USENIX Association, pp. 255–266 (2018).
[10] Yarom, Y. and Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, USENIX Association, pp. 719–732 (2014).
[11] Jake Edge: Kernel address space layout randomization, https://lwn.net/Articles/569635/ (2013). (accessed 2020-10-03).
[12] Microsoft Corporation: DEP/NX Protection, https://docs.microsoft.com/en-us/windows/win32/win7appqual/dep-nx-protection. (accessed 2020-10-07).
[13] paboldin: meltdown-exploit, https://github.com/paboldin/meltdown-exploit. (accessed 2020-10-03).
[14] crozone: SpectrePoC, https://github.com/crozone/SpectrePoC. (accessed 2020-10-17).
[15] kdlucas: byte-unixbench, https://github.com/kdlucas/byte-unixbench. (accessed 2020-10-03).
[16] The Apache Software Foundation: ab - Apache HTTP server benchmarking tool, http://httpd.apache.org/docs/2.4/programs/ab.html. (accessed 2020-10-22).
[17] Microsoft: PsSetCreateProcessNotifyRoutine function (ntddk.h), https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine. (accessed 2020-10-27).
[18] Process Hacker: KProcessHacker/include/ntfill.h File Reference, https://processhacker.sourceforge.io/doc/ntfill_8h_source.html#l00308. (accessed 2020-10-27).
[19] Fortinet: Meltdown/Spectre Update, https://bit.ly/3gE2lMq (2018). (accessed 2020-10-03).