Vol.2019-DPS-178 No.10
Vol.2019-CSEC-84 No.10
2019/3/4

カーネルに対する攻撃における独自の仮想記憶空間の切替え手法の検出能力と防御手法

葛野 弘樹[1,2]　山内 利宏[1]

概要：オペレーティングシステム（OS）の特権奪取，さらにセキュリティ機構のバイパスを目的としたカーネル脆弱性を介した攻撃への対策が求められている．攻撃者は攻撃により，本来許可されていないカーネル管理の仮想記憶空間を改ざんし，任意のプログラムコードを動作させる．OS の攻撃対策として，仮想記憶空間上にカーネルコード・データをランダム配置する KASLR，カーネルモードとユーザモードの仮想記憶空間分離の KPTI，および制御フロー監視の CFI が導入された．また，特権最小化の強制アクセス制御やケイパビリティも利用可能である．CPU では，カーネルへの仮想記憶空間のデータ読込・実行を制御する SMAP/SMEP がある．これら対策手法により，ユーザモードからカーネルへの攻撃を緩和，および特権制限を行える．しかし，カーネルモードでのみ攻撃を行う ret2dir も提案され，カーネルの仮想記憶空間を改ざんされる可能性はある．我々は，カーネルに対し独自の仮想記憶空間を用意し，カーネルの仮想記憶空間を監視するセキュリティ機構を提案している．本稿では，独自の仮想記憶空間とカーネルの仮想記憶空間の切替えパターン毎の監視処理での攻撃検出タイミングの差異，提案手法による既存のセキュリティ機構保護，ならびに提案手法へのカーネルモードにて受ける攻撃と対策について提案する．Linux にてこれら提案を実現し，攻撃検出性能と提案手法での攻撃対策の有効性および仮想記憶空間の切替えパターン毎の性能オーバヘッドを評価し，考察を行う．

キーワード：カーネル脆弱性，カーネル保護，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

攻撃者は，カーネル脆弱性を利用し攻撃を行うことでオペレーティングシステム（OS）カーネルの特権奪取やセキュリティ機構の回避を試みる．カーネル脆弱性による攻撃により，カーネルの仮想記憶空間に置かれた権限情報を改ざんされた場合，特権奪取は成立する．また，アクセス制御に対するセキュリティ機構の呼出し回避はカーネルの仮想記憶空間上のカーネルコード改ざんにより行われる．

OS においては，特権の最小化を目的とした強制アクセス制御 [1] ならびにケイパビリティ [2] により，攻撃者の特権奪取後の被害低減は可能である．カーネルでの攻撃の防止として，スタック領域の監視 [3]，カーネルの仮想記憶空間上にある特権情報の監視 [4]，カーネルの関数呼び出し制御フローの完全性を検証する CFI（Control Flow Integrity）[5]，および，カーネルの仮想記憶空間におけるカーネル関数とデータ部配置をランダム化する KASLR (Kernel Address Space Layout Randomization) がある [6]．また，CPU においてはユーザモードの仮想記憶に配置された攻撃コードの参照および実行を防ぐ SMAP/SMEP[7] がある．

カーネルの仮想記憶空間へのサイドチャネル攻撃である Meltdown は，ユーザモードからカーネルの仮想記憶空間に対する参照可能性を示した．これに対し，ユーザモードとカーネルモードにおける仮想記憶空間を分離する手法 [8] が提案され，KPTI（Kernel Page Table Isolation）として Linux など各種 OS にて実装された．

カーネル脆弱性による攻撃の特徴として，ユーザモードとカーネルモードの相互の利用があり，既存の対策により攻撃成功率の低減ならびに被害最小化は可能である．しかし，ret2dir などカーネルモードのみ利用した攻撃も提案され [9]，カーネルの仮想記憶空間上にあるカーネルコード・データを保護し，攻撃対策を行うことは重要である．

我々は，カーネルの仮想記憶空間を監視するためのセキュリティ機構として独自の仮想記憶空間をカーネルに用意し，その仮想記憶空間上において監視処理を動作させる手法を提案した [10]．先の研究では，カーネルモードにお

[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] セコム株式会社 IS 研究所
Intelligent Systems Laboratory, SECOM CO., LTD., Japan

いてカーネル処理中に独自の仮想記憶空間からカーネルモジュール領域を監視し，不正なモジュールの検出可能性を示した．本稿では，独自の仮想記憶空間への複数の切替えパターンと攻撃の検出対応能力，モジュール領域以外にカーネルのセキュリティ機構の保護ならびに我々のセキュリティ機構への攻撃対応手法について提案する．

(1) カーネルのシステムコール実行前後，実行中における独自の仮想記憶空間への切替えパターンの違いによる攻撃の検出タイミングならびに検出可能な攻撃の差異
(2) 提案手法による任意のカーネルコード・データの保護として，カーネルの仮想記憶空間上に配置されたアクセス制御などカーネルの備える既存のセキュリティ機構のデータの改ざん検出
(3) 仮想記憶空間管理において処理速度の高速化のため物理記憶空間を仮想記憶空間にマッピングする手法があり，提案手法も攻撃対象となる．このため，物理記憶空間の一部を仮想記憶空間へのマッピングから除外し，提案手法のセキュリティ機構を攻撃から防ぐ手法

提案を KPTI に対応した Linux に実現し，攻撃検出能力の評価実験，および性能オーバヘッドを評価した．本稿での研究の貢献ならびに得られた結果は以下の通りである：

- カーネルの仮想記憶空間に対し，独自の仮想記憶空間から監視するセキュリティ機構において仮想記憶空間の切替えパターンによるカーネルへの攻撃の検出タイミングの差異を示した．この方式により既存のセキュリティ機構への攻撃を検出できることを示し，また物理記憶空間の仮想記憶空間へのマッピング手法を改良し，提案手法への攻撃可能性の低減を実現した．
- 最新の Linux にて KPTI と組み合わせた提案手法の実現方式，および事例としてカーネル脆弱性を利用した攻撃に対する仮想記憶空間の切替えパターン毎の検出能力，既存のセキュリティ機構への攻撃の検出，ならびに物理記憶空間の仮想記憶空間へのマッピング処理の変更による攻撃低減について評価した．ベンチマークによる性能評価により，オーバヘッドはシステムコール実行前に監視を行う仮想記憶空間の切替えパターン 1 が 0.729 ms から 1.99 ms および，システムコール実行後に監視を行う仮想記憶空間の切替えパターン 3 が 0.406 ms から 5.029 ms であることを示した．

2. 背景知識

2.1 カーネル脆弱性を用いた攻撃

カーネル脆弱性には，実装上の不具合から複数に分類される [11]．攻撃者はカーネル脆弱性を起点とし，カーネルに対し任意のプログラムコードを不正な処理として実行させ，特権奪取あるいはセキュリティ機構の回避を行う．Linux では，特権奪取のため権限情報の格納される `struct cred` 構造体の `uid` 変数を書換える．セキュリティ機構の回避として，LSM（Linux Security Module）のフック関数を管理する `security_hook_list` 変数を書換える．

書換え対象のカーネルコードやデータはカーネルの仮想記憶空間に配置され，通常は OS での特権レベル制御によりユーザモードからはアクセスできない．しかし，仮想記憶空間に物理記憶空間をマッピングする領域（ダイレクトマッピング）を持つ仮想記憶空間管理方式では，ret2dir のように，ダイレクトマッピングを利用した攻撃や，カーネルモードにおいてカーネル脆弱性を介し不正なプログラムコードを配置した場合，カーネルに対し攻撃は可能である．

2.2 ユーザモードとカーネルモードの仮想記憶空間の分離

カーネルの仮想記憶空間は，CPU およびカーネルにて実行時の特権レベルの保護を前提に，仮想記憶管理の効率性や割込み処理の高速化のためにユーザモードとカーネルモードで同一の仮想記憶空間を利用していた．しかし，Meltdown サイドチャネル攻撃により，ユーザモードからカーネルの仮想記憶空間に対する参照可能性が示された．

Meltdown 対策として，ユーザモードとカーネルモードの仮想記憶空間を分離が提案された [8]．分離は，ユーザモードで動作するプロセス向けの仮想記憶空間（ユーザの仮想記憶空間）とカーネルモードで動作するカーネル向けの仮想記憶空間（カーネルの仮想記憶空間）を用意し行い，ユーザモードからカーネルモードに遷移した際に，ユーザの仮想記憶空間からカーネルの仮想記憶空間に切替える．ユーザの仮想記憶空間におけるカーネルコードは仮想記憶空間の切替え部分のみマッピングし，Meltdown による参照範囲を最小限に低減している． Linux においては KPTI [8] として，また他の OS にも同等の機能が実装された [12].

2.3 想定する脅威モデル

本稿での脅威モデルとして，カーネル脆弱性を介した攻撃によるカーネルの仮想記憶空間上の改ざん対象は，セキュリティ機構の配置された領域，および物理記憶空間のマッピングされた領域のみとする．なお，BIOS ，CPU，MMU，TLB，ならびにその他ハードウェアは安全とする．

3. 提案手法と実現方式

3.1 提案手法による仮想記憶空間の切替えと監視

我々の提案しているセキュリティ機構は，カーネルの仮想記憶空間上のカーネルコード・データの改ざん検出を目的とし，カーネルモードにおいて独自の仮想記憶空間を用いてカーネルの仮想記憶空間を監視し，改ざん検出を可能である（**図 1** を参照）．攻撃の検出事例として，ルートキット等のカーネルの仮想記憶空間を操作を意図した不正なモジュールを挿入中に検出し，挿入自体の未然防止を可能であることを示した [10].

本稿で提案する項目は次の通りである．

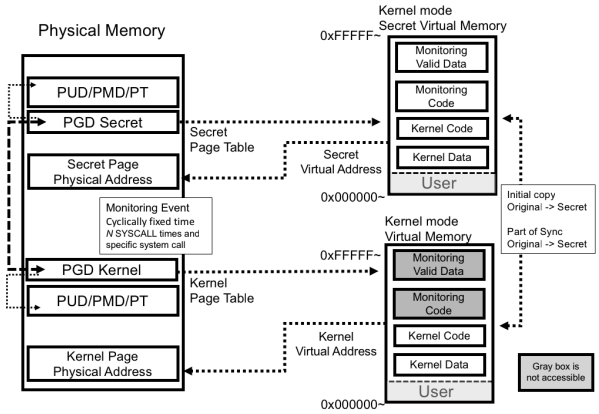

図 1　独自の仮想記憶空間を用いた監視機構

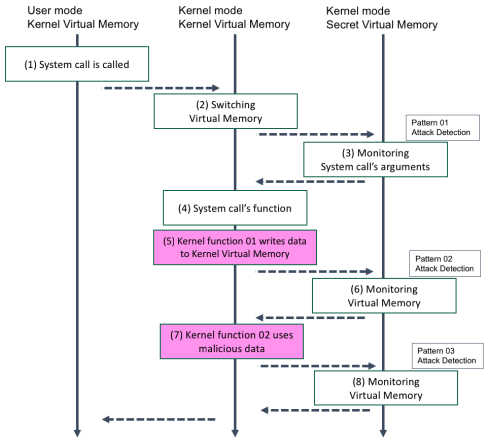

図 2　仮想記憶空間の切替えパターン 1，2 および 3

（提案 1）仮想記憶空間の切替えパターンと検出能力
　カーネルの仮想記憶空間と監視用の仮想記憶空間の切替えパターンを複数検討し，カーネルの処理において，パターン毎に監視を実行する箇所の異なること，ならびにカーネルの仮想記憶空間を改ざんに対する検出タイミングの差異を明らかにする
（提案 2）仮想記憶空間の切替えによるカーネルの保護
　従来までの監視対象であるカーネルモジュール領域に加え，独自の仮想記憶空間による新たなカーネルの保護として，カーネルのセキュリティ機構の監視を行う
（提案 3）仮想記憶空間管理における提案手法の保護手段
　カーネルの仮想記憶空間にて，ダイレクトマッピング領域にある監視用のカーネルコード・データの物理記憶空間を除外し，ダイレクトマッピング経由での提案手法への攻撃手法を無効化する手段を提案

3.2　仮想記憶空間の切替えパターンと検出能力

提案するセキュリティ機構での仮想記憶空間の切替えパターンとして，ユーザモードからカーネルモードへの遷移後のカーネル処理との関連により，3 つのパターン（**図 2** を参照）を挙げる．

パターン 1：カーネルの処理前に監視用仮想記憶空間に切替え監視

パターン 2：カーネルの処理中に監視用仮想記憶空間に切替え監視

パターン 3：カーネルの処理後に監視用仮想記憶空間に切替え監視

いづれもユーザモードからカーネルモードに遷移したのち，カーネル処理に割込む形式である．監視タイミングは異なるため，カーネル脆弱性を利用した攻撃の検出にあたり，利用可能な情報はパターン毎に違う．パターン 1 の場合はシステムコール引数を監視対象，パターン 2，およびパターン 3 の場合はカーネルの仮想記憶空間上のカーネルコード・データを監視対象とする．

また，監視タイミングの違いとして，パターン 1 ないしパターン 3 はカーネルの処理前，処理後の 1 度のみ監視を実行可能である．パターン 2 はカーネルの処理中の任意のタイミングにおいて監視を実行可能である．監視にあたり，複数パターンを適用し監視タイミングを組合わせることは可能である．しかし，実現方式におけるセキュリティ機構への攻撃を考慮し，独自の仮想記憶空間への切替え処理の設置場所とオーバヘッドの影響に基づき決定しなければならない．

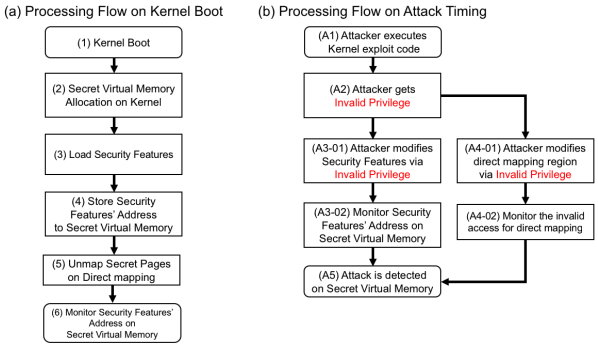

図 3　独自の仮想記憶空間を用いた監視データの確保とアンマップ処理および監視の流れ

3.3　仮想記憶空間の切替えによるカーネルの保護

提案するセキュリティ機構にて，従来の保護対象としていたカーネルモジュール領域に加え，新たにカーネルの仮想記憶空間上にある指定されたカーネルコード・データの格納領域を保護対象とする．監視の前処理にて，正当データは仮想記憶空間上においてアクセス可能な仮想アドレス自体，ないし物理記憶空間から読出し可能なデータとする．監視用の正当データ確保流れを**図 3** に示し，説明する．

(1) カーネル起動中に監視用の仮想記憶空間を物理記憶空間に割当て確保
(2) カーネルの仮想記憶空間において監視対象カーネルコード・データを設定
(3) 監視対象カーネルコード・データを監視用の仮想記憶空間に正当データとして保存

(4) 監視用の仮想記憶空間および監視処理をダイレクトマッピング領域よりアンマップ
(5) 監視用の仮想記憶空間を用いて監視処理を開始

監視処理中の攻撃検出の流れを説明する（図 3 を参照）．

(A1) 攻撃者はカーネル脆弱性を利用しカーネルへの攻撃を行う
(A2) 攻撃者は特権奪取を行う
(A3-01) 攻撃者による特定のカーネルの仮想記憶空間上のカーネルコード・データの改ざん
(A3-02) 監視用の仮想記憶空間による改ざんされたカーネルコード・データを正当データと比較し検査
(A4-01) 攻撃者によるダイレクトマッピング領域を介した監視用の仮想記憶空間・監視処理への攻撃
(A4-02) アンマップされたダイレクトマッピング領域への参照に対しカーネルにより例外を発生
(A5) 監視用の仮想記憶空間上にて攻撃検出と判定

攻撃検出の判定は，仮想記憶空間の切替えパターン毎において監視用の仮想記憶空間に切替え，監視処理を実行した際，監視対象のカーネルコード・データが正当なデータと異なる場合，およびアンマップされたダイレクトマッピング領域への参照を攻撃と判断することで行う．監視処理を終了し，カーネルの仮想記憶空間に切替え，改ざんの判定結果を処理する．攻撃検出時には，カーネルとしての動作の継続性を損なわず，適切な処理を行う．例として，システムコールや割込み処理の中断，あるいはカーネルの処理を中断しユーザモードからの要求であるシステムコールに対しエラー値を返すなど動作に失敗した場合として処理する．また，必要に応じ，改ざんされた仮想記憶空間の書き戻し処理も可能とする．

3.4 仮想記憶空間管理における提案手法の保護手段

仮想記憶空間の管理において，物理記憶空間の割当てや解放処理の高速化のためにダイレクトマッピングとしてカーネルの仮想記憶空間に物理記憶空間をマッピングする方式がある．ret2dir やカーネルモードにて完結するカーネル脆弱性を用いた攻撃では，ダイレクトマッピング領域を利用してカーネルコード・データを改ざんする．

我々のセキュリティ機構は，監視用の仮想記憶空間をカーネルの仮想記憶空間とは異なる物理記憶空間に割当てる．カーネルの仮想記憶空間からは独立しており，カーネルへの攻撃による影響は受けない．新たな保護手段として，ダイレクトマッピング領域に含まれる監視用の仮想記憶空間を攻撃対象から除外を行う．ダイレクトマッピング領域より監視用の仮想記憶空間および切替え処理や監視処理の領域をあらかじめ特定し，アンマッピングを行う．これにより，カーネルの仮想記憶空間において行われたカーネル脆弱性の攻撃により，セキュリティ機構の用いる物理記憶空間へ直接影響を及ぼすことを回避し，監視処理の阻害を防止する．カーネルの仮想記憶空間から提案するセキュリティ機構の利用している物理記憶空間をアンマップする処理を説明する．

(1) カーネルの起動中にカーネルの仮想記憶空間に物理記憶空間をマッピング
(2) 監視用の仮想記憶空間を確保し，物理記憶空間における領域を確定
(3) 物理記憶空間における監視用の仮想記憶空間および監視用コードの領域を特定
(4) ダイレクトマッピング領域より監視用の仮想記憶空間および監視用コード領域をアンマップ

一連の処理はカーネルの起動中に実行される．物理記憶空間のアンマップ処理は，カーネルの仮想記憶空間に対して物理記憶空間をマッピング，および監視用の仮想記憶空間の用いる物理記憶空間を確保した直後に行う（図 3 を参照）．これにより，カーネルの仮想記憶空間において，ダイレクトマッピング領域を介してセキュリティ機構で用いる物理記憶空間の操作を防ぎ，監視用の仮想記憶空間に影響を与えることを避ける．

3.5 実現方式

実現環境として OS は Linux，CPU アーキテクチャは x86_64 を想定する．

3.5.1 仮想記憶空間の切替えと監視処理

パターン 1 から 3 の実現にあたり，実現方式において，監視用の仮想記憶空間は，Linux カーネルの仮想記憶空間を指す `init_mm` 変数の `pgd` 変数から 4 ページサイズ分（x86_64 では 16Kbytes）を論理和した物理アドレスとする．また，監視用の仮想記憶空間に，カーネル起動時にカーネルの仮想記憶空間が確保された後，カーネルコードおよびデータを `pgd` 変数から複製する．

実現方式における，各パターンでの仮想記憶空間の切替えと監視処理を説明する．

パターン 1：`entry_SYSCALL_64` 関数におけるシステムコール関数呼出し前，監視対象となるシステムコール番号および引数をスタックに退避し，監視用仮想記憶空間の物理アドレスを CR3 レジスタに書込み仮想記憶空間を切替え，監視処理にてシステムコール番号および引数を `pt_regs` 変数を介して検査

パターン 2：カーネル処理中に監視用仮想記憶空間の物理アドレスを CR3 レジスタに書込み，仮想記憶空間を切替え，監視対象データと正当なデータの差異を検査

パターン 3：`entry_SYSCALL_64` 関数におけるシステムコール関数呼出し後，監視用仮想記憶空間の物理アドレスを CR3 レジスタに書込み仮想記憶空間を切替え，監視対象データと正当なデータの差異を検査

パターン 1 から 3 において，監視処理の終了後，カーネルの仮想記憶空間への切替えは `current` 変数における

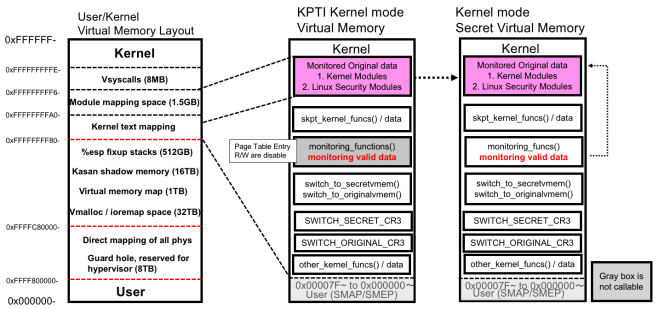

図 4　Linux カーネルでの提案機構および監視対象の仮想記憶空間領域

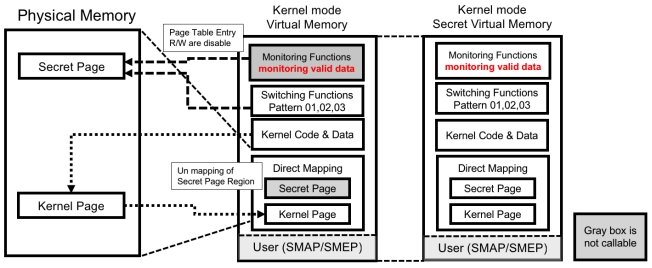

図 5　Linux カーネルでのアンマップ処理対象の仮想記憶空間領域

（task_struct 構造体）内にある active_mm 構造体の pgd 変数の物理アドレスを CR3 レジスタに書込むことで行う．監視用の仮想記憶空間にて監視の実行中に割込み要求を受けた際は，割込みハンドラ関数にてカーネルの仮想記憶空間に切替え，割込み処理を行う．

3.5.2　実現方式におけるカーネルの保護領域

実現方式においては，従来のカーネルモジュールに加え，Linux での強制アクセス制御を提供するセキュリティ機構のカーネルコードである LSM を監視し保護対象とした（図 4 を参照）．Linux ではカーネル起動時，security_init 関数にて LSM を設定するための各種強制アクセス制御の初期化関数を呼出す．初期化関数においては，security_add_hooks 関数を利用して security_hook_list に強制アクセス制御で利用するフック関数群を登録する．SELinux の場合，フック関数群の selinux_hooks 変数を security_hook_list へ登録する．

監視の前処理として，カーネルの起動時，security_init 関数の終了後において，security_hook_list に登録される selinux_hooks などのフック関数群の関数アドレスを監視用の仮想記憶空間の領域に正当なデータとして複製する．実現方式における LSM の監視処理では，監視用の仮想記憶空間に切替え後，カーネルの仮想記憶空間上にある security_hook_list 変数に登録されたフック関数群の関数アドレスと正当なデータとして保存したフック関数群の関数アドレスの比較を行い，カーネル脆弱性を用いた攻撃などの不正な操作による改ざんの有無を検出する．

3.5.3　実現方式における仮想記憶空間管理手法

Linux 4.4 系列（x86_64）では，ダイレクトマッピング領域としてカーネルの仮想記憶空間のうち 0xFFFF880000000000 - 0xFFFFC7FFFFFFFFFF，64TBytes の範囲に物理記憶空間を割当て，ページ単位に管理された物理記憶空間のうち，64TBytes の範囲はダイレクトマッピング経由で操作される．カーネルの仮想記憶空間上のダイレクトマッピング領域外のカーネルコード・データにも，ダイレクトマッピング領域の仮想アドレスを経由し，参照可能である．

Linux ではカーネルの起動時，ダイレクトマッピング領域は init_mem_mapping 関数にて作成され，kernel_physical_mapping_init 関数にて，カーネルの仮想記憶空間に物理記憶空間をマッピングする．提案手法に対するダイレクトマッピング領域の仮想アドレスを利用した参照を防ぐため，実現方式では，カーネル起動時に監視処理のカーネルコード・監視に用いる正当なデータの物理アドレスを確定した後，remove_pagetable 関数を用い，カーネルの仮想記憶空間において，該当する物理アドレスを管理するページをダイレクトマッピング領域より削除し，アンマップ処理を行う．（図 5 を参照）．アンマップした領域への参照した場合，カーネルにおいてはページフォルトが発生し，パニックとして処理される．

4.　評価

4.1　評価の目的と評価環境

提案手法に対する評価の目的と内容を以下に示す．

（評価 1）システムコールの引数監視判断の実験
提案手法を適用したカーネルにおいて，特定のシステムコールの実行前に，システムコールの引数に対し，監視用の仮想記憶空間の切替えパターン 1 にて正しく監視処理可能か評価した．

（評価 2）セキュリティ機構に対する攻撃監視判断の実験
提案手法の適用環境にて，カーネルの強制アクセス制御を管理する LSM への攻撃に対し，監視用の仮想記憶空間の切替えパターン 2 および 3 にて正しく検出可能か評価，また，攻撃検出にかかる時間を測定した．

（評価 3）ダイレクトマッピング領域のカーネルデータに対する攻撃防止の実験
提案手法の導入後，ダイレクトマッピング領域に置かれる監視用の正当なデータに対してアンマッピング処理を行うことで参照不可能となるか評価した．

（評価 4）提案手法の適用後カーネルのオーバヘッド測定
提案手法を適用したカーネルにおいて，ベンチマークソフトウェアを動作させ，仮想記憶空間の切替えパターン毎のオーバヘッドを測定した．

評価用計算機は，Intel(R) Core(TM) i7 7700HQ (2.80GHz, 4 コア，メモリ 16GB, OS は Debian 9.0 (Linux Kernel 4.4.165, x86_64) とした．また，ハイパースレッディング機能は無効化し，カーネルの利用 CPU は 1 つとした．

```
// Switching to secret virtual memory and monitoring, pattern01
[  58.690804] target system call
[  58.690821] system call number: 0000000000000139
[  58.721702] module name: malicious_module
[  58.721731] Invalid malicious_module
// Switching to kernel virtual memory, pattern01
[  58.727898] malicious_module: module license 'unspecified' taints kernel.
[  58.728542] Disabling lock debugging due to kernel taint
[  58.772438] Attack Module Init
```

(a) Monitoring of Linux System Call's Arguments

```
[ 148.612904] lsm_malicious_module: module license 'unspecified' taints kernel.
[ 148.613605] Disabling lock debugging due to kernel taint
[ 148.695219] calling change_lsm_address_attack()
[ 148.696546] Original Address ffffffff812d5c70
[ 148.696574] Malicious Address ffffffffa0000000
[ 148.696592] Override Address ffffffffa0000000
[ 148.696606] finishing change_lsm_address_attack()
// Switching to secret virtual memory and monitoring, pattern02
[ 148.713330] Invalid lsm function is detected
[ 148.713354] Address ffffffff812d5c70    (Valid), ffffffffa0000000    (Invalid)
// Switching to kernel virtual memory, pattern02
// Switching to secret virtual memory and monitoring, pattern03
[ 251.950933] Invalid lsm function is detected
[ 251.950957] Address ffffffff812d5c70    (Valid), ffffffffa0000000    (Invalid)
// Switching to kernel virtual memory, pattern03
```

(b) Monitoring of Linux Security Module's Function

図 6 提案手法によるシステムコール引数の監視時のログ，およびセキュリティ機構への攻撃検出時のログ

4.2 システムコールの引数監視判断の実験

評価において，ルートキットの挿入を想定し，モジュール追加に使われるシステムコール `init_module` および `finit_module` を監視対象とし，監視用の仮想記憶空間の切替えパターン 1 において，正当なモジュール名かどうかを比較し，検出結果をログに出力した．監視対象システムコールは “target system call”，そして不正なモジュール検出時は “Invalid module name” と文字列表示する．

提案手法によるシステムコール引数の監視時のログ出力を**図 6** に示す．左から，カーネル起動時からの時刻，ログ出力文字列となる．ログ出力結果より，提案手法のシステムコール引数監視は正しく行われている．カーネルでのシステムコール実行ならびにモジュール初期化関数の実行 0.05 ms 前に，引数の監視として提案手法はモジュールを読込み，モジュール名に対する監視判断を処理している．

4.3 セキュリティ機構に対する攻撃監視判断の実験

評価では，強制アクセス制御を提供する LSM のフック関数を上書きするモジュールを自作し，監視用の仮想記憶空間の切替えパターン 2 およびパターン 3 にて監視の際に正当なデータと比較し，差異の検出結果をログ出力することで行った．正規の関数アドレスはカーネル起動時に監視用の仮想記憶空間に保持させ，不正なモジュールは挿入後，`selinux_hooks` 変数に格納される SELinux の関数アドレスの一つをモジュールの関数アドレスにて上書き処理を行う．差異検出時のログでは，正当なデータと異なる関数アドレスの検出時に “Invalid lsm function is detected”，および “Virtual Address (Invalid)” と文字列表示する．

実験における提案手法による差異の検出時のログ出力を図 6 に示す．ログ出力結果より，不正なモジュールによるセキュリティ機構の上書きに対し提案手法の検出は正しく行われている．挿入処理開始から 0.018 ms 後にモジュール内のセキュリティ機構の関数アドレス上書き関数が呼ばれ，0.1 ms でパターン 2 の上書き検出，103.25 ms でパターン 3 の上書き検出が行われている．

```
[ 143.610533] calling change_kernel_physmap_data_attack()
[ 143.612688] valid data 1st virtual address: ffffffff820de600 (address), ffffffff812d5c80 (data)
[ 143.612922] valid data pfn: 00000000000020de
[ 143.612989] valid data pfn phys: 00000000020de000
[ 143.613009] valid data pfn virtual address: ffff8800020de000
[ 143.613218] valid data 2nd virtual address: ffff8800020de600 (address)
[ 143.613234] valid data 2nd virtual address: ffffffff812d5c80 (data)
// Unmapping valid data on direct mapping
[ 143.614117] BUG: unable to handle kernel paging request at ffff8800020de6005>] string.isra.4+0x65/0xd0
```

(c) Unmapping of the Valid Data Variable from Direct Mapping Region

図 7 提案手法によるダイレクトマッピング領域のカーネルデータに対する攻撃防止時のログ

4.4 ダイレクトマッピング領域のカーネルデータに対する攻撃防止の実験

提案手法の効果を実験するため，監視に用いる正当なデータをダイレクトマッピング経由で上書きするモジュールを自作し，カーネルの仮想記憶空間上での正当なデータへの参照を可能とした．モジュールの挿入後，正当なデータのアンマッピング処理，およびダイレクトマッピング経由の上書きを試行し，その結果をログに出力した．

提案手法においてアンマッピング処理後のカーネルにおけるログ出力を**図 7** に示す．ログ出力において，不正なモジュールは挿入後，正当なデータの仮想アドレスからダイレクトマッピング上の仮想アドレスを計算し，参照を行う．その後，提案手法で利用するアンマッピング処理を行った結果，ダイレクトマッピング上の仮想アドレスを利用した値の上書き時にカーネルへのページリクエストが発生し，正当なデータへの上書きは失敗している．

4.5 仮想記憶空間の切替えパターンのオーバヘッド測定

オーバヘッド測定では，提案手法適用前の Linux カーネルと仮想記憶空間の切替えパターン 1，および 3 適用後の Linux カーネルにて Apache 2.4.25 ならびに ApacheBench 2.4 を用いて評価した．評価条件として，通信経路 100Mbps，同時接続数 1 とし，1KB，10KB，100KB のファイルに 10 万回アクセスした際の 1 リクエストの平均値を算出した．クライアントは，Intel(R) Core(TM) i5 4200U （1.6GHz，2 コア），メモリ 8GB，OS は Windows 8 の計算機とした．

提案手法での仮想記憶空間の切替えパターンの差異によりカーネルへ与える影響を測定するため監視処理は行わない．パターン 1 はシステムコール実行前に都度，監視用の関数を呼出し，システムコール 10 回毎に仮想記憶空間の切替えを行う．パターン 2 はパターン 3 と同様の監視処理のため，パターン 3 のみシステムコール実行後に都度，監視用の関数を呼出し，システムコール 1000 回毎に仮想記憶空間の切替えを行う．評価結果について**表 1** に示す．

提案手法の適用後においてのオーバヘッドは仮想記憶空間の切替えパターン 1 が 0.729 ms から 1.99 ms，およびパターン 3 が 0.406 ms から 5.029 ms となった．Apache Bench のオーバヘッドは対象となる Web サーバプロセスのシステムコール発行回数に比例して変化する．このため，オーバヘッドの差異は提案手法による監視用の関数呼出し，および仮想記憶空間の切替えの負荷を示している．

表 1　ApacheBench による提案手法を適用した Linux における仮想記憶空間切替えパターン 1 および 3 のオーバヘッド（単位：ms）

ファイルサイズ（KB）	適用前（T0）	適用後		オーバヘッド	
		パターン 1 （T1）	パターン 3 （T3）	（T1-T0）	（T3-T0）
1	1.089	3.079	1.495	1.99 (64.63%)	0.406 (27.15%)
10	1.895	2.266	2.413	0.371 (16.37%)	0.518 (21.46%)
100	10.024	10.753	15.053	0.729 (6.77%)	5.029 (33.4%)

5. 考察

5.1 評価に対する考察

提案手法の適用環境にて，監視用の仮想記憶空間への切替えパターン 1，2，および 3 に対し，いづれも適切な監視タイミングにて検出処理を行えることを確認できた．評価より，カーネルの脆弱性を利用した攻撃に対し，パターン 1 によりシステムコール引数を利用した場合は攻撃発生前，パターン 2 ないし 3 により直接カーネルの仮想記憶空間を操作された場合は攻撃発生後の任意のタイミング，また遅くともシステムコール終了後に攻撃を検出可能である．

ApacheBench を用いた評価にて，提案手法のオーバヘッドはパターン 1 が 0.729 ms から 1.99 ms，およびパターン 3 が 0.406 ms から 5.029 ms である．パターン 1 では，システムコールの呼出し回数に対しファイルの転送時間などアプリケーション処理時間が増加し，相対的にオーバヘッドコストは低減していると考えられる．パターン 3 では，処理時間は増加しているが，全体に対するオーバヘッドの割合は大きく変化せず，仮想記憶空間の切替え処理のみ一定の負荷があると考えられる．仮想記憶空間の切替え処理には TLB Flush を伴う， 影響低減のため TLB キャッシュタグ付けを行う ASID （Address Space Identifier，x86_64 では PCID （Process Context ID））利用を検討している．

システムコール引数監視のオーバヘッドはシステムコール引数の種別毎により異なる負荷を伴うと考えている．文字列やアドレス情報の負荷は少ないが，一定量のデータ読込みを伴うと負荷は高いと言える．カーネルの仮想記憶空間の監視オーバヘッドは監視対象数に依存する．監視対象カーネルコードやデータを増加した場合，監視用の正当データとの比較に必要な処理時間は増加すると考えている．

5.2 仮想記憶空間の切替えパターンに対する考察

仮想記憶空間の切替えとして 3 つのパターンの検討と監視実験を行った．各パターンを組合わせることでカーネルへの攻撃の起点時から仮想記憶空間への影響を与え，攻撃を完了するまでの全体の処理を捕捉できる．

パターン 1 ないしパターン 2 は，システムコール処理完了前に攻撃を検出可能であることから，攻撃の未然防止に利用可能と考えている．特にパターン 1 では，監視結果をもとに実際のシステムコールの実行可否を判断でき，カーネルの仮想記憶空間への影響を与えずに攻撃を防止できる．パターン 2 では，カーネルの仮想記憶空間は改ざんに対し，システムコールから呼出される一連のカーネル処理の任意のタイミングで監視を行える．. このため，監視タイミングの挿入箇所を適切に配置しシステムコールの処理を中断させるなど攻撃の影響を最小限に抑えることできる．

パターン 3 では，システムコール処理後に監視処理を行う，一度のシステムコールでカーネルへ攻撃を行う場合，攻撃完了までは検出困難である．一方，システムコール処理中に発生した仮想記憶空間の書換えは確実に検出できる．

5.3 カーネルのセキュリティ機構の保護に対する考察

提案手法により，既存のカーネルのセキュリティ機構として強制アクセス制御を担うカーネルコードのうち，LSM フック関数として SELinux の関数アドレス保護を実現した．保護対象のカーネルコードはカーネルの仮想記憶空間に配置されており，任意のカーネルコード・データに対し提案手法による保護を適用可能であると考えている．これにより，提案手法は既存のセキュリティ機構と共存し，さらに連携により最適な監視タイミングを実現可能といえる．

5.4 ダイレクトマッピング領域の提案手法保護に対する考察

カーネルの仮想記憶空間と監視用の仮想記憶空間は個別に管理されており，仮想アドレスを利用しては相互に参照できない．計算機上の物理記憶空間は共有していることから，カーネルの仮想記憶空間のダイレクトマッピング領域に監視用の仮想記憶空間もマッピングされ，物理ページの仮想アドレスを辿ることで監視用の正当データへ参照は可能であった．提案手法においては，ダイレクトマッピング領域から，監視用に用いる正当なデータに関するページ等をカーネルの仮想記憶空間からアンマッピングすることで，ダイレクトマッピング領域経由での参照防止を実現した．これにより，カーネルの仮想記憶空間より監視用の仮想記憶空間は参照できず確実に分離できたと考えている．

今後，提案手法への攻撃の困難化のため，物理記憶空間の適切な管理と利用方法を検討することを考えている．

6. 関連研究

OS のセキュリティ機構は複数レイヤにてアクセス主体とアクセス対象を細かく分離し管理する [13], [14]．Linux では SELinux[1] , ケイパビリティ [2] , KASLR[6], CFI[5] および

関数の戻り番地を検証する Code Pointer Integrity（CPI）[15] の適用が進み，CPU の NX-bit[16] や SMAP/SMEP[7] によりユーザモードからカーネル脆弱性を利用した攻撃は困難化された．カーネルの仮想記憶空間へのサイドチャネル攻撃 Meltdown[12] が示され，ユーザモードとカーネルモードの仮想記憶空間は KPTI などで分離された [8], [17]．

カーネルモードで完結する攻撃 ret2dir [9] もあり，ダイレクトマッピング，ROP（Return Oriented Programming）[18] ，および CFI 対策 [19] 等での，カーネルでのデバイスドライバに対する脆弱性の危険性が示されている [20]．

カーネルの完全性を起動時や実行中に担保する手法として，カーネルを仮想マシンモニタ（VMM）で監視するセキュリティ機構 [21], [22], [23], [24]，カーネルや VMM と同一レイヤで監視するセキュリティ機構がある [25], [26]．これらはカーネルの外部監視やハイパーバイザ監視を想定し，ハードウェア仮想化支援や一部 CPU 命令の捕捉を前提とする．提案手法は，カーネル内部で監視を行うためベアメタルからクラウド上の仮想マシンにも適用可能である．

仮想記憶空間に対して細粒性の高い分割と実行環境を提供する手法も提案されている [27], [28], [29], [30], [31], [32], [33]．また，カーネルの仮想記憶空間の保護として，カーネルのページテーブル位置のランダム化 [34] や R^X の排他制御を行う機構 [35] が提案されている．これらの手法は提案手法のセキュリティ機構保護に組合せられると考えている．カーネルの障害を分離し管理する手法 [36] を元にカーネルモードのデバイスドライバの分離実行やの監視のためのセキュリティ機構 [37], [38] も提案されている．提案手法では，実行中カーネルに影響を与えることなく監視処理は行えるため，今後，デバイスドライバ単位での監視などカーネルの仮想記憶空間に対して影響を与える可能性の高い処理に監視点を挟むなど，既存手法と連携し，効率的なカーネル保護についての検討を進める予定である．

7. おわりに

カーネル脆弱性を介した攻撃による仮想記憶空間の改ざんの緩和策として，OS では，KASLR，KPTI，および CFI などがあり，CPU では，NX-bit や SMAP/SMEP の実装が進んだ．しかし，カーネル空間のみでカーネル脆弱性を介した攻撃を成功させる可能性は依然として存在する．

我々は，カーネルモードにて独自の仮想記憶空間を用いカーネルの仮想記憶空間の監視を可能とするセキュリティ機構を提案している．本稿では，複数の仮想記憶空間の切替えパターンによるカーネルへの攻撃の検出タイミングの差異，既存のセキュリティ機構の保護，および提案手法へのカーネル脆弱性を介した攻撃からの保護を提案した．Linux にて提案を実現し，カーネルへの攻撃検出の有効性を示した．また，オーバヘッド評価にて，仮想記憶空間の切替えパターン 1 が 0.729 ms から 1.99 ms および，パターン 3 が 0.406 ms から 5.029 ms であることを示した．

参考文献

[1] Security-enhanced Linux, available from http://www.nsa.gov/research/selinux/, (accessed 2019-01-22).
[2] Linden, A. T.: Operating System Structures to Support Security and Reliable Software, ACM Computing Surveys (CSUR), vol. 8, no. 4, pp. 409445, (1976).
[3] Kemerlis, P, V., et al.: kGuard - Lightweight Kernel Protection against Return-to-User Attacks, the 21st USENIX Conference on Security symposium, (2012).
[4] Yamauchi, T., et al.: Additional Kernel Observer to Prevent Privilege Escalation Attacks by Focusing on System Call Privilege Changes, The 2018 IEEE Conference on Dependable and Secure Computing (DSC), (2018).
[5] Abadi, M., et al.: Control-Flow Integrity Principles, Implementations, the 12th ACM Conference on Computer and Communications Security (CCS), pp. 340–353, (2005).
[6] Hund, R., et al.: Practical Timing Side Channel Attacks against Kernel Space ASLR, 2013 IEEE Symposium on Security and Privacy, pp. 191–205, (2013)
[7] Mulnix D.: Intel Xeon Processor D Product Family Technical Overview, https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview, (2015), (accessed 2019-01-22).
[8] Lipp, M., et al.: KASLR is Dead - Long Live KASLR, 2017 International Symposium on Engineering Secure Software and Systems (ESSoS), vol. 10379, no. 3, (2017).
[9] Kemerlis, P, V., et al.: ret2dir - Rethinking Kernel Isolation, the 23rd USENIX Conference on Security Symposium, pp. 957–972, (2014).
[10] 葛野弘樹, 山内利宏.: 独自のカーネル用仮想記憶空間を用いたカーネルモジュール監視手法, コンピュータセキュリティシンポジウム 2018, pp.971–978 (2018).
[11] Chen, H., et al.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems, the 2nd Asia-Pacific Workshop on Systems (APSys), (2011).
[12] Lipp, M., et al.: Meltdown - Reading Kernel Memory from User Space, the 27th USENIX Conference on Security Symposium, (2018).
[13] Shu, R., et al.: A Study of Security Isolation Techniques, ACM Computing Surveys (CSUR), vol. 49, no. 3, (2016).
[14] Zhang, F. and Zhang, H.: SoK A Study of Using Hardware-assisted Isolated Execution Environments for Security, the Hardware and Architectural Support for Security and Privacy 2016, pp. 1-8, (2016).
[15] K. Volodymyr., et al.: Code-Pointer Integrity, 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), (2014).
[16] Ingo Molnar, [announce] [patch] NX (No eXecute) support for x86, 2.6.7-rc2-bk2, available from http://lkml.iu.edu/hypermail/linux/kernel/0406.0/0497.html, (2004). (accessed 2018-08-10).
[17] Hua, Z., et al.: EPTI - Efficient Defence against Meltdown Attack for Unpatched VMs, 2018 USENIX Annual Technical Conference (ATC), 2018.
[18] Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), the 14th ACM Conference on Computer and Communications Security (CCS), pp. 552-561, (2007).
[19] Carlini, N., et al.: Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, the 24th USENIX Security Symposium, pp. 161-176, (2015).
[20] Song, D., et al.: PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary, the 26th Annual Network and Distributed System Security Conference (NDSS), (2019).
[21] Trusted computing group. tpm main specification. "http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2003, (accessed 2018-08-10).
[22] Seshadri, A., et al.: SecVisor - a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, the 21st ACM SIGOPS symposium on Operating systems principles (SOSP), pp. 335–350, (2007).
[23] McCune, M, J., et al.: TrustVisor - Efficient TCB Reduction and Attestation, 2010 IEEE Symposium on Security and Privacy, (2010).
[24] Zhang, Z., et al.: KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels, The 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), (2018)
[25] Sharif, I, M., et al.: Secure in-VM monitoring using hardware virtualization, the 16th ACM Conference on Computer and Communications Security (CCS), (2009).
[26] Deng, L., et al.: Dancing with Wolves: Towards Practical Event-driven VMM Monitoring, the 13th ACM SIGPLAN/SIGOPS International Conference, (2017).
[27] Litton, J., et al.: Light-Weight Contexts - An OS Abstraction for Safety and Performance, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), (2016).
[28] Hsu. C, T., et al.: Enforcing Least Privilege Memory Views for Multithreaded Applications, the 2016 ACM Conference on Computer and Communications Security (CCS), pp. 393–405, (2016).
[29] Koning, K., et al.: No Need to Hide: Protecting Safe Regions on Commodity Hardware, the Twelfth European System Conference (EuroSys), pp.437-452, (2017)
[30] Vahldiek-Oberwagner, A., et al.: ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys, CoRR abs/1801.06822, (2018).
[31] Mogosanu, L., et al.: MicroStache - A Lightweight Execution Context for In-Process Safe Region Isolation, The 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID), pp.359–379(2018).
[32] Frassetto, T., et al.: IMIX - In-Process Memory Isolation EXtension, the 28th USENIX Security Symposium, (2018).
[33] Kim, H, C., et al.: Securing Real-Time Microcontroller Systems through Customized Memory View Switching, the 25th Network and Distributed System Security Symposium (NDSS), (2018).
[34] Davi, L., et al.: PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables, the 23th Network and Distributed System Security Symposium (NDSS), (2016).
[35] Pomonis, M., et al.: “kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse, the Twelfth European Conference on Computer Systems (EuroSys), pp. 420–436, (2017).
[36] Castro, M., et al.: Fast byte-granularity software fault isolation, the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP), pp. 45–58, (2009).
[37] Boyd-Wickizer, S., et al.: Tolerating Malicious Device Drivers in Linux, USENIX Annual Technical Conference, (2010).
[38] Tian, J, D., et al.: LBM: A Security Framework for Peripherals within the Linux Kernel, 2019 IEEE Symposium on Security and Privacy, (2019).