Computer Security Symposium 2020
26 - 29 October 2020

コンテナ向けカーネル仮想記憶空間の分離制御機構

葛野 弘樹[1,a)]　山内 利宏[2]

概要：複数の計算機環境を単一計算機上に実現するため，オペレーティングシステムのカーネルが提供する仮想化機能やコンテナ機能が利用される．仮想化機能の仮想ハードウェアやコンテナ機能の資源管理に関するデータはカーネルの仮想記憶空間に配置し制御される．カーネルの仮想記憶空間は全プロセスで共有されるため，サイドチャネル攻撃として，攻撃プロセスから CPU キャッシュなどへサイドチャネルを行い，本来は参照不可能な他のプロセスの利用するカーネルデータを推測可能なことが指摘されている．サイドチャネル攻撃への対策として，Kernel page table isolation は，カーネルモード用とユーザモード用の 2 つに仮想記憶空間のページテーブルを分離した．さらに Address space isolation では，カーネルの仮想記憶空間から仮想化機能を分離する手法が検討された．本稿では，カーネルにおける CPU キャッシュへのサイドチャネル攻撃の緩和を実現するため，コンテナ機能の利用プロセス毎にページテーブルを備えるカーネル仮想記憶空間の分離制御機構を提案する．提案手法を Linux にて実現し，Foreshadow の PoC コードによる実際のサイドチャネル攻撃への有効性を評価した．また，性能評価として，システムコール呼出しに対して最大 7.864 μs のオーバヘッドであることを示した．

Design and Implementation of Kernel Address Isolation for Container

Hiroki Kuzuno[1,a)]　Toshihiro Yamauchi[2]

Abstract: Operating system kernel provides the virtualization and the container technology support the cloud service to create multiple environments on the one physical computer. The virtual machine process or the container process store their management data on the kernel memory region. Recent software-based cache side-channel attacks target CPU and MMU caches to speculate the data on the protected memory region belongs to kernel. To tackle with side-channel, kernel page table isolation separates virtual address space for user mode and kernel mode. The address space isolation also supports the multiple page tables for dedicated feature (e.g., virtualization). In this paper, we provide a novel kernel virtual address space isolation mechanism for the kernel data of container process to reduce the attack surface of the cache side-channel. Our mechanism is realized to the latest Linux that can protect the actual side-channel attack and indicates better overhead performance at the evaluation.

1. はじめに

オペレーティングシステム（OS）の新たな脅威として，仮想マシンやコンテナにより複数の計算機資源を単一の計算機にて実現した環境（以下，マルチテナント環境）において，攻撃プロセスから，他のプロセスやカーネルの利用する CPU や MMU の各種キャッシュに対し，ソフトウェアによるサイドチャネルを行いデータを推測する攻撃（以下，サイドチャネル攻撃）が提案されている [1], [2], [3], [4].

Meltdown では，仮想記憶空間への参照に対し，攻撃プロセスからカーネルの管理するページテーブル全体を推測可能なサイドチャネル攻撃が実証された [1]．Kernel page table isolation（KPTI）は Meltdown 対策として有効とされ，仮想記憶空間はユーザモード用（ユーザの仮想記憶空間），およびカーネルモード用（カーネルの仮想記憶空間）として，2 つのページテーブルに分割された [5].

Foreshadow では，攻撃プロセスから CPU L1 キャッシュ

[1] セコム株式会社 IS 研究所
Intelligent Systems Laboratory, SECOM Co., Ltd., Japan
[2] 岡山大学 大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University, Japan
[a)] kuzuno@s.okayama-u.ac.jp

に保持されたデータを参照するサイドチャネル攻撃が可能とされた [3], [4]． Foreshadow 対策として，ページテーブルエントリの無効化時に仮想アドレス部分を反転処理する PTE inversion [6]，ならびに仮想マシンの動作切替時の L1 キャッシュ初期化 [7] が導入された．また，カーネルの仮想記憶空間から，仮想化機能を分離し，実行する Address space isolation（ASI）が提案されている [8]．

攻撃プロセスからのサイドチャネル攻撃に対し，動作中の他のプロセスをカーネルにて安全に維持するため，より汎用的なサイドチャネル攻撃緩和策を図ることが重要であり，従来手法には以下の課題が存在する．

- **カーネルの仮想記憶空間から仮想化機能のみ分離可能**

  従来手法 ASI はカーネルの仮想化機能を利用する仮想マシンに専用のページテーブルを備え，仮想マシンの実行時にのみ分離するため，同じカーネルの上のプロセス間でカーネル機能を保護することはできない．

特にコンテナ機能を利用するマルチテナント環境において，各コンテナプロセスはカーネルを共有し，細かに資源管理制御されることから，カーネルレイヤにおけるプロセス単位でのサイドチャネル攻撃への対応は必要といえる．

本稿では，この課題に対し，プロセスへのサイドチャネル攻撃対策として，特定のカーネル機能に対し，複数のページテーブルからなるカーネルの仮想記憶空間の新たな適用手法を提案する．具体的な提案手法は以下の通りである：

- 提案手法によるカーネルのセキュリティ機構として，特定のカーネル機能に対して新たなページテーブルの具備を可能とし，他のカーネル機能のページテーブルと合わせ，カーネルの仮想記憶空間を構成する．従来手法とは異なり，保護対象プロセスがシステムコールを介して特定のカーネル機能を呼出す際のみカーネルデータを動的に利用可能とする．カーネルレイヤにて，攻撃プロセスから他プロセスの利用するカーネルコード・データの推測など，カーネルに対するサイドチャネル攻撃の困難化を実現し，加えて，カーネル動作への影響を最小化する．

提案手法を Linux に実現し，適用例であるコンテナプロセスに対するサイドチャネル攻撃防止の有効性評価，および性能評価を行った．本稿での研究貢献ならびに得られた結果は以下の通りである：

(1) カーネルにおけるサイドチャネル攻撃対策として，特定のカーネル機能を利用するプロセスのカーネルコード・データに対し，専用のページテーブルを割当可能なセキュリティ機構を設計し，コンテナ機能への実現方式を提案した．この方式は，コンテナやユーザ単位でのサーバホスティングなど，カーネルを共有する環境でのサイドチャネル攻撃による影響を緩和し，カーネル動作の安全性の向上を実現する．

(2) 提案方式を Linux にて実現し，実際のサイドチャネル攻撃に対してカーネルデータ保護の有効性評価した．また，性能評価を行い，システムコール処理に対し，最大 7.864 μs のオーバヘッドであることを示した．

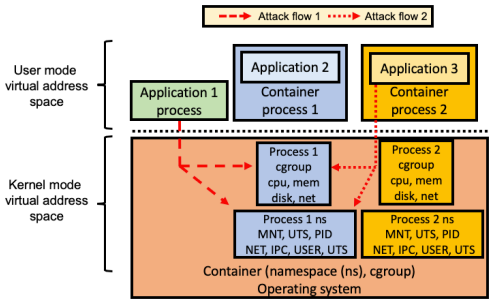

図 1　想定するマルチテナント環境，および脅威モデル

2. 背景知識

2.1 マルチテナント環境

クラウドサービスなどの構築と提供のために，コンテナ機能を用いて単一の計算機にて複数の計算機資源を実現したマルチテナント環境を図 1 で示す．

コンテナ機能

カーネルにてコンテナに対し，計算機資源の細粒度割当てを行う管理機能を提供する．複数のコンテナを作成し，プロセスとして動作させるため，カーネルは共有される．コンテナ内部で動作するアプリケーションは，コンテナプロセスの子プロセスとして動作し，割当てられた制限内にて計算機資源利用する．Linux カーネルにおいては，namespace として，マウントポイント，ホスト名，プロセス ID，ユーザ ID，ネットワーク，ならびにプロセス間通信をコンテナ毎に割当て可能である．また，cgroup として，CPU，メモリ，ブロック IO，ネットワークの資源制御をコンテナ毎に行なわれる．

2.2 ソフトウェアによるサイドチャネル攻撃

ソフトウェアによるキャッシュへのサイドチャネルは，図 2（a）に示すように，攻撃プロセスからアクセスが許可されていない攻撃対象プロセスやカーネルの利用するデータを CPU や MMU の各種キャッシュを介して間接的に推測する攻撃である [1], [3], [4]．キャッシュに格納されたデータを推測するサイドチャネル手法は複数あり [9]，代表的な手法を図 2（b）に示し，以下に述べる：

- Flush+Reload 攻撃

(1) Flush 段階：攻撃プロセスは，特定の記憶領域を共有キャッシュから削除

(2) 攻撃対象プロセスがデータにアクセスし，共有キャッシュにデータが配置される

(3) Reload 段階：攻撃プロセスは，特定の記憶領域にア

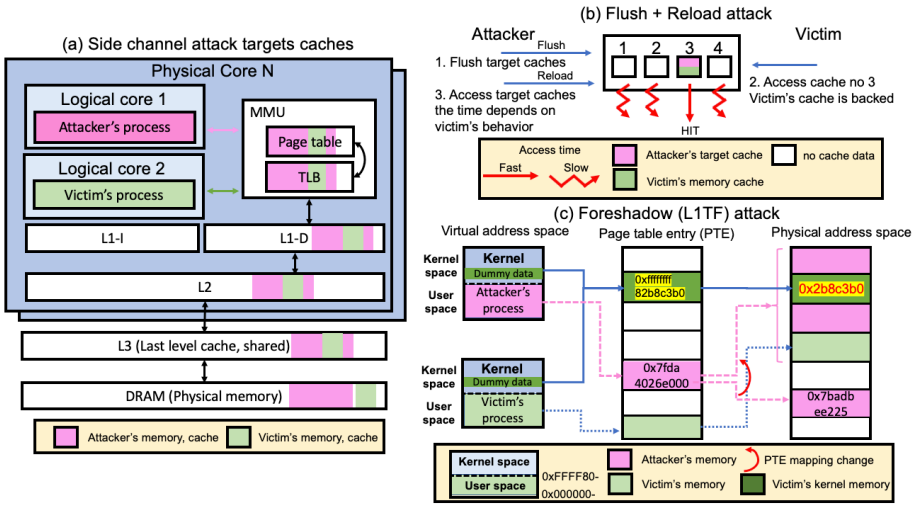

図 2　ソフトウェアによるサイドチャネルの攻撃対象キャッシュと手法の概要

```
1.  $ cat /boot/System.map-`uname -r` | grep -i dummy_data
2.  ffffffff82b8c3b0 B dummy_data01
3.  $ sudo cat /proc/iomem | grep 'Kernel code'
4.  01000000-01e00efc : Kernel code
5.  $ cat /boot/System.map-`uname -r` | grep -i _stext
6.  ffffffff81000000 T _stext

7.  // dummy data physical address calculation
8.  ffffffff82b8c3b0 - ffffffff81000000 = 1b8c3b0
9.  1b8c3b0 + 01000000 = 2b8c3b0

10.// PoC execution before container boot
11.$ sudo ./doit 0x2b8c3b0 0x400
12.Looking for the PTE for VA 0x7f0f09dff000 in RAM...
13.Our PTE now mapped. Value: 7badbee225
14.Dumping from VA 0x7f0f09dff3a0
15.00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

16.  // container execute
17.$ ./container cat
18.parent pid: 2403
19.child pid: 2404

20.// container log
21.[  411.921574] pid: monitor process 2403
22.[  411.925962] pid: monitoring process 2404
23.[  411.925969] dummy data01 1st virtual address (pB):
0xffff88847c90a900

24.  // PoC execution after container boot
25.$ sudo ./doit 0x2b8c3b0 0x400
26.Looking for the PTE for VA 0x7fda4026e000 in RAM...
27.Our PTE now mapped. Value: 7badbee225
28.Dumping from VA 0x7fda4026e3a0
29.00 00 00 00 00 00 00 00 00 a9 90 7c 84 88 ff ff
```

図 3　サイドチャネル攻撃成功時のログ

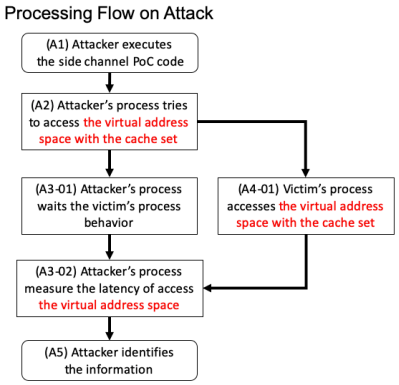

図 4　攻撃者のサイドチャネル攻撃フロー

クセスし，共有キャッシュ利用有無の時間を計測し，内容を推測

Flush+Reload 攻撃では，攻撃プロセス，攻撃対象プロセスにおいて，仮想記憶空間の共有を必要としている [10].

2.2.1　Foreshadow によるサイドチャネル攻撃例

Foreshadow は CPU L1 キャッシュへのサイドチャネル攻撃として，CVE-2018-3615 [11]，CVE-2018-3620 [12]，および CVE-2018-3646 [13] と登録されている [14].

プロセスからカーネルへ攻撃可能な Foreshadow の Proof of concept（PoC）コード [15] によるサイドチャネル攻撃の概要を図 2（c）に示す．攻撃プロセスは Present bit を無効化したページテーブルエントリ（PTE）を予め作成し，PTE が任意の物理アドレスを指すように Flush+Reload 攻撃を繰返す．攻撃時，CPU L1 キャッシュへの情報有無により PTE を介したアクセスによるページフォルトの時間変化を計測し，攻撃対象カーネルデータの値を推測する．

PoC コード [15] による攻撃成功時のログを**図 3** に示す．1 行目から 9 行目にて，攻撃対象コンテナプロセスの利用するカーネルデータ `dummy_data` の配置される仮想アドレスから物理アドレス `2b8c3b0` を特定している．11 行目にて攻撃プロセスとして PoC コードを実行し，該当の物理アドレスへの攻撃を行うが，15 行目にて推測できる値はないことを確認できる．17 行目にて，攻撃対象コンテナプロセスを起動させる．この際，カーネルにて，カーネルデータ `dummy_data` に `kmalloc` を利用して確保した記憶領域の仮想アドレスを格納する．23 行目にて，実際の仮想アドレスの値 `0xffff88847c90a900` を確認している．25 行目にて，再度攻撃プロセスとして PoC コードを実行し，物理アドレスへの攻撃を行う．29 行目にてカーネルデータ `dummy_data` の格納した値の推測成功を確認でき，PoC コードにより，攻撃プロセスからカーネルの仮想記憶空間上にあるカーネルデータの値を特定できている．

3.　脅威モデル

脅威モデルにて想定するマルチテナント環境を図 1 で示す．攻撃者は通常プロセスあるいはコンテナプロセスを用い，攻撃対象はコンテナプロセスとする．攻撃プロセスはサイドチャネル攻撃により，カーネルの仮想記憶空間配置の攻撃対象コンテナプロセスに関するカーネルデータを推測する．**図 4** に攻撃者の攻撃フローを示し，以下に述べる．

（A1）　攻撃者はマルチテナント環境にて，サイドチャネルを行う攻撃プロセスを起動

（A2）　攻撃プロセスは特定の記憶領域を対象にサイドチャネルを開始

（A3-01）　攻撃プロセスは攻撃対象プロセスの動作を待機

（A3-02）　攻撃プロセスは特定の記憶領域へのアクセス時間を測定しサイドチャネルを行う

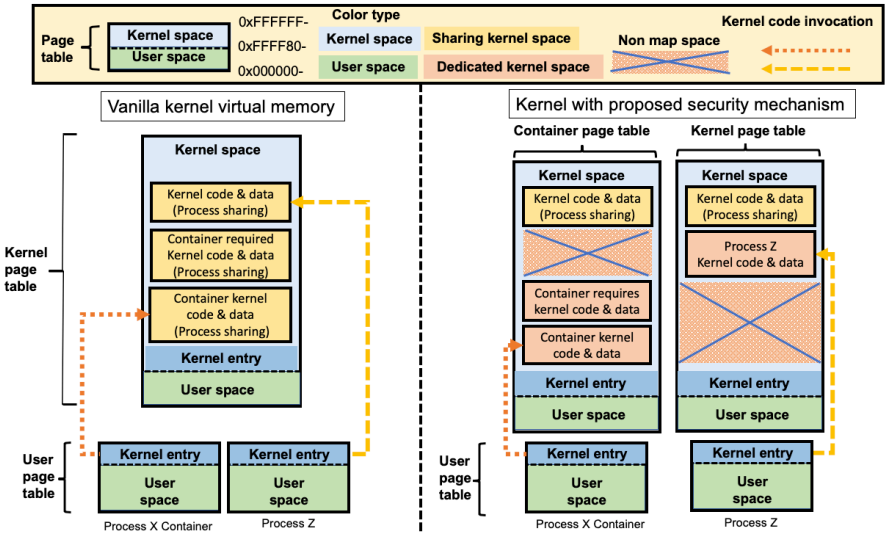

図 5　提案手法の適用前，および適用後の仮想記憶空間の構成

（A4-01）　攻撃対象プロセスは通常動作を継続的に行う

（A5）　攻撃プロセスは特定の記憶領域へのアクセス時間から情報を推測し入手

4.　提案手法と実現方式

4.1　提案手法の目的と要件

本稿において提案する特定のカーネル機能に対するカーネルの仮想記憶空間の分離制御機構では，カーネルレイヤにて特定のカーネル機能を利用するプロセスのカーネルデータをサイドチャネル攻撃からの保護を目的とする．サイドチャネル攻撃を緩和するための要件を以下に挙げる：

要件：　カーネルの仮想記憶空間において，サイドチャネル攻撃において攻撃対象となるカーネルコードとカーネルデータを，サイドチャネル攻撃を試みるプロセスのカーネルコードから分離し管理する

4.2　提案手法の設計

提案手法の設計として，要件を満たすため，分離対象とするカーネル機能を利用するプロセスに対し，新たなページテーブル（特定機能のページテーブル）を用意し，専用のカーネルの仮想記憶空間（特定機能の仮想記憶空間）を構築する．分離対象のカーネルコード，およびデータは特定機能の仮想記憶空間に配置し，実行可能とする．一部，分離対象とするカーネル機能を動作させるのに必要となるカーネルコードならびにデータのみ，他のカーネルの仮想記憶空間を構成するページテーブルと共有する．

4.3　提案手法の構成

カーネルの仮想記憶空間における提案手法の適用前後の構成を図 5 に示す．図 5 において，プロセス X はコンテナ機能，プロセス Z は通常のプロセスとしている．提案手法適用前の従来のカーネルの仮想記憶空間では，グローバルな領域として全てのプロセスがカーネルコード・カーネルデータを共有しており，利用可能である．提案手法適用後，プロセス X は特定機能のページテーブルを備え，コンテナ機能のカーネルコードは特定の仮想記憶空間上において実行する．一方，プロセス Z は従来のページテーブルのみ利用し，プロセス Z より呼び出されるカーネルコードは従来のカーネルの仮想記憶空間上にて実行される．

4.4　提案手法のプロセス管理

分離対象のカーネル機能を利用するプロセスに対して，カーネルの仮想記憶空間の分離制御の適用時のシーケンスを図 6 に示す．まず，プロセス生成時，および分離対象のカーネル機能実行時の流れを説明する．

（1-1）　プロセス生成時，ページテーブルを作成

（1-2）　分離対象のカーネル機能に対し特定機能のページテーブルを作成

（1-3）　特定機能の仮想記憶空間の構築として，必要とするカーネルデータ，およびデータをマッピング処理

続いて，分離対象のカーネル機能を利用するプロセスの実行時の流れを説明する．

（2-1）　プロセス実行時，カーネルの仮想記憶空間を利用

（2-2）　分離対象のカーネル機能を実行時，特定機能のページテーブルから構成される仮想記憶空間に切替え

（2-3）　特定機能のページテーブルから構成される仮想記憶空間にてカーネル機能を実行し，関連するカーネルデータを利用

（3-1）　プロセス実行に伴うコンテキストスイッチ時，カーネルの仮想記憶空間のページテーブルを利用

（3-2）　分離対象のカーネル機能の実行開始時，特定機能のページテーブルから構成される仮想記憶空間を利用

（3-3）　分離対象のカーネル機能の実行終了時，カーネル

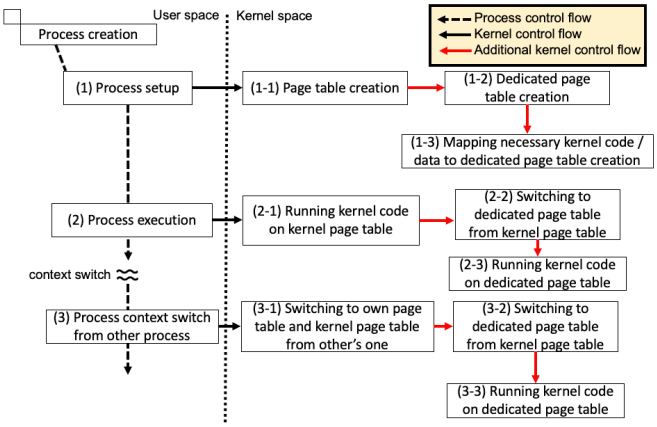

図 6　提案手法におけるプロセス管理フロー

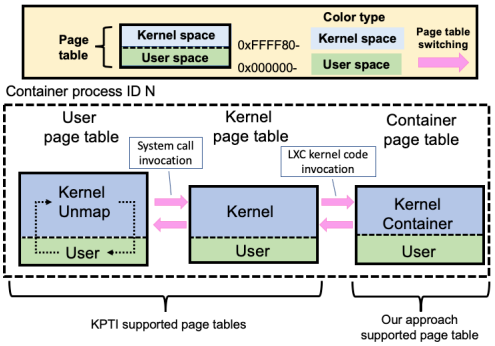

図 7　提案手法におけるコンテナプロセスのページテーブル

の仮想記憶空間のページテーブルを利用

一連の処理により，カーネルの仮想記憶空間の分離制御を特定のカーネル機能を利用するプロセスに適用する．

4.5　サイドチャネル攻撃に対する効果

提案手法において，サイドチャネル攻撃を試みるプロセスの利用するカーネルコードの参照する仮想アドレス空間には分離対象のカーネルデータは存在しない．このため，プロセス間でカーネルデータを CPU，ならびに MMU の各種キャッシュにて共有される可能性を削減し，サイドチャネル攻撃によるデータの推測成功率の低減につながる．

4.6　実現方式

本稿での実現方式では，特定のカーネル機能をコンテナ機能とし，要件を満たすために，提案手法を適用する．提案手法の実現環境は Linux とし，CPU アーキテクチャは x86_64 を想定する．Linux カーネルにてコンテナ機能に対して提案手法を適用した際のコンテナプロセスを図 7 に示す．コンテナプロセスと関連するカーネルデータに対して，コンテナ用の特定機能のページテーブル（コンテナのページテーブル）を特定機能の仮想記憶空間として利用可能とすることでカーネルの仮想記憶空間の分離制御を行う．

コンテナプロセスは，コンテナ作成として資源制御の namespace 設定を引数指定した clone システムコールによる生成プロセスとし，プロセス ID を記録，識別可能する．

4.6.1　実現方式の構成

提案手法を Linux に実現した際のページテーブルと仮想記憶空間でのカーネルコードおよびカーネルデータの配置を図 8 に示す．実現方式でのカーネルの仮想記憶空間は，コンテナのページテーブル，ならびにカーネル機能全体で共有する従来のページテーブルから構成する．

4.6.2　実現方式での分離制御

実現方式では，コンテナプロセスに対し，コンテナのページテーブルとして `mm_struct` 構造体の `cpt_pgd` 変数を追加する．コンテナプロセス生成時，図 8 に示すように，カーネルのページテーブルからコンテナのページテーブル

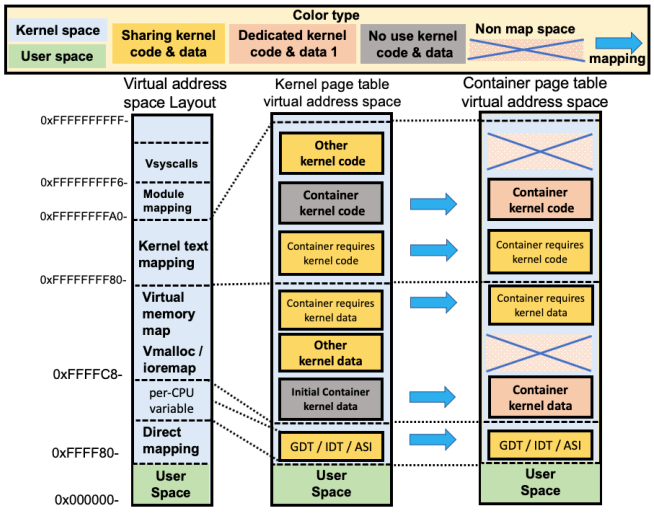

図 8　提案手法を適用したページテーブルと仮想記憶領域の配置

に対し，コンテナ機能の動作に必要なカーネルコード，ならびにカーネルデータのマッピングを以下の手順で行う．

(1) 実行中プロセスの変数 `current` に含まれる `pgd` から，`cpt_pgd` 変数にコンテナ機能に関するカーネルコードをマッピング

(2) CPU 毎の変数である GDT（Global Descriptor Table）を `cpt_pgd` 変数にマッピング

(3) コンテナプロセスの動作に必要なカーネルデータ，ならびにコンテナ設定に関するカーネルデータを作成し，`cpt_pgd` 変数にマッピング

4.6.3　実現方式での実行制御

コンテナプロセス実行時の制御を説明する．

(1) コンテナプロセスからのシステムコール呼出しか判定

(2) コンテナプロセスならばコンテナのページテーブルである変数 `current` の `cpt_pgd` を CR3 レジスタに書込み，特定機能の仮想記憶空間に切替える

(3) システムコールを実行

(4) システムコール終了後，変数 `current` の `pgd` を CR3 レジスタに書込み，カーネルの仮想記憶空間に切替える

コンテナプロセスの実行中，割込み，例外，ならびに他プロセスへのコンテキストスイッチが発生した場合，コンテナプロセスの `pgd` 変数を CR3 レジスタに書込み，カー

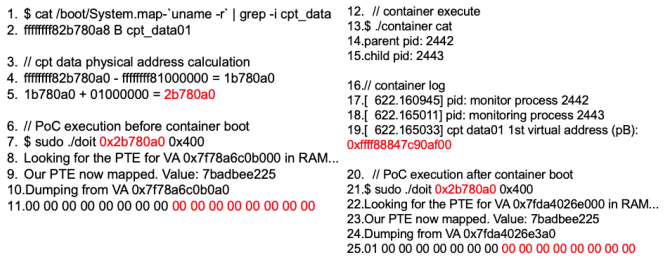

図 9　提案手法によるサイドチャネル攻撃防止時のログ

表 1　提案手法を適用した Linux におけるオーバヘッド (μs)

System call	Vanilla kernel	Our kernel	Overhead
fork+/bin/sh	525.088	562.496	37.408
fork+execve	135.527	152.326	16.799
fork+exit	123.938	139.665	15.727
open/close	2.985	3.041	0.056
write	0.222	0.249	0.027
read	0.263	0.294	0.031
stat	1.008	1.051	0.043
fstat	0.285	0.311	0.026

ネルの仮想記憶空間に切替え後，カーネル処理を継続する．

4.6.4　適用対象のカーネル機能

コンテナのページテーブルに配置されるカーネルコードはコンテナプロセスの利用するシステムコール関数とし，カーネルデータは，コンテナに関する namespace など資源分離データを格納する `nsproxy` 構造体に関する情報，ならびに cgroup の資源制御データを格納する `cgroup_subsys` 構造体に関する情報とする．

5.　評価

5.1　評価の目的と評価環境

本稿における評価の目的と内容を以下に示す．

（評価 1）　分離対象カーネルデータの制御実験
提案手法の適用後のカーネルにて，分離対象のカーネル機能にコンテナ機能を指定し，コンテナプロセスのカーネルデータを保護可能か評価．

（評価 2）　オーバヘッド評価
提案手法の適用後のカーネルにて，コンテナプロセス上でベンチマークソフトウェアを動作させ，システムコール実行時のオーバヘッドを測定．

評価には CPU Intel(R) Core(TM) i7-7700HQ (2.80GHz，4 コア)，Memory 16 GBytes を備えた計算機を用い，OS は Debian 9.0 (Linux Kernel 5.0.0, x86_64) とした．提案手法の実装を Linux kernel 5.0.0 に行い，41 個のファイルにおいて，2,794 行で実現した．また，Foreshadow サイドチャネル攻撃に PoC コード [15] を用い，攻撃の有効化のため，PTE inversion は無効とした．

5.2　分離対象カーネルデータの制御実験

実験では，分離対象のカーネル機能をコンテナ機能とし，攻撃対象はコンテナプロセスの実行時に作成するカーネルデータとした．Root 権限にて攻撃プロセスとして PoC コードを実行，カーネルデータの推測結果を出力する．

提案手法による PoC コード [15] に対する攻撃防止時のログを**図 9** に示す．2.2.1 節の PoC コード攻撃成功時を参考に 1 行目から 5 行目にて，コンテナプロセスの利用するカーネルデータ `cpt_data` の配置される仮想アドレスから物理アドレス `2b780a0` の特定を行っている．7 行目にて攻撃プロセスとして PoC コードを介して物理アドレスへの攻撃を行い，11 行目にて推測できる値はないことを確認できる．13 行目にて，コンテナプロセスを起動させる．この際，提案手法を適用したカーネルでは，特定機能のページテーブルに切替えた後，カーネルデータ `cpt_data` を配置し，`kmalloc` を利用して確保した記憶領域の仮想アドレスを格納する．19 行目にて，実際の仮想アドレスの値は `0xffff88847c90af00` と確認できる．21 行目にて，再度攻撃プロセスとして PoC コードを実行し，物理アドレスへの攻撃を行うが，25 行目でカーネルデータの推測に失敗していることを確認できる．

5.3　オーバヘッド評価

提案手法の適用前後の Linux kernel にてコンテナ内プロセスとしてベンチマークソフトウェア lmbench をそれぞれ 15 回実行し，平均値からオーバヘッドを算出した．

評価結果を表 1 に示す．lmbench におけるシステムコール呼出回数は，fork+/bin/sh は 54 回，fork+execve は 4 回，fork+exit は 2 回，open/close は 2 回，および，その他は 1 回である．表 1 から，提案手法を適用したコンテナ内プロセスにおいて，最も影響を受けたシステムコールは fork+exit (7.864 μs)，最も少ないオーバヘッドは write (0.027 μs) である．提案手法によるシステムコール 1 回あたりのオーバヘッドは 0.027 μs から 7.864 μs となった．

6.　考察

6.1　評価に対する考察

評価結果より，提案手法を適用したカーネルでは，PoC コードを利用した攻撃プロセスからの CPU L1 キャッシュに対するサイドチャネル攻撃の防止の実現を確認した．また，提案手法の分離対象としたコンテナ機能，およびコンテナプロセスの動作に対し，影響のないことを確認した．コンテナプロセスのカーネルデータの推測は困難であることから，カーネル機能をページテーブル単位で分離することで，PoC コード [15] の利用する PTE からの参照を防ぎ，サイドチャネルの脅威を緩和できたと考えられる．

提案手法の性能評価では，lmbench におけるシステムコール毎のオーバヘッドとして，コンテナ機能においては，0.027 μs から 7.864 μs を示した．lmbench はシステムコール発行にかかる時間計測を行うため，提案手法では，分離対象とするコンテナプロセスからのシステムコール呼出し，ならびに，カーネルタスクとして該当カーネル機能の処理を行う際，ページテーブルの切替えが発生し，処理

時間は増加すると考えられる．また，fork や exit など，コンテナ内部でプロセス生成や破棄を伴うシステムコールでは，新たに特定機能のページテーブルの作成と解放処理を行うため，負荷が高くなると考えている．

今後，提案手法による各種キャッシュへの影響，および他のサイドチャネルへの有効性について検討し，Web アプリケーションなど実用性の高いソフトウェアを動作させた際の性能評価を行う予定である．

6.2 提案手法の制限

提案手法を実現するにあたり，ページテーブルの切替え，ならびに切替前後において分離対象のカーネル機能を正常動作させるための処理が必要であり，分離対象とするカーネル機能の追加毎にオーバヘッドは増加する．

カーネルの仮想記憶空間を分離するため，複数のページテーブルを介したカーネルデータのやり取りは困難となる．新たにカーネル機能をサイドチャネル攻撃から保護するには，分離対象とすべきカーネルデータ，および共有を行うカーネルデータを選別し，カーネルの仮想記憶空間の分離後も動作するように再設計，および実現する必要がある．

6.3 移植可能性

提案手法の他 OS への移植可能性として， Linux KPTI と同等の機能を FreeBSD [16] ，および Windows [17] は実現しており，これらの OS では，カーネルレイヤにてページテーブルを新たに導入することで提案手法を適用可能であると考えている．また，他のアーキテクチャへの移植可能性として，ARM においては，Translation Table Base Registers (TTBR0/TTBR1) により，プロセス，およびカーネルの実行時に利用可能な物理記憶空間の範囲を制御可能である [18]．提案手法においてもカーネル機能毎に TTBRs を制御することで実現の可能性はある．

7. 関連研究

ソフトウェアによるサイドチャネル攻撃

ソフトウェアによるサイドチャネル攻撃は多数提案されており [19]，メモリ操作と特定の CPU 命令の応答時間を利用し，保護された記憶領域を推測する手法 [10], [14], [20]，ならびに Meltdown，Spectre や Foreshadow など CPU の投機的実行とソフトウェア実装を利用し，CPU や MMU の各種キャッシュへの攻撃が考案されている [1], [2], [3], [4]．

サイドチャネル攻撃への脆弱性としての対策

サイドチャネル攻撃への脆弱性対策としては，CPU，カーネル，ならびにコンパイラなどにて対策技術が導入される．CPU では，マイクロコードの更新が行われ [21]．カーネル，ならびにコンパイラでは，Meltdown に対し，KPTI [5] によるユーザの仮想記憶空間とカーネルの仮想記憶空間のページテーブル分離機構 [22]，Spectre に対し，CPU の投機的実行による先読み発生箇所への緩和コード [23] が導入された．また，Foreshadow に対し，PTE invertion でのプロセス間の攻撃対策 [6]，L1 キャッシュの初期化による仮想マシン間の攻撃対策 [7] が導入された．

キャッシュ分割によるサイドチャネル攻撃対策

サイドチャネル攻撃への対策として，攻撃プロセスから参照不可能なキャッシュ領域の生成手法が提案されている．STEALTHMEM では，仮想マシンモニタによる仮想マシン単位での CPU コア割当てによる Last level cache（LLC）の参照領域の制御を行う [24]．SecDCP では，プロセス単位での動的な LLC サイズの調整を可能としている [25]．CATalyst では，Intel Cache Allocation Technology により LLC にセキュア領域を確保し，仮想マシンへのページ割当てを実現している [26]．また，DAWG では，CPU コア，およびキャッシュデータを紐付けし，アクセス可否を制御する機構を提案している [27]．

CPU の投機的実行制御によるサイドチャネル攻撃対策

サイドチャネルの影響緩和手法として，InvisiSpec では，CPU の投機的実行命令の処理時，命令の進行度に応じ，データ領域の参照可否を制御するための一時バッファ機構を検討している [28]，SafeSpec では，L1 キャッシュおよび Translation Lookaside Buffer（TLB）へのデータ登録を投機的実行終了まで回避する制御機構を提案している [29]．

カーネルにおけるサイドチャネル攻撃対策

ASI では，カーネルの仮想記憶空間から仮想化機能をページテーブル単位での分離 [8]，Process-local memory では，特定プロセスのみカーネルの仮想記憶空間の一部を参照可能とするページ割当て [30]，また，Core Scheduling では，攻撃・攻撃対象プロセスの同一 CPU コアでの実行回避スケジューリング [31] を実現している．Safehidden では，TLB ヒットミス時，カーネルの仮想記憶空間上のカーネルデータ配置の再ランダマイズ化を提案している [32]．

関連研究との比較

提案手法はカーネルレイヤにおけるサイドチャネル攻撃緩和を図っている．サイドチャネル攻撃は複数レイヤでの対策が必要と考えており，ハードウェア機能を活用したキャッシュ分割技術，ならびに CPU の投機的実行制御での命令実行順に応じたデータ参照可否操作とは組合せ可能といえる．カーネルにおける対策のうち，ASI は仮想化機能である KVM のカーネルコード，およびカーネルデータのみカーネルの仮想記憶空間からの分離を行う．提案手法は実行中のカーネル機能に加え，プロセスに適用可能な設計であり，より汎用性を有すると考えている．また，Linux への実現性の観点から， ASI と連携し，高速かつ信頼性の高い実装方式を検討している．

8. おわりに

本稿では，サイドチャネル攻撃対策として，特定のカー

ネル機能にページテーブルを備え，カーネルの仮想記憶空間を分離可能な新たなセキュリティ機構を提案，評価した．

提案するセキュリティ機構の適用先として，Linux のコンテナ機能に関するカーネルコードおよびデータの分離を実現した．また，評価として，提案手法の有効性とカーネル動作への影響調査のため，実際のサイドチャネル攻撃によるコンテナプロセスの利用するカーネルデータの不正参照を防止可能なことを確認し，オーバヘッド評価にて，システムコール毎の負荷は最大 7.864 μs なことを示した．

謝辞 本研究の一部は，JSPS 科研費 JP19H04109 の助成を受けたものです．

参考文献

[1] Lipp, M. et al.: Meltdown: Reading Kernel Memory from User Space, *Proc. 27th USENIX Security Symposium*, pp.973–990, USENIX (2018).
[2] Kocher, P. et al.: Spectre Attacks: Exploiting Speculative Execution, *Proc. 2019 IEEE Symposium on Security and Privacy*, pp.1–19 (2019).
[3] Bulck, V, J. et al.: FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution, *Proc. 27th USENIX Security Symposium*, pp.991–1008 (2018).
[4] Weisse, O. et al.: Foreshadow-NG: Breaking the Virtual Memory Abstraction with TransientOut-of-Order Execution, https://foreshadowattack.eu/. (accessed 2020-08-05).
[5] Gruss, D. et al.: KASLR is Dead: Long Live KASLR, *Proc. 2017 International Symposium on Engineering Secure Software and Systems (ESSoS)*, vol.10379, no.3, pp.161–176 (2017).
[6] LWN.net, Meltdown strikes back: the L1 terminal fault vulnerability, https://lwn.net/Articles/762570/. (accessed 2020-08-06).
[7] The Linux kernel user’s and administrator’s guide, L1TF - L1 Terminal Fault, https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html. (accessed 2020-08-06).
[8] Chartre, A.: Kernel address space isolation, https://lwn.net/Articles/813393/. (accessed 2020-06-12).
[9] 嶋田有佑，河野健二.: キャッシュのモニタリングによるキャッシュサイドチャネル攻撃の検知．コンピュータセキュリティシンポジウム 2018 論文集, vol.2018, no.2, pp.46–53（2018）.
[10] Yarom, Y. et al.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, *Proc. 23rd USENIX Security Symposium*, pp.719–732 (2014).
[11] CVE-2018-3615, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3615. (accessed 2020-07-31).
[12] CVE-2018-3620, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3620. (accessed 2020-07-31).
[13] CVE-2018-3646, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3646. (accessed 2020-07-31).
[14] Schwarz, M.: Software-based Side-Channel Attacks and Defenses in Restricted Environments. *PhD Thesis, Graz University of Technology*, (2019).
[15] L1TF (Foreshadow) VM guest to host memory read PoC, https://github.com/gregvish/l1tf-poc. (accessed 2020-07-31).
[16] Tetlow, G.: Response to Meltdown and Spectre, https://lists.freebsd.org/pipermail/freebsd-security/2018-January/009719.html. (accessed 2020-05-21).
[17] Ionescu, A.: Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER), https://twitter.com/aionescu/status/930412525111296000. (accessed 2020-08-06).
[18] CMSIS-Core (Cortex-A), Translation Table Base Registers (TTBR0/TTBR1), https://arm-software.github.io/CMSIS_5/Core_A/html/group__CMSIS__TTBR.html. (accessed 2020-08-06).
[19] Canella, C. et al.: A Systematic Evaluation of Transient Execution Attacks and Defenses, *Proc. 28th USENIX Security Symposium*, pp.249-–266 (2019).
[20] Jang, Y. et al.: Breaking Kernel Address Space Layout Randomization with Intel TSX, *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp.380–392, ACM (2016).
[21] Intel Corporation .: Addressing Hardware Vulnerabilities, https://www.intel.com/content/www/us/en/architecture-and-technology/facts-about-side-channel-analysis-and-intel-products.html. (accessed 2020-08-06).
[22] Hanse, D.: KAISER: unmap most of the kernel from userspace page tables, https://lwn.net/Articles/738997/. (accessed 2020-08-06).
[23] LLVM .: Introduce the ”retpoline” x86 mitigation technique, https://reviews.llvm.org/D41723. (accessed 2020-08-06).
[24] Kim, T. et al.: STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud, *Proc. 21st USENIX Security Symposium*, pp.189–204 (2012).
[25] Wang, Y. et al.: SecDCP: Secure dynamic cache partitioning for efficient timing channel protection, *Proc. 53nd ACM/EDAC/IEEE Design Automation Conference*, pp.1–6 (2016).
[26] Liu, F. et al.: CATalyst: Defeating last-level cache side channel attacks in cloud computing, *Proc. IEEE International Symposium on High Performance Computer Architecture*, pp.406–418 (2016).
[27] Kiriansky, V. et al.: DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors, *Proc. 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp.974–987 (2018).
[28] Yan, M. et al.: Invisispec: Making speculative execution invisible in the cache hierarchy, *Proc. of 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp.428–441 (2018).
[29] Khasawneh, N, K., et al.: SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *Proc. 56th ACM/IEEE Design Automation Conference*, pp.1–6 (2019).
[30] Hillenbrand, M. et al.: Process-local memory allocations for hiding KVM secrets, https://lwn.net/Articles/791069/. (accessed 2019-08-08).
[31] Zijlstra, P.: Core scheduling, https://lwn.net/Articles/780703/. (accessed 2020-06-12).
[32] Wang, Z. et al.: Safehidden: An efficient and secure information hiding technique using re-randomization, *Proc. 28th USENIX Security Symposium*, pp.1239–1256 (2019).