CL-002

カーネルにおけるサービス妨害攻撃緩和手法の提案

葛野 弘樹[1)]
Hiroki Kuzuno

概要

オペレーティングシステム（OS）のカーネル脆弱性を利用したサービス妨害攻撃は，計算機を管理する OS において対策が必要な課題である．カーネルにおいて，計算処理や計算資源の排他制御管理はロック処理で行われており，サービス妨害攻撃の一つは，カーネル脆弱性を利用し，排他制御に利用されるロック変数等を意図的に奪取することで，デッドロックを発生させカーネルを停止させる．カーネル脆弱性を検出する既存研究も存在し，カーネルのロック変数への参照調査手法によるデッドロック検出が行われている．しかし，カーネル脆弱性を利用した攻撃では，動作中カーネルに対して，デッドロックを発生させるため攻撃防止は困難である．本稿では，動作中カーネルに対するカーネル脆弱性を利用したサービス妨害攻撃の緩和のために，攻撃検出を行うセキュリティ機構を提案する．提案手法では，特定のロック変数のカーネルデータを制御するカーネル関数を対象とし，カーネル脆弱性を利用した強制的なカーネル関数呼出しの検出を実現する．提案するセキュリティ機構では，カーネル関数の呼び出し元検査を動作中カーネルにて行い，ロック変数を制御するカーネル関数呼出しを攻撃と判定し，サービス妨害攻撃の検出を可能とする．提案するセキュリティ機構の評価として，実現方式を備えた Linux にて，動作中の特定のカーネル関数に対する呼出し元検査を行い，サービス妨害攻撃を検出可能なことを確認した．また，性能評価にて，オーバヘッドは 0.015%から 5.134%であることを示した．

1　はじめに

オペレーティングシステム（OS）カーネルに対する攻撃として，サービス妨害攻撃がある．サービス妨害攻撃の一つでは，カーネルの排他制御に機構に用いられるロックを専有し，カーネルの停止を引き起こす．

カーネルの計算処理や計算資源の管理は，排他制御等により行われる．動作中カーネルに対するサービス妨害攻撃の一つでは，カーネル脆弱性を利用し，排他制御に利用されるロック変数等を意図的に奪取し，デッドロックを発生させカーネルを停止させる．

カーネル脆弱性の検出手法として，カーネルデータの参照調査を行い，ロック変数の追跡によるデッドロック検出がある [1]．しかし，動作中カーネルにおいて，カーネル脆弱性を利用した攻撃が行われた場合，デッドロック防止は困難であり，次の課題がある．

- 課題：ロック変数を操作するカーネル関数の呼出し元の検査ユーザプロセスによるカーネル脆弱性を利用したサービス妨害攻撃において排他制御に利用されるロック変数の奪取によるデッドロックは防止が困難である．サービス妨害発生前にロック変数を操作するカーネル関数の呼出し元を検査し，サービス妨害攻撃の検出を確実に行うことは課題である．

本稿では，課題解決として，サービス妨害攻撃の緩和のため，強制的なロック変数を制御するカーネル関数呼出しを対象として，サービス妨害攻撃を検出可能とするセキュリティ機構を提案する．図 1 に提案するセキュリティ機構の概要を示す．提案するセキュリティ機構では，最初にカーネルのソースコードの静的解析を行い，対象カーネル関数についての関数呼出しグラフを生成する．動作中カーネルにて，生成した関数呼出しグラフを用いて，対象カーネル関数の呼出しが適切か，カーネル関数呼出し元の検査を行う．

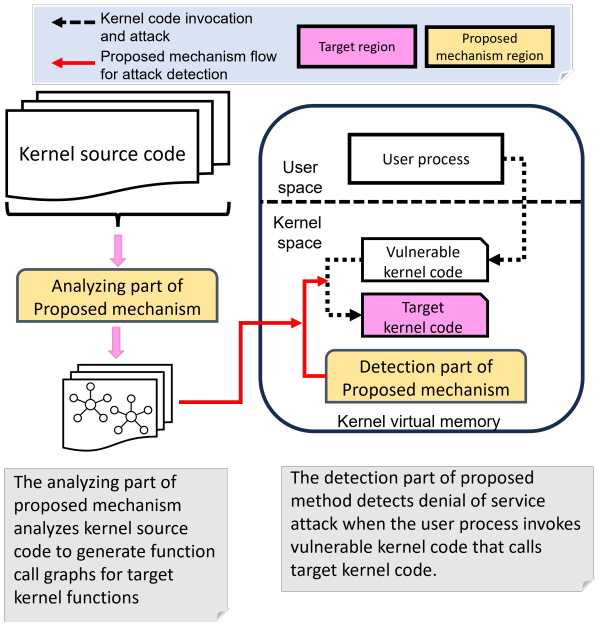

図 1　提案するセキュリティ機構の概要

提案するセキュリティ機構の実現方式では，Linux カーネルに対し，静的解析に `cflow` を用いて関数呼出しグラフを生成する．カーネル関数呼出し元の検査は，eBPF（extended Berkley Packet Filter）を用いる．対象カーネル関数の呼出し時，カーネル関数呼出し元のカーネル関数名の検査として，関数呼出しグラフと一致しているか比較し，一致しない場合は攻撃の可能性と判定することで，サービス妨害攻撃の検出を行う．

提案するセキュリティ機構では，対象カーネル関数にロック変数を制御するカーネル関数を指定することで，対象カーネル関数にて事前に決められたカーネル関数呼出し以外は，カーネル脆弱性を利用したサービス妨害攻撃として検出可能とする．

本稿での研究貢献は以下の通りである：

1. ロック変数を制御するカーネル関数を対象都市，呼出し元を検査することでサービス妨害攻撃の緩和のために攻撃を検出可能とするセキュリティ機構の設計と実装を行った．提案手法を Linux に適用し，特定のロック変数処理を行うカーネル関数を監視，強制的なカーネル関数の呼出しを，サービス妨害攻撃として検出可能とした．

1)　神戸大学 大学院 工学研究科

表 1　カーネル脆弱性を利用した攻撃による影響 [2]
(✓：提案手法の検出対象)

Effect	Description	Target
Memory corruption	カーネルコードやカーネルデータの改ざん	
Policy violation	カーネルにおける権限判定処理部分の実装不備	
Denial of Service	カーネルの強制停止	✓
OS information leakage	カーネルデータの初期化漏れによるデータ漏洩	

表 2　PoC の利用可能な Linux カーネル脆弱性リスト

CVE ID	Types	Description
CVE-2016-4997[3]	DoS, Mem. Corr.	Boundary check error
CVE-2016-9793[4]	DoS, Mem. Corr.	Boundary check error
CVE-2017-6074[5]	DoS	Use after free
CVE-2017-7533[6]	DoS, Mem. Corr.	Race condition
CVE-2017-16995[7]	DoS, Mem. Corr.	Boundary check error

2. 提案するセキュリティ機構の評価にて，サービス妨害攻撃を行うユーザプロセスによる強制的なロック変数を制御するカーネル関数呼出しを検出可能なことを確認した．また，オーバヘッドは 0.015%から 5.134%であることを示した．

2　背景知識

2.1　サービス妨害攻撃

カーネル脆弱性は，カーネルへの攻撃に利用可能な実装不備である [2]．カーネル脆弱性を利用した攻撃種別を表 1 に示す．カーネル関数を強制的に呼出し可能な脆弱性は，影響範囲が広く，ロック変数を制御するカーネル関数を呼出し，サービス妨害攻撃が可能となる．

サービス妨害攻撃攻撃として，Proof of Concept（PoC）が利用可能な Linux カーネルの脆弱性を表 2 に示す．サービス妨害攻撃で利用されるカーネル脆弱性として，カーネルにおけるメモリ破壊を行う脆弱性 [3, 4, 5]，ならびにデッドロックを発生させる脆弱性がある [6, 7]．

2.2　ロック変数の管理

図 2 に Linux での排他制御を行うスケジューラを示す．12 行目にて，CPU 制御のロックを管理する `rq` 構造体のロック変数に排他制御の情報を格納している．

排他制御は 16 行目から 37 行目にて行われる．ロックの取得は 16 行目の `rq_lock_irqsave` のカーネル関数にて行われる．17 行目から，34 行目にかけて，排他制御が必要な処理が行われる．ロックの開放は 37 行目の `rq_unlock_irqrestore` のカーネル関数にて行われる．

サービス妨害攻撃が成功した場合，攻撃時にロック取得が行われ，ロック開放は行われない．本来，排他制御が必要な処理にて，ロック取得が出来ないため，カーネルが停止し，再起動が求められる可能性がある．

3　脅威モデル

3.1　攻撃対象環境

提案するセキュリティ機構において，想定する脅威モデルとしての攻撃対象環境は次の通りである．

- 攻撃者：計算機に一般ユーザ権限にて侵入し，ユーザプロセスを実行可能とする．攻撃者のユーザプロセスは攻撃コードを介して脆弱なカーネル関数を呼出し，サービス妨害攻撃を行う．
- カーネル：カーネル脆弱性を含んでおり，攻撃者のユーザプロセスからサービス妨害攻撃に利用可能な脆弱なカーネル関数を呼出し可能とする．
- カーネル脆弱性：サービス妨害攻撃に利用可能なカーネル脆弱性とする．ロック変数を制御するカーネル関数を呼出し可能とする．
- 攻撃対象：攻撃対象は，カーネルにおいて排他制御を行う際に利用するロック変数を制御するカーネル関数とする．

```
1  /* File: kernel/sched/core.c */
2  /* Scheduler runqueue lock control example */
3  static void __sched notrace __schedule(bool preempt)
4  {
5      struct task_struct *prev, *next;
6      unsigned long *switch_count;
7      struct rq_flags rf;
8      struct rq *rq;
9      int cpu;
10
11     cpu = smp_processor_id();
12     rq = cpu_rq(cpu);               // Get CPU's runqueue
13     prev = rq->curr;
14
15     // Acquire runqueue lock (spinlock + IRQ disable)
16     rq_lock_irqsave(rq, &rf);
17
18     // Critical section: scheduling decision processing
19     update_rq_clock(rq);
20
21     switch_count = &prev->nivcsw;
22     if (prev->state && !(preempt & PREEMPT_ACTIVE)) {
23         if (unlikely(signal_pending_state(prev->state, prev
               ))) {
24             prev->state = TASK_RUNNING;
25         } else {
26             deactivate_task(rq, prev, DEQUEUE_SLEEP);
27             prev->on_rq = 0;
28         }
29         switch_count = &prev->nvcsw;
30     }
31
32     next = pick_next_task(rq, prev, &rf);  // Select next
           task
33     clear_tsk_need_resched(prev);
34     clear_preempt_need_resched();
35
36     // Release runqueue lock
37     rq_unlock_irqrestore(rq, &rf);
38 }
```

図 2　Linux カーネルにおける排他制御を行うスケジューラの例 [8]

3.2　攻撃シナリオ

攻撃シナリオにて，攻撃者は，カーネル脆弱性を利用したサービス妨害攻撃を行う．攻撃者は計算機に一般ユーザ権限で侵入し，攻撃用のユーザプロセスを実行する．攻撃者のユーザプロセスは脆弱なカーネル関数を介してロック変数を制御するカーネル関数を強制的に呼出す．続いて，攻撃対象とするロックの取得を行う．

強制的なロックの取得後，攻撃者のユーザプロセスによるロックの開放は行われない．該当ロックを用いるカーネルの排他制御は，ロックの取得が行えず，排他制御が必要な処理を実行できない．攻撃により，カーネルが停止した場合，サービス妨害攻撃は成功する．

4　提案手法

4.1　提案手法の要件

提案するセキュリティ機構は，動作中カーネルに対し，対象カーネル関数の呼出し元カーネル関数を検査し，サービス妨害攻撃を検出可能とする．次の要件を満たすよう設計を行う．

- 要件 1：指定する対象カーネル関数は，ロック変数を制御するカーネル関数とする．
- 要件 2：動作中カーネルに対して，カーネル関数の呼出し元カーネル関数名の取得，呼出し元の検査を行う．
- 要件 3：カーネル関数の呼出し元カーネル関数名を異常とみなした場合，攻撃と判定する．

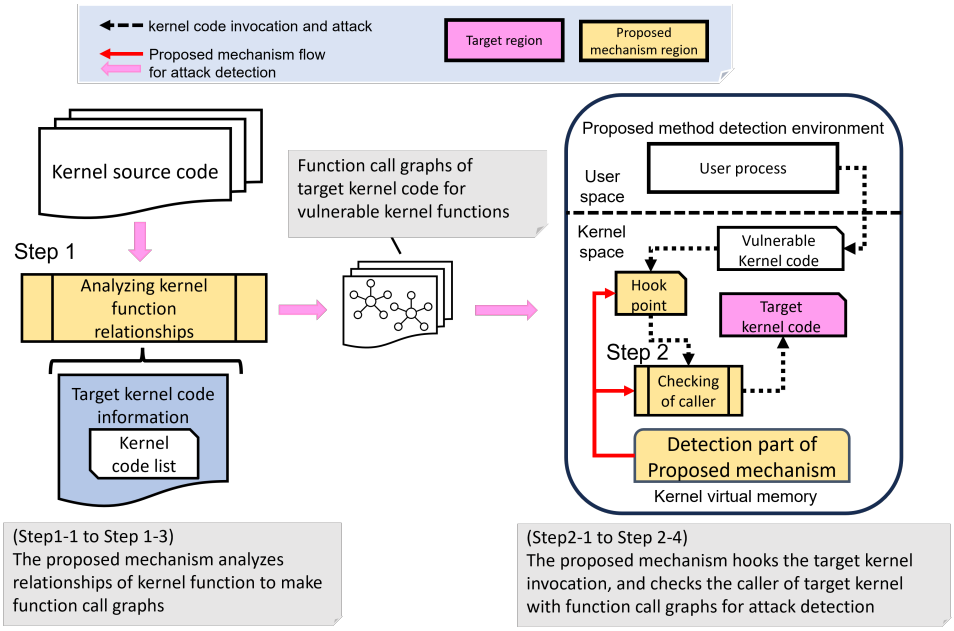

図 3　提案するセキュリティ機構の設計概要

4.2　提案手法の設計

4.2.1　設計方針

提案するセキュリティ機構の設計方針は次の通りとする．

- 設計方針 1：ロック変数を制御するカーネル関数を特定するため，カーネルのソースコードを静的解析し，対象カーネル関数に関する関数呼び出しグラフを生成する．
- 設計方針 2：ロック変数を制御するカーネル関数において関数呼出し元の検査を行うために指定されたカーネル関数の呼出し直後において，関数呼出しグラフを利用し，検査する．
- 設計方針 3：ロック変数を制御するカーネル関数での関数呼出し元の検査において，関数呼出しグラフに含まれない関数呼出しを攻撃と判定する．

4.2.2　設計概要

提案するセキュリティ機構の設計概要を図 3 に示す．要件を満たすために，設計方針に基づき，カーネルのソースコードの静的解析により，カーネル関数の関数呼出しグラフを生成する．続いて，カーネルにおいて，指定した対象カーネル関数において関数呼出し元を検査可能なセキュリティ機構を設ける．

提案するセキュリティ機構では，あらかじめソースコードの静的解析により，カーネル関数の直接呼出しを関数呼出しグラフとして把握する．加えて，動作中のカーネルにて，カーネル関数の実行時に関数呼出し元を検査し，攻撃者のユーザプロセスによる対象カーネル関数の呼出しが，あらかじめ決められたカーネル関数の呼出しと異なるかを検査し，攻撃有無を判定する．

4.2.3　関数呼出しグラフ

関数呼出しグラフは，ソースコードを静的解析して生成されるグラフである．関数呼出しグラフ g に対し，プログラムの関数を頂点 v，関数呼出しを辺 e とする．提案するセキュリティ機構では，カーネルの関数呼出しグラフ $g = \{V, E\}$，V として，頂点 v の集合，E は辺 e の集合，および L を頂点 v, 辺 e のラベルを返す関数とする．頂点 v のラベルはカーネル関数の名称，辺 e のラベルは関数呼出しと戻り規則とする．関数呼出し規則は，直接呼出し `direct`，とし，戻り規則は `ret` とする．直接呼出しはカーネル関数名称を用いた呼出しとする．

4.2.4　関数呼出しグラフの生成

提案するセキュリティ機構におけるカーネル関数呼出しグラフは，カーネルのソースコードを静的解析し，生成する．カーネルのソースコードに含まれるカーネル関数の集合 $K = \{k_1, \ldots, k_n\}$ に対するカーネル関数呼出しグラフの集合 $G = \{g_1, \ldots, g_n\}$ を生成する．一連の流れを以下で述べる．

- (Step 1-1)：カーネルのソースコードからカーネル関数を抽出
- (Step 1-2)：起点となる指定したカーネル関数（起点カーネル関数）k_i を特定
- (Step 1-3)：起点カーネル関数の呼び出しグラフを生成するため Step 1-3-a から Step 1-3-d を行う
  - (Step 1-3-a)：起点カーネル関数 k_i に対応する関数呼出しグラフ $g_i = \{V_i, E_i\}$ を生成
  - (Step 1-3-b)：起点カーネル関数の名称をラベルとする頂点 v_n を作成，V_i に追加
  - (Step 1-3-c)：起点カーネル関数を呼出すカーネル関数を特定，同関数の名称をラベルとする頂点 v_m を作成し，V_i に追加
  - (Step 1-3-d)：関数呼出し関係 e_n を作成．直接呼出し `direct`，または戻り規則 `ret`，を辺 e_n のラベルとし，E_i に追加
  - (Step 1-3-e)：起点カーネル関数毎に呼出し元のカーネル関数を列挙し，頂点と辺の追加を指定した深さで再帰的に行う

4.2.5　関数呼出し元の検査

提案するセキュリティ機構におけるカーネル関数呼出し元の検査は，カーネル関数の集合 K，ならびにカーネル関数呼出しグラフの集合 G を利用する．手続きを以下で述べる．

- (Step 2-1)：カーネル関数呼出し元の検査を実施する指定されたカーネル関数を k_j とする
- (Step 2-2)：カーネル関数 k_j が，カーネル関数の集合 K に含まれているか確認
- (Step 2-3)：含まれていた場合，起点カーネル関数の呼び出しグラフが存在する Step 2-3-a から Step 2-3-f にてカーネル関数呼出し元の検査を行う
  - (Step 2-3-a)：カーネル関数 k_j を，起点カーネル関数とし，対応する関数呼出しグラフ $g_j = \{V_j, E_j\}$ を選択
  - (Step 2-3-b)：カーネル関数 k_j の呼出し元カーネル関数を取得し，k_m とする．ただし $k_m \neq k_j$.
  - (Step 2-3-c)：起点カーネル関数グラフ g_j の頂点集合 V_j に対し，カーネル関数 k_j を名称とする頂点 v_j，および呼出し元カーネル関数 k_m を名称とする頂点 v_m が含まれているか探索
    - v_m が存在する場合，頂点 v_j と v_m 間に辺 e_jm が関数呼出しグラフの辺集合 E_j に存在するか探索
    - e_{jm} が存在する場合，頂点 v_j と v_m 間のラベルが直接呼出し `direct` であるか確認
  - (Step 2-3-d)：次の条件のいずれかの場合，異常と判定
    - v_m が存在しない場合：カーネル関数 k_j は呼出し元カーネル関数 k_m から呼ばれることはないと判定
    - e_{jm} が存在しない場合：カーネル関数 k_j は呼出し元カーネル関数 k_m から直接呼ばれることはないと判定
    - e_{jm} のラベルが `direct` でない場合：：カーネル関数 k_j は呼出し元カーネル関数 k_m から直接呼出し以外で呼ばれたと判定
- (Step 2-4)：カーネル関数 k_j に対する起点カーネル関数の呼び出しグラフが存在しない場合，カーネル関数呼出し元の検査は行わない．

4.2.6 対象カーネル関数

提案するセキュリティ機構によるサービス妨害攻撃検出のため，カーネル関数の呼出し元検査を行うカーネル関数を対象カーネル関数とし，次の通りとする．

- 対象カーネル関数：ロック変数を制御するカーネル関数（例．ユーザプロセススケジューラ，ファイル制御，デバイス制御等）

4.2.7 サービス妨害攻撃の検出

攻撃者の攻撃用ユーザプロセスにより，サービス妨害攻撃が成功した場合，攻撃用ユーザプロセスを実行している動作中カーネルは停止する．提案するセキュリティ機構では，対象カーネル関数におけるカーネル関数の呼出し元検査において，次の条件を満たした場合，サービス妨害攻撃が発生したとみなす．

- サービス妨害攻撃とみなす条件
  対象カーネル関数の実行時において，関数呼出し元検査を行い，関数呼び出しグラフに含まれないカーネル関数の呼出しと判定された場合，サービス妨害攻撃が行われたとみなす．

5 実現方式

提案するセキュリティ機構の実現方式は x86_64 CPU アーキテクチャ Linux を対象とした．

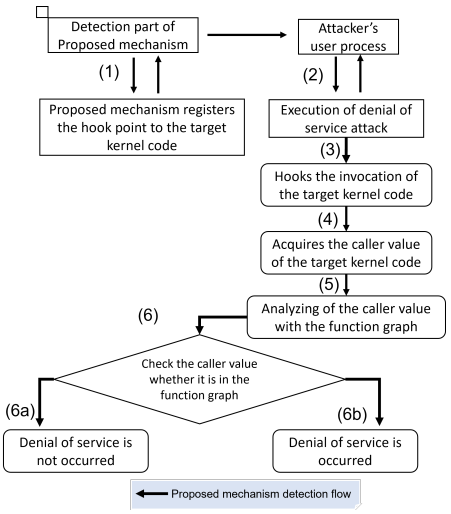

図 4 実現方式におけるサービス妨害攻撃検出処理フロー

5.1 実現方式の概要

実現方式において，サービス妨害攻撃に利用されることを想定し，対象カーネル関数に関係するソースコードに対して静的解析を行い，関数呼出しグラフを生成する．動作中カーネルにおいて，専用のセキュリティ機構を設けることで，対象カーネル関数の関数呼び出しグラフの読込み，および対象カーネル関数呼出し時の関数呼出し元の検査を実現する．サービス妨害攻撃の検出として，関数呼出し元の検査結果の値を取得し，ユーザプロセスによるサービス妨害攻撃の発生有無を判定する．

5.2 関数呼出しグラフ

実現方式での関数呼出しグラフの生成は，対象カーネル関数に対して，`cflow` を用いてソースコードの静的解析を行う． `cflow` により，対象カーネル関数からの呼出し元の逆順を探索し，対象カーネル関数を直接呼出しするカーネル関数を特定，関数呼出しグラフを生成する．

5.3 関数呼出し元の検査

実現方式における関数呼出し元の検査処理，および関数呼出し元の検査処理登録は次の通りである．

- 関数呼出し元の検査処理：eBPF を用いて対象カーネル関数の実行前に，関数呼び出しグラフを用いて関数呼出し元の検査を行う処理を実施する．
- 関数呼出し元の検査処理の登録：動作中カーネルにおける，関数呼出し元の検査処理の登録は Kprobes（Kernel Probes）を用いる． Kprobes は Linux カーネルの関数呼出し前後の処理登録に利用可能である．eBPF における，Kprobes の処理登録機能を利用し，対象カーネル関数の呼出し前の実行関数として，関数呼出し元検査処理の動的追加を実現する．

5.4 サービス妨害攻撃の検出処理

実現方式でのサービス妨害攻撃の検出処理のフローを図 4 に示す．サービス妨害攻撃の検出は，対象カーネル関数の実行前において，関数呼出しグラフを利用した関数呼出し元検査にて行う．検査は次の手順で行う．

1. 攻撃用のユーザプロセスの実行前，対象カーネル関数に対して関数呼出しグラフを利用した関数呼出し元の検査処理を登録
2. 攻撃用のユーザプロセスによるサービス妨害攻撃を実行

3. 対象カーネル関数の呼出しを捕捉
4. 対象カーネル関数の呼出し元関数の値を取得
5. 関数呼出しグラフを利用して，対象カーネル関数の呼出し元関数が含まれているか検査
6. サービス妨害攻撃の検出処理を開始
   (a) 関数呼出しグラフにおいて，関数呼出し元の関数から対象カーネル関数への直接呼出しの辺が存在する場合：サービス妨害攻撃ではないと判定
   (b) 関数呼出しグラフにおいて，関数呼出し元の関数から対象カーネル関数への直接呼出しの辺が存在しない場合：サービス妨害攻撃と判定

6 評価

6.1 評価項目と目的

提案するセキュリティ機構に対し，セキュリティ機能の評価として，サービス妨害攻撃の検出能力，ならびにカーネルに与える性能負荷を評価した．評価項目と内容を以下に示す．

1. サービス妨害攻撃の検出能力の評価
   提案するセキュリティ機構を適用したカーネルにおいて，サービス妨害攻撃に利用可能なカーネル脆弱性をカーネルに導入し，攻撃用のユーザプロセスによるサービス妨害攻撃を検出可能か評価した．
2. 提案するセキュリティ機構の性能負荷評価
   提案するセキュリティ機構を適用した Linux カーネルにおいて性能負荷についてベンチマークソフトウェアを用いて測定した．

6.2 評価環境

セキュリティ評価，ならびに性能評価に用いた評価用計算機は Intel(R) i9-13900H（2.60 GHz，20 コア，メモリ 64 GB）とし，仮想環境として ProxMox 8.1 を利用する．評価対象の仮想マシン（CPU 4 コア，メモリ 8 GB）を用意した．仮想マシンの OS は Debian 12，Linux カーネル 6.1.0-20-amd64 とした．提案するセキュリティ機構における関数呼び出しグラフの生成は，`cflow` 1.7 を用い，Python により 719 行で実現した．また，サービス妨害攻撃の検出は Linux eBPF で行い，268 行で実現した．

サービス妨害攻撃に利用可能なカーネル脆弱性

提案するセキュリティ機構のサービス妨害攻撃の検出能力の評価のため，次の通り，サービス妨害可能な脆弱性を備えるシステムコールを導入した．

- 独自システムコール：独自システムコールでは，`sys_dos_vuln` 関数にてロック取得を行う関数とする．通常は `sys_dosvuln_valid` 関数からのみ呼出されるが，`sys_dosvuln_invalid` 関数から，強制的に呼出し可能とした．

6.3 サービス妨害攻撃の検出能力の評価

サービス妨害攻撃の検出能力の評価結果として，サービス妨害攻撃を行う攻撃用ユーザプロセス実行結果を図 5 に示す．2 行目にて，攻撃用ユーザプロセスの `uid` の値は 1,000 であり，一般ユーザである．300 秒後，4 行目にて，独自システムコール 1 を呼出し，サービス妨害攻撃を行う，11 行目にて，提案するセキュリティ機構によりサービス妨害攻撃として検出されている．

5 行目から 8 行目は，提案するセキュリティ機構におけるロック変数を制御するカーネル関数の呼出し元の検査の登録を示している．9 行目，および 10 行目は，提案するセキュリティ機構におけるロック変数を制御するカーネル関数の呼出し元の検査の結果を示している．

```
// PoC code running, process id is 928
1.  user $ ./a.out
2.  uid=1000(user) gid=1000(user) groups=1000(user)
3.  // wait 300 seconds
4.  // forcibly call sys_dosvuln function invocation

// Register hook point for dos attack detection
// Import kernel function call graph for target kernel code
5.  root $ bpftrace sys_dos_vuln_filtered.bt a.out
6.  Attaching 1 probe …
7.  Monitoring a.out, register kprobe for sys_dosvuln
8.  Import kernel function call graph for sys_dosvuln

// Kernel function call is valid
9.  a.out[928] <- sys_dosvuln is called from sys_dosvuln_valid
// Kernel function call is invalid
10. a.out[928] <- sys_dosvuln is called from sys_dosvuln_invalid
11. DoS attack was detected, attack user process pid=928

Red text is the points of kernel denial of service information
```

図 5 提案するセキュリティ機構によるサービス妨害攻撃の検出結果

提案するセキュリティ機構は 9 行目では，関数呼び出しグラフに登録されたカーネル関数からの呼出しとして，サービス妨害攻撃と判定していない． 10 行目では，強制的にロック変数を制御するカーネル関数が呼出され，関数呼び出しグラフに含まれないカーネル関数の呼出し元であり，サービス妨害攻撃と判定している．

提案するセキュリティ機構は，事前に生成した関数呼び出しグラフに基づき，サービス妨害攻撃の検出に成功している．

6.4 性能負荷評価

ベンチマークソフトウェア UnixBench を用いて提案するセキュリティ機構の実現方式を備えたカーネルの性能負荷評価を行った．対象カーネル関数は `schedule` 関数とし，カーネル関数呼出し元検査を実施した．

性能負荷評価では，提案するセキュリティ機構の適用前の Linux カーネル，および提案するセキュリティ機構の適用後の Linux カーネルにて 5 回実行し，平均値から性能スコアを算出した．UnixBench は数値計算，ファイルコピー，プロセス処理，ならびにシステムコールに関する性能を測定し，カーネルの性能が高いほど大きなスコア値を示す．

UnixBench によるカーネル性能の評価結果を表 3 に示す．表 3 から，提案するセキュリティ機構により，最も性能への影響が大きいのは File Copy 4096 bufsize の 5.134 %であり，最も性能への影響が小さいのは数値計算の Double-Precision Whetstone の 0.015 % であることを確認した．評価では，`schedule` 関数 を対象カーネル関数としており，ユーザプロセスの制御に関係する指標として，Execl Throughput は 2.196%，Shell Scripts は 2.096%，2166%であることを確認した．また，カーネルの性能スコア全体への影響は 1.518 %であることを確認した．

7 考察

7.1 評価に対する考察

提案するセキュリティ機構を適用した Linux カーネルにおけるサービス妨害攻撃攻撃の検出能力評価として，対象カーネル関数にロック変数を制御するカーネル関数とした場合，呼出し元の検査は可能であることを確認した．提案するセキュリティ機構を用いることで攻撃

表 3　UnixBench を用いた提案手法の性能スコア

	Vanilla kernel	Proposed mechanism
Dhrystone 2	19500.02	19454.38 (0.234%)
Double-Precision Whetstone	6217.60	6216.62 (0.015%)
Execl Throughput	3989.38	3902.90 (2.167%)
File Copy 1024 bufsize	6814.34	6676.34 (2.025%)
File Copy 256 bufsize	4264.94	4189.24 (1.774%)
File Copy 4096 bufsize	14348.72	13612.00 (5.134%)
Pipe Throughput	2865.02	2846.36 (0.651%)
Pipe-based Context Switching	1804.32	1796.84 (0.414%)
Process Creation	4211.98	4132.42 (1.888%)
Shell Scripts (1 concurrent)	10459.36	10240.04 (2.096%)
Shell Scripts (8 concurrent)	10149.26	9929.34 (2.166%)
System Call Overhead	1490.44	1483.20 (0.488%)
System Benchmarks Index Score	5453.14	5370.32 (1.518%)

者のユーザプロセスによるカーネル脆弱性を利用したロック取得を行うカーネル関数の強制的な呼出しを捕捉し，サービス妨害攻撃の緩和のための攻撃検出が可能である．

提案するセキュリティ機構を適用した Linux カーネルの性能評価結果より，動作中カーネルにて， eBPF によるカーネル関数の呼出し処理への割込によるオーバヘッド発生を確認した．動作中カーネルでの負荷として，対象カーネル関数にてカーネル関数の呼出し元を検査する場合，動作中カーネルおよびネットワークへの負荷はカーネル関数の呼出し毎に発生している． 評価より，eBPF による個々のカーネル関数への負荷は限定的と考えられる．一方，ロック制御に関する対象カーネル関数の増加，および対象カーネル関数の呼び出し回数の増加により，性能負荷は高くなると考えている．

7.2　提案手法の考察

7.2.1　提案手法の設計ならびに実現方式

提案するセキュリティ機構の設計では，カーネルのソースコード静的解析により生成した関数呼び出しグラフをカーネル関数呼出し元の検査に用い，ロック変数を制御するカーネル関数の呼出しがソースコードでの直接呼出しに含まれていることの確認を可能とした．

関数呼び出しグラフは関数の直接呼出しからのみ生成しているため，ポインタ関数等の間接呼出しへの対応は行われておらず，対応が必要である．また，動作中カーネルにおけるカーネル関数呼出し元の正確な取得においては，処理による影響を調査する必要があり，カーネル性能や安定性に影響を及ぼすと考えている．

提案するセキュリティ機構の実現方式により，カーネル脆弱性を利用したサービス妨害攻撃は検出可能となる．対象カーネル関数を増加させ，カーネル脆弱性を利用した特権昇格攻撃など，他の種別のカーネルへの攻撃を防止できる可能性があると考えている．

7.2.2　限界

実現方式では，特定のロック変数を制御するカーネル関数への適用としており，サービス妨害攻撃の検出は限定的である．任意のカーネル関数を対象とし，カーネル関数の呼出し元を検査する場合，性能負荷は検査対象毎に増加する．関数呼出し順序の検査を自動的に行う Control Flow Integrity（CFI）[10, 11] の導入など，提案するセキュリティ機構において，よりカーネル変更を最小限にとどめる方式との組合せを検討している．

また，提案するセキュリティ機構では，ユーザプロセスからのサービス妨害攻撃を想定しており，ネットワークを介したサービス妨害攻撃は対象ではない．サービス妨害攻撃の種別を調査し，提案するセキュリティ機構にて網羅的に攻撃を検出可能かの検討を予定している．

7.3　移植可能性

提案するセキュリティ機構の実現方式では，カーネルの静的解析に `cflow` を用いる． C 言語で記述され，ソースコードを参照可能なカーネルには適用可能である．

動作中カーネルにおけるカーネル関数の呼出し元の検査には，eBPF を利用している．移植においては，eBPF と同等のカーネル関数への追加的処理を行える機構が必要と考えている．FreeBSD では，eBPF の移植が検討されており，提案するセキュリティ機構を実現できる可能性はある [12]．また，Windows においても eBPF の導入が検討されており，カーネル関数の呼出し元検査は将来的に実現できると考えている [13].

8　関連研究

カーネル解析機能

カーネル解析研究として，LockDoc では，カーネル関数のロック処理をソースコードから抽出し，文書化する手法を提案している [15]．また，カーネルデータの調査研究として，ProbeBuilder では，カーネルデータ種別毎にプローブを設置，追跡する手法を提案している [16].

カーネル脆弱性の検出

カーネル脆弱性の検出手法として，DIFUZE はカーネルのソースコードを静的解析し，変数操作に基づく入出力データをを生成，カーネル脆弱性を特定する手法を提案している [1]. また，カーネルのソースコードから抽象構文木と制御フローグラフを生成，脆弱性の特徴のある箇所を Code Property グラフとして，カーネル脆弱性を検出する手法も提案されている [17].

不正カーネルコード実行防止

不正カーネルコード実行防止手法として， KCoFI ではカーネルコード呼出し順を検査し，不正カーネルコードの実行防止を可能としている [10]．また，ARM ポインタ認証を利用し，低負荷でのカーネルコード呼出し検証を実現する手法も提案されている [11].

カーネルへの攻撃制限

カーネルへの攻撃制限手法として，sysfilter では，プログラムのシステムコール呼出し推定から，関数呼出しグラフを作成し， BPF フィルタを生成，システムコールの制御を行う手法が提案されている [18]．また，VulShield はカーネルを含めたソフトウェアの脆弱な処理に対して，割込みを行い，脆弱な処理を実行制御する手法を提案している [19].

9　まとめ

本研究では，動作中カーネルに対して，サービス妨害攻撃の緩和のために攻撃検出を行うセキュリティ機構を提案した．提案手法では，排他制御に関するロック変数を操作するカーネル関数を対象とし，カーネル脆弱性を利用した攻撃による不正呼出しを，カーネル関数の呼出し元検査にて攻撃と判定，サービス妨害攻撃として検出可能としている．

提案するセキュリティ機構の実現方式では，対象カーネル関数に対し，動的なカーネル関数の呼出し元検査の適用を実現した．評価結果において，サービス妨害攻撃

を試みるユーザプロセスに対し，ロック変数を操作するカーネル関数の強制的な呼出しを検出可能なことを検証した．また，性能評価として，提案手法を適用した Linux カーネルでのオーバヘッドは 0.015%から 5.134%であることを示した．

謝辞

本研究の一部は，JSPS 科研費 JP23K16882，JP25K03119，電気通信普及財団 2024 年度研究助成，ならびに JST ACT-X JPMJAX24M4 の助成を受けたものです．

参考文献

[1] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, V., Hao, S., Kruegel, C. and Vigna, G.: DIFUZE: Interface Aware Fuzzing for Kernel Drivers, *In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, (2017).

[2] Chen, H., Mao, Y., Wang, X., Zhow, D., Zeldovich, N. and Kaashoek, F, M.: Linux kernel vulnerabilities - state-of-the-art defenses and open problems. *In: Proceedings of the Second Asia-Pacific Workshop on Systems*, pp. 1–5, (2011).

[3] CVE-2016-4997, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4997. (accessed 2025-05-12).

[4] CVE-2016-9793, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9793. (accessed 2025-05-12).

[5] CVE-2017-6074, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6074. (accessed 2025-05-12).

[6] CVE-2017-7533, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533. (accessed 2025-05-12).

[7] CVE-2017-16995, available from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995. (accessed 2025-06-10).

[8] The Linux Kernel Archives, available from https://www.kernel.org/. (accessed 2025-06-10).

[9] eBPF, available from https://ebpf.io/. (accessed 2025-06-10).

[10] Criswell, J., Dautenhahn, N. and V. Adve.: KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels, *In: Proceedings of the 35th IEEE Security and Privacy*, pp. 292–307, (2014).

[11] Sungbae Y., Jinbum P., Seolheui K., Yeji K. and Taesoo K.: In-Kernel Control-Flow Integrity on Commodity OSes using ARM-Pointer Authentication, *In: Proceedings of the 31st USENIX Conference on Security Symposium*, pp. 89–106, (2019).

[12] Hayakawa, Y.: eBPF Implementation for FreeBSD, available from https://papers.freebsd.org/2018/bsdcan/hayakawa-ebpf_implementation_for_freebsd/. (accessed 2025-06-10).

[13] eBPF for Windows, available from https://github.com/microsoft/ebpf-for-windows. (accessed 2025-06-10).

[14] Petrol, N, L., Hicks, M.: Automated detection of persistent kernel control-flow attacks, *In: Proceedings of the 14th ACM SICSAC Conference on Computer and Communications Security*, pp. 103–115, (2007).

[15] Alexander L., Horst S., Hendrik B. and Olaf Spinczyk.: Lochman, A., et al.: LockDoc: Trace-Based Analysis of Locking in the Linux Kernel, *In: Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–15, (2019).

[16] C. W. Wang, and S. Shieh.: ProbeBuilder: Uncovering Opaque Kernel Data Structures for Automatic Probe Construction, *IEEE Transactions on Dependable and Secure Computing*, vol. 13, pp. 568–581, (2016).

[17] Yamaguchi, F., Golde, N., Arp, D. and Rieck, K.: Modeling and Discovering Vulnerabilities with Code Property Graphs, *In: Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 590–604, (2014).

[18] DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R. and Kemerlis, P., V.: sysfilter: Automated System Call Filtering for Commodity Software, *In: Proceedings of the 2020 International Conference on Research in Attacks, Intrusions, and Defenses*, (2020).

[19] Li, Y., Zhang, C., Zhu, J., Li, P., Li, C., Yang, S. and Tan, W.: VulShield: Protecting Vulnerable Code Before Deploying Patches, *In: Proceedings of the Network and Distributed System Security Symposium 2025*, (2025).