Vol.2010-OS-113 No.16
2010/1/28

プロセスグループ別LSMのための LSMフレームワーク拡張

中田晋平[†1]　倉光君郎[†2,†3]

近年、プロセスグルーピングによるアイソレーションやリソースアカウンティングは、システムの信頼性、安定性向上のため重要な技術になってきている．これに伴い、プロセスグループ単位でのオペレーションが増えると、プロセスグループ単位のセキュリティが必要になる．Linux オペレーティングシステムでは、セキュリティフレームワークとして Linux Security Modules (LSM) フレームワークがあるが、LSM フレームワークではシステム全体でひとつのセキュリティモデルの適用しかできず、プロセスグループ毎にセキュリティモデルを変更したい場合に問題となる．そこで我々は、プロセスグループ毎に独立したアクセス制御モデルを適用させるため、Security Cube LSM を提案し、また、LSM だけでは解決できない問題に関しては LSM フレームワークの拡張を行った．　本稿では我々の提案する Security Cube LSM と、LSM フレームワーク拡張のプロトタイプについて Linux 上で実装を行い、動作性能を測定した．測定の結果、オーバヘッドは 8%未満で収まることを確認した．

Multiplexing Linux Security Modules Framework for Applying Different Modules to Grouped Processes

Shinpei Nakata[†1] and Kimio Kuramitsu[†2,†3]

Recently, process grouping have become more important technology for the appearance of virtualization with container, resource accounting technology. In Linux, they have cgroups for process grouping. However, they have a difficulty in the security. Linux Security Modules Framework is a flexible, generic access control framework, which have been implemented in Linux, and have exploited since its appearance. Although, when system is partitioned with containers, LSM Framework will burden them with unnecessary overhead. In this paper, we propose Security Cube LSM, which is a LSM practitioner for grouped processes. Our Security Cube showed lower than 8% overhead for normal usage.

1. はじめに

近年、ハードウェアの高性能化に伴い、オペレーティングシステム (以下、OS) は豊富な資源を効率的に利用するための仮想化技術を実装するものが登場してきた[1)]．仮想化技術の一手法としてコンテナによる仮想化がある[12)]．コンテナによる仮想化はハイパーバイザによる仮想化に較べ、比較的オーバヘッドが少ないことが利点として挙げられており、注目を集めている．Linux においてコンテナによる仮想化を支える基盤技術として、バージョン 2.6.27 から Control Groups[6)](以下、`cgroups`) がマージされた．`cgroups` はプロセスを最小単位としたグループを構成する機能を持ち、グループ内のプロセスにリソース制約など、一定条件下での動作を強制することができる．現在、`cgroups` の主な用途としては Linux Containers[14)](以下、LXC) によるコンテナの構成に使用されている．

ここで、`cgroups` によるグルーピングはリソース制約を加えることができても、セキュリティを分離することはできないことが問題となる．Linux にはカーネルレベルのアクセス制御を実現する為に、Linux Security Modules Framework (LSM フレームワーク) が実装されている．LSM フレームワークでは、開発者が独自のアクセス制御モデルを Linux Security Module (LSM) としてモジュール化して、カーネル起動時にロードして使用することができるため、現在では数種類の LSM が用途に応じて使い分けられている．

ところが、コンテナとしてシステムから分離されたプロセスグループが存在していても、LSM フレームワークは全システムを対象として動作するため、コンテナに対して不適切なアクセス制御モデル、およびそのオーバヘッドが負荷されてしまうことになる．このため、管理者はコンテナを分離するためにセキュリティポリシーを再設定する必要があり、さらにコンテナは複数立ち上がるにも関わらず、LSM をそれぞれに分離することはできない.[15)]

そこで我々は `cgroups` などのプロセスグルーピングによって分離されたコンテナに対し、独立して LSM を適用することを研究の目標とし、セキュリティキューブを提案した．セキュリティキューブはプロセスグループごとに別の LSM にアクセス可否を問い合わせる．しかし、前述のとおり LSM は静的に決まることを前提に設計されているため、複数の LSM を

†1 横浜国立大学大学院
Graduate School of Yokohama National University
†2 横浜国立大学
Yokohama National University
†3 科学技術振興機構
Japan Science Techonology Agency

カーネル内で動作させると、特に LSM 専用のデータ構造を各 LSM が独占することが問題になる．そこでさらに、我々はカーネルリソースへのアクセスを抽象化した Kernel Object Accessor (以下、KOA) を提案し、各 LSM はそれぞれの KOA を通じて専用データへアクセスを行うこととした．

本稿では、Linux Kernel 2.6.30 上にセキュリティキューブと KOA のプロトタイプを実装した．また、カーネルメインラインに含まれる LSM である TOMOYO Linux、 Smack に KOA を実装し、セキュリティキューブ上でのパフォーマンス測定を行った．本論文の貢献は次の通りである．

(1) LSM フック関数を切り替えてプロセスグループごとに LSM を変更するセキュリティキューブを提案した
(2) カーネルオブジェクトへアクセスするための抽象化インターフェースである KOA を設計した
(3) セキュリティキューブのプロトタイプを Linux Kernel 2.6.30 に実装しパフォーマスを測定した

本論文の構成は、以下の通りである．第 2 節では問題定義を行う．第 3 節では提案するシステムの設計について述べる．第 4 節では実装の詳細を述べ、第 5 節では評価実験、第 6 節で関連研究と比較し、第 7 節で結論とした．

2. 問題定義

前節に述べた通り、プロセスグルーピングによるコンテナ構成は軽量な OS 仮想化を提供する一方、システムが既存のセキュリティフレームワークでは想定されていない設計になるため、セキュリティポリシーを書く際に不必要な手間をかけてしまうことが問題となる．

そこで我々は Linux の標準セキュリティフレームワークである LSM フレームワークを拡張し、複数の LSM を扱うことを考えた．しかし、既存の LSM フレームワークは単一の LSM しか使わないという前提があるため、分割、多重化には工夫が必要である．そこで本節では、LSM フレームワークの動作原理を説明し、問題箇所を指摘する．

2.1 LSM フレームワークの動作

LSM フレームワークは Linux におけるアクセス制御によるセキュリティ確保の為のフレームワークとしてバージョン 2.6 から追加されたものである[10]．LSM フレームワークを用いることで、開発者は独自のアクセス制御モデルをモジュールとしてカーネルに組み込むことができる．Linux カーネルへのモジュール組み込みのための仕組みは Loadable Kernel Modules (LKM) として既に確立されており、LSM も一種の LKM として作成する．LSM フレームワークはソースコード中に埋め込まれたアクセス制御のためのフックポイントを提供し、そのフック関数でありアクセスの可否を計算する関数（アクセス制御関数）を持つ LKM(LSM) の作成、登録をサポートするための機能から構成される．その動作の様子を、よく使われるシステムコール open を例に、**図 1** に示した．フックポイントは主にファイルの読み書きなど、カーネルが管理するリソース (カーネルオブジェクト**表 2.2**) へのアクセスを伴うシステムコール中の、特にリソースアクセスの直前に埋め込まれる．これは不必要な TOCTTOU 脆弱性[7] を生まない為である．図 1 で open システムコールはファイルの管理情報である *inode* へアクセスを行う．*inode* からファイルの実体を探すことはできるので、アクセス制御の対象は *inode* となる．*inode* へのアクセスを行う関数の直前にフックポイントが埋め込まれ、アクセスの可否を LSM にあるアクセス制御関数へと問い合わせる仕組みである．

また、LSM フレームワークは、アクセス制御の対象となるカーネルオブジェクト (表 2.2) のそれぞれについて、各 LSM が独自の管理構造への参照を保持するためのポインタ (セキュ

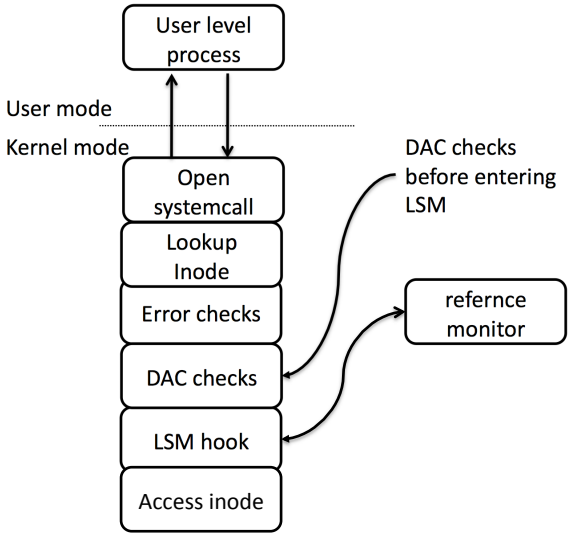

図 1 LSM の仕組み
Fig. 1 Linux Security Module Framework

リティコンテキストポインタ) を用意している．現在、LSM として Linux でメインライン化がなされているものとして、SELinux[2], Smack[5], TOMOYO Linux[4] などが存在する．

2.2 問 題 点

LSM フレームワークは単一の LSM を動作させるためのフレームワークである．そのため、複数の LSM をカーネルで動作させようとした場合、以下に示す２点が問題となる．

(1) 単一の LSM しか登録できない設計

(2) カーネルオブジェクトのセキュリティコンテキストポインタが独占される

まず、第一の問題は、LSM フレームワークへの LSM 登録で発生する．LSM の登録には `register_security` 関数を用いるが、すでに登録されている LSM があった場合、それ以降の LSM の登録は行うことができない実装になっている．したがって、複数の LSM を登録でき、かつ、フックポイントからそれぞれの LSM へアクセス可否を問い合わせるための拡張が必要になる．

次に、第二の問題点について述べる．カーネルオブジェクトのセキュリティコンテキストポインタは各 LSM が独自の管理データへの参照を保持させるために用いられる．そのため、仮に第一の問題を解決し、複数の LSM への問い合わせを行えたとしても、各 LSM はカーネルオブジェクトのセキュリティコンテキストポインタへ直接アクセスを行ってしまう．そのため複数の LSM が動作していた場合、それぞれが管理データへの参照を上書きしてしまい、別の LSM のものへアクセスしてしまう．したがって、このセキュリティコンテキストポインタへの各 LSM からのアクセスも参照先をそれぞれで変更する必要がある．

表 1 カーネルオブジェクトの例 (Linux v2.6.30)
Table 1 An example of Kernel Object (Linux v2.6.30)

データ	構造体名	フィールド
プロセス	struct task	void * security
ソケット	struct socket	void * sk_security
i-node	struct inode	void * i_security
ファイルシステム	struct super_block	void * s_security

3. システム設計

上述の問題を解決するために我々は新しい LSM であるセキュリティキューブと、そのために必要な LSM フレームワークの拡張である Kernel Object Accessor(KOA) を提案する．

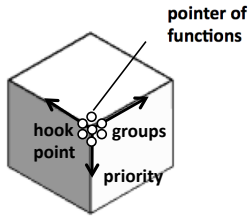

図 2 セキュリティキューブ

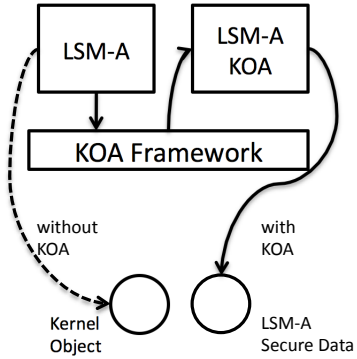

図 3 Kernel Object Accessor

以下、それぞれの設計を述べる．

3.1 セキュリティキューブ

LSM をプロセスグループごとに分割するためには、LSM のフックポイントをさらにフックし、システムコール発行元のプレセス ID(以下、PID) からプロセスグループ ID(PGID) を取得する．取得された PGID と LSM のマッピングから、該当のアクセス制御関数を呼び出し、アクセスの可否（バイナリで表現され、0 がアクセス許可、1 がアクセス不許可）をそのまま LSM のフックポイントへ返す機能が必要である．我々は要求される機能を２つに分割した．ひとつは PGID、LSM フックポイント、優先順位の３つを軸としたマトリクスであり、これをセキュリティキューブと名付けた．セキュリティキューブを**図 2** に示す．

実際のセキュリティキューブは関数ポインタの集合である．ここで、PGID と LSM フックポイントから検索可能な、アクセス制御関数へのポインタの集合をアクセス制御関数ベクタと定義する．PGID、LSM フックポイントの２軸ではなく、優先度の３軸目を加えたのは、ひとつのグループに対し、複数 LSM を指定するためである．これは、一部の LSM コミュニティにおいて、セキュリティモデルが他の LSM と異なるため、同時に併用して使っている例があるためである．この場合、一方の LSM は LSM としてではなく、カーネルパッチとして適用される．しかし、本稿では複数 LSM を合成した場合のセキュリティについては議論しない．

次に、セキュリティキューブへの変更を行うキューブコントローラについて説明する．セキュリティキューブには PGID と LSM の対応、LSM の優先度をアクセス制御関数の粒度

で指定することができる．この変更をおこなうのがキューブコントローラである．キューブコントローラはユーザ空間とカーネル空間に API をもち、キューブへの変更を実際に行う．また、PID は実行時に変更されるので、PID と PGID の対応表を管理する機能も持つ．キューブコントローラはセキュリティキューブの唯一のインタフェースとなり、外部からの変更はできない．

3.2 Kernel Object Accessor

LSM フレームワークはセキュリティコンテクキストポインタを独占することを前提に設計されている．そのため、セキュリティキューブによりアクセス制御関数の振り分けが行われても、それぞれが参照するカーネルオブジェクト内のセキュリティコンテキストポインタは同じものになってしまう．この問題を解決するためには、カーネルオブジェクトの要素を増やすか、セキュリティコンテキストポインタへのアクセスをフックして、アクセス元の LSM ごとにアクセス先を変更するかのどちらかの方法が考えられる．我々は、将来的に Linux 開発コミュニティへ提案することを考え、カーネルオブジェクトの要素を増やすことは、現在広く実用化されているカーネルのメモリレイアウトを著しく崩してしまうため、ソースコードの変更がない後者を選択した．セキュリティコンテキストポインタへのアクセスをフックするため、我々は抽象化層を設け、Kernel Object Accessor と名付けた．

KOA はカーネルオブジェクトへのアクセサを抽象化したものである．各 LSM からのカーネルオブジェクトへのアクセスは KOA を用いて行う．各 LSM はそれぞれの KOA を用意することで、別々の場所にアクセスする．アクセス先はコンパイル時には決まらないため、実行時に KOA が判断することになる．**図 3** に KOA の動作の様子を示した．開発者の負担としては、それぞれの KOA を用意する必要があること、KOA を介することの関数呼び出しオーバヘッドが増えることなどがあげられるが、頻繁に変わるカーネルのコードシンタクスへの柔軟性があがるなどのメリットもでてくる．

3.3 プロセスグルーピング

セキュリティキューブは簡単のため、単純なプロセスグルーピング機構を持っている．管理者は指定した PID を任意のグループに属させることができる．任意のプロセスを指定できるため、子プロセスや親プロセスなどの関係も自由に指定できる．**図 4** は割り当てられたプロセスグループに所属するプロセスのシステムコールがどのようにセキュリティキューブによって制御されるかを示したものである．図の通り、複数の LSM をひとつのプロセスグループに対応させることも可能であり、LSM の実行順番は優先度により制御できる．

他のプロセスグルーピング手法とは自動的にマッピングをとることで対応可能であると考

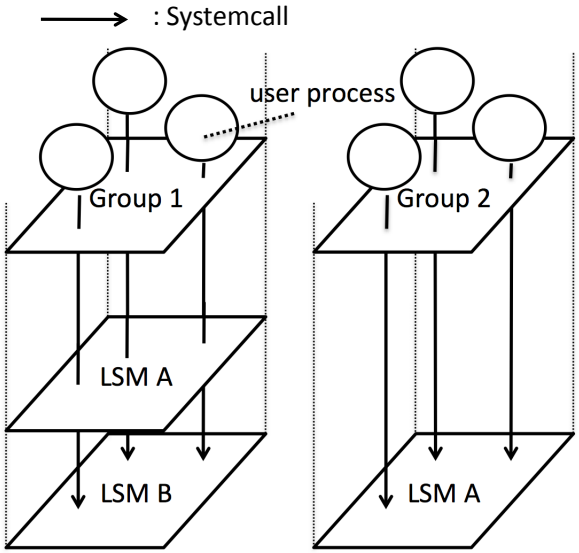

図 4 プロセスグループとセキュリティキューブによるアクセス制御
Fig. 4 A layered access controlling by Security Cube

えている．以下、具体的に cgroups を例に、マッピングの方法を示す．cgroups は任意のプロセスを任意のグループに属させることができる．グルーピングは多重化も可能で、その場合は双方のグループからの制約がかかる．今、プロセス P_1, P_2, P_3 が動作しているとし、これらを要素とする次の 2 つのグループを定義したとする．

$$G_1 = \{P_1, P_2, P_3\} \qquad G_2 = \{P_2, P_3\}$$

これをプロセス毎にみてみると、以下の通り、グループを要素とするグループを作ることができる．

$$G_{P_1} = \{G_1\} \qquad G_{P_2} = \{G_1, G_2\} \qquad G_{P_3} = \{G_1, G_2\}$$

G_{P_2}, G_{P_3} は同じ集合であるので、cgroups でグルーピングされた集合 G_1, G_2 には G_{P_1}, G_{P_2} を定義し、プロセス P_1 を G_{P_1} に、P_2, P_3 を G_{P_2} に属させることで等価な動作をさせることができる．

4. プロトタイプ実装

我々は Linux 2.6.30 上の LSM としてセキュリティキューブのプロトタイプを実装した．**図 5** にプロトタイプ実装のシステム図を示す．セキュリティキューブ上で動作させる LSM はカーネルメインラインに入っている Smack, TOMOYO Linux の 2 つを用いた．

システム図より、実装は大きく 3 つのコンポーネントから成る．

(1) セキュリティキューブ LSM
(2) キューブマネージャ
(3) KOA フレームワークとアクセサメソッド

以下にそれぞれ、実装の詳細を述べる．

4.1 セキュリティキューブ LSM

セキュリティキューブ LSM は、各 LSM へアクセス制御の可否を問い合わせる役割を果たす．そのためにはまず、LSM フックポイントをフックする必要がある．そのため、我々はセキュリティキューブを LSM として実装した．セキュリティキューブは LSM フレームワークの提供する 180 箇所以上のフックポイントすべてへのフック関数を実装した．これらフックポイントはカーネルのバージョンごとに異なるので、簡単なスクリプトを用意し、アクセス制御関数は自動生成することでバージョンの違いによる問題は解決した．各 LSM はすべてのフックポイントに対し、アクセス制御関数を用意しているとは限らない．その場合は、予め Linux に用意されている `default_security_ops` を用いることとした．また、セキュリティキューブは特定のプロセスグループに対し、優先度に沿って異なる LSM へアクセス可否を問い合わせることができる．本実装では、アクセス制御ベクタは、内部に登録されている LSM の数から 4 次元までとした．

説明のため open システムコール例に、実際のアクセス制御がおこる詳細を述べる．open システムコールが呼び出されると CPU はスーパバイザモードに移行し、カーネルへと制御が移る．カーネルは `sys_open` 関数を呼び出し、さらに `do_open` 関数を実行する．`do_open` 関数には LSM フックである `dentry_open` が埋め込んである．`dentry_open` は現在の LSM フック関数を管理する構造体から `dentry_open` ハンドラを起動する．この `dentry_open` フック関数がセキュリティキューブへのエントリポイントになっており、キューブへと制御を移す．

セキュリティキューブはシステムコールを起動したプロセスの PGID、およびこの場合は LSM フックに割り当てられた LSM フックポイント ID から該当の関数を探し、対応するアクセス制御ベクタを見つけたら、中でも優先度の高いアクセス制御関数、すなわちインデックス 0 のものを実行する．アクセス制御関数はアクセスの許可、不許可をバイナリで返すので、それをそのまま LSM フック元まで返し、`dentry_open` は処理を終え、アクセス許可ならば処理はそのまま実行を続け、不許可ならばその旨がシステムコール発行元へ知らされる．

4.2 キューブマネージャ

セキュリティキューブは、キューブマネージャに実行中の PID をもとに PGID と、対応する LSM を問い合せる．実行中の PID はグローバル変数である `current` 変数から取得する．

4.2.1 インタフェース

アクセス制御関数はユーザが任意の関数をカーネルモジュールとして用意することができる．アクセス制御関数の登録はセキュリティキューブ API をもちいておこなう．API には以下のメソッドが用意されている．

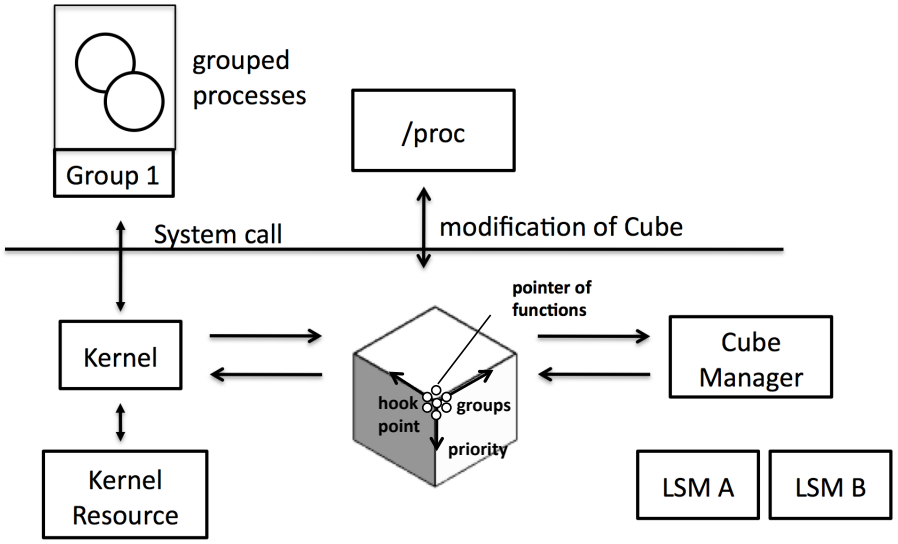

図 5 プロトタイプ実装のシステム図
Fig. 5 Prototype system

- POST:新規登録
- PUT:変更
- DELETE:削除
- GET:情報取得

アクセス関数は API を用いて作成されたカーネルモジュールを `insmod` することで有効になる．

もうひとつのインタフェースは proc ファイルシステム上にマウントされる．これは、登録したいアクセス制御関数がカーネルモジュールとして組み込まれるだけとは限らず、また、モジュールの切り替えの度に `rmmod` する手間を省くためである．

本システムではこのように２つの方法でアクセス制御関数のセキュリティキューブへの登録を行う．また/proc/securitycube 以下にはグループ ID に対応したファイルがあり、それぞれから登録されているアクセス制御関数の情報の一覧を取得することができる．

4.3 KOA

KOA はカーネルオブジェクト内のセキュリティポインタへのインタフェースとなり、以下の２つのアクセサが定義される．

- `koa_get()`
- `koa_set()`

実装では既存 LSM である TOMOYO Linux と Smack について、カーネルリソースのセキュリティコンテクストへ直接アクセスをしていた部分を修正した．今回は KOA の書き換え先の変更ではなく、それぞれに上記メソッドを用意してアクセス先を変更した．LSM 開発者の負担として、ソースコード変更は負担となるが、参考までに今回の実装で行ったそれぞれの LSM の変更行数を**表 2** に示す．LSM 自体は 5000 行を超えるので、その割合は 5%未満である．

4.4 グルーピング制御

セキュリティキューブにおけるプロセスのグループの変更は PID を proc ファイルシステムに以下のように書き込むことで切り替えられる．

表 2 KOA による修正
Table 2 KOA Modification LOC of each LSM

LSM	修正箇所 (Line Of Code)
TOMOYO	43
Smack	103

```
echo (PID) > /proc/securitycube/(PGID)
```

今回は手動での変更になるが、他グルーピング機構とマッピングをとることで既存のグルーピングと対応付けが可能である（3.3）．

今回実験の対象とした２つの LSM はどちらもシステムの状態遷移をセキュリティモデルの一部として持っている．そのため、LSM を利用してアクセスがある度に状態が変更されれば、その変更を記録していた．このような場合、グループごとに LSM フック関数を切り分けて、あるグループだけフック関数が呼ばれない場合、LSM で管理しているシステムの状態に不整合が生じる．

そこで我々はグループに関係なくシステムの状態遷移を監視させるためアクセス制御計算のみをグルーピングの対象とした．つまりグルーピングできまるのは、アクセス制御がおこるか否かであり、実際の関数が呼ばれるか否かではない．

5. 実　　験

本節ではセキュリティキューブのプロトタイプ実装の性能評価のためにおこなった評価実験の手法と結果について述べる．

5.1 評 価 実 験

我々はプロトタイプ実装の評価のために以下の実験を行った．まず、セキュリティキューブのオーバヘッドを測定するため、よく使われるとおもわれるシステムコール呼び出しにかかる時間を LSM 単体時、セキュリティキューブ経由時に対して測定した．

次に TOMOYO Linux, Smack を単体で呼び出した場合、セキュリティキューブを通じた場合の２つを比較してセキュリティキューブの実用的なオーバヘッドを測定した．

最後に TOMOYO Linux, Smack を KOA 化してセキュリティキューブにのせた場合と、それぞれを修正なく LSM フレームワーク上で動かした場合の比較を行い、オーバヘッドを測定した．実験環境は KVM 上の Linux kernel 2.6.30, Intel Xeon 2.53GHz, 2GB RAM でおこなった．システムコールのベンチには lmbench-3.0-a5[13] を用いた．

5.2 実 験 結 果

表 3 よりセキュリティキューブを介することでのオーバヘッドは 10%以内でありオーバヘッドは許容範囲内であるとわかった．それぞれのシステムコールで結果に差があるのは、実際に通過する LSM フックの数が違うからであると考えられる．また、**表 4**〜**表 7** において、セキュリティキューブを使用した時と LSM を使用したときのオーバヘッドも 10%前後であり、グルーピングによるオーバヘッドも LSM と大差ない．TOMOYO と Smack で差

表 3 セキュリティキューブのオーバヘッド
Table 3 Overhead of Security Cube

システムコール	LSM	LSM+SC	オーバヘッド
open	2.65	2.82	+6.4%
write	0.39	0.41	+5.1%
read	0.43	0.45	+4.6%
stat	1.70	1.83	+7.6%

表 4 TOMOYO Linux 適用時
Table 4 TOMOYO Linux on Security Cube

システムコール	LSM	SC	オーバヘッド
open	3.96	4.0	+1.0%
write	0.39	0.41	+5.1%
read	0.43	0.45	+4.7%
stat	1.68	1.76	+4.8%

表 5 Smack 適用時
Table 5 Smack on Security Cube

システムコール	LSM	SC	オーバヘッド
open	2.70	2.90	+7.4%
write	0.39	0.42	+7.8%
read	0.42	0.45	+7.1%
stat	1.80	1.90	+5.6%

が開いたのも、TOMOYO がフックする関数は 12 個に対し、Smack は 101 個のフックを行うためであると考えられる．

6. 関 連 研 究

アクセス制御の研究は古くより行われてきた．SELinux は米国 NSA が研究開発したセキュア OS であるが、SELinux がもつ特徴の一つに、ドメインがある．SELinux はセキュリティモデルとして SVO モデルを用いてポリシーを記述する．ドメインはグループ化されたプロセスの集合であり SVO モデルでは Subject として振る舞う．また、ドメイン毎にセキュリティポリシーを記述できる．あるドメインから派生したり起動されたプロセスはそのドメインのポリシーを継承するが、必要に応じて権限の変更がおこなわれる（Type Enforcement）．SELinux はプロセスのグルーピングにドメインによるモデル化を用いているので我々の研究と類似するが、異なるのはグルーピングを行うレベルの違いである．我々は OS レベルでグルーピングを行い、セキュリティモデルの切り替えに利用するのに対し、SELinux では SVO モデル内で使えるグルーピングを提供している．我々のグルーピングは他の、たとえばモニタリングなどの機能も取り込めるが、SELinux では SELinux がサポートしている機能以外はドメインに対して使うことはできない．（モニタリングは最初からサポートしているが）セキュリティモデル中のモデルであり、我々はセキュリティモデルの切り替えのグループとしてプロセスグルーピングを用いた．

表 6 KOA 化した TOMOYO
Table 6 TOMOYO with KOA

システムコール	修正なし	KOA	オーバヘッド
open	3.96	4.01	+1.1%
write	0.39	0.41	+5.1%
read	0.43	0.45	+4.5%
stat	1.68	1.76	+4.8%
p			

表 7 KOA 化した Smack
Table 7 Smack with KOA

システムコール	修正なし	KOA	オーバヘッド
open	2.70	2.85	+5.6%
write	0.39	0.41	+5.1%
read	0.42	0.45	+7.1%
stat	1.80	1.91	+6.1%

WhiteList[9] は LSM と `cgroups` を対応させてグループごとのデバイスアクセスの可否を設定できる LSM である．アクセス制御とプロセスグルーピングを対応させている点で我々と同じだが、異なる点として WhiteList では LSM を WhiteList が独占してしまい、既存の LSM をグループの上で用いることはできない．我々のセキュリティキューブでは LSM としてキューブが入るので、キューブ上で他の LSM を動かすことが可能になる点で異なる．

セキュリティモデル記述に関する研究として [11] などがあげられる．セキュリティモデルの複雑さに、時間軸を区切ってポリシーの記述量を減らすことで取り組んでいる．我々はアクセス制御モデルではなく、アクセス制御モデルをコントロールできるレイヤーを追加している為、このような研究をそのまま用いることができると考えられる．

7. 結　　論

我々は cgroups 等によるプロセスグルーピングでセキュリティを分離できない問題を解決することを目標に、LSM フレームワークの問題点を指摘し、プロセスグループと LSM の対応付けを行うセキュリティキューブを提案、設計した．その際問題となるカーネルオブジェクトアクセスの抽象化を行う為に KOA を設計した．さらに Linux 上でプロトタイプ実装を行い、実行性能評価を行った．我々の実験ではセキュリティキューブのオーバヘッドは LSM に比べても、8%以内に収まった．しかし、既存の cgroups などプロセスグルーピングに対応させるためには、動的に構成されるプロセスグループに対し、LSM の構成を書き換えていく必要がある．また、セキュリティの面で、動的な書き換えが行われることでセキュリティが損なわれないかなどの検証も必要であると考えている．

謝辞　本研究は、JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた．

参 考 文 献

1) Kernel Based Virtual Machine, http://www.linux-kvm.org
2) N. S. Agency. Security-enhansed linux. "http://www.nsa.gov/selinux/"
3) Boebert, W.E.and Kain, R.Y. "A practical alternative to hierarchical integrity policies," Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD, 1985
4) Tomoyo Linux "http://tomoyo.sourceforge.jp/"
5) Smack "http://www.schaufler-ca.com/"
6) Paul B. Menage, "Adding Generic Process Containers to the Linux Kernel," Proceedings of the Ottawa Linux Symposium, 2007.
7) M. Bishop and M. Digler, "Checking for Race Condition in File Accesses," Computing Systems, 9(2):131-159, Spring 1996.
8) Trent Jaeger, Reiner Sailer, and Xiaolan Zhang, "Analyzing integrity protection in the SELinux example policy," Proceedings of the USENIX Security Symposium, pages 59-74, August 2003
9) Serge E. Hallyn, "cgruops:Implement Device Whiltelist LSM," http://lwn.net/Articles/273208/, 2008
10) Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman,"Linux Security Modules: General Security Support for the Linux Kernel", 11th USENIX Security Symposium 2002.
11) Mahesh V. Tripunitara, Ninghui Li, "Comparing the Expressive Power of Access Contorl," CSS '04 October 25-29, 2004, Washington DC, USA.
12) Stephen Soltesz, et al, "Container-based Operating System Virtualization: A Scalable, Hight-performance Alternative to Hypervisors," EuroSys 2007, March 21-23, 2007 Lisboa.
13) LM Bench Project, "http://sourceforge.net/projects/lmbench"
14) Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, "Virtual Severs and Checkpoint/Restart in Mainstream Linux", ACM SIGOPS Operating System Review, Vol 42, Issue 5, July 2008.
15) Serge E. Hallyn, "Secure Linux containers cookbook", http://www.ibm.com/delveloperworks/linux/libary/l-lxc-security, Feb 2009.