システムソフトウェアと
オペレーティング・システム 90－12
（２００２．６．２７）

細粒度保護ドメインによる軽量サンドボックスの実現

品川 高廣 †　　河野 健二 ††,†††　　益田 隆司 ††

† 東京大学大学院理学系研究科情報科学専攻
†† 電気通信大学情報工学科，††† 科学技術振興事業団さきがけ研究 21
電子メール：shina@is.s.u-tokyo.ac.jp，{kono,masuda}@cs.uec.ac.jp

要旨

本稿では，細粒度保護ドメインを利用して軽量なサンドボックスを実現する手法について述べる．従来の外部プロセスで実現したサンドボックスでは，プロセス間通信のコストが大きいため，軽量なサンドボックスを実現することが難しかった．本稿で述べる手法では，プロセスの代わりに細粒度保護ドメインを用いることでプロセス間通信のコストを削減し，軽量なサンドボックスを実現する．実験により，この手法によるサンドボックスのオーバーヘッドは，外部プロセスで実現した場合と比べて，約 5 分の 1 程度に削減できていることを確認した．

A Lightweight Sandbox based on the Fine-grained Protection Domain

Takahiro Shinagawa†　　Kenji Kono††,†††　　Takashi Masuda††

†Department of Information Science, Graduate School of Science, University of Tokyo
††Department of Computer Science, University of Electro-Communications,
†††PREST, Japan Science Technology Corporation
E-mail: shina@is.s.u-tokyo.ac.jp, {kono,masuda}@cs.uec.ac.jp

Abstract

This paper shows a technique that realize a lightweight sandbox based on the fine-grained protection domain. Usual sandbox based on the process is difficult to be lightweight because of the heavy cost of inter-process communication. This paper shows a design of a lightweight sandbox that eliminates the cost of inter-process communication by using fine-grained protection domains instead of processes. Experimental results show that the lightweight sandbox is one-fifth the overhead of usual one.

1 はじめに

インターネットにアクセスする計算機のセキュリティを向上させる手法として，**サンドボックス**という技術が一般的に用いられている．サンドボックスとは，信頼のできないプログラムによる不正アクセスを防止するために，アクセスできる計算機資源を一定の範囲内に制限した環境下でプログラムを動作させる技術の総称である．たとえば Java アプレットでは，Java 仮想機械を用いてファイルやネットワークなどへのアクセスを非常に制限したサンドボックス内で Java アプレットのプログラムを動作させる．また最近では，アプリケーションのバグを突いて制御を乗っ取るなどの不正アクセスに対処するために，インターネットからダウンロードしたコンテンツを扱う一般のアプリケーションをサンドボックスの中で動作させる技術が数多く提案されている [1, 2, 3, 5, 9, 12, 16, 20].

一般のアプリケーションに対してサンドボックスを行う手法としては，外部プロセスで監視する方式 [1, 2, 9, 16] がある．この方式では，多くのオペレーティングシステムで用意されているシステムコールトレース機能を利用して，サンドボックスを行うアプリケーションが発行するシステムコールを別プロセスで監視することでアクセス制御を行う．この方式ではアクセス制御をシステムコールレベルで行なうため，アプリケーションからは透過的にサンドボックスを実現できる．また，アクセス制御をユーザレベルのコードで行うので，必要に応じて様々なセキュリティポリシーを実現できる．しかし，システムコールの監視を監視対象のアプリケーションとは別のプロセスで行うので，システムコールを発行するたびにプロセス間通信が必要になり，オーバヘッドが大きくなる．

本稿では，我々が提案している**細粒度保護ドメイン** [17, 21] の機構を利用して，軽量なサンドボックスを実現する手法を示す．本稿で述べる手法では，外部プロセスで監視する方式と同様に，監視対象のアプリケーションが発行するシステムコールを監視することでサンドボックスを実現する．ただしサンドボックスを軽量に実現するために，監視を行なうコード（**監視モジュール**と呼ぶ）を監視対象のアプリケーションと同じプロセスに入れて，監視モジュールの保護を細粒度保護ドメインで行う．細粒度保護ドメインとは，一つのプロセス内に複数個持つことができる保護ドメインで，保護ドメイン間のメモリ保護と，保護ドメインで発行されたシステムコールを予め登録した保護ドメイン内のモジュールでフックする機能を持っている．細粒度保護ドメイン間の切り替えは非常に軽量に実現されており，監視モジュールでシステムコールを監視するコストを低く抑えることができる．

また本稿では，細粒度保護ドメインで実現したサンドボックスの性能を測定した実験の結果を示す．１回のシステムコールをフックするために必要なコストは，外部プロセスで実現したサンドボックスと比べて，約 12 分の１に抑えることができた．また，一般のアプリケーションとして ghostscript にサンドボックスを行って実行時間を測定する実験では，外部プロセスで実現した場合と比べてオーバーヘッドをおよそ 5 分の１に抑えることができた．

以下2章では，サンドボックスの要求仕様とその実現方針について述べる．3章では，サンドボックスの実現に利用する細粒度保護ドメインの概要について説明し，4章で細粒度保護ドメインを用いたサンドボックスの実現手法について説明する．5章では，サンドボックスのオーバーヘッドを測定した実験の結果を示す．6章で関連研究に触れ，7章 で本稿をまとめる．

2 目標と方針

本章では，一般のアプリケーションに対してサンドボックスを行う機構が満たすべき要件について説明し，本稿で述べる手法でその要件を満たすためのアプローチについて述べる．本章では，サンドボックスが満たすべき要件として，以下の 3 つの点について述べる．

- 軽量なサンドボックス機構
- 変更可能なポリシー機構
- 既存のプログラムへの透過的な適用

2.1 軽量なサンドボックス機構

サンドボックスを実現する機構は，アプリケーション本来の動作速度をできるだけ損なわないことが望ましい．そのためには，保護のオーバーヘッドを抑えて軽量なサンドボックスを実現する必要がある．

従来の外部プロセスで実現したサンドボックスでは，システムコールの監視に伴うオーバーヘッドが比較的大きくなってしまう．これは，システムコールの監視を行うプログラムを保護するために，監視対象のアプリケーションとは別のプロセスで監視を行うので，監視対象のアプリケーションが発行したシステムコールをフックするためにプロセス間通信が必要となるためである．プロセス間通信は比較的重い処理なので，サンドボックスを軽量に実現することは難しい．

本稿で述べる方式では，プロセス間通信のコストを削減するために，プロセスの代わりに細粒度保護ドメインを用いて軽量なサンドボックスを実現する．細粒度保護ドメインは一つのプロセスの中で複数個持つことができる保護ドメインで，従来のプロセスの概念から保護ドメインの概念だけを分離したものである．細粒度保護ドメイン間の通信はプロセス間に比べて軽量に実現されており，監視を行うプログラムを保護しつつ，システムコールをフックするための通信コストを低く抑えることができる．細粒度保護ドメインについては，3章で説明する．

2.2 変更可能なポリシー機構

アプリケーションの種類やユーザの方針に柔軟に対応できるようにするために，サンドボックスでは必要に応じて様々なセキュリティポリシーを実現できることが望ましい．セキュリティポリシーはサンドボックス内でアクセスできる資源を定めるもので，アプリケーションの動作に必要な資源はアクセスを許可しつつ，保護すべき資源を保護できるように設定する必要がある．セキュリティポリシーは，それを実施する部分であるポリシー機構と密接に関連しており，単一のポリシー機構の実装で様々なセキュリティポリシーを実現することは困難である．

本稿で述べる方式では，様々なセキュリティポリシーを実現できるようにするために，セキュリティポリシーを実現するポリシー機構をサンドボックスの機構からは独立したモジュールとして実装する．ポリシー機構を分離することによって，サンドボックスの機構を修正することなくセキュリティポリシーを変更できるようにする．また必要に応じてセキュリティポリシーを変更できるようにするために，ポリシー機構を実行時に選択して動的に組み込めるようにする．

2.3 既存のプログラムへの透過的な適用

サンドボックスの機構は，アプリケーションに対して修正を施さずにサンドボックスを行えるようにするために，アプリケーションからは透過的に行えることが望ましい．アプリケーションからは透過的に実現することで，すでに数多く存在する既存のアプリケーションをそのまま再利用することができる．

本稿で述べる方式では，アプリケーションから透過的にサンドボックスを実現するために，プログラムローダを改造してサンドボックスを実現する機構を組み込む．プログラムローダを使用することで，アプリケーション自体のバイナリには変更を加えることなくサンドボックスを構成することができ，アプリケーションに制御が移ると同時にサンドボックス内で動作させることができる．

3 細粒度保護ドメイン

本章では，まず軽量なサンドボックスの実現に利用する細粒度保護ドメインについて説明する．なお，細粒度保護ドメインの詳細については，文献 [17, 21] を参照されたい．

3.1 細粒度保護ドメインの概要

細粒度保護ドメインとは，一つのプロセスの中に複数個持つことができる保護ドメインである．保護ドメインはアクセス可能な資源の集合を表す概念で，動作中のプログラムのアクセス権限を定義するものである．たとえば従来のオペレーティングシステムではプロセスが保護ドメインの役目を果たしており，アクセス可能なメモリ領域やファイルなどを決めているほか，プロセス間の保護の役割を果たしている．細粒度保護ドメインは，従来のプロセスから保護ドメインの概念を分離して複数個持てるようにしたもので，一つのプロセスの中に複数のアクセス権限を持たせることができる．それぞれの細粒度保護ドメインは，プロセスによる保護ドメインの部分集合となっている．つまり，細粒度保護ドメインでアクセス可能な資源は，プロセスでアクセス可能な資源の一部を制限したものとなっている．

細粒度保護ドメインは，一つのプロセスを構成する複数のモジュール（プログラムの構成要素）に対して，お互いを保護しつつも効率のよい連携を実現

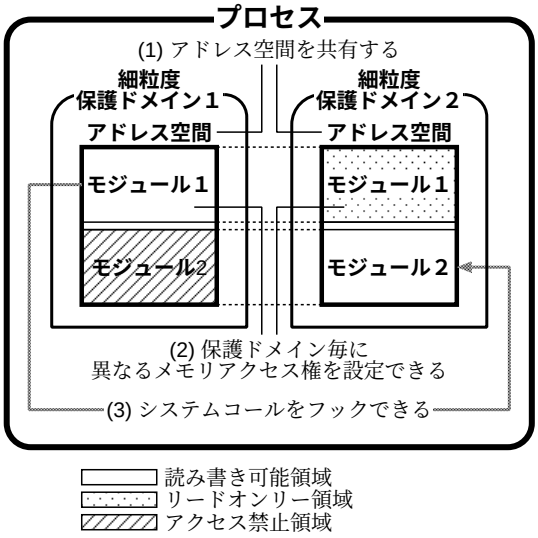

図 1: 細粒度保護ドメイン

することを目指している．すなわち，それぞれのモジュールに対して細粒度保護ドメインを割り当てることで，一つのプロセスの中でモジュールごとに異なるアクセス権限を持たせ，モジュール間の保護を実現する．一方で，モジュール間の連携を効率よくおこなうために，細粒度保護ドメインを割り当てられたモジュールは，アクセス権限以外の実行環境はプロセスで定義されるものを共有する．たとえばメモリ上のデータ共有を効率よくおこなうために，プロセスの仮想アドレス空間を共有して，ポインタ変換などを不要にする．また，細粒度保護ドメイン間の手続き呼び出しを高速におこなうために，タイムスライスやページテーブルなどプロセス単位で割り当てられた資源を共有して，プロセス切替えに伴うスケジューリングや TLB フラッシュなどのコストを削減する．これによって，細粒度保護ドメイン切替えのコストを，アクセス権の切替えに必要な最小限度に抑えている．

3.2 細粒度保護ドメインの機能

細粒度保護ドメインを実現する機構は，以下の三つの機能を持っている．

- ページ単位のメモリ保護
- システムコールのフック機能
- 専用の保護ドメイン切替え命令

以下ではそれぞれの機能について説明する．

3.2.1 ページ単位のメモリ保護

一つのプロセス内のそれぞれの細粒度保護ドメインは，アドレス空間を共有しつつも，メモリのアクセス権をページ単位で細粒度保護ドメインごとに異なる設定にすることができる．たとえば図1には細粒度保護ドメイン１と細粒度保護ドメイン２の二つの細粒度保護ドメインがあるが，(1) に示すように同じアドレス空間を共有している．一方で，このアドレス空間内のモジュール１のメモリ領域は，(2) で示すように細粒度保護ドメイン１では読み書き可能なのに対して，細粒度保護ドメイン２ではリードオンリーとなっている．このようにアドレス空間を共有することでポインタを含むようなデータの共有を容易にしながら，メモリ保護を行うことができる．

それぞれの細粒度保護ドメインのメモリアクセス権の設定は，`mmap()` や `mprotect()` を拡張したシステムコールでおこなう．これらのシステムコールでは，アクセス権を設定するメモリ領域と新しいアクセス権に加えて，細粒度保護ドメイン識別するための ID を指定することができる．勝手に保護すべきメモリ領域のアクセス権が変更されないようにするために，次節で説明するシステムコールフック機能でチェックする．

3.2.2 システムコールのフック機能

ファイルやネットワークなどシステムコールでアクセスする資源に対して，細粒度保護ドメインごとに異なるアクセス権を持たせるために，細粒度保護ドメイン内で発行されたシステムコールをフックすることができる．たとえば図1では (3) で示すように細粒度保護ドメイン１で発行されたシステムコールを細粒度保護ドメイン２のモジュール２でフックしている．一つのプロセス内では一つの細粒度保護ドメインだけがシステムコールをフックする特権を持っており，システムコールが発行されたときに呼び出されるコールバックルーチンをカーネルに登録することができる．コールバックルーチンには，システムコールが発行された細粒度保護ドメインの ID，システムコールの種類およびその引数が渡される．コールバックルーチンはそれらの内容をチェックして，そのシステムコールの発行の可否を決定する．

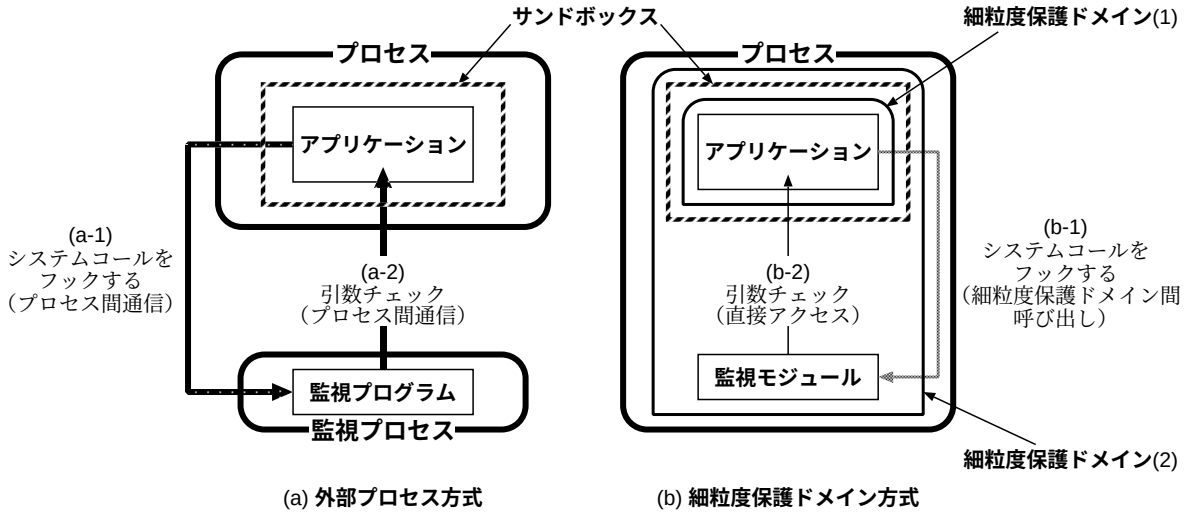

図 2: サンドボックス実現方式の比較

3.2.3 専用の保護ドメイン切替え命令

高速な保護ドメイン切替えを実現するために，細粒度保護ドメイン切替え専用のソフトウェアトラップ命令を用意している．不正な領域を呼び出すことを防止するために，細粒度保護ドメインごとに保護ドメイン間呼び出しのエントリポイントをあらかじめ登録しておく．異なる細粒度保護ドメインの手続きを保護ドメイン間呼び出しで呼び出すときは，呼び出し先の細粒度保護ドメインの ID と手続きごとに割り当てられた番号を指定して，この保護ドメイン切替え命令を発行する．

4 軽量サンドボックスの実現

本章では，細粒度保護ドメインを利用して軽量なサンドボックスを実現する手法を説明する．

4.1 軽量なサンドボックス機構

従来の図2 (a) のような外部プロセスで実現したサンドボックスでは，以下の２つの理由からシステムコール監視のオーバヘッドが大きくなる．まず第一に，図2 (a-1) のようにアプリケーションが発行するシステムコールをフックするためにプロセス間通信が必要になる．第二に，図2 (a-2) のようにシステムコールの引数をチェックするために，監視対象のアプリケーションのメモリデータをプロセス間通信で取得する必要がある．

本稿で述べる手法では，プロセス間通信によるオーバーヘッド軽減するために，図2 (b) のように監視を行うコード（監視モジュール）を監視対象のアプリケーションと同じプロセスに入れる．これによってアプリケーションと監視モジュールの間のプロセス間通信を不要にする．監視モジュールを保護するために，アプリケーションと監視モジュールのそれぞれに対して一つの細粒度保護ドメインを割り当て，アプリケーションから監視モジュールのメモリ領域にアクセスできないようにする．アプリケーションが発行するシステムコールを監視するために，細粒度保護ドメインの機能を使用して図2 (b-1) のようにアプリケーションが発行したシステムコールを監視モジュールでフックする．また引数チェックを効率よく行うために，図2 (b-2) のように監視モジュールがアプリケーションのメモリ領域に直接アクセスできるように細粒度保護ドメインのアクセス権限を設定する．

このように細粒度保護ドメインを利用することで，外部プロセスで実現したサンドボックスと同等の機能を，外部プロセスより軽量に実現することができる．

4.2 変更可能なポリシー機構

サンドボックスのポリシー機構を容易に変更できるようにするために，ポリシーを実現する機構であ

る監視モジュールは，細粒度保護ドメインでサンドボックスを実現する機構とは独立した設計にする．監視モジュールのインターフェイスは，システムコールの内容をチェックしてその可否を返すだけのものとし，システムコールのフックを実現する機構とは分離する．監視モジュールの実装は，動的にロード可能な共有オブジェクトとして提供し，プログラムの起動時に指定できるようにする．

4.3 既存のプログラムへの透過的な適用

既存のプログラムからは透過的にサンドボックスを実現できるようにするために，細粒度保護ドメインの割り当てやアクセス権の設定をプログラムローダで行う．プログラムローダはアプリケーション本体の起動前に実行されるプログラムで，アプリケーション本体や動的リンクライブラリを読み込むなどの初期化作業をおこなう．プログラムローダで細粒度保護ドメインの割り当てやアクセス権の設定を行うことで，アプリケーション本体のバイナリをいっさい改変せずにサンドボックスを実現できる．

5 性能実験

本章では，細粒度保護ドメインで実現したサンドボックスのオーバヘッドを測定した実験の結果を示す．まずシステムコールをフックするためにかかるコストを測定する実験を行った．次に一般のアプリケーションでサンドボックスを行うことによるオーバーヘッドを測定する実験を行った．

実験に使用したマシンは，PentiumIII 1GHz，メモリ 128M バイト，ハードディスク 75GB (DTLA-307030) を搭載した PC である．使用したオペレーティングシステムは，Linux 2.2.18 を改造して細粒度保護ドメインの機構を組み込んだカーネルで動作する Vine Linux 2.1.5 である．

5.1 システムコールフック

まずシステムコール１回あたりにかかる保護のコストを計測した実験を行った．実験には `getpid()` システムコールを使用し，一回のシステムコール発行にかかる CPU サイクル数を計測した．計測は，サンドボックスが無い場合，細粒度保護ドメイン方式で実現したサンドボックスを使用した場合，外部プロセス方式で実現したサンドボックスを使用した場合の３通りで行った．測定には PentiumIII に搭載されているサイクルカウンタを利用した．表1 に結果を示す．

表 1: getpid() 一回あたりのコスト

方式	サイクル数	時間
サンドボックスなし	472	0.47μs
細粒度保護ドメイン	995	1.00μs
外部プロセス	6,602	6.60μs

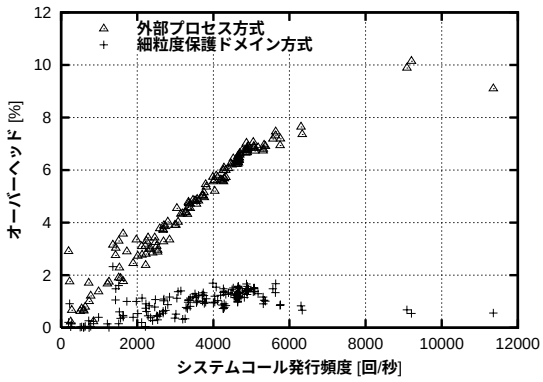

図 3: オーバヘッド (ghostscript)

サンドボックスを行わない通常の `getpid()` にかかる時間は 472 サイクルである．細粒度保護ドメイン方式でサンドボックスを行った場合は 995 サイクルで，サンドボックスのコストは 523 サイクルであった．外部プロセス方式でサンドボックスを行った場合は 6,602 サイクルで，サンドボックスのコストは 6,130 サイクルであった．細粒度保護ドメイン方式で実現したサンドボックスは，外部プロセスで実現したサンドボックスと比べて，システムコール１回あたりに必要なサンドボックスのコストを約 12 分の 1 に抑えられていることが分かる．

5.2 アプリケーション

次にアプリケーションとして Ghostscript を使用して，サンドボックスのオーバーヘッドを測定する実験を行った．Ghostscript に対してサンドボックスを行って様々な大きさの Postscript ファイルを解釈させ，全体の実行時間を計測した．サンドボックス

を行わなかった場合の実行時間も同様に測定して，実行時間の比からオーバーヘッドを求めた．比較のために，外部プロセスで実現したサンドボックスのオーバーヘッドも計測した．

本稿で述べたサンドボックスの実現方式では，システムコールが発行されるたびにフックのコストがかかるので，サンドボックスのオーバーヘッドはシステムコールの発行頻度に比例すると考えられる．そこで，Ghostscript の実行中に発行されたシステムコールの総数を測定して，全体の実行時間で除算して Postscript ファイルごとのシステムコールの発行頻度を求めた．

実験に使用した Postscript ファイルは，大きさが 112Byte から 175MB までの 158 個のファイルである．誤差を減らすために，実行はコマンドラインから非対話的に行ない，出力先は`/dev/null`として，ページの最初から最後まで一気に解釈させた．使用した Ghostscript のバージョンは 5.50 である．

実験結果を図3に示す．横軸は Postscript ファイルごとのシステムコール発行頻度で，縦軸はサンドボックスによる実行時間のオーバーヘッドである．外部プロセスで実現したサンドボックスは，システムコール発行頻度が 1 秒間に 5000 回の時でおよそ 7%であるのに対し，細粒度保護ドメインで実現したサンドボックスはおよそ 1.5%であった．したがって，オーバヘッドをおよそ５分の１に抑えられていることが分かる．

以上の結果から，細粒度保護ドメインで実現したサンドボックスは，外部プロセスで実現した場合と比べて軽量なサンドボックスが実現されていることが分かる．

6　関連研究

アプリケーションに対してサンドボックスを実現する手法としては，外部プロセスで監視する手法 (Janus [9], MAPbox [1], SecurePot [16] など) がある．この手法ではオペレーティングシステムの機能であるシステムコールトレース機能を利用して，サンドボックスを行うプログラムが発行するシステムコールを監視する．しかしシステムコールの監視を別プロセスで行うため，システムコールを発行するたびにプロセス切り替えが発生してオーバヘッドが大きくなる．

オペレーティングシステムのみでサンドボックスを実現する手法 [3, 5, 20] では，カーネルに新たなアクセス制御機構を組み込んでサンドボックスを実現する．オペレーティングシステムのみで実現したサンドボックスでは，セキュリティポリシーを実現する機構がカーネルに組み込まれるため，アプリケーションに合わせて柔軟に変更することが難しい．

プロセスの中のモジュールに対してサンドボックスを行う手法としては，言語処理系で実現する手法 (Java [18])，ソフトウェアで実現する手法 (SFI [19], PCC [15, 14])，カーネルで実現する手法 (Palladium [7], PSL [4]) など様々なものがある．これらの手法ではアプリケーションそのものをサンドボックスすることは想定しておらず，既存のアプリケーションを改変せずにサンドボックスを実現することは難しい．

従来のプロセス間通信を高速化する仕組みとしては，さまざまな手法が提案されている (LRPC [6], Spring [10], Mach [8], L4 [11, 13])．しかしこれらの手法では，サンドボックスを効率よく実現するための手法については触れられていない．

7　まとめ

本稿では，細粒度保護ドメインを利用して一般のアプリケーションに対して軽量なサンドボックスを実現する手法について述べた．軽量なサンドボックス機構を実現するために，システムコールの監視を外部プロセスで行う代わりに，細粒度保護ドメインで保護された同じプロセス内の監視モジュールで行って，プロセス間通信のコストを削減し，効率の良いシステムコール監視機構を実現した．必要に応じて様々なポリシーを実現できるように，セキュリティポリシーを実現する機構をサンドボックスの機構からは独立させて動的に読み込めるようにした．アプリケーションを改変せずにサンドボックスを行うために，プログラムローダを改変してサンドボックスの機能を実現するようにした．

実験によって，実際に軽量なサンドボックスの機構を実現できていることを実証した．１回のシステムコールをフックするために必要なコストは，外部プロセスで実現したサンドボックスと比べて，約 12 分の１に抑えることができた．また，ghostscript にサンドボックスを行った実験では，外部プロセスで実現した場合と比べてオーバーヘッドをおよそ 5 分の１に抑えることができた．

参考文献

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proc. of the 9th USENIX Security Symposium*, Aug. 2000.

[2] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: Confined execution environment for internet computations. Available at http://www.cs.ucsb.edu/ berto/papers/99-usenix-consh.ps, 1998.

[3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A domain and type enforcement UNIX prototype. In *Proc. of the 5th USENIX UNIX Security Symposium*, June 1995.

[4] A. Banerji, J. M. Tracey, and D. L. Cohn. Protected Shared Libraries – A New Approach to Modularity and Sharing. In *Proc. of the1997 USENIX Annual Technical Conference*, pp. 59–75, Oct. 1997.

[5] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):36–61, 2002.

[6] B. N. Bershad, T. E. Anderson, E. D. Lanzowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Comuputer Systems*, 8(1):37–55, February 1990.

[7] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proc. of the 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), pp. 140–153, Dec. 1999.

[8] B. Ford and J. Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proc. of theUSENIX Winter 1994 Technical Conference*, Jan. 1994.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Enviroment for Untrusted Helper Applications. In *Proc. of the 6th USENIX Security Symposium*, July 1996.

[10] G. Hamilton, M. L. Powell, and J. G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pp. 69–79, Dec. 1993.

[11] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), pp. 66–77, Oct. 1997.

[12] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE)*, 2000.

[13] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance. In *Proc. of the 6th Workshop on Hot Topics in Operating Systems* (*HOTOS '97*), pp. 28–31, May 1997.

[14] G. C. Necula. Proof-carrying code. In *Proc. of the 24th ACM Symposium on Principles of Programing Languages* (*POPL '97*), pp. 106–119, Jan. 1997.

[15] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pp. 229–243, Oct. 1996.

[16] Y. Oyama and K. Kato. SecurePot: Secure Software Execution System Based on System Call Hooks. In *18th JSSST Conference*, Sept. 2001.

[17] M. Takahashi, K. Kono, and T. Masuda. Efficient kernel support of fine-grained protection domains for mobile code. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 64–73, May 1999.

[18] J. Team, J. Gosling, B. Joy, and G. Steele. *The Java[tm] Language Specification*. Addison Wesley Longman, 1996. ISBN 0-201-6345-1.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pp. 203–216, Dec. 1993.

[20] 榮樂, 新城, 板野. システム・コールに対するラッパ／リファレンス・モニタ sysguard の設計と実現. 情報処理学会論文誌, 43(6), June 2002. 掲載予定.

[21] 品川, 河野, 高橋, 益田. 拡張コンポーネントのためのカーネルによる細粒度軽量保護ドメインの実現. 情報処理学会論文誌, 40(6):2596–2606, June 1999.