実行可能コンテンツの安全な実行環境

品川高廣[†] 河野健二[††,†††] 益田隆司[††]

プログラムコードを含んだコンテンツである実行可能コンテンツと呼ばれる形態が普及するにつれて，不正アクセスの被害が増加している．実行可能コンテンツは保護機構の実装の不備を攻撃して不正アクセスを行うので，これを完全に防ぐことは難しい．本論文では，実行可能コンテンツの不正アクセスに対処するための保護機構のモデルとして，階層保護モデルを導入する．階層保護モデルでは，不正アクセスの被害を最小限に抑えることを目的として，実行可能コンテンツのアクセス制御を行う1次保護ドメインと，実行可能コンテンツを処理するアプリケーション自体のアクセス制御を行う2次保護ドメインの2つの保護ドメインを組み合わせることで安全性を高める．本論文では，我々が提案してきた細粒度保護ドメインの機構を応用して，階層保護モデルを効率良く実現する手法を示す．

A Secure Execution Environment for Executable Contents

Takahiro Shinagawa,[†] Kenji Kono[††,†††] and Takashi Masuda[††]

Executable contents, containing program code, increasingly pose the threat of unauthorized access. Protecting against unauthorized access is difficult because of the inevitable flaws in implementaion of protection mechanisms. This paper introduces a hierarchical protection model to provide a fail-safe mechanism. A hierarchical protection model improves reliability of protection mechanisms by nesting two protection domains: a level-1 protection domain to control access from executable contents and a level-2 protection domain to control access of application processing the executable contents. This paper shows an efficient implementation of a hierarchical protection model which incorporates fine-grained protection domains proposed in our privious paper.

1. はじめに

WWW やメールなどで配布されるコンテンツとして，実行可能コンテンツと呼ばれる形態が一般的になっている．実行可能コンテンツとは，プログラム・コードを含んだコンテンツのことで，ユーザが閲覧する際にプログラムが自動的に実行され，内容が動的に生成されるコンテンツである．実行可能コンテンツを処理するアプリケーションは，プログラムをユーザの計算機上で実行させることによって，表示内容を動的に変化させたり，ユーザとの対話的な処理を行ったりすることができる．実行可能コンテンツの例には，ActiveX や Java applet，JavaScript，VBScript，PostScript，PDF などがあげられる．

実行可能コンテンツの実行環境が普及するにつれて，不正アクセスの被害が深刻な問題になっている．実行可能コンテンツはユーザの計算機で直接実行されるため，外部から侵入する場合に比べて比較的容易に不正アクセスが行われてしまう．また，実行可能コンテンツはインターネットを通じて配布されるため，ウィルスやワームなどのように不正アクセスを行う実行可能コンテンツが容易に配布されて被害が広範囲に及んでしまう．実行可能コンテンツによる不正アクセスは近年多く報告されており，その数も増加傾向にある[1)~6)]．

実行可能コンテンツによる不正アクセスを完全に防ぐことは難しい．不正アクセスを防止するためには実行可能コンテンツに対して細かい粒度でアクセス制御を行う必要があるが，アクセス制御を細かい粒度で行うと保護機構が複雑になって正しく実装することが困難になる．実行可能コンテンツは，保護機構の実装における脆弱性を利用して間接的に不正アクセスを行うことが可能であり，実際に実行可能コンテンツを処理するアプリケーションに対して，保護機構の実装の不

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Faculty of Science, University of Tokyo
†† 電気通信大学情報工学科
Department of Computer Science, University of Electro-Communications
††† 科学技術振興事業団さきがけ研究 21
PREST, Japan Science Technology Corporation

備を利用した不正アクセスが多く報告されている．

本論文では，実行可能コンテンツの不正アクセスに対処するための保護機構のモデルとして，**階層保護モデル**を導入する．この保護モデルの目的は，仮に不正アクセスが行われても，その被害を最小限に抑えられるフェイルセーフを実現することである．階層保護モデルでは，実行可能コンテンツによる不正アクセスを直接的に防止するために，実行可能コンテンツに対して**1 次保護ドメイン**と呼ぶ保護ドメインを割り当てる．これによって，実行可能コンテンツのアクセス制御を細かい粒度で行う．さらに，実行可能コンテンツを処理するアプリケーションを攻撃するという間接的な不正アクセスに対処するために，アプリケーション自身に対して，より堅牢性の高い**2 次保護ドメイン**を割り当てる．これによって，アプリケーションとは関係のない資源を保護し，不正アクセスの被害を最小限に抑えることができる．

このモデルの特徴は，保護の粒度と堅牢性がそれぞれ異なる保護ドメインを組み合わせることで，保護機構の信頼性を高めている点である．保護の粒度と堅牢性はトレードオフの関係にあるので，単一の保護機構で粒度が細かく堅牢性も高い保護を実現するのは困難である．階層保護モデルでは，堅牢性は低いが細かい粒度のアクセス制御を行える 1 次保護ドメインと，保護の粒度は粗いが実装が単純で堅牢性の高い 2 次保護ドメインを組み合わせることで，細粒度のアクセス制御と堅牢性の高い保護を両立して安全性を高めることができる．

階層保護モデルの実現には，我々が提案してきた**細粒度保護ドメイン**[7),8)] の機構を利用する．細粒度保護ドメインとは，1 つのプロセス内に複数の保護ドメインを共存させる技術であり，この共存する各保護ドメインのことを細粒度保護ドメインと呼ぶ．細粒度保護ドメインを用いると，プロセス内の各モジュールごとに異なるアクセス権限を持たせることができ，従来のオペレーティングシステムに比べ，より粒度の細かいアクセス制御を行うことができる．文献 7)，8) では，細粒度保護ドメインの実現方式を報告するだけにとどまっており，実行可能コンテンツによる不正アクセスを防止する手法については述べられていない．

本論文では，細粒度保護ドメインを利用して，階層保護モデルを実際のアプリケーションに適用する例について述べる．例として Web ブラウザと電子文書ビュアーを使用して，既存のアプリケーションを改変せずに保護機構を組み込む．Web ブラウザにおける実装では，プラグインを利用してバイナリ型の実行可能コンテンツに 1 次保護ドメインを割り当てる手法を説明する．また，電子文書ビュアーにおける実装では，プログラムローダを利用してアプリケーションを改変せずに 2 次保護ドメインを割り当てる手法を説明する．

以下，2 章では実行可能コンテンツによる不正アクセスの脅威について説明し，実行可能コンテンツの形式別に具体例を示す．3 章では階層保護モデルのデザインを説明し，4 章で細粒度保護ドメインを利用した階層保護モデルの実現手法を説明する．5 章では階層保護モデルを実際に実行可能コンテンツに適用した例について述べ，6 章で細粒度保護ドメインを利用した階層保護モデルの評価実験の結果を示す．7 章で関連研究を述べ，8 章で本論文をまとめる．

2. 実行可能コンテンツ

実行可能コンテンツとは，プログラム・コードやその一部を含むコンテンツのことである．コンテンツとはインターネットでやりとりされる情報の中身のことで，Web ブラウザやメールリーダなどのアプリケーションで表示するなどの処理が行われるものである．従来の HTML 形式などのコンテンツでは，その内容は静的で作成された時点で決まっているのに対して，実行可能コンテンツは，処理する際にプログラムが自動的に実行されて，その内容が動的に生成される．プログラムをユーザの計算機上で実行することで，表示するたびに内容を変化させたり，ユーザとの対話的な処理を行ったりすることができる．なお本論文中で述べる実行可能コンテンツは，プログラムがサーバ側ではなくクライアント側，つまり受け取り側の計算機上で実行されるものを指している．

本章では，まず実行可能コンテンツによる不正アクセスが脅威となっている背景について説明する．次に実行可能コンテンツをその形式によって 2 種類に分類し，不正アクセスが行われるメカニズムについてそれぞれ具体的に説明する．

2.1 実行可能コンテンツの脅威

実行可能コンテンツによる不正アクセスが脅威となる背景には，まず実行可能コンテンツの作者の匿名性が高いという点がある．実行可能コンテンツはインターネットを通じて配布されるため，その作者を特定することは困難である．悪意を持った作者が，ユーザの計算機に不正アクセスを行うような実行可能コンテンツを作成して，身元を知られることなく密かに配布することが比較的容易である．

また，実行可能コンテンツは，不正アクセスの手段として利用しやすい側面を持っている．まず第 1 に，実

行可能コンテンツは不正アクセス自体が技術的に容易である．実行可能コンテンツはユーザの計算機上で直接実行されるため，わざわざ外部から侵入しなくても，不正アクセスを行うプログラムを動作させられる．第2に実行可能コンテンツは実行される機会が多い．実行可能コンテンツはユーザがコンテンツを閲覧するだけでプログラムが実行されるので，インストールなどの人手を介する作業を必要とせず，多くのコンピュータ上で実行される．第3に実行可能コンテンツは拡散が容易である．インターネットを利用することで，広範囲の計算機で不正アクセスを行う機会を得ることができる．インターネットに接続されるコンピュータの数はますます増加しており，それに比例して被害の規模も大きくなっている．

実行可能コンテンツの不正アクセスが増加しているにもかかわらず，不正アクセスを完全に防止する機構を実現することは難しい．実行可能コンテンツに対して適切なアクセス制御を行ったとしても，実行可能コンテンツは保護機構の実装におけるバグなどを利用することによって，許可されていない資源へ間接的にアクセスすることが可能である．たとえば，最も多く利用されている攻撃方法として，バッファ溢れ攻撃（buffer overflow attacks）[9] によってアプリケーションの制御を乗っ取る攻撃がある．保護機構は複雑な実装を必要とするために完全に正しく実装することは困難であり，実行可能コンテンツによる不正アクセスが繰り返されているのが現状である．

2.2 実行可能コンテンツの分類

実行可能コンテンツには，2つの形式がある．本論文中では，それぞれ**バイナリ型**および**インタプリタ型**の実行可能コンテンツと呼ぶ．ここでは，それぞれの形式について，その特徴と不正アクセスが行われるメカニズムについて具体的に説明する．

2.2.1 バイナリ型

バイナリ型の実行可能コンテンツとは，プログラムコードがユーザの計算機のCPUで直接実行できる形式で格納されているコンテンツのことである．プログラムコードはアプリケーションにリンクされて実行され，アプリケーションと同じプロセス内で動作する．バイナリ型の実行可能コンテンツは，システムコールを直接発行したり，アプリケーションのAPIを呼び出したりするなどして，資源アクセスを行う．

バイナリ型の実行可能コンテンツでは，基本的にアクセス制御を十分に行うことができない．プログラムコードがアプリケーションと同じプロセス内で動作するため，従来のオペレーティングシステムでは，そのコードはアプリケーションと同じアクセス権限で実行されてしまう．バイナリ型の実行可能コンテンツは，アプリケーションのメモリ領域にアクセスして情報を盗んだり，直接システムコールを呼び出してユーザのファイルを破壊したりするなどの不正アクセスを自由に行うことができてしまう．

バイナリ型の実行可能コンテンツの例には，ActiveXコントロールがある．ActiveXではデジタル署名や配布元ホストなどに基づいて，コンテンツの信頼性を保証することで安全性を確保している．しかしアプリケーションの実装の不備を利用して，信頼できないActiveXを実行させる攻撃が報告されている[3]．

2.2.2 インタプリタ型

インタプリタ型の実行可能コンテンツとは，アプリケーションに内蔵されたインタプリタによって解釈実行する形式のコンテンツのことである．プログラムコードは特定の言語で記述されており，その言語処理系であるインタプリタが逐次解釈実行することでコンテンツを生成する．インタプリタ型の実行可能コンテンツの例としては，JavaScriptやVBScriptなどのほか，PostScriptやPDFなどの電子文書として用いられているコンテンツも言語としての性質を持っており，インタプリタ型の実行可能コンテンツと見なすことができる．さらに，Java appletも仮想機械によって解釈実行されるという点で，インタプリタ型に分類することができる．

インタプリタ型の実行可能コンテンツは，バイナリ型に比べると不正アクセスは比較的困難である．インタプリタ型では言語レベルで型安全性が保証されていることが多く，不正なメモリ領域へのアクセスはできない．また計算機資源へのアクセスも，インタプリタによってアクセス制御が行われるため，実行可能コンテンツが直接的に不正アクセスを行うことはできない．

しかし，実行可能コンテンツは，インタプリタの実装上の不備を利用することで不正アクセスが可能である．実行可能コンテンツは，アクセス制御が正しく実装されていない部分を攻撃したり，アプリケーションが予期していないような入力を与えたりする攻撃などで不正アクセスを行うことができる．たとえばPostScriptの処理系であるghostscriptでは，ファイルアクセスができてしまう不備が報告されている[5]．また，PDFの処理系であるAcrobatReaderでは，バッファ溢れ攻撃に対する脆弱性が報告されている[6]．

3. 階層保護モデル

実行可能コンテンツの不正アクセスを防止するため

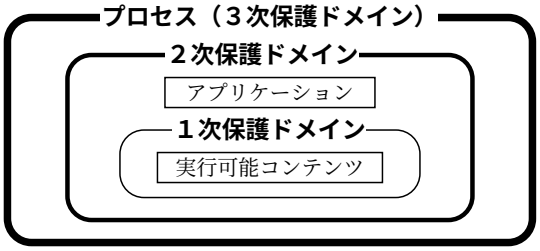

図 1 階層保護モデル
Fig. 1 A hierarchical protection model.

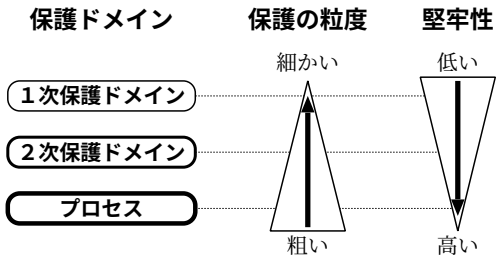

図 2 保護の粒度と堅牢性のトレードオフ
Fig. 2 A trade-off between granularity and reliability.

の保護モデルとして，**図 1** のような階層保護モデルを導入する．階層保護モデルでは，保護機構の信頼性を高めるために複数の保護ドメインを階層的に構成する．図 1 から分かるように，内側の保護ドメインは外側の保護ドメインの一部になっており，内側の方が外側よりアクセス権が制限された構成になっている．最も内側で実行可能コンテンツに対して直接割り当てる保護ドメインを **1 次保護ドメイン**，実行可能コンテンツを含むアプリケーション全体に割り当てる保護ドメインを **2 次保護ドメイン**と呼ぶ．

1 次保護ドメインの役割は，実行可能コンテンツの不正アクセスを未然に防ぐことである．最小特権の原則[10] に基づいて，実行可能コンテンツがコンテンツを生成するのに最低限必要な資源のみアクセスを許可する．これを実現するために，1 次保護ドメインでは実行可能コンテンツのアクセス制御を細かい単位で行って不正アクセスを防止する．これによって，アプリケーション自身やユーザの計算機資源を保護する．

2 次保護ドメインの役割は，1 次保護ドメインで不正アクセスを防ぎ切れなかったときの安全装置（フェイルセーフ）となることである．不正アクセスによる被害を最小限に抑えるために，2 次保護ドメインではアプリケーションの実行に必要のない資源にはアクセスできないようにする．また不正アクセスを確実に防止できるようにするために，2 次保護ドメインでは 1 次保護ドメインよりも堅牢な保護機構を使用する．これによって，仮に 1 次保護ドメインが攻撃されてアプリケーションの制御が乗っ取られたとしても，その被害を限定的なものにすることができる．

この保護モデルの特徴は，粒度の異なる複数の保護ドメインを併用することで，保護の粒度と堅牢性を両立している点にある．**図 2** に示すように，保護の粒度と堅牢性はトレードオフの関係にある．保護の粒度を細かくして柔軟なアクセス制御を行おうとすると，保護を実現する機構が複雑になって正しく実装することが難しくなり，保護の堅牢性が低下する．保護の堅牢性を高くするためには，保護機構をできるだけシンプルにして不具合が生じないようにする必要がある．そこで多少複雑でも細かい粒度で保護が行える 1 次保護ドメインで実行可能コンテンツの不正アクセスを防止しつつ，多少粒度が粗くても堅牢性の高い 2 次保護ドメインでさらに保護を行う．これによって，細粒度のアクセス制御と堅牢性の高い保護を同時に実現し，保護を二重にして安全性を高めている．

4. 保護の実現

階層保護モデルの実現には，我々が提案してきた細粒度保護ドメインの機構を利用する．細粒度保護ドメインとは，プロセス内に複数の保護ドメインを共存させる技術である．本章では，まず細粒度保護ドメインについて簡単に説明し，次に細粒度保護ドメインを利用して，1 次保護ドメインと 2 次保護ドメインを実現する手法についてそれぞれ説明する．細粒度保護ドメインの詳細については，文献 7), 8) を参照されたい．

4.1 細粒度保護ドメイン

細粒度保護ドメインとは，1 つのプロセス内に複数共存させることができる保護ドメインである．プロセス内の各モジュールに対してそれぞれ細粒度保護ドメインを割り当てることによって，モジュールごとに異なるアクセス権限を持たせることができる．モジュールのアクセス権限を制限することによって，モジュール間の保護や計算機資源の保護を実現できる．

細粒度保護ドメインのアクセス制御は，プロセスに割り当てられたメモリや，ファイル・ネットワークなどの計算機資源に対して，細かい粒度で行うことができる．メモリに対するアクセスは，細粒度保護ドメインごとにページ単位で異なる保護モードを設定することができる．ファイルやネットワークなどの計算機資源へのアクセスは，システムコールのレベルでアクセスの可否を制御できる．

それぞれの細粒度保護ドメインでどの資源のアクセスを許可するかといった保護ポリシーは，ユーザレベ

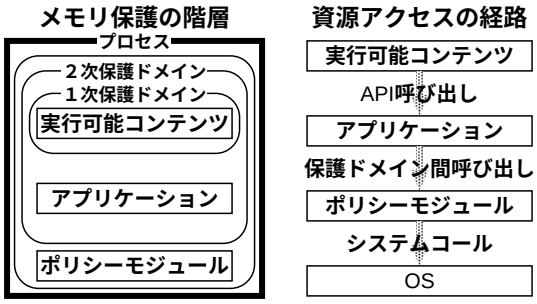

図 3 保護ドメインの構成
Fig. 3 Configuration of protection domains.

ルのポリシーモジュールと呼ばれるモジュールで実現する．ポリシーモジュールはカーネルが提供する機能を利用して，それぞれの細粒度保護ドメインのページ保護モードを設定する．また，システムコールをインターセプトして，その発行の可否を保護ポリシーに基づいて決定する．ポリシーモジュール自身も細粒度保護ドメインによって保護されている．

細粒度保護ドメインが割り当てられたモジュール間の連係は，プロセス間の連係と比べて効率良く行うことができる．細粒度保護ドメインは，プロセスに割り当てられた仮想アドレス空間を共有しているので，ポインタを含む複雑なデータ構造を容易にやりとりできる．また，ページテーブルやタイムスライスなどのプロセスの資源を共有することで TLB フラッシュやスケジューリングのコストをなくし，細粒度保護ドメイン間呼び出しを効率良く行うことができる．

4.2 1 次保護ドメイン

1 次保護ドメインを実現するためには，実行可能コンテンツに対して 1 つの細粒度保護ドメインを直接割り当てる．これによって，実行可能コンテンツによるメモリアクセスや，計算機資源へのアクセスを制御できるようになる．

1 次保護ドメインは実行可能コンテンツの不正アクセスを未然に防ぐためのものであるから，1 次保護ドメインにおけるアクセス権限は，最小特権の原則に従って必要最小限に制限するべきである．

そこで，1 次保護ドメインの保護ポリシーとしては，原則として以下のような方針を適用する．まず，**図 3** のメモリ保護の階層で示すように，実行可能コンテンツに割り当てられたメモリ領域以外へのアクセスを禁止する．また，ファイルなどの資源へのアクセスを制御するために，実行可能コンテンツからのシステムコールの発行は原則としてすべて禁止する．実行可能コンテンツの実行に必要な資源アクセスは，図 3 の資源アクセスの経路で示すように，アプリケーションが提供する安全な API を呼び出すように強制する．API を適切に設計することで，任意の粒度で保護を行うことができる．

4.3 2 次保護ドメイン

2 次保護ドメインを実現するためには，アプリケーション全体に対して細粒度保護ドメインを割り当てる．2 次保護ドメインの保護ポリシーは，アプリケーションから独立したポリシーモジュールで実装する．ポリシーモジュール自身を保護するために，ポリシーモジュールに対してアプリケーションとは別の細粒度保護ドメインを割り当てる．

アプリケーションはシステムコールによって計算機資源にアクセスするので，2 次保護ドメインにおけるアクセス制御はシステムコールのレベルで行う．アプリケーションが発行するシステムコールをチェックするために，図 3 の資源アクセスの経路で示すように，システムコールの発行は保護ドメイン間呼び出しによるポリシーモジュールの呼び出しに置き換える．ポリシーモジュールはシステムコールの内容をチェックして，アクセスの可否を決定する．

システムコールのチェックを容易にするために，ポリシーモジュールに割り当てる保護ドメインは，図 3 のメモリ保護の階層で示すように，アプリケーションに割り当てる 2 次保護ドメインを含むように構成する．これによって，アプリケーションが発行したシステムコールのパラメータがポインタを含んでいる場合に，ポリシーモジュールがアプリケーションのメモリ領域を直接アクセスしてチェックすることができる．

不正アクセスの被害を最小限に抑えるためには，2 次保護ドメインを実現する保護機構は堅牢性を高くする必要がある．1 次保護ドメインは細かい粒度で保護を行うために実装が複雑になりがちで，堅牢性を高めることが難しい．実行可能コンテンツは 1 次保護ドメインの保護機構の実装の不備やアプリケーションが提供する API の実装の不備などを利用した予想外の攻撃で不正アクセスを行うので，1 次保護ドメインだけで不正アクセスを完全に防止するのは困難である．2 次保護ドメインの役割は，そのような予想外の攻撃によって 1 次保護ドメインで不正アクセスを防ぎきれなかったときでも，その被害を最小限に抑える安全装置となることである．したがって，その実装は 1 次保護ドメインよりも堅牢である必要がある．

2 次保護ドメインの堅牢性を高くするために，ポリシーモジュールの実装をできるだけ単純かつ再利用可能なものとする．ポリシーモジュールの実装を単純にするためには，2 次保護ドメインの保護ポリシーは不

正アクセスによる被害が大きくならない範囲でできるだけシンプルなものとする．これによってポリシーモジュールを正しく実装しやすいようにして，保護機構の堅牢性を高くする．また，ポリシーモジュールを再利用可能とするために，アプリケーションから独立した外部モジュールとして実装して，アプリケーションの起動時に読み込むようにする．これによって，アプリケーションごとにポリシーモジュールを作成し直す必要がないようにして，実装の信頼性を高める．これによって，堅牢な 2 次保護ドメインを実現することができる．

5. 適 用 例

本章では，階層保護モデルを実際のアプリケーションに適用した例について述べる．アプリケーションとしては Web ブラウザと電子文書ビュアーを使用し，既存のアプリケーションを改変せずに階層保護モデルを実装する手法について説明する．

5.1 Web ブラウザ

階層保護モデルの適用例として，まず Web ブラウザにおいて階層保護モデルを実現する手法について述べる．この手法では実行可能コンテンツとして 2.2.1 項で説明したバイナリ型の実行可能コンテンツを採用して，1 次保護ドメインを割り当てて実行する．2.2.1 項で説明したように，バイナリ型の実行可能コンテンツは従来のオペレーティングシステムでは保護を行うことが難しいが，この手法では 4.2 節で述べたように細粒度保護ドメインを利用することで 1 次保護ドメインを実現することができる．

バイナリ型実行可能コンテンツの例として，我々は新しい MIME タイプのコンテンツを定義した．このコンテンツは ELF 形式のバイナリで提供されるモジュールで，Web ブラウザとリンク可能な形式になっている．このコンテンツの利用形態は，Java applet などと同様に HTML 文書に埋め込まれて表示されるような状況を想定しており，HTML 文書とともに Web サーバからダウンロードされて，クライアントの Web ブラウザでプログラムコードが実行されることによって内容を生成する．

Web ブラウザ自体を改変せずに保護を実現するために，プラグインを利用して保護機構を Web ブラウザに組み込んだ（**図 4** 参照）．このプラグインは，Web ブラウザの代わりに実行可能コンテンツに対する 1 次保護ドメインの割り当てる処理を行う．プラグインは Web ブラウザがダウンロードした ELF 形式の実行可能コンテンツを Web ブラウザのアドレス空間に読み込んでから，1 次保護ドメインを実現するために新しく細粒度保護ドメインを作成して割り当てる．また，プラグインは保護ドメイン間呼び出しで Web ブラウザと実行可能コンテンツの仲介を行う．図 4 の右側に示すように，プラグインは Web ブラウザからの描画要求を中継して，実行可能コンテンツに対するコンテンツ生成要求を細粒度保護ドメイン間呼び出しで行う．実行可能コンテンツが Web ブラウザで表示する画像を生成できるようにするために，プラグインは描画 API を用意して，図 4 の左側のように細粒度保護ドメイン間呼び出しで API を利用できるようにする．

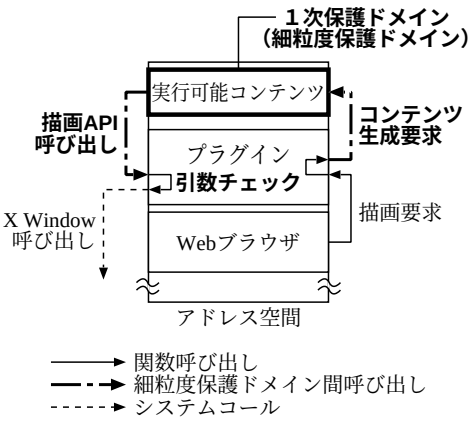

図 4 Web ブラウザにおける 1 次保護ドメインの実現
Fig. 4 A level-1 protection domain incorporated in a Web-browser.

1 次保護ドメインにおける細かい粒度での保護の実現は，細粒度保護ドメインとプラグインでの API の実装によって行う．メモリアクセスの制御は細粒度保護ドメインの機構を利用してページ単位で行う．また資源アクセスの制御を行うために，細粒度保護ドメインの機構を利用して実行可能コンテンツによるシステムコールの発行をすべて禁止し，プラグインが用意した API のみを呼び出せるようにする．プラグインは API の実装において引数が不正な値になっていないかなどのチェックを厳密に行って細粒度でのアクセス制御を実現する．これによって，Web ブラウザが指定した領域をはみ出して描画するなどの不正アクセスを防止する．

この手法による 1 次保護ドメインの実装では，アクセス制御機構は実行可能コンテンツが必要とする機能に応じて開発する必要がある．実行可能コンテンツに対するアクセス制御はプラグインで提供する API によって実装されるため，この部分は保護したい実行可能コンテンツの機能に依存したものとなる．一方，細粒度のメモリ保護や API 呼び出しを強制する機構の

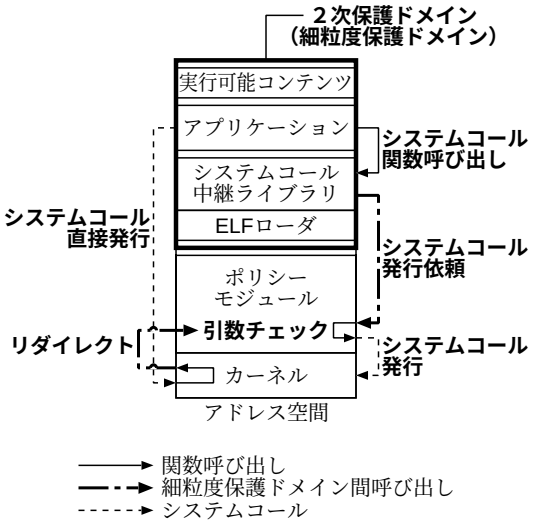

図 5 2 次保護ドメインの実現
Fig. 5 Implementation of level-2 protection domain.

実現手法はバイナリ型の実行可能コンテンツ一般に適用することができる．

このような手法によって Web ブラウザにおいて 1 次保護ドメインを実現することができる．2 次保護ドメインの実現に関しては，次説で説明する電子文書ビュアーの場合と同様の手法を利用しているので，次説でまとめて説明する．

5.2 電子文書ビュアー

次に電子文書ビュアーにおいて階層保護モデルを実現する手法について述べる．2.2.2 項で説明したように，PostScript や PDF などの電子文書はインタプリタ型の実行可能コンテンツに分類される．インタプリタ型の実行可能コンテンツの場合は，インタプリタによってある程度アクセス制御が行われるので，すでに 1 次保護ドメインが実現されていると見なすことができる．したがって，ここではすでに 1 次保護ドメインを実現しているアプリケーションに対して，さらに 2 次保護ドメインを適用する手法について説明する．

既存の電子文書ビュアーを改変せずに 2 次保護ドメインを実現するために，アプリケーションをロードするプログラムローダを改変して保護機構を組み込む．プログラムローダは，アプリケーション自体をアドレス空間に読み込むために，アプリケーションの起動に先だって実行されるプログラムである．2 次保護ドメインを実現するために，改変した ELF ローダを利用して**図 5** のようにアプリケーション全体に対して 2 次保護ドメインとなる細粒度保護ドメインを割り当てる．プログラムローダを利用することで，アプリケーションからは透過的に保護機構を実現できる．ここでは，ELF 形式のプログラムを読み込むプログラムローダである ELF ローダを使用した．

2 次保護ドメインのアクセス制御を行うために，図 5 のようにポリシーモジュールをアプリケーションのアドレス空間に読み込んで，アプリケーションによるアクセスをチェックする．ポリシーモジュールは，アプリケーションとは独立したモジュールとして提供して，アプリケーションの起動時に改変した ELF ローダで外部ファイルから読み込む．また，ポリシーモジュール自体を不正アクセスから保護するために，ポリシーモジュールに対しても別の細粒度保護ドメインを割り当てる（ただし図 5 では煩雑になるため省略してある）．

アプリケーションはシステムコールによって計算機資源にアクセスするので，2 次保護ドメインにおけるアクセス制御も基本的にシステムコールのレベルで行う．アプリケーションが発行するシステムコールをチェックできるようにするために，2 次保護ドメインで発行されたシステムコールはすべてポリシーモジュールを経由するようにする．アプリケーションがプロセッサのトラップ命令を使用して直接発行したシステムコールは，図 5 の左側に示すように，細粒度保護ドメインの機能によってポリシーモジュールにリダイレクトされる．ライブラリ関数を使用して発行されるシステムコールは，チェックを効率良く行うために，図 5 の右側に示すように細粒度保護ドメイン間呼び出しでシステムコールの発行をポリシーモジュールに依頼するように変更する．これはシステムコール中継ライブラリを使用してライブラリ関数を置き換えることによって実現する．これによって，ライブラリ関数を利用したシステムコールでは，カーネルによってリダイレクトされるコストを省くことができる．

2 次保護ドメインの保護ポリシーは，正しく実装できるようになるべくシンプルにする必要がある．今回の実装例では，2 次保護ドメインによる保護の目的を限定したものにすることで，シンプルな保護ポリシーを実現する．まずファイルに対するアクセスは，ファイルが破壊されないことを目的とする．ファイルが破壊さえされなければ，不正アクセスが行われた後にシステムを復旧することは比較的容易に行える．したがってファイルアクセスは，原則として読み込みのみを許可し，書き込みはアプリケーションごとのテンポラリディレクトリのみ許可する．ネットワークを含むプロセス間通信は，アプリケーションの実行に必要のないものは禁止する．電子文書ビュアーでは X Window への通信以外を禁止する．これより細かい粒度の保護

の実現は，5.1 節で説明した 1 次保護ドメインの機構や，インタプリタのアクセス制御による保護機構にまかせることで，2 次保護ドメインの保護機構をシンプルに保ち堅牢性を高めることができる．

6. 評価実験

本章では階層保護モデルに対する評価として，モデルの正当性を示すための実証実験と，実用性を示すための性能実験について述べる．実証実験の目的は，階層保護モデルが既存の攻撃手法に対して有効であることを示すことである．この実験では，実行可能コンテンツによる攻撃として実際に報告されている手法を用いた不正アクセスを再現し，階層保護モデルに基づく保護機構によって正しく不正アクセスを防止できることを確認する．また，性能実験の目的は，階層保護モデルが実用に耐えうる性能で実現可能であることを示すことである．この実験では，5 章で説明した手法による実装を用いて，実際のアプリケーションにおいて保護をすることによるオーバヘッドを測定する．

実験に使用したマシンは，CPU が PentiumIII 1 GHz，メモリは 512 MB（PC133 CL2），ハードディスクに IBM Deskstar 75GXP DTLA-307075（75 GB，7200 rpm），グラフィックカードに Matrox Millennium G450 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，X Window System のバージョンは 4.0.1 である．

6.1 不正アクセスの防止

階層保護モデルの正当性を示すために，実行可能コンテンツによる不正アクセスに対して階層保護モデルによる保護機構が有効であることを実証する実験を行った．この実験では，アプリケーションに対して攻撃を行う実行可能コンテンツを実際に作成し，実行可能コンテンツによる不正アクセスを再現した．階層保護モデルに基づく保護機構で保護した場合と保護していない場合のそれぞれに対して攻撃を行い，保護機構としての有効性を検証した．

実験では，不正アクセスを行う実行可能コンテンツの例として PDF 形式の実行可能コンテンツを作成し，Acrobat Reader に対する攻撃を行った．この攻撃では，Acrobat Reader におけるバッファ溢れ攻撃に対する脆弱性[6] を利用する．Acrobat Reader 4.05 では，PDF ファイル中の特定の箇所に非常に長い文字列を格納しておくことで Acrobat Reader のバッファを溢れさせ，任意のコードを送り込んで実行させることができる．我々はこの攻撃手法を利用して，`exec()` システムコールを発行してコマンドインタプリタを起動する PDF を作成した．この PDF はコマンドインタプリタを起動するだけなので実害はないが，コマンドインタプリタが起動できれば様々な不正アクセスを行うのに十分な能力を持っているといえる．

```
1  > ls
2  acroread shell.pdf
3  > ./acroread shell.pdf
4  $ ls
5  acroread shell.pdf
6  $
```

図 6 不正アクセスが行われる例
Fig. 6 Example of an attack.

```
1  > ls
2  acroread shell.pdf
3  > /lib/ld-pd.so -pm /lib/pm.so \
4    ./acroread shell.pdf
5  pm.so: exec: permission denied
6  pm.so: program terminated
7  >
```

図 7 不正アクセスを防止した例
Fig. 7 Example of protection against the attack.

実際に作成した PDF によって不正アクセスが行われる例を**図 6** に示す．2 行目で表示されている `shell.pdf` がコマンドインタプリタ（`/bin/sh`）を起動する PDF である．3 行目で示すように `acroread` コマンドで `shell.pdf` を閲覧しようとすると，4 行目で示すようにプロンプトが `>` から `$` に変わってしまう．これはコマンドインタプリタが起動したことを示しており，PDF によって不正なアクセスが行われてしまっていることが分かる．

次に 5 章で説明した実装を利用して，階層保護モデルに基づく保護機構で不正アクセスを防止した例を**図 7** に示す．3 行目で示されている `/lib/ld-pd.so` は，5.2 節で説明した 2 次保護ドメインを割り当てるための ELF ローダである．引数としてプログラムを指定することで，そのプログラムに 2 次保護ドメインを割り当てて起動することができる．また，`-pm` はポリシーモジュールを指定するためのオプションであり，ここでは `/lib/pm.so` を指定している．2 次保護ドメインを割り当てた環境では，5 行目に示すように，ポリシーモジュールによって `exec()` システムコールが検出されている．保護ポリシーに基づいて，この `exec()` システムコールは却下され，6 行目で示すようにアプリケーションは終了させられている．このように，アプリケーションが正常に実行を続けることはできないが，2 次保護ドメインによって不正な `exec()` を防止

表 1 保護ドメイン間呼び出しのコスト
Table 1 Cost of a cross-domain call.

方式	時間	サイクル数
細粒度保護ドメイン間呼び出し	0.38 μs	378
関数呼び出し（保護なし）	7 ns	7
プロセス間通信（パイプ）	4.05 μs	4,046

して，フェイルセーフを実現している．

6.2 保護のオーバヘッド

階層保護モデルの実用性を示すために，実際のアプリケーションに適用した場合のオーバヘッドを測定する実験を行った．この実験では，保護によるオーバヘッドを調べるために，5 章で説明した手法による実装を用いて，1 次保護ドメインと 2 次保護ドメインのそれぞれについてオーバヘッドを測定した．以下 6.2.1 項で 1 次保護ドメインのオーバヘッド，6.2.2 項で 2 次保護ドメインのオーバヘッドを測定した実験について述べる．

なお，階層保護モデルとして 1 次保護ドメインと 2 次保護ドメインを組み合わせたときのオーバヘッドは，それぞれの保護ドメインにおけるオーバヘッドの算術和で求められる．再帰的仮想計算機[11] などでは，内側の保護ドメイン切替えを外側の保護ドメインで仮想化するためにオーバヘッドが指数関数的に積算されるが，5 章で述べた手法による実装では，保護ドメイン切替えはカーネルレベルで行われるため，外側の保護ドメインが内側の保護ドメインの性能に影響を与えることはない．したがって，全体のオーバヘッドは，単純に 1 次保護ドメインと 2 次保護ドメインのオーバヘッドを足し合わせたものとなる．

実験について述べる前に，まず参考として，階層保護モデルの実装においてオーバヘッドの主な要因となる保護ドメイン間呼び出しにかかる時間の計測値を**表 1** に示す．表 1 には，5 章で述べた手法で利用した細粒度保護ドメイン間の呼び出しにかかる時間のほか，比較として関数呼び出しとプロセス間通信（パイプ）にかかる時間も示す．呼び出し時間の計測には，PentiumIII に装備されているサイクルカウンタを使用した．表 1 に示すように，細粒度保護ドメイン間の呼び出しは 0.38 μs で行うことができる．これはプロセス間通信と比べて約 11 倍速く，保護のオーバヘッドも低く抑えられることが予想される．

6.2.1 1 次保護ドメイン

1 次保護ドメインのオーバヘッドを測定するために，5.1 節で説明した Web ブラウザにおける階層保護モデルの実装を利用して，1 次保護ドメインを割り当てた実行可能コンテンツの実行時間を計測する実験を行った．実験では，Web ブラウザのウィンドウに描画を行う実行可能コンテンツを作成して，保護をした場合と保護がない場合の描画時間の違いを比較した．描画する図形はフラクタル図形の一種であるマンデルブロ集合で，正方形の一辺の長さを 1 ピクセルから 150 ピクセルまで変化させながら，それぞれの大きさの図形を描画するのにかかる時間を計測した．なお，この実験では 1 次保護ドメイン単独でのオーバヘッドを測定することが目的なので，2 次保護ドメインは使用していない．

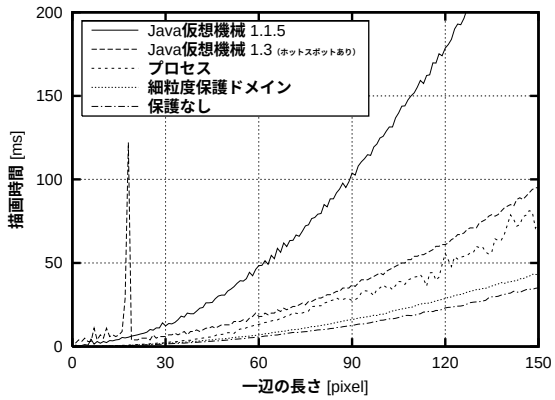

図 8 1 次保護ドメインのオーバヘッド
Fig. 8 Overhead of level-1 protection domain.

実験では，5.1 節で述べたバイナリ型の実行可能コンテンツを利用して，保護がない場合と細粒度保護ドメインで実現した 1 次保護ドメインで保護した場合のそれぞれの描画時間を測定した．また，対照実験として，プロセスで保護した場合と Java の仮想機械で保護した場合の実験も行った．プロセスで保護した場合は，実行可能コンテンツを子プロセスで動作させ，プロセス間通信で実行可能コンテンツを呼び出した．プロセス間通信の機構としては，親子間で閉じた通信路を確立できる機構の代表例としてパイプを使用した．Java の仮想機械で保護した場合は，バイナリ型と同じフラクタル図形を描画する実行可能コンテンツを Java applet として作成して実行した．使用した Java 仮想機械は，Netscape に内蔵された Java 1.1.5 および Sun のホットスポット対応 Java 1.3 である．Web ブラウザには，Netscape 4.76 を使用した．

図 8 に実験結果を示す．横軸は描画した図形の一辺の長さ [pixel]，縦軸は描画にかかった時間 [ms] である．このプラグインは 1 ドット描画するたびに描画 API を呼び出すので，一辺の長さの 2 乗に比例して実行時間が増加している．細粒度保護ドメインで実現した 1 次保護ドメインのオーバヘッドは，保護がない

場合に比べて平均 22.6%程度に抑えられている．プロセス間通信を用いた場合のオーバヘッドは，保護がない場合と比べて平均 101%，Java 仮想機械では，Java 1.3 は 31〜150 ドットでの平均☆で 199%，Java 1.1.5 は平均 755% であった．細粒度保護ドメインで実現した 1 次保護ドメインのオーバヘッドは，ほかの保護機構と比べて低く抑えられており，実用的な性能で 1 次保護ドメインが実現可能であるといえる．

6.2.2 2 次保護ドメイン

2 次保護ドメインのオーバヘッドを測定するために，5.2 節で説明した電子文書ビュアーにおける階層保護モデルの実装を利用して，2 次保護ドメインを割り当てたアプリケーションの実行時間を計測する実験を行った．実行可能コンテンツを処理するアプリケーションに対して，2 次保護ドメインを割り当てた場合と割り当てなかった場合のそれぞれの実行時間を計測して，実行時間の差から 2 次保護ドメインのオーバヘッドを求めた．ここでいうオーバヘッドとは，保護を行うことによるプログラム全体の実行時間の増加率である．すなわち，保護を行っていないプログラムの実行時間を T_{noprot}[秒]，保護を行った場合の実行時間を T[秒] とすると，オーバヘッドは以下の式で求められる．

$$overhead = \left(\frac{T}{T_{noprot}} - 1 \right) \times 100[\%] \qquad (1)$$

実行可能コンテンツとしては PostScript 形式と PDF 形式を使用し，処理系としては Ghostscript 5.50 と Acrobat Reader 4.05 をそれぞれ使用した．5.2 節で説明したように，電子文書ビュアーでは実行可能コンテンツの処理系が 1 次保護ドメインを実現していると考えられる．したがって，この実験ではすでに 1 次保護ドメインを使用しているアプリケーションに対して，さらに 2 次保護ドメインを割り当てることによるオーバヘッドを測定する．

Ghostscript を使用した実験では，PostScript ファイルを解釈して画面に表示するのにかかる時間を計測した．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い，ページの最初から最後まで一気に表示させた．PostScript ファイルとしては，大きさが 280Byte から 4.55 MB までの 160 個のファイルを使用して，それぞれの PostScript ファイルに対して実行時間を測定した．実行時間の増加分からそれぞれのファイルにおけるオーバヘッドを求め，その分布を表すヒストグラムを作成した．

☆ 20pixel 付近のスパイクの原因は，ホットスポットが作動してコンパイルが行われているためと思われる．

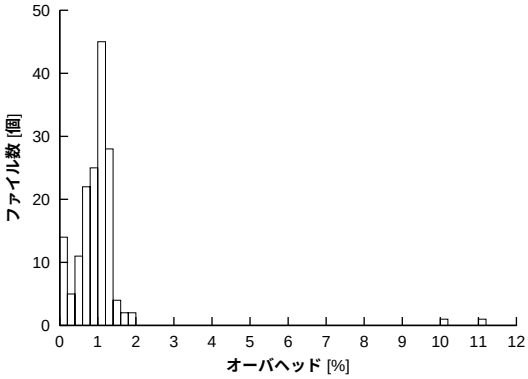

図 9 2 次保護ドメインのオーバヘッド（Ghostscript）
Fig. 9 Overhead of level-2 protection domain (Ghostscript).

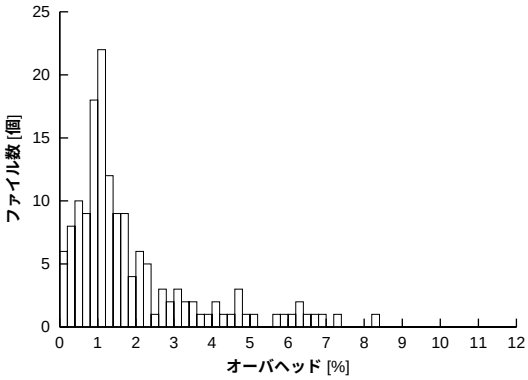

図 10 2 次保護ドメインのオーバヘッド（Acrobat Reader）
Fig. 10 Overhead of level-2 protection domain (Acrobat Reader).

図 9 に実験結果を示す．横軸は 0.2%きざみのオーバヘッド [%] で，縦軸はそれぞれの階級に属する PostScript ファイルの数 [個] である．Ghostscript における 2 次保護ドメインのオーバヘッドはほぼ 2%以下に抑えられており，最大でも 11%程度であることが分かる．オーバヘッドの平均は 1.05%であった．絶対時間で比べても，実行時間の増加分は最大で 160 ms 以下であった．

Acrobat Reader を使用した実験では，PDF 形式のファイルを Postscript 形式に変換するのにかかる時間を計測した．誤差を減らすために，PDF から Postscript への変換はコマンドラインから非対話的に行った．実験には，大きさが 758 Byte から 8.13 MByte までの PDF ファイル 152 個を使用して，PostScript 形式の場合と同様にしてヒストグラムを作成した．

図 10 に実験結果を示す．Acrobat Reader では Ghostscript に比べてオーバヘッドの分布が広くなっているが，最大でも 8%程度に抑えられており，平均

では 1.88%であった．絶対時間で比べても，実行時間の増加分は最大で 1 秒程度であった．

以上の実験から，Ghostscript や Acrobat Reader では 2 次保護ドメインのオーバヘッドは平均 1.05〜1.88%に抑えられており，実用的な性能で 2 次保護ドメインが実現可能であるといえる．

7. 関連研究

実行可能コンテンツを安全に実行する環境としては，Java アプレット[12]による方式が普及している．Java は安全性を確保するために，型安全な言語の使用や，バイトコードベリファイアによるコードの検証，Java 仮想マシンによる実行時チェックなどの言語処理系の機能を利用して，粒度の細かい保護ドメインを実現している．Java アプレットの保護機構は，階層保護モデルの 1 次保護ドメインのみを実現したものと見なすことができる．Java の保護モデルは JavaVM による一枚岩的なもので，階層保護モデルのような階層的な保護は提供されていない．

プロセス内に保護ドメインを形成する仕組みとしては，OS に依存しない方式と，OS の機能として提供する方式がある．OS に依存しない方式でプロセス内に保護ドメインを形成する仕組みとしては，SFI（Software Fault Isolation）[13]や PCC（Proof Carrying Code）[14),15)]などがある．SFI ではバイナリ形式のコードを直接改変してアクセスチェックのための命令を挿入する．また，PCC は，バイナリコードに添付された証明を静的に検証することにより，実行時のチェックを行わずに安全性を保証する．OS の機能としてプロセス内に保護ドメインを形成する仕組みとしては，Palladium[16]や PSL（Protected Shared Libraries）[17]などがある．Palladium は Intel x86 CPU のリング保護機構を利用して，プロセス内に 2 段階の保護ドメインを形成する．PSL は POWER プロセッサ上でアドレス空間を部分的に切り替えて，共有ライブラリのデータを安全に共有する．これらのプロセス内の保護ドメインは，階層保護モデルにおいてメモリ保護のみの 1 次保護ドメインを実現したものと見なすことができる．これらの保護ドメインは，主にメモリ保護を目的としたものであり，ファイルなどの一般の資源に対するアクセス制御は想定されていない．

従来の保護ドメインであるプロセスの切替えを高速化する仕組みとしては，様々な試みがなされている（LRPC[18]，Spring[19]，Mach[20]，L4[21),22)]）．しかしプロセスによる保護ドメインでは，そのままでは OS の資源に対して細かい粒度でアクセス制限を行うことは難しい．

Janus[23]では，実行可能コンテンツを処理するヘルパアプリケーションを安全に実行する環境を実現している．Janus は，Solaris のシステムコールトレース機能を利用して，監視プロセスがヘルパアプリケーションを子プロセスとして実行し，ヘルパアプリケーションが発行するシステムコールをチェックしている．Janus は，階層保護モデルの 2 次保護ドメインのみを実現したものと見なすことができる．Janus はシステムコールの監視を別プロセスで行っており，システムコールを発行するたびにプロセス切替えが発生するので，細粒度保護ドメインで実現した 2 次保護ドメインと比べて保護のオーバヘッドが大きくなる．

Multics では，リングプロテクションによる階層的なメモリ保護を提供している．Multics は GE-645 プロセッサのメモリ保護機構を利用して，最大 8 段階の特権レベルを提供して階層的な保護を行う．本論文で導入した階層保護モデルは，リングプロテクションの概念を一般の保護ドメインに拡張し，実行可能コンテンツの保護に応用したものと考えることができる．Multics のリングプロテクションはプロセス間のメモリ保護を目的としたもので，そのままでは実行可能コンテンツの保護に使うことは難しい．また，リングプロテクションの概念には，保護の粒度と堅牢性の関係についての考察は含まれていない．

8. まとめ

本論文では，実行可能コンテンツの安全な実行環境を実現する手法について述べた．実行可能コンテンツの不正アクセスに対処するための保護モデルとして階層保護モデルを導入して，不正アクセスによる被害を最小限に抑える仕組みを示した．階層保護モデルでは，堅牢性は低いが細かい粒度のアクセス制御を行える 1 次保護ドメインと，保護の粒度は粗いが実装が単純で堅牢性の高い 2 次保護ドメインを組み合わせることで，細粒度の保護と保護機構の堅牢性を両立して安全性を高めることができる．本論文では，我々が提案してきた細粒度保護ドメインの機構を応用して，階層保護モデルを効率良く実現する手法を示した．1 次保護ドメインと 2 次保護ドメインのそれぞれを細粒度保護ドメインで実装し，オーバヘッドの低い保護機構を実現した．また，既存のアプリケーションを改変せずに階層保護モデルを実装する手法を示し，実際にこの手法を用いた実装例として，Web ブラウザと電子文書ビュアーに対して階層保護モデルを実装した．この実装を用いた実験を行って，階層保護モデルが既存の攻

撃手法に対して有効であることを示した．また，保護によるオーバヘッドを測定して，階層保護モデルが実用的な性能で実現可能であることを示した．我々の実装では，Web ブラウザにおける 1 次保護ドメインのオーバヘッドは 22.6%程度，電子文書ビュアーにおける 2 次保護ドメインのオーバヘッドは平均で 1.05〜1.88%程度，最大でも 8〜12% 程度であった．

参 考 文 献

1) CERT Advisory: CA-2001-26 Nimda Worm (2001).
2) CERT Advisory: CA-2001-22 W32/Sircam Malicious Code (2001).
3) CERT Advisory: CA-2000-07 Microsoft Office 2000 UA ActiveX Control Incorrectly Marked "Safe for Scripting" (2000).
4) CERT Advisory: CA-1997-20 JavaScript Vulnerability (1999).
5) CERT Advisory: CA-1995-10 Ghostscript Vulnerability (1995).
6) SPS Advisory: #39: Adobe Acrobat Series PDF File Buffer Overflow (2000).
7) 品川高廣，河野健二，高橋雅彦，益田隆司：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，Vol.40, No.6, pp.2596–2606 (1999).
8) Takahashi, M., Kono, K. and Masuda, T.: Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code, *Proc. 19th IEEE International Conference on Distributed Computing Systems*, pp.64–73 (1999).
9) Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, *Proc. DARPA Information Survivability Conference and Exposition* (1999).
10) Saltzer, J.H. and Schroeder, M.D.: The Protection of Information in Computer Systems, *Proc. IEEE*, Vol.63, No.9, pp.1278–1308 (1975).
11) Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G. and Clawson, S.: Microkernels Meet Recursive Virtual Machines, *Proc. 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pp.137–151 (1996).
12) Team, J., Gosling, J., Joy, B. and Steele, G.: *The Java[tm] Language Specification*, Addison Wesley Longman (1996). ISBN 0-201-6345-1.
13) Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L.: Efficient Software-Based Fault Isolation, *Proc. 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pp.203–216 (1993).
14) Necula, G.C. and Lee, P.: Safe Kernel Extensions without Runtime Checking, *Proc. 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pp.229–243 (1996).
15) Necula, G.C.: Proof-Carrying Code, *Proc. 24th ACM Symposium on Principles of Programing Languages* (*POPL '97*), pp.106–119 (1997).
16) Chiueh, T., Venkitachalam, G. and Pradhan, P.: Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions, *Proc. 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), pp.140–153 (1999).
17) Banerji, A., Tracey, J.M. and Cohn, D.L.: Protected Shared Libraries — A New Approach to Modularity and Sharing, *Proc. 1997 USENIX Annual Technical Conference*, pp.59–75 (1997).
18) Bershad, B.N., Anderson, T.E., Lanzowska, E.D. and Levy, H.M.: Lightweight Remote Procedure Call, *ACM Trans. Comuput. Syst.*, Vol.8, No.1, pp.37–55 (1990).
19) Hamilton, G., Powell, M.L. and Mitchell, J.G.: Subcontract: A flexible base for distributed programming, *Proc. 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pp.69–79 (1993).
20) Ford, B. and Lepreau, J.: Evolving Mach 3.0 to a Migrating Thread Model, *Proc. USENIX Winter 1994 Technical Conference* (1994).
21) Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *Proc. 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), pp.66–77 (1997).
22) Liedtke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N. and Jaeger, T.: Achieved IPC Performance, *Proc. 6th Workshop on Hot Topics in Operating Systems* (*HOTOS '97*), pp.28–31 (1997).
23) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.A.: A Secure Environment for Untrusted Helper Applications, *Proc. 6th USENIX Security Symposium* (1996).

(平成 13 年 12 月 20 日受付)
(平成 14 年 4 月 16 日採録)

品川 高廣（学生会員）

1974 年生．1998 年東京大学工学部電子工学科卒業．2000 年東京大学大学院理学系研究科情報科学専攻修士課程修了．現在，同大学院博士課程在学中．2000 年 12 月から電気通信大学特別研究学生．オペレーティングシステム等のシステムソフトウェアに興味を持つ．IEEE/CS, ACM 各学生会員．平成 11 年度情報処理学会論文賞受賞．

河野 健二（正会員）

1970 年生．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．2000 年 1 月から電気通信大学情報工学科助手，現在に至る．1999 年 10 月から科学技術振興事業団さきがけ研究 21「情報と知」領域・研究員を兼務．オペレーティングシステム等のシステムソフトウェア，分散および並列処理に興味を持つ．理学博士．平成 11 年度情報処理学会論文賞受賞．

益田 隆司（正会員）

1939 年生．1963 年東京大学工学部応用物理学科卒業．1965 年東京大学大学院修士課程修了．同年（株）日立製作所入社．1977 年から筑波大学，1988 年 3 月から東京大学に勤務．2000 年 4 月から電気通信大学情報工学科教授．専門はオペレーティングシステム．