Vol.2009-OS-112 No.9
2009/8/6

クラス階層型セキュリティポリシーによるアクセス制御

横川 晃[†1] 品川高廣[†1] 加藤和彦[†1]

現在の SecureOS では，セキュリティポリシーを記述しアクセス権をコントロールする方法が取られているが，そのセキュリティポリシーの記述が難しい事が問題となっている．その原因として設定項目の多さ故記述が複雑になる点，一度記述したポリシーの再利用が難しい点が挙げられる．ポリシー記述が難しいと，不適切なアクセス権を与えてしまう可能性を大きくし，ユーザが持っている重要なファイルを破壊するなど不正アクセスを引き起こす可能性がある．本研究では，ポリシーの記述を容易化するためクラス階層型のセキュリティポリシーを提案する．ポリシー記述の容易化のため，似たようなポリシー記述をクラスとしてまとめる．これにより個々のファイルやプロセス単位でセキュリティポリシーを記述する必要がなくなる．また一度記述したポリシーの再利用性を高めるため，クラスを継承して新たなクラスを作成できるようにする．さらにポリシーの再利用性を高めるため，動的にアクセス権を決定できるよう条件式を記述できるようにする．提案方式に基づいた実装を行い，実験を通して故意な書き換えが抑えられる事を確認した．

Access control through class-hierarchized security policy

Akira Yokokawa,[†1] Takahiro Shinagawa[†1]
and Kazuhiko Kato[†1]

SecureOS recently performs its security function by enforcing a security policy, However the complex method of writing its security policy represents a problem. This is partly due to too many complex items to configure and the difficulty in reusing a previously written security policy. These factors may cause illegal access due to improperly granted privileges. In this paper, we present a design and an implementation of class-hierarchized security policy, which defines security policy as a class, includes condition expressions into the security policy and enables inheriting a class. This lets the user ignore the privileges granted to each process or file and makes security policy reusable. we have implemented our design and confined illegal access to file through an experiment.

1. 序　　論

現在の SecureOS では，アクセス権設定のためのセキュリティポリシーの記述方法が難しい事が問題になっている．その原因のひとつに，複雑で多数の項目を設定しなければならないことが挙げられる．記述が複雑になると誤ったポリシーを記述してしまう恐れがある．もうひとつの原因として，現在のセキュリティポリシーの記述方法が特定のシステムやアプリケーションに依存した記述になり，一度記述したポリシーの再利用が難しいことが挙げられる．例として Word 向けに作成されたポリシーは，たとえ Excel 向けに作成されたポリシーと似たような記述が多かったとしても，ファイル名やパス名といった Word とアクセスされるファイルの関係に依存した記述が入ってきてしまうため，そのまま Excel 向けのポリシーとして使用することは難しい．

従来からもセキュリティポリシーの記述を容易にするために，様々なアクセス制御モデルやセキュリティポリシーの記述方法が提案されている．例えば SELinux[1] などに用いられている DTE(Domain and Type Enforcement)[8)9)10)] では，ドメインと呼ばれるアクセスの主体側であるプロセスのタイプと，アクセスされる対象であるファイルのタイプの 2 つに分けてポリシー記述をまとめることができる．しかし SELinux には多くの複雑な設定項目があり，セキュリティポリシーの記述は容易ではない．また，MAPBox[3] ではアプリケーションの動作を元に behaviors クラスと呼ばれるポリシーの雛形が用意されている．雛型を編集することによってセキュリティポリシーを記述することができる．しかし一度編集したポリシーを再利用することはできない．

本研究では，セキュリティポリシーの記述を容易化するために，クラス階層型のセキュリティポリシーを提案する．本手法では，ポリシー記述を容易にするために，個々のアプリケーションやファイルごとにポリシーを記述するのではなく，ポリシーをクラスとして予め定義しておき，アプリケーションやファイルにはアクセス制御を適用すべき適切なクラスを割り当てる．これにより，具体的なアクセス制御の内容ではなくより抽象的なクラスという単位でポリシーを記述できるようにし，アプリケーションごとのポリシー記述の手間を削減する．またポリシーの再利用性を高めるために，クラスを継承して新たなポリシーを設定できるようにする．これにより上位のクラスで定義されたポリシーを再利用して，新たなクラ

†1 筑波大学大学院システム情報工学研究科
University of Tsukuba, Graduate School of Systems and Information Engineering

スを作成する際のポリシー記述の手間を削減する．またポリシーの再利用性を高めるために，プロセスやファイルごとに異なる属性値を設定できるようにして，ポリシー記述の際にこの属性値に基づく条件式によって動的にアクセス権を決定できるようにする．これにより似たようなポリシー記述を更にまとめることを可能にする．

本稿では，提案したクラス階層型のセキュリティポリシーの適用例として，アプリケーションのアップデートを安全に行う仕組みを示す．アップデートを安全かつ自動的に行うために，ローカル PC 内のファイルに適切なクラスを割り当て，アップデートプログラムがアクセスできるファイルの範囲を制限する．本稿では，本適用例を実現する実装を行い，実験を行った結果を示す．

本稿の構成は以下である．2 章では想定する不正アクセスの例について述べる．3 章では本稿で提案するクラス階層型のセキュリティポリシーの概要について述べる．4 章では提案方式に基づいた実装方法について述べる．5 章では実装したシステムを用いて，実際にアクセス制御を行った結果について述べる．6 章では関連研究について述べ，7 章で結論を述べる．

2. 想定する不正アクセスの例

本研究では，アプリケーションのアップデートを行う際に，悪意をもったアップデートプログラムがユーザの重要なファイルを不正に操作する例を考える．アプリケーションのアップデートを行う際，通常は制作元のアップデートプログラムを通じてアプリケーションをアップデートする．その際にアップデートプログラムは，アプリケーションが動作するために必要なファイルをローカル PC に配置する．

しかし多くのアップデートの場合，どんなファイルがローカルに配置され，アップデートプログラムがどんな操作を行うのか全てを把握することは困難である．またアップデートの際，ユーザは管理権限を委譲することを求められる場合が多く，アップデートプログラムが故意にユーザの重要なファイルを操作していたとしても制限することができない．それ故，悪意をもったアップデートプログラムが，アプリケーションのアップデート中に，ユーザの重要なファイルを故意に書き換えたりするなどの不正アクセスが発生する可能性がある．本研究では，アップデートプログラムがユーザの重要なファイルを不正に書き換えることを防ぐ．

```
class Object {
  var user, maker, domain, name
}

class Executable extends Object {
  allow read any
  allow exec any
}

class Contents extends Object {
  allow read any
}
```

```
class Normal extends Executable {
  allow write if subject.domain == object.domain
}

class NormalContents extends Contents {
  allow write if subject.domain == object.domain
}

class Confidential extends Contents {
  allow read if subject.name == object.maker
  allow write if subject.domain == object.domain
}
```

図 1 セキュリティポリシーの例
Fig. 1 Example of security policy

3. 提案システムの概要

本研究で提案するクラス階層型のセキュリティポリシーについて述べる．提案システムでは，セキュリティポリシーの記述を容易にするために，似たようなセキュリティポリシーをクラスとしてまとめる．これにより個々のプロセスやファイル単位でのセキュリティポリシーの記述の必要性をなくす．また一度記述したポリシーの再利用性を高め，新たにクラスを作成する際の労力を削減するために，クラスを継承して新しくクラスを定義できるようにする．実行時のシステムの状態を考慮して動的にアクセス権を決定できるようにするために，プロセスやファイルに属性値を持たせる．さらにポリシーの再利用性を高めるため，ポリシーに条件式を記述できるようにする．

以下本章では，3.1 節でクラスによるポリシー定義について述べる．3.2 節ではクラスの継承について述べる．3.3 節ではファイルやプロセスに持たせる属性値と条件式について述べる．そして 3.4 節では提案システムによりどのようにファイルが保護されるのか述べる．

3.1 クラスによるポリシー定義

クラス階層を用いたセキュリティポリシーの例を図 1 に示す．図 1 のポリシーでは Object, Executable，Normal，Contents，NormalContents, Confidential という名前を用いて 6 つのクラスを定義している．それぞれのクラスは，中括弧の間に記述されているポリシーを

classdef	::=	`class` *classname* [`extends` *classname*] { *policy* }
policy	::=	*permission operation parameter*
permission	::=	`allow` | `deny`
operation	::=	`read` | `write` | `exec`
parameter	::=	`any` | `condition`
condition	::=	`if subject.`*var equation* `object.`*var*
var	::=	`name` | `user` | `domain` | `maker` | `class`
equation	::=	== | !=

図 2 ポリシーの生成規則
Fig. 2 Syntax for security policy

持つ．例として Executable クラスでは”allow read any”，”allow exec any”の 2 つのポリシーを持っている．クラス内に記述されるそれぞれのポリシーは図 2 の生成規則に従って記述される．

まず記述したいクラスのクラス名を決定する．クラス名は図 2 の classname にあたる．続けて extends の記述を用いて親クラスを継承するかしないかを記述する．親クラスの継承については，3.2 節で説明する．その後クラス内の各ポリシーを記述する．ポリシーの先頭には，アクセスの可否を決めるため allow か deny を記述する．続けて制限したい操作を read, write, exec から選んで記述する．文末には条件式を記述する．条件式の記述については 3.3 節で述べる．もし条件式を記述しない場合は，アクセス権の決定に条件を設けない旨の any を記述する．例として図 1 の ”allow read any” は read の権限を持ってアクセスする事を誰でも許可するというポリシーになる．

またポリシーを記述する際に，明示的に記述されていない権限に関しては全て deny であるとする．その例を図 3 に示す．図 3 上段左側の Object クラスには”var user, maker, domain, name”の記述のみがあるのに対し，上段右側には”deny read any”, ”deny write any”, ”deny exec any”の 3 つの記述が加わっている．上段右側と左側，2 つのクラスの記述は同じ内容のセキュリティポリシーとして評価される．”var user, maker, domain, name” はファイルの属性値に関する記述である．属性値に関しては 3.3 節で述べる．

3.2 クラスの継承について

提案手法では，新しくクラスを作成しポリシーを記述する際にかかる労力を削減するために，以前記述したクラスを継承して新しくクラスを作成する．例として図 1 には Normal と

実際の記述	等価な記述

`class Object {`
` var user, maker, domain, name`
`}`
	`class Object {`
	` var user, maker, domain, name`
	` deny read any`
	` deny write any`
	` deny exec any`
	`}`

`class Normal extends Executable{`
` allow write`
`  if subject.domain == object.domain`
`}`
	`class Normal {`
	` var user, maker, domain, name`
	` allow read any`
	` allow write`
	`  if subject.domain == object.domain`
	` allow exec any`
	`}`

図 3 ポリシー記述の等価関係
Fig. 3 Equivalence relationship between policy descriptions

いうクラスが記述されている．Normal クラスのクラス名には続けて”extends Executable”と記述されている．この記述を行うことにより，Executable というクラスを継承する．本手法では多重継承は許可しない．継承を行って作成された新しいクラス（以下子クラスと明記する）は，継承した親クラスのポリシーを持つ．したがって，図 3 の下段左側の記述と右側の記述は等価である．

記述された子クラスでは新しくポリシーを定義することが可能である．その際に親クラスのポリシーとの整合性を考える必要がある．例えば親クラスで”allow read any”と記述されているのに対して，子クラスで”deny read any”と記述すると，ポリシーの整合性に不都合が生じる．本手法では子クラスに書かれているポリシーの優先度を挙げて対処する．

3.3 ポリシー内の条件式と属性値

図 1 では Normal クラスに 1 つのポリシーが記述されており，”if” で始まる記述を含んでいる．この”if”以降の記述が条件式である．このポリシーは，アクセス権が決定される際 if 文で指定されている条件を満たした場合に適用される事を示している．

ポリシー内の条件式を記述する際，システムの実行時の状態を考慮してポリシーを記述できるようにするため，ファイルごとに属性値を持たせる．ファイルの属性値というのは，

表 1 属性値とその説明
Table 1 Definition of attributes

属性名	説明
user	ファイルを所持しているユーザ名
name	ファイル名
maker	作成元もしくは作成したプログラムの名前
domain	ファイルが所属するドメイン

ファイルの名前や所持しているユーザ名などファイルにまつわる情報を利用して決定している．提案システムで現在用いている属性値を表 1 に示す．

属性値 user はそのファイルを所持しているユーザ名を表す．name はファイルの名前であり，domain はそのファイルが所属しているドメイン名を示す．インターネット上からダウンロードされたファイルの domain 属性は，ダウンロード元のドメイン名を取る．ファイルがプログラムの場合は，実行するといくつかのファイルを生成する場合がある，そのプログラムから生成されたファイルは生成元のプログラムの domain 属性を引き継ぐ．この属性値を用いて，ポリシーの条件式を記述する．図 1 の Object クラスには，”var user, maker, domain, name”という記述がある．この記述より，クラス内でポリシーの記述者が使用したい任意の属性値を宣言する．この記述もクラスの継承で子クラスに引き継がれる．Normal クラスではアクセスの主体であるプロセスと，アクセスの対象であるファイルの属性値を用いて ”allow write if subject.domain == object.domain” と条件式を記述している．これはアクセスの主体であるプロセスが所属するドメインと，アクセスされる側であるファイルが所属するドメインが同等の場合には，write の権限をもってアクセスする事を許可する，というポリシーである．subject はアクセスの主体であるプロセスを示しており，object はアクセスされるファイルを示している．プロセスやファイルに割り当てられている属性値を取得したい場合は”.”に続けて，取得したい属性値を記述する．例として”subject.domain” と記述するとそれは，実行中のプロセスが所持している domain 属性を参照する．また”object.domain”と記述するとそれは，アクセスされるファイルが所持している domain 属性を参照する．各ファイルに対する属性値の割り当ては，ある特権をもったプロセスが行う．プロセスが実行中に生成したファイルは属性値を持たない．生成したファイルに属性値を持たせるためある特権をもったプロセスが生成された新しいファイルに対して，属性値の付与を行う．

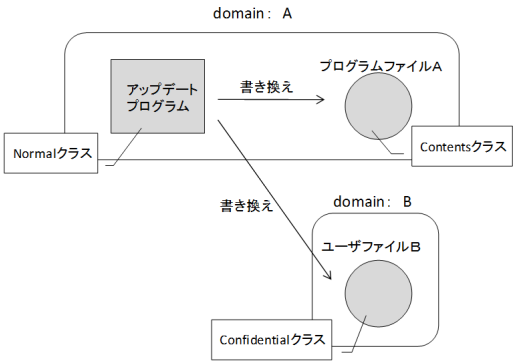

図 4 プログラムとファイルの構成図
Fig. 4 Program and file diagram

3.4 重要ファイルの保護

図 1 を用いて，ユーザの重要なファイルを保護することを考える．ファイルやプログラム構成の概略図に関して，図 4 に示す構成を例に考える．図 4 はあるアップデートプログラムが，同じドメインに所属しているプログラム設定ファイル A と別ドメインに所属しているユーザファイル B の 2 つに対して書き換えを行う際のアクセス関係を表している．この例でユーザファイル B をアップデートプログラムによる書き換えから保護したいとする．それぞれのプログラムやファイルに与える domain 属性値として，アップデートプログラムとプログラム設定ファイル A には A，ユーザファイル B には B という domain 属性値を与える．また図 1 に示したクラスの割り当てとして，アップデートプログラムに Normal クラス，プログラムファイル A に NormalContents クラス，ユーザファイル B に Confidential クラスを割り当てる．

属性値やクラスの割り当てを行った後，ポリシーに従い実際のアクセスを制限する．図の例で，アップデートプログラムが同じドメインに所属するプログラム設定ファイル A の書き換えを行う場合，その書き換えに関するアクセス権は，プログラム設定ファイル A に割り当てられている NormalContents クラス内のセキュリティポリシーによって決定される．図 1 を見ると，NormalContents クラスには”allow write if subject.domain == object.domain” という write の権限に関するセキュリティポリシーが記述されている．このセキュリティポリシーに従い，アクセスの主体側であるアップデートプログラムの domain 属性と，アクセ

```
class Executable {
  allow read any
  allow exec any
}

class Contents {
  allow read any
}
```

```
class Normal extends Executable {
  allow write if subject.domain == object.domain
}

class NormalContents extends Contents {
  allow write if subject.domain == object.domain
}

class Confidential extends Contents {
  allow write if subject.domain == object.domain
}
```

図 5 実装したセキュリティポリシー例
Fig. 5 Example of security policy implemented

スの対象であるプログラム設定ファイル A の domain 属性を比較する．先に挙げた属性値の設定によると，アップデートプログラムの domain 属性は A であり，プログラム設定ファイル A の domain 属性は A である．両者を比較した結果 subject.domain == object.domain の関係を満たすため，NormalContents クラス内のポリシーが有効になる．従ってアップデートプログラムのアクセスは許可される．

一方アップデートプログラムがユーザファイル B の書き換えを行おうとした場合，プログラム設定ファイル A のアクセス権の決定と同様にユーザファイル B に割り当てられている Confidential クラスのポリシーによってアクセス権が決まる．Confidential クラスには ”allow write if subject.domain == object.domain” という記述があるため，プログラムとファイル両者の domain 属性を比較する．その結果，条件式が成り立たないためプログラムのアクセスは許可されない．

4. 実　　装

前章で述べた提案方式に基づいて図 5 に示すポリシーを実行するための実装を行った．ファイルへの domain 属性を格納するため，記述したポリシーをカーネル側で保持させるため，システムコールの追加を行った．また属性値の取得や比較を行うためカーネルモジュールを作成した．ポリシー内で使用する属性値は表 1 とした．実装環境は以下である．

- Kernel:　Linux2.6.18-6

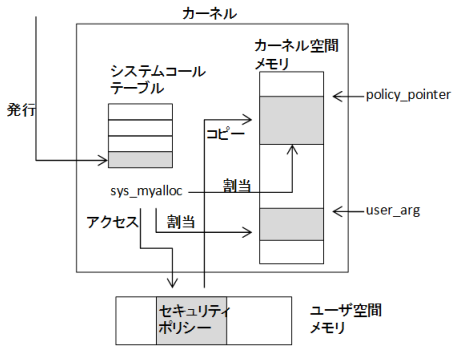

図 6 カーネル空間のメモリ図
Fig. 6 Kernel memory map

- Distribution:　Debian4.0

4.1 システムコールの追加

本実装ではシステムコールを 2 つ追加した．一つ目は，記述したポリシーをカーネル側で保持させるためのメモリと，ファイルに与える domain 属性値を保存しておくためのメモリを割り当てるシステムコールである．そのシステムコールの動作概要を図 6 に示す．追加したシステムコール `sys_myalloc` は，発行されるとカーネル空間のメモリに `policy_pointer` で参照されるセキュリティポリシーを保持しておくためのメモリ領域と，`user_arg` で参照されるファイルに与える domain 属性値を保存しておくためのメモリ領域を割り当てる．メモリ割り当て後は，ユーザ空間のメモリ上にあるセキュリティポリシーを，`policy_pointer` で参照される領域にコピーする．二つ目は，指定した domain 属性値を `user_arg` で参照されるメモリ領域にコピーするためのシステムコールである．図 7 に動作概要を示す．追加したシステムコールは `sys_attribute` である．本実装ではファイルのダウンロードに wget コマンドを使うこととした．図 7 では wget が `sys_attribute` を発行している．wget により新しく生成されたファイルに domain 属性値を付けるため，ダウンロード元のドメイン名から domain 属性値を決定し `sys_attribute` に渡すよう wget コマンドを改変した．改変した wget がこのシステムコールを発行すると，システムコールが引数に格納されている domain 属性値を `user_arg` で参照されるメモリ領域にコピーする．

4.2 カーネルモジュールの作成

提案手法を用いてアクセス制御を行う際に，記述されたポリシーをカーネル側で解釈し，

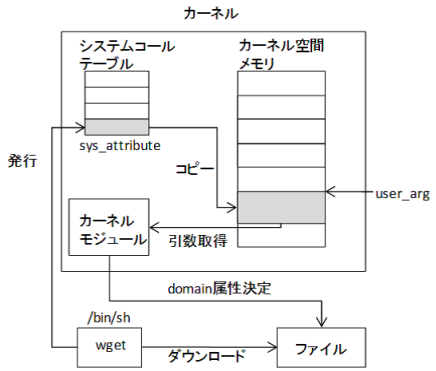

図 7 ファイルの domain 属性の決定
Fig. 7 Desiding a domain attribute for a file

プロセスとファイルの両方から取得した属性値の比較を行う必要がある．この作業を行うプログラムを LSM(Linux Security Module) を用いたカーネルモジュールとして作成した．LSM はカーネルのセキュリティ機能を拡張するために提供されているフレームワークである．カーネルモジュールが行う作業は大きく分けると以下に分類される．

- 属性値の取得や付与
- 記述されたポリシーの解析，アクセス権の決定

提案システムの属性値は，拡張属性を用いて実装した．実装した属性を表 1 に示す．それぞれのファイルやプロセスは表 1 に示す属性値を持っているが，それら属性値の中でも domain 属性は実装したカーネルモジュールを通じて与えられる．domain 属性がファイルに与えられるタイミングは，プロセスが新しくファイルを生成した時である．図 7 を用いてファイルがダウンロードされた時の domain 属性の付与の流れについて示す．wget が `sys_attribute` を発行すると，`user_arg` で参照されるメモリ領域に指定した domain 属性が格納される．カーネルモジュールはこのメモリ領域から domain 属性を取得する．取得した後 wget によって生成された新しいファイルに，domain 属性を与える．

記述されたポリシーはカーネルモジュールがアクセス権を決定する際に参照される．プロセスがファイルにアクセスをしようとした際に，カーネルモジュールが処理をフックし `policy_pointer` で参照されるセキュリティポリシーを読み込み，ポリシーに従いプロセスとファイルの両方から属性値取得する．取得した属性値をポリシーに従い比較し，その結果を用いてアクセス権を決定する．

表 2 ファイルとその属性値
Table 2 Files and attributes

ファイル	domain 属性
confidential.txt	sample.com
`fake_ocaml_3.09.2-9_i386.deb`	www.osss.cs.tsukuba.ac.jp

5. 動作実験

実装したシステムを用いて，あるアプリケーションのアップデートを行った際の動作実験を行った．ここで想定する不正アクセスは，あるアプリケーションのアップデートを装ったアップデータがユーザの重要なデータを故意に書き換える場合である．本検証では，Distribution に Debian4.0 を用いているため dpkg を想定するアップデートプログラムとして確認を行った．

表 2 に動作実験で使用するファイルとその domain 属性を示す．confidential.txt (/home/user に配置される）はユーザが所持している重要なファイルを示す．インストールするパッケージ，`fake_ocaml_3.09.2-9_i386.deb` は Debian 公式サイトから配布されている ocaml のパッケージと同等のものであるが，ocaml システムの提供とは別に，confidential.txt を書き換えようとする．クラスの割り当ては，通常の ocaml パッケージで提供されるバイナリファイルには Normal クラス，ライブラリ等のファイルは NormalContents クラスを割り当て，confidential.txt には Confidential クラスを割り当てた．従来のシステム上でパッケージをインストールすると以下の結果が得られた．

```
debian:~# cat /home/user/confidential.txt
confidential

debian:~# dpkg -i fake_ocaml_3.09.2-9_i386.deb
(Reading database ... 138505 files and directories currently installed.)
Preparing to replace ocaml 3.09.2-9 (using fake_ocaml_3.09.2-9_i386.deb) ...
Unpacking replacement ocaml ...
Setting up ocaml (3.09.2-9) ...

debian:~# cat /home/user/confidential.txt
```

```
written by fake_ocaml
```

/home/user/confidential.txt には，あらかじめ”confidential”という文字が記述されている．fake_ocaml_3.09.2-9_i386.deb をインストールすると正常にインストールが終了するが，confidential.txt は"written by fake_ocaml"という内容のファイルに書き換えられた．また，提案システム上でインストールを行うと以下の結果が得られた．

```
debian:~# cat /home/user/confidential.txt
confidential

debian:~# dpkg -i fake_ocaml_3.09.2-9_i386.deb
-su: /usr/bin/dpkg: Operation not permitted

debian:~# cat /home/user/confidential.txt
confidential
```

パッケージのインストールは，アクセスが許可されないファイルへのアクセスを含んでいるため許可されない結果になった．インストールが行われないため confidential.txt の書き換えも制限することができた．

6. 関 連 研 究

Janus[4] は記述されたセキュリティポリシーによって，サンドボックス上でコードを実行するシステムである.[2] ユーザは事前に，実行しようとしているコードが読み込んで良いファイルやネットワーク上のマシンの IP アドレスなどを指定する．それらをセキュリティポリシーとして記述し保存する．Janus はサンドボックス上のコードがシステムコールを発行するたびにその実行を止め，そのシステムコールの発行が許可されているかを逐一確認する．もし許可されていればシステムコールの発行を行い，許可されていなければエラーメッセージを返す．Janus は特定のプログラムやファイルに関してセキュリティポリシーが書けるのが特徴であるが，本研究は特定のプログラム名やファイル名を直接ポリシー内に記述しない点について相違である．

MAPbox[3] は Janus に似たシステムであるが，アプリケーションが必要とするリソースや動作を元に browsers クラス，editors クラス，filters クラスなどの behaviors クラス（振る舞いクラス）を作成する.[2] またそのクラスに応じてセキュリティポリシーの雛形を用意しており，ユーザはその雛形のパラメータを編集することによってセキュリティポリシーを記述することができる．MAPBox はセキュリティポリシーの雛型を編集してポリシーを設定できることが特徴であるが，本研究で提案するクラスの継承を用いてセキュリティポリシーを書くことはできない．

LOMAC[6] モデルでは，”group”という概念を導入して，連携して動作する一連のプロセスを”job”と呼ばれる枠組みで分類している．group の概念は本研究にあるクラスと似ている点が多い．LOMAC ではプロセスを分類するのに対し，本研究ではクラスや属性を通じてプロセスだけでなくファイルも分類される．

Chinese Wall モデル[7] では，ある特定の商業用途のアクセス制御モデルを定義し，各オブジェクトを”企業グループ”と”競合クラス”に分けている．企業グループは競合クラスに包括され，あるユーザが企業グループのデータにアクセスするとそれ以降は同競合クラスに所属する別の企業グループのデータにはアクセスできないというモデルである．Chinese Wall モデルでは，初めはユーザはどの企業グループにもアクセスできる権限を持っているが，本研究ではクラスや属性によって初めからアクセスできる範囲が限定されている点が相違である．

DTE(Domain and Type Enforcement)[8][9][10] では，実行しているプロセスを”Domain”，ファイルに対しては”Type”とよばれるグループに分けアクセス制御を行う．”Domain”にはどの”Type”を操作して良いかアクセス権の定義が付加される．それらのアクセス権はドメイン定義テーブルと呼ばれる表に明記されている．本研究では，アクセス権を明記する表を別途用意せずアクセス先のファイルに与えられているセキュリティポリシーに従う点で異なる．

RBAC(Role base access control)[11] では，必要以上より多くの権限を与えてはならないという考えのもと，Role と呼ばれる役割に基づいてアクセス制御を行う．アクセス権も Role に基づいて決定されるが，あるデータに対するアクセス権を設定するのではなく Role が行う操作そのものに制限をかける．RBAC 内ではあるサブジェクトは複数の Role を持つことが可能だが，本研究では 1 つのサブジェクトは 1 つのクラスのみを持つ点で異なる．

7. 結　　論

現在の SecureOS のセキュリティポリシーでは，複雑かつ多数の項目を設定しなければいけないこと，ファイル名などを用いてある特定のシステムに依存した記述になることなどが原因で，記述が難しい事が問題となっている．記述が難しいと間違ったポリシーを記述する可能性が高くなり，その結果ユーザの重要なファイルが破壊されるなどの不正アクセスが発

生する可能性がある．本研究では，セキュリティポリシーの記述を容易にするためにクラス階層型のセキュリティポリシーを提案した．提案手法ではポリシーの記述を簡単にするために似たようなセキュリティポリシーをクラスとして定義しまとめる．一度記述したポリシーの再利用性を高めるために，継承を用いて新しいクラスを記述できるようにする．またシステムの実行時の状態を考慮して動的にアクセス権を決定できるようにするため，更なるポリシーの再利用性の向上のため，ファイルに属性値を与えセキュリティポリシーに条件式を記述できるようにする．

提案方式に基づいてシステムコールの追加とカーネルモジュールの実装を行った．記述したポリシーをカーネル側で持たせるため，ファイルへの domain 属性を格納するため，また新しいファイルに domain 属性を与えるため，システムコールの追加を行った．記述されたセキュリティポリシーによってアクセス権を決定するため，またアクセス権の決定の際にプロセスやファイルから属性値を取得するため，LSM を用いてカーネルモジュールを作成した．

提案システムを用いたアクセス制御として，ocaml のインストールを安全に行う仕組みを示した．ocaml を装った別のパッケージである `fake_ocaml_3.09.2-9_i386.deb` は ocaml システムをインストールする他 confidential.txt を書き換える．適正なクラスをファイルに割り当て，パッケージのインストールを行ったところ．従来のシステム上では confidential.txt の書き換えが発生したのに対し，提案システム上では書き換えは発生しないことが確認された．

今後の課題としては，詳細なポリシーの設計やアクセス制御に必要な属性値の決定，クラスや属性値を割り当てるプログラムの設計やアップデート問題以外の提案方式の適用について取り組みたい．

参 考 文 献

1) Security-Enhanced Linux, http://www.nsa.gov/research/selinux/
2) 大山恵弘, ”ネイティブコードのためのサンドボックスの技術”, コンピュータソフトウェア Vol.20, No.4, pages 55-72, 2003 年 7 月.
3) A. Acharya, M. Raje, ”Mapbox: Using parameterized behavior classes to confine applications”, In Proceedings of the USENIX Security Symposium, pp. 1-17, Aug. 2000.
4) I. Goldberg, D. Wagner, R. Thomas, E. Brewer, ”A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker”, In Proceedings of the 6th USENIX security symposium, pp. 1-13, 1996.
5) 情報処理振興事業協会セキュリティセンター, ”オペレーティングシステムのアクセスコントロール機能におけるセキュリティポリシーモデル”.
6) T. Fraser, ”LOMAC: Low Water-Mark Integrity Protection for COTS Environments”, In Proceedings of the IEEE Symposium on Security and Privacy, 2000.
7) F. C. David, M. J. Nash, ”The Chinese Wall Security Policy”, The Symposium on research in security and privacy, 1-3 May 1989, OAKLAND, CALIFORNIA, pp. 206-214.
8) P. Kearns, S. Hallyn, ”Deriving Tools to Administer Domain and Type Enforcement”, Proceedings of the 15th USENIX System Administration Conference, 2001, pp 151-155.
9) K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, K. A. Oostendorp, ”Confining root programs with domain and type enforcement (DTE)”, Sixth USENIX UNIX Security Symposium, 1996.
10) L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, S. A. Haghighat, ”A Domain and Type Enforcement UNIX Prototype”, Fifth USENIX UNIX Security Symposium Proceedings, Salt Lake City, Utah, 1995.
11) D. F. Ferraiolo, D. R. Kuhn, ”Role Based Access Control”, 15th National Computer Security Conference, 1992, pp. 554-563.