平成12年度

筑波大学第三学群情報学類

卒業研究論文

題目 SunRPCの付加的なアクセス制御のシステムコールレベルでの実現

主 専 攻	情報工学
著 者 名	中田 吉法
指導教員	電子・情報工学系 板野肯三
	電子・情報工学系 新城靖
	電子・情報工学系 千葉滋

要 旨

Sun RPC は，遠隔手続き呼び出し (Remote Procedude Call) の一方式であり，NIS や NFS などのサービスにおいて，その基幹技術として利用されている．しかし SunRPC 自体は，現代のネットワーク技術に求められる十分な情報生存能力が達成されているとは言い難い．

本研究では Sun RPC にの安全性を高めることを目的として付加的なアクセス制御を実現する．これを Sun RPC に対し直接変更を加えるのではなく，システムコールに対するラッパ *SysGuard* を利用することで実現する．アクセス制御には，従来の IP アドレスやポート番号に加えて SunRPC の手続き番号などを利用できる．また、このための判定機構として仮想機械を利用する。

目 次

第1章　はじめに

Sun RPC[6] は，Sun Microsystems 社の提唱による，遠隔手続き呼び出し (Remote Procedude Call) の一方式であり，遠隔地のコンピュータ上にある手続きをネットワーク経由で呼び出すための通信規約である．

SunRPC は，その仕様が公開されたことに加え，SunRPC を応用した技術としてネットワーク透過なファイルシステムである NFS(Network File System) やローカルエリアネットワーク単位のユーザー認証に用いられる NIS(Network Information System) などでも使われており，事実上の標準技術として広く普及するに至った．特に，NFS や NIS は現在においても Unix 系 OS でのネットワーク環境構築の際には標準的な技術として，欠かすことのできない重要な位置を占めているものである．その基幹技術であるところの SunRPC も，現代のネットワーク構築において，重要な一角を占めているものであると言える．

一方で，近年インターネットが爆発的な普及を見せ，ネットワーク環境が一般的なものになりつつある．職場や学校だけでなく，自宅からもパーソナルコンピュータを使用してインターネットのサービスを利用することができるようになってきた．それに伴うように，不正アクセスの件数も増加している．

不正アクセスとは，様々なコンピュータの弱点（セキュリティホール）を利用して，利用権限のないコンピュータへのアクセスを行うことである．たとえば，コンピュータ内の情報の改竄や，パスワードファイルの取得などがそれにあたる．

このような事由から，現代のコンピュータには，ネットワークに対する十分なセキュリティ機能が要求される．しかしながら，Sun RPC はその設計時点では現代のようなネットワークの広範な普及が見込まれていなかったという背景もあり，セキュリティ面においては脆弱と判断すべきレベルの機能しか備えていない．

これに対しては，十分なセキュリティ機能を有する代替技術の登場とその普及こそが望ましいものであるが，少なくともそれが実現されるまでの期間は今後も SunRPC が広範に利用される状況が続くものであると思われる．

また，Linux などのパーソナルコンピュータ上で動く Unix 系の OS も普及しつつあるが，これらのシステムが十分なセキュリティ管理のなされないまま運用されている事例も多い．

このような背景を受け，本研究ではシステムコールレベルでアクセス制御を行うことで，SunRPC のアクセス制御機能を付加的に強化する手法を提案する．こ

れは，SunRPC の実行に関するシステムコールが発行されたときに，それが許可できるかどうかを調べるということである．

本研究ではこのようなアクセス制御を行うために，システムコールに対するラッパである *SysGuard* を利用して，これを実現する．*SysGuard* ではガード・モジュールと呼ばれるカーネル内のモジュールを利用してアクセス制御を行う．本研究での実装物は，*SysGuard* のガード・モジュールとして実装される．

本論文の構成は，以下の通りである． 2 章では，本研究のアクセス制御の対象となる SunRPC について紹介し，また SunRPC に対して適用可能な既存のアクセス制御技術についてその問題点を論じる．3 章では，本研究でのアクセス制御の実行に利用される，*SysGuard* を紹介する．4 章では，SunRPC に対するアクセス制御を実現するための仕様を述べる．5 章では，4 章に基づいてガード・モジュールの実装内容について述べる．6 章では，5 章で実装したシステムの評価と考察を行う．最後に 7 章で以上についてまとめ，結論を述べる．

第2章　関連研究

この章では，関連研究として，本研究におけるアクセス制御対象である SunRPC について紹介する．また同時に，既存の SunRPC に対するアクセス制御技術についても紹介し，その問題点を取り上げる．

2.1　SunRPC

この節では，本研究における前提として，SunRPC についてそのプロトコル仕様や技術的な背景を，本研究に関わりの深い事項に重点を置いて紹介する．

2.1.1　遠隔手続き呼び出し

遠隔手続き呼び出し (Remote Procedure Call / RPC) とは，遠隔地のコンピュータに対し，なんらかの処理を要求する際に，その処理要求全体を 1 つの手続き呼び出し (Procedure Call) とみなして，これを実行するという概念・方式である．

RPC プログラムは，呼び出し側（図 2.1 中のホスト B）をクライアント，手続き実行側（同ホスト C）をサーバとする，クライアント/サーバ モデルに基づいたネットワークプログラムである．

図 2.1 に，RPC の概念図を示す．図の上部が通常の手続き呼び出しの場合，図の下部が遠隔手続きの場合についてのことを示している．

通常の手続き呼び出しの場合，ホスト A のプログラム 1 では，procedureI() にて手続き I を呼び出す．呼び出された手続き I は，同一のホスト A にて所定の処理を行い，結果をプログラム 1 に戻す．

遠隔手続き呼び出しの場合，ホスト B のプログラム 2 では，procedureII() にて，遠隔地であるホスト C にある手続き II を呼び出す．呼び出された手続き II は，ホスト C にて所定の処理を行い，結果をホスト B のプログラム 2 に戻す．

SunRPC は，RPC の一方式であり，Sun Microsystem 社の実装によるものである．このソースコードは現在では無料で配布されている．

このほかの RPC の代表的な実装としては，Open Software Foundation の DCE RPC[2]，Object Management Group による CORBA[3]，Sun Microsystems の JavaRMI[4] などが存在する．

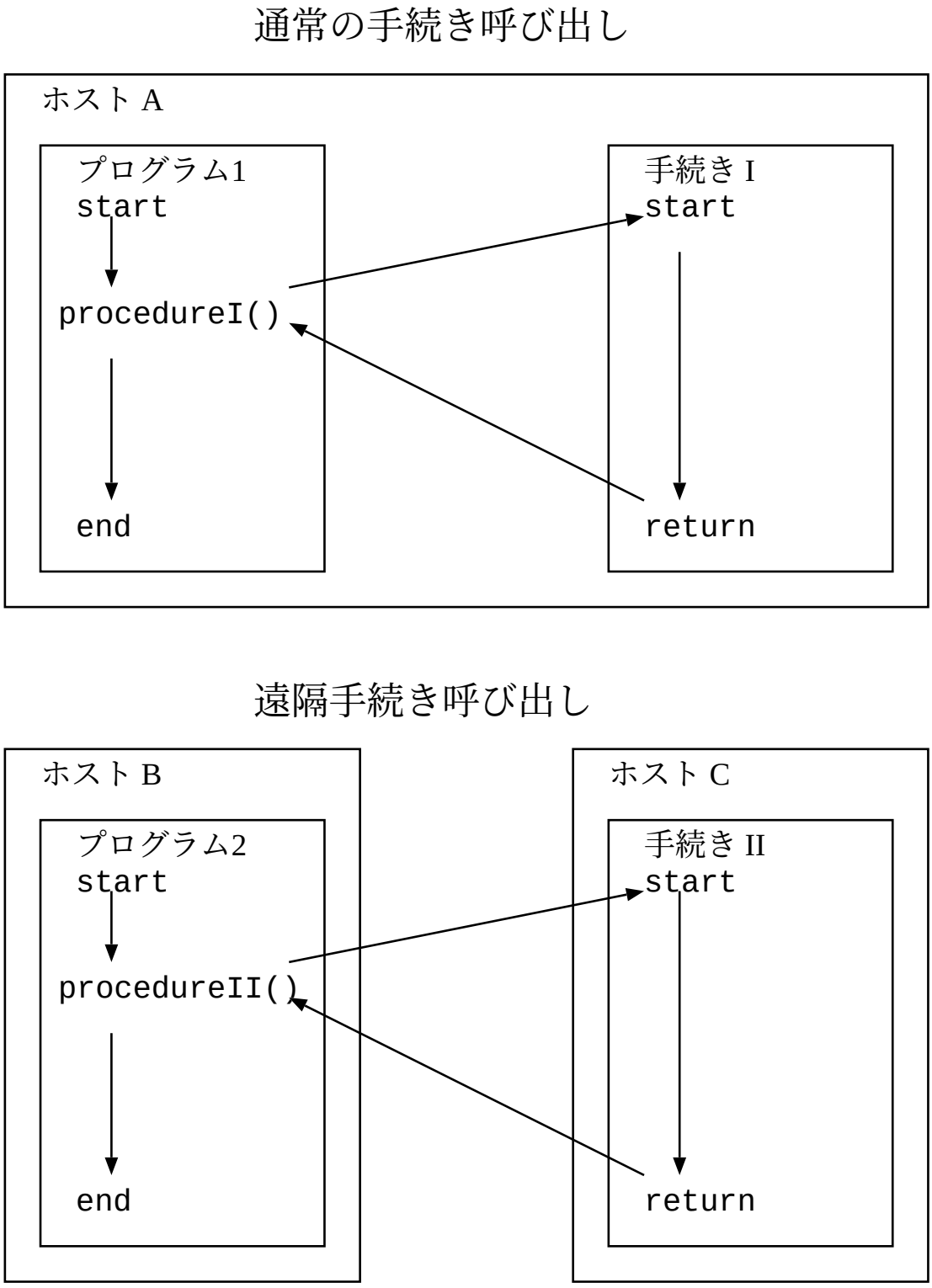

図 2.1: 手続き呼び出しと遠隔手続き呼び出し

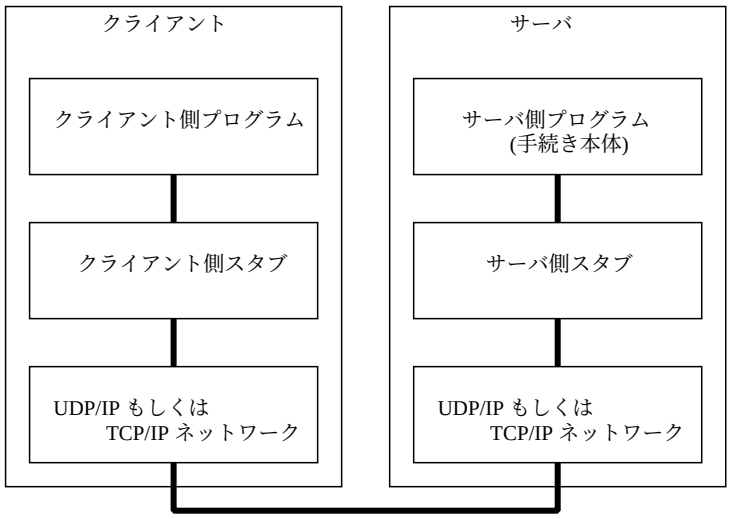

図 2.2: SunRPC の概念図

2.1.2　SunRPC の動作

実際の SunRPC による手続き呼び出しについての概念図を，図 2.2 に示す．

クライアント側プログラムでは，通常の手続き呼び出しと同様の手順で，クライアント側スタブを手続きとして実行する．

クライアント側スタブでは，渡された引数をとりまとめて，規約に基づいた呼び出しメッセージを作成し，これをサーバに送信する．送信には，UDP/IP または TCP/IP が用いられる．サーバ側スタブではメッセージを受け取るとそこから引数を取り出し，サーバ側プログラムである手続き本体を実行する．手続きの実行結果はサーバ側スタブが返信メッセージとしてまとめ，これがクライアントに返信される．実行結果を受け取ったクライアント側スタブでは，メッセージから返戻値を取り出し，これを実行結果としてクライアント側プログラムに返す．

通常，スタブ部分はインターフェース記述から `rpcgen` コマンドによって自動生成される．従って，SunRPC プログラムを作成する場合にユーザが作成する必要があるのは，サーバ，クライアントそれぞれのプログラム部分とインターフェース記述のみでよい．

サービスの特定

Sun RPC では，Sun RPC に基づいて提供される各サービスを識別するために，各サーバに，**プログラム番号**を割り当てる．また，異なるバージョンの複数のサーバを動作させるために各サーバには**バージョン番号**も割り当てられる．更に，単一のサーバが複数の機能を提供する場合のために，実行可能なそれぞれの手続きについて**プロシージャ番号**を割り当てる．

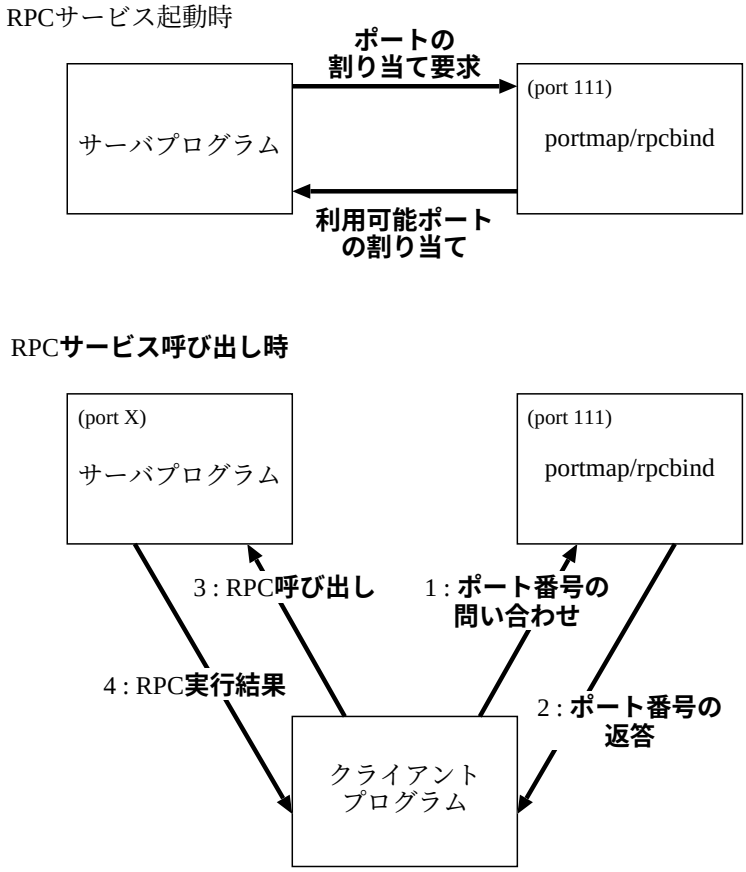

図 2.3: portmap/rpcbind を含む RPC 呼び出しの様子

SunRPC に基づくクライアント/サーバを作成する場合，プログラム制作者が取り扱うのはこれらの番号と，サーバの IP アドレス（もしくはホスト名）のみである．実際には，SunRPC のサーバプログラムは，その動作しているホストにおいて IP ポート番号を割り当てられる．通常，サーバプログラムに割り当てられるポート番号は不定である．

そこで，プログラム番号／バージョン番号で識別されるサーバと，その実際のポート番号との対応を検索するために portmap[8] と呼ばれるサービスが提供されている．また，このサービスを提供するサーバプログラムは portmapper と呼ばれる．

portmap を含む RPC 呼び出しの様子を図 2.3 に示す.

portmap は，プログラム番号 100000 番を持つ SunRPC に基づくサービスである．

portmap は SunRPC によるサービスのうち，唯一ポート番号が固定されたサービスである．具体的には，UDP/IP 及び TCP/IP のポート 111 番にて機能している．

あるサーバに存在する RPC サービスを利用する場合，クライアントはまず portmap

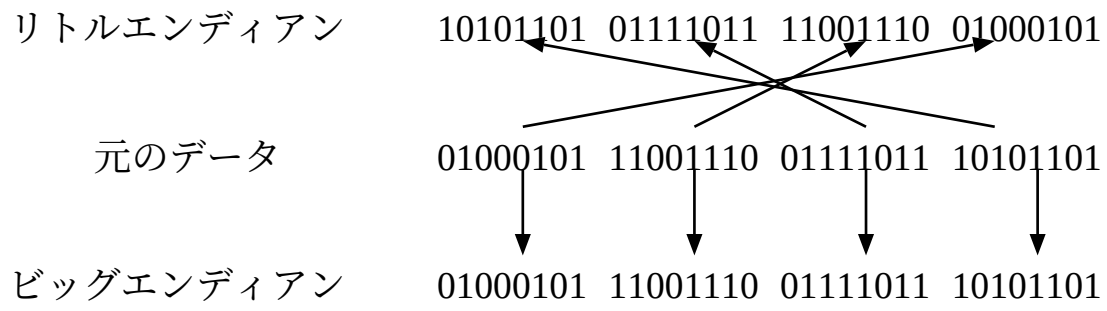

図 2.4: ビッグエンディアンとリトルエンディアン

に対して，ポート番号通知の手続きを呼び出す．このとき，呼び出したい RPC サービスの プログラム番号，バージョン番号，プロシージャ番号が引数として渡される．呼び出しを受けた portmap は，プログラム番号，及びバージョン番号からサービスを特定し，これに対応するアドレスとポート番号を返信する．クライアントは portmap の返信内容から，目的とするサービスのポート番号を知り，改めてそのポートに対して本来の呼び出し手続きを行う．

XDR

異なる機種間で相互に RPC 呼び出しを行う場合，引数の受け渡し方が問題となる．CPU の種別により，2 バイト以上の数値データのメモリ上での保持方法に異なる場合があるので，メモリ上の引数のイメージをそのまま渡すのでは不都合が生じるためである．たとえば，整数という単純なデータについても，リトルエンディアンとビッグエンディアンという 2 つの方式がある (図図 2.4)．リトルエンディアン (little endian) では 2 バイト以上の数値データを下位バイトから順に記録/送信する．ビッグエンディアン (big endian) では上位のバイトから順に記録/送信する．整数以外にも，文字コード，構造体の格納順序など，様々な点で問題が生じる要因となる．

SunRPC では，XDR(External Data Representation Standard)[7] というデータ交換フォーマットを利用することで，この問題を解決している．

SunRPC で実際にメッセージを送信する場合，引数は必ず XDR 形式に変換された上でメッセージに組み込まれる．メッセージを受け取った側は，XDR 形式のメッセージを CPU 種別に合わせた形式に変換し，ローカルのメモリ上に展開する．

なお，一般にこのようにしてデータ交換のためにデータ形式を変換することをマーシャリング，マーシャリングされたデータをローカル用の形式に変換し直すことをアンマーシャリングと呼ぶ．

SunRPC メッセージ形式

SunRPC のメッセージ形式は，XDR 形式のデータ型として定義されている．この定義は，XDR 形式のデータ定義のための言語である，XDR 言語によって書かれている．RPC メッセージの定義の抜粋を図 2.5 に示す．

XDR 言語で定義されたデータ型をマーシャリングする場合，一定の規則に基づいてデータ列へと変換される．たとえば，RPC 呼び出しメッセージの場合，図 2.6 に示したような構造のデータ列へと変換される．

2.1.3　Sun RPC のセキュリティ機能

SunRPC は，そのメッセージのヘッダ部分に認証情報を収めておくためのフィールドとして，opaque_auth 構造体が用意されている（opaque_auth 構造体の定義は前出の図 2.5 にて示した）．

opaque_auth 構造体のうち，flavor は認証方式を，body は認証情報そのものが収められるフィールドである．

認証方式は，Sun RPC 呼び出しの実行前にクライアントプログラム内で設定される．このため，認証方式を変更する場合には，クライアントのプログラムを修正する必要がある．また，認証情報の取り扱いもサーバに任されているため，新たな認証方式を取り入れる場合にはサーバでもプログラム修正が必要である．

Sun RPC で使用できる認証方式には，以下のようなものがある．

AUTH_NONE

: 無認証．body には何も記述されない（長さ 0 だけが記述される）．デフォルトでは，この方式が使用される．この場合，信用できないユーザも含め誰もがサービスを利用可能である．

AUTH_SYS

: 認証情報として，UNIX におけるユーザ情報を送付する．送付される内容は，呼び出しを行ったユーザーのクライアント側ホストにおけるユーザー ID，グループ ID，及びホスト名である．AUTH_SYS ではユーザに関する情報は入手可能であるが，送られる認証情報が信用できるものであるかを保証する機能はない．また，AUTH_SYS による認証情報そのものも偽造が容易である．

AUTH_DES

: DES 暗号方式による認証．この方式を用いた通信は，Secure RPC とも呼ばれる．公開鍵暗号と秘密鍵暗号とを組み合わせた，比較的安全度の高いユーザー認証機能を実現する．詳しくは，2.2.3 節で述べる．

```
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

enum msg_type {
    CALL  = 0,
    REPLY = 1
};

struct call_body {
    unsigned int rpcvers;       /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth  cred;
    opaque_auth  verf;
    /* procedure specific parameters start here */
};

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

図 2.5: RPC メッセージの定義 (抜粋)

xid		
cmsg	mtype (= 0)	
	rpcvers(= 1 or 2)	
	prog	
	vers	
	proc	
	cred	auth_flavor
		body
	verf	auth_flavor
		body
	（以降、サービス固有の引数） ⋮ ⋮	

図 2.6: RPC 呼び出しメッセージの XDR 形式での構造

2.2　既存の技術による SunRPC へのアクセス制御

2.2.1　IP ルータによるネットワークに対するアクセス制御

IP ルータは，LAN と外部ネットワークの接点として設置されるものであり，IP(Internet Protocol) パケットのヘッダ部分の情報を元に，パケットをどのネットワークに転送するか決める装置である．

一般に，送信元/宛先アドレス，サービスのタイプ (UDP,TCP,ICMP など) に基づく転送に関するルールリストを記述し設定することができる．

ルータには，専用の機器も存在するが，Unix マシンをその用途にあてることもできる．IPFW は，Unix 上で主に IP ルータとして使われることを想定して作られたものである．IPFW はカーネル内で IP 層に実装されている．

IPFW では，次のようなルールリストを扱う．

```
# Stop spoofing
ipfw add deny all from 192.168.1.0:255.255.255.0  to any in
# Allow access to our DNS
ipfw add pass tcp from any to 192.168.1.1 53 setup
# Allow access to our WWW
ipfw add pass tcp from any to 192.168.1.1 80 setup
# Allow DNS queries out in the world
ipfw add pas udp from any 53 to 172.16.1.1
ipfw add pass udp from 172.16.1.1 to any 53
# Everything else is denied as default.
```

このリストは，実際にはシェルスクリプトとして実行され，ipfw コマンドを通じて，各ルールを転送先決定のためのルールリストに加える．IPFW では，IP 層の情報に加え，上位レイヤである UDP や TCP の情報である，ポート番号をアクセス制御に利用できる．

IPFW を用いて SunRPC の使用を制限するには，まず，ポート番号 111 へあてられた外部からのパケットの通過を禁止して，外部からの portmapper へのアクセスを不可能にする．これにより，呼び出しに portmapper を必要とする一般的な Sun RPC プログラムを外部から実行することを阻止できる．

しかし悪意のある利用者が，portmapper を利用せずに，アクセス可能なあらゆるポートに対して呼び出しメッセージを送信してくるようなクライアントを利用する場合が考えられる．なお，このような全てのポートに対して実行を試みる攻撃法を一般にポートスキャンと呼ぶ．ポートスキャンを行う SunRPC クライアントに対しては，本節に挙げた方法では十分なアクセス制御が可能とは言えない．

これに対しては，一般的に未使用とされるポート群について，外部からのパケットの通過を禁止する．ただしこの場合，SunRPC を初めとして，これらのポートを用いたサービスの利用がまったくできなくなる．

また，IP ルータによるアクセス制御では，内部のネットワークからのアクセスに対する制御は行えない．

2.2.2　portmapper の置換によるアクセス制御

標準の portmapper を使用する場合，その動作は単純にプログラムとポートの割り当てを解決するだけであり，セキュリティの確保は各サービスのサーバで行うことになる．これに対し，通常の portmapper の代わりに，アクセス制御を機能を付加した portmapper[5] を設置することで，セキュリティを向上させることができる．

しかしこの方式では，portmapper へのアクセスは管理できるが，呼び出しメッセージそのものに対してアクセス制御を実行することはできない．従って，ある Sun RPC プログラムについて，許可/禁止の制御を行うことはできるが，プロシージャによって制御を変更するような方針を採ることはできない．また，許可/禁止の制御自体も，portmapper に依存するクライアントに対してしか有効でなく，portmapper に依存しない悪意のあるクライアントに対しては無防備である．

2.2.3　Secure RPC

Secure RPC は，Sun Microsystems 社による，SunRPC の認証方式の一実装である．Sun RPC の認証方式として AUTH_DES を使用することで利用することができる．

Secure RPC では，実際の Sun RPC 呼び出しを行う前にクライアントとサーバ

側ホストの間で暗号を利用してユーザー認証とセッション鍵の交換を行う．

Sun RPC 呼び出しのときは，このセッション鍵を用いて暗号化された認証情報を送る．これにより，送られてきたメッセージが，どこから来たものであるかについての保証を得ることができる．

しかし，Secure RPC を用いても，通信に用いる伝送路の安全は保障されない．そのため，引数やプロシージャ番号が書き換えられたような場合には対処できない．

また，Secure RPC を用いるためには，クライアント・サーバの双方を AUTH_DES を認証に用いるように設定する必要があるため，プログラムの書き換えと再コンパイルが必要になるという問題もある．

第3章　*SysGuard*

SysGuard は，アクセス制御機能を強化するための，システムコールに対するラッパである．*SysGuard* では，ガードと呼ばれるデバイスドライバ同様に簡単に組み込むことができるカーネル内モジュールを利用する．ガードは，システムコール処理の実行前後での付加的なアクセス制御を実現する．

本研究では，この *SysGuard* を利用して目的とするアクセス制御を実現した．この章では，その利用法に重点を置きながら，*SysGuard* の特徴・機能について紹介する．

3.1　ガード

ガード (guard) は，カーネル内のモジュールであり，指定されたシステムコールが実行された時に，カーネル内のシステムコールの入口及び出口から呼び出される．図 3.1 に，ガードの呼び出しの様子を示す．ガードは，システムコール本体の実行前，または，実行後において呼び出され，付加的にアクセス権をチェックする．各ガードはそれぞれ許可，もしくは拒否を返す．全てのガードが許可したものについてのみ，システムコールの本体が実行される．どれか 1 つのガードでも拒否を返した場合は，システムコールの本体の実行はされない．

SysGuard では，少数の種類の汎用のガードではなく，多数の種類の専用のガードを用いてアクセス制御を強化する．この方式の利点は，各ガードの実現が単純化される点にある．

ガードは，システム全体に対するグローバルな単位の他に，ユーザー，グループ，プロセス，などといった単位ごとに設定することができる．

ガードを操作するためには，以下のようなシステムコールをが用いる．

- `guard_create()`：ガードの生成
- `guard_destroy()`: ガードの破棄
- `guard_ctl()`：既存のガードの制御
- `guardtbl_set()`：ガード表へのガードの登録（活性化，非活性化）
- `guardtbl_get()`：ガード表の内容の取得

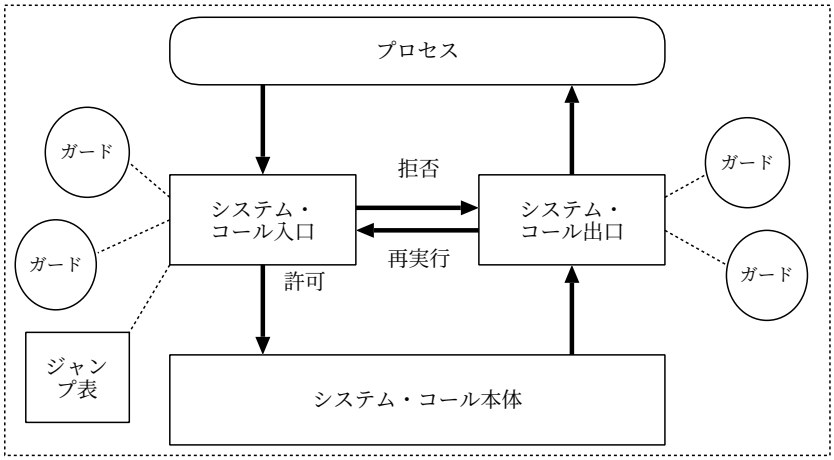

図 3.1: システムコールの入口と出口におけるガードの呼び出し

3.2　ガードの開発

ガードを実現するモジュールのことを，ガード・モジュールと呼ぶ．ガードを開発するためには，このガード・モジュールを作成しカーネルに組み込むこととなる．

3.2.1　ガード開発キット

カーネル内のモジュールであるガードの開発を容易にするために，*SysGuard* ではカード開発キットが用意されている．

ガード開発キットを用いてガードを開発する場合，次の 2 段階の手順をとる．

(1) ユーザー空間における開発

ガード開発者は，まずユーザー空間内で動作するガードを開発する．この段階では，ある特定のターゲットプログラムを，開発中のガード，およびシステムコールのスタブと共にリンクして実行する．スタブは本来のシステムコールの代わりに呼び出され，図 3.1 に示したシステムコール入口，および出口の部分に相当する処理を行う．

また，ガード開発キットでは，表 3.1 に示すような各手続きが用意されている．これらは，本来カーネル空間でのみ利用可能な手続きであるが，これにより，ガードを擬似的にユーザー空間内で動作させることができる．

(2) カーネルへの組み込み

ユーザー空間内でデバッグが完了したガードを，カーネル内に組み込む．カーネル内のガードのデバッグには，gdb のリモートデバッグ機能が利用できる．

手続き	機能
fdtosockname()	getsockname() 相当
fdtopeername()	getpeername() 相当
kmalloc(), kfree()	メモリの確保と解放
copy_from_user()	ユーザー空間からカーネル空間へのコピー（アクセス権チェック付き）

表 3.1: ガード・モジュール内で利用できる手続き

```
struct guardmod
{
  struct guardmod *next;
  int guardtype;
  char *name;
  struct guardbody *bodies; /* array */
  int (*g_create)  (struct guardid *idp,
    char *argval, size_targlen);
  int (*g_destroy) (struct guardid *idp);
  int (*g_ctl)  (struct guardid *idp,
 char *argval, size_targlen);
  int (*g_fork) (struct guardid *oldidp,
 struct guardid *newidp,
 struct task_struct *newtask);
  int (*g_exit) (struct guardid *idp);
};
```

図 3.2: guardmod 構造体の定義

これにより，ガード開発者は行単位のステップ実行含めて，一般のユーザプログラムとほぼ同じ環境でガードをデバッグすることができる．

3.2.2　guardmod 構造体

3.1 節で述べたように，*SysGuard* では多種多様のガードを実現する．それぞれのガードを管理・識別するためのデータ構造として，guardmod 構造体が用意される．

guardmod 構造体の定義を，図 3.2 に示す．

ガード・モジュールの開発は，適切な guardmod 構造体を記述することによって行われる．

guardmod 構造体に含まれる各要素について，ガード・モジュールの開発に必要

な事項と共に説明する．

guardtype guardtype は，ガードの型を示す．guardtype は，そのガードにのみ与えられる固有の整数値である．各ガードは，guardtype の値によって区別される．

name そのガード・モジュールの名前を示す．そのガードの名前である文字列へのポインタを収める．

bodies guardbody 構造体は，チェック用関数とシステムコールの関連づけを保持するための構造体である．guardbody 構造体には，そのガードが対象とするシステムコールの番号と，各システムコールに一対一で対応したアクセス権をチェックする関数へのポインタとが含まれている．

ガードを構成する際には，まず，チェック対象となる各システムコールに対応した関数を作成する．たとえば，connect() システムコールの実行前に呼び出される関数であれば，before_connect() のようなものを作成する．

次に，チェックを行う関数と，各システムコールを関連づけるために，guardbody 構造体の配列として両者の対応を記述し，その配列へのポインタを bodies に収める．

g_create, g_destroy, g_ctl, g_fork,g_exit

ガードが動作するのに必要な関数へのポインタである．

g_create, g_destroy, g_ctl は，それぞれguard_create(), guard_destroy(), guard_ctl() の各システムコールに対応した関数へのポインタで，これらのシステムコールの発行時に呼び出される．g_fork および g_exit は，fork() システムコールによるプロセスの複製時，および，exit() や kill() システムコールによるプロセスの終了時に呼び出される関数へのポインタである．

ガード・モジュールを構成する際には，g_create(), g_destroy(), g_ctl(), g_fork(),g_exit() の各関数を実装する必要がある．

3.2.3　ガード・モジュールの登録

各ガード・モジュールを容易にシステムに組み込むために，Linux の他のモジュール（デバイス，実行形式など）に倣った，次のようなカーネル内の手続きが用意されている．

```
int register_guardmod( struct guardmod *module )
```

register_guardmod() は，引数として与えられた module を，システムによって管理される guardmod 構造体のリストに登録する．

新たなガード作成した場合，通常は，register_guardmod() による登録手続きを，起動時手順の一部となっているガード・モジュール登録部分に記述する．

3.2.4　ユーザインタフェースの作成

完成したガード・モジュールを実際に動作させるにはガードの生成やガード表への登録が必要である．そこで，guard_create() や guardtbl_get() などを用いて，ガードを使用可能な状態にするためのユーザインタフェースの作成が求められる．

第4章　*SysGuard*による アクセス制御の付加

第 3 章では，システムコールのレベルでアクセス制御を付加することのできるラッパとして *SysGuard* を紹介した．以下では，実際に SunRPC 呼び出しに対し，アクセス制御を行うガードとして，SunRPC ガード・モジュールを構成していく．この章では，SunRPC ガード・モジュールを構成するにあたっての方針および仕様を述べる．

4.1　Sun RPC呼び出しの捕捉

この節では，Sun RPC にセキュリティチェックを行うには，どのシステムコールに対し，どのようなチェックを加えればよいかについて述べる．

SunRPC のライブラリでは，UDP/IP による送信の場合には `sendto()` システムコール及び `recvfrom()` システムコールが，TCP/IP による送信の場合には `write()` システムコール及び `read()` システムコールが用いられている．具体的には，これらのシステムコールについて，クライアント側ではその実行直前に，サーバ側ではその実行直後に *SysGuard* によってその実行を中断させ，アクセス制御に関するチェックを行うようにすればよい．

SysGuard でこれを実現するには，`before_sendto()`，`after_recvfrom()`，`before_write()`，`after_read()` の各手続きを実装し，それぞれを `sendto()`，`recvfrom()`，`write()`，`read()` と関連づけることで達成される．

ただし，これらのシステムコールのすべてが SunRPC のメッセージを送受信にに使われるわけではない．これらのシステムコールは，UDP/IP や TCP/IP における通信時に，UNIX で標準的に用いられるシステムコールである．また，`write()` はファイルやコンソールなど，各種ストリームに対する書き出し/読み込みの際に一般的に用いられるシステムコールでもある．

従って，SunRPC の送信に対するアクセス制御チェックを行う前に，以下のようなチェックを行うことが求められる．

- 送出されようとしているメッセージが SunRPC のメッセージであるかどうか

- `write()` システムコールの場合，TCP/IP ストリームに対する書き込みであるかどうか

以下，これらのチェック項目の実現の方法について述べる．

4.1.1　Sun RPC メッセージの選別

システムコールが捕捉されたときに，SunRPC 呼び出しとは関係のないメッセージ送信についてこれを許可するためには，送信されようとしているメッセージが，SunRPC 呼び出しのものであるかどうかを判定して，その結果に従った動作をすればよい．

しかし，送信されようとしているメッセージが，あるいは受信されたメッセージが，Sun RPC 呼び出しのものであるかどうかを一意に決定できる情報は，存在しない．

各システムコールの引数にはそのような情報は含まれていない．また，多くのプロトコルで判別に用いることのできる通信先ポート番号が判明しても，SunRPC では，サーバプログラムのポート番号は，一部の例外を除けば常に不定である．この他，メッセージ中のいかなる情報を用いても，確実と呼べる判別は不可能である．

これに対する方策としては，以下の 2 つの方針が考えられる．

- portmapper から，SunRPC プログラムのために用いられているポート番号の一覧を取得し，該当するポートに対してアクセス制御を行う
- 全ての送信されようとしているメッセージに対し，それが SunRPC のメッセージに見られる特徴を含んでいるかどうかを調べる

このうち，本研究では後者の方法を用いることとした．具体的には，2.1.2 節で示した SunRPC メッセージヘッダのうち以下のフィールドを判別に用いる．

- RPC メッセージ種別 (`mtype`)

  RPC メッセージ種別は，呼び出しメッセージと返答メッセージの 2 種類がある．呼び出しに対しアクセス制御を行う場合，0(呼び出しメッセージ) のみを取る．返答メッセージに拡張する場合，1(返答メッセージ) も取りうる．

- RPC プロトコルバージョン (`rpcvers`)

  RPC プロトコルバージョンは，現在のところ 1 もしくは 2 である．

  将来的に SunRPC の仕様が拡張されることも考えられるが，その場合に本方式をそのまま適用するのは不適当であると思われるので，考慮しないこととする．

なお，この方式による誤動作の危険性については 6.3.3 節で論じる．

4.1.2　`write()` における，TCP/IP ストリームの選別

UNIX システムにおいて `write()` システムコールが用いられる頻度は非常に高い．たとえばコンソールやファイルに対する出力にも `write()` システムコールが用いられている．だが，仮に全ての `write()` システムコールに対し Sun RPC メッセージの選別チェックが行われた場合でも，大半は SunRPC のメッセージでないと判別されるものと予想される．従って，本節で述べる `write()` についての非 TCP/IP ストリーム選別を行わない場合でも，誤動作する可能性は十分に低い．

むしろ問題となるのは，そのチェックの実行回数の多さそのものにある．特に *SysGuard* には，カーネル空間で動作するものであるため，ユーザー空間にあるメッセージの内容を見るためには，ユーザー空間からカーネル空間へとメモリ内容をコピーしてから見なければならない，という制約がある．

実際に全ての `write()` システムコールに対してそれが TCP/IP ストリームであると仮定してチェックを行う場合，メモリ内容をコピーするコストが必要となる．`write()` システムコールの発行頻度は高いので，コストが微少なものであっても全体としては大きな影響が発生することになる．

そこで，これに対して以下に述べるような対策を行う．

- `connect()` システムコールの直前でも実行を中断させ，ネットワーク通信に用いられるソケットについてそのファイル記述子を保持しておく．
- 実際に `write()` システムコールを中断させた場合には，まずその書き込み対象となっているファイル記述子が，ネットワーク通信に用いられているものであるかどうかを確認する．もしもネットワーク通信に用いられている場合には UDP/IP での `sendto()` システムコールと同様のチェックを行う，そうでない場合には，それ以上の処理を行わずに実行を再開させる．

この方式を採る場合，新たにオーバーヘッドが増えることになるが，`write()` のたびに前述のオーバヘッドがかかるよりは，十分に小さなもので済む．

また更に，この方式を応用すれば，一度 SunRPC のメッセージ送信でないと判明した TCP/IP ストリームについても，余計なオーバーヘッドを低減させることが可能となる．具体的には，SunRPC に用いられていないと判明した TCP/IP ストリームのファイル記述子を，リストから削除すればよい．リストから削除することによって，以後当該 TCP/IP ストリームは非 TCP/IP ストリームと同様に扱われるようになる．そのため，そのストリームについてのメッセージのメモリ転送が行われるのは最初の 1 回のみで済む．

ただし，以上の方策は，*SysGuard* でまだ未実装であるファイルディスクリプタに対するガードが実装されるまでの，過渡的なものである．*SysGuard* の側でファイルディスクリプタに対するガードの機能が実装された場合には，以上の方策に相当する処理は *SysGuard* の側で行われることになる．

4.2　アクセス制御のための情報取得

アクセス制御のための判定を行うためには，判定の基準となる情報を取得しておく必要がある．この節では，*SysGuard* によって中断されたシステムコールの引数から，必要な情報を取得する方法について述べる．

4.2.1　呼び出しメッセージからの情報の取得

呼び出しメッセージ中から，セキュリティチェックの対象とするフィールドを取り出す際の方針について述べる．2.1.2 節で述べたように，SunRPC のメッセージは，XDR 形式を取っている．XDR 形式にマーシャリングされたメッセージを，ホストに適した形式に復元するには，通常 rpcgen で生成される関数を用いて行う．

しかし，SunRPC にて XDR 形式にマーシャリングされたメッセージの全体を解読するには，実行されるプログラムに固有の引数についての情報が必要となる．SunRPC のメッセージそのものには，引数についての情報は記述されておらず，想定される全てのプログラムについて，デコードするための関数を用意しておく必要がある．そのような方針で構成を行えば，モジュールが肥大化するし，メッセージの解読にかかるコストも大きくなる．

そこで，全呼び出しメッセージに共通する部分のうち，必要最低限のフィールドについてのみ解読を行うこととして，そのための簡単な実装だけを行うこととする．

全ての呼び出しメッセージに共通するフィールドとしては，以下のものが挙げられる．

- **xid（呼び出し固有の id）**
- **RPC メッセージ種別**
- **RPC プロトコルバージョン**
- **プログラム番号**
- **プロシージャ番号**
- **バージョン番号**

- **認証情報**

このうち，RPC メッセージ種別と RPC プロトコルバージョンは 4.1.1 節で述べたように全メッセージからの Sun RPC メッセージの選別のために用いる．

残りのフィールドのうち，**xid** は，呼び出し毎に無作為に変動する値である．この値は，呼び出し／返答の全メッセージを継続的に調査するような場合には有用である．しかし，今回のように呼び出しを事前に差し止めるという目的のみの場合には重要な情報とはなりえない．
また，**認証情報**は 2.1.3 節で述べたように，サーバやクライアントの設定に依存したユーザー認証に関する情報が含まれている．しかし，設定によって収められている内容もまちまちであるため，アクセス制御に用いる情報としては適切でない．

以上を踏まえ，アクセス制御に用いるフィールドとしては，**プログラム番号**，**プロシージャ番号**，**バージョン番号**の各フィールドを用いることとする．これらのフィールドを取得し判定に用いることができれば，既知の SunRPC サービスのうち，危険性を含むものを判別することが可能である．これに，SunRPC メッセージの選別のために用いる，RPC メッセージ種別と RPC プロトコルバージョンとを加えた各フィールドを，呼び出しメッセージのヘッダから取り出す．

4.2.2　パケット外からの情報の取得

また，アクセス制御に用いる情報として，接続先の IP アドレス，ポート番号を利用する．

4.3　アクセス制御リストの保持方式

この節では，実際のアクセス制御チェックを行うための，アクセス制御リストの保持方法について述べる．

4.3.1　アクセス制御リスト

アクセス制御リストを，仮想機械のコードとして保持することとした．具体的には，アクセス制御リストの 1 要素を，簡単な仮想機械に対する命令コード列として記述する．この概念図を，図 4.1 に示す．
本研究で提案する，アクセス制御リストとしての仮想機械コードは，and，or などの構文を持たない非常に単純なものである．この方式を採る利点として，以下のものを挙げる．

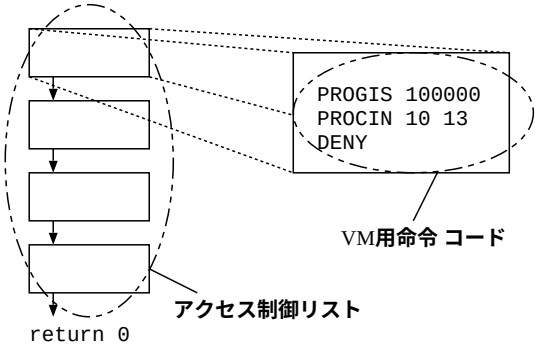

図 4.1: 本方式におけるアクセス制御リストの概念図

- **仮想機械の単純化**

  and，or などの構文に対応する場合，仮想機械をスタックマシンとして設計する必要がある．この場合，記述の表現能力は向上するが，スタック領域の確保とスタック管理のための処理が必要となる．

  だが，これらの構文を持たなければ，仮想機械自体の状態が非常に単純化される．具体的には，プログラムカウンタ以外の状態を持たないレジスタマシンとして構成することが可能になる．これは速度の点からも，あるいは複雑な設計自体がセキュリティホールの原因となりうる点から考慮しても，優位な点であると考える．

- **機能拡張の容易性**

  判定項目を追加する場合，データ構造そのものを変化させる必要がなく，判定用情報取得部分と，仮想機械部分に若干の変更を施すだけで済む．その場合にも，仮想機械そのものの構成は単純なままであり，セキュリティホールの要因とはなり難い．

仮想機械が備える命令として，以下のものを実装する．

- 条件命令

  対象メッセージに関する情報（ヘッダ情報，通信先情報）のいずれかを対象として，引数との比較を行う．比較に使える条件としては次のものを用意する．

  – 同値
    { 対象値 = 引数} の場合に真となる．
  – 非同値
    {対象値 ≠ 引数 } の場合に真となる．用

- 範囲
  {引数 1 < 対象値 < 引数 2} の場合に真となる．

結果が真であった場合，次の命令の評価を行う．

結果が偽であった場合，仮想機械は NULL 値を返して動作を終了する．else に相当する構文は持たない．

具体的には，条件命令として対象・比較条件別に，以下の各命令を実装する．

- 対象：プログラム番号
  `PROGEQ`, `PROGNE`, `PROGIN`
- 対象：バージョン番号
  `VERSEQ`, `VERSNE`, `VERSIN`
- 対象：プロシージャ番号
  `PROCEQ`, `PROCNE`, `PROCIN`
- 対象：通信先 IP アドレス
  `IPADDREQ`, `IPADDRNE`, `IPADDRIN`
- 対象：通信先ポート番号
  `IPPORTEQ`, `IPPORTNE`, `IPPORTIN`

- PASS

  この命令を評価した場合，仮想機械部は PASS(許可) 値を返して動作を終了する．

- DENY

  この命令を評価した場合，仮想機械部は DENY(不許可) 値を返して動作を終了する．

4.4　ユーザインターフェース

SunRPC ガード･モジュールの，ユーザインタフェースを作る補助として，`rpcgvmc` を作成する．

ユーザは，`guard_create` や `guard_ctl` 等のシステムコールを使って SunRPC ガード・モジュールを操作するための独自のコマンドを用意することもできる．

4.4.1　コマンドの構文

rpcgvmc コマンドの構文を以下に示す.

rpcgvmc *FILE*.rpcg

rpcgvmc は，*FILE*.rpcg に記述された仮想機械用のソースコードから，SunRPC ガード・モジュールにアクセス制御リストとして登録する C 言語の関数を生成する.

出力は，*FILE*.c というファイルに対して行われる.

出力されたファイルは，SunRPC ガード・モジュールに関する定数値などを定義した，guard_rpc.h ファイルと共にインクルードして使用する.

4.4.2　ルールファイルの記述

FILE で指定したファイルには，仮想機械用プログラムコードの形で，アクセス制御リストを記述する.

- **開始行**

  一つのルールは，次のいずれかの行から始まる.

  clientrule:　*label*　serverrule:　*label*

  前者はクライアント側での制御の場合、後者はサーバ側での制御の場合に用いる。ただし，*label* は任意の文字列である．出力される関数の名称は addrule_*label*() となる.

- **終了行**

  一つのルールは，次に示すどちらかの行で終わる.

  deny

  pass

  これは，開始行と終了行に挟まれた条件に適合するときの結果を指定する．deny の場合，その呼び出しは不許可とされ，RPC は失敗する．pass の場合，その呼び出しは許可とされ，通常通りに RPC が実行される.

- **条件**

  開始行と終了行の間には，条件文を記述する．条件文は，4.3.1 節で示した各条件命令に対応する.

– 対象：プログラム番号

PROGNUM には，プログラム番号を指定する．

* `progeq` *PROGNUM*
  PROGNUM とプログラム番号が一致する．
* `progne` *PROGNUM*
  PROGNUM とプログラム番号が一致しない．
* `progin` *PROGNUM1*,*PROGNUM2*
  プログラム番号が，*PROGNUM1* と *2*で指定された範囲の中にある．

– 対象：バージョン番号

VERSION には，プログラム番号を指定する．

* `verseq` *VERSION*
  バージョン番号と *VERSION* が一致する．
* `versne` *VERSION*
  バージョン番号と *VERSION* が一致しない．
* `versin` *VERSION*`1`,*VERSION2*
  バージョン番号が，と *VERSION1* と *2*で指定された範囲の中にある．

– 対象：プロシージャ番号

PROCNUM には，プロシージャ番号を指定する．

* `proceq` *PROCNUM*
  プロシージャ番号と *PROCNUM* が一致する．
* `procne` *PROCNUM*
  プロシージャ番号と *PROCNUM* が一致しない．
* `procin` *PROCNUM1*,*PROCNUM2*
  プロシージャ番号が *PROCNUM1* と *2*で指定された範囲の中にある．

– 対象：通信先 IP アドレス

IPADDR には，IP アドレスを指定する．IP アドレスの指定には，”192.168.1.1” のような数値による指定と，”hostname” のようなホスト名の記述による指定とが使える．ただし，ホスト名からアクセス制御リスト内の定数への変換は，C ソースの生成時に行なうため，ホストのアドレスが変化した場合には対応できない．

– `ipaddreq` *IPADDR*

IP アドレスと IPADDR が一致する．

– `ipaddrne` *IPADDR*

IP アドレスと *IPADDR* が一致しない．

  - `ipaddrin` *IPADDR1*,*IPADDR2*
    IP アドレスが，*IPADDR1* と *2* で指定した範囲の中にある．

- 対象：通信先ポート番号
  IPPORT には，IP ポート番号を指定する．

  - `ipporteq` *IPPORT*
    IP ポート番号と *IPPORT* が一致する．
  - `ipportne` *IPPORT*
    IP ポート番号と *IPPORT* が一致しない．
  - `ipportin` *IPPORT1*,*IPPORT2*
    IP ポート番号が，*IPPORT1* と *2* で指定した範囲の中にある．

4.4.3　実際の利用例

`rpcgvmc` の実際の利用例については，6.1 節で示す．

第5章　実装

第4章では，SunRPCに対する付加的なアクセス制御を行うための実現方針や仕様について述べた．この章では，第4章に基づき，SunRPCに対する付加的なアクセス制御を実現するガード・モジュールであるSunRPCガードについて，実際にどのような実装を行ったかを述べる．

5.1　実装内容

SunRPCガードは，図5.1に示したような構成を持つ．

rpcguard構造体，rpc_aclist構造体，rpc_fdlist構造体は，SunRPCガード・モジュールの動作に必要な情報が収めるためのデータ構造である．また，rpc_cmsg構造体，rpc_env構造体は，チェック対象のSunRPC呼び出しに関する情報を収めるためのデータ構造である．これらのデータ構造には，それぞれ操作用の関数が用意されている．詳細は，5.3節で述べる．

before_sendto(), after_recvfrom() before_write() , after_read(), before_connect(), は，*SysGuard*の機能によって，それぞれsendto(), recvfrom(), write(), read(), connect()の実行前に呼び出されるチェック用の関数である．詳細は，5.4節で述べる．

rpc_check()，rpc_check_vm() は，チェック対象のメッセージとアクセス制御リストとを照合する，制御判定の実行部である．詳細は，5.5節で述べる．

rpcgvmcは，SunRPCガードを利用するためのユーザインタフェースを作成する補助となるスタブである．詳細は，5.6節で述べる．

5.2　環境

本研究において，実装を行った環境は以下の通りである．

機体: VMware 2.0.3 build-799

OS: Linux

カーネル: バージョン 2.2.16 に，*SysGuard*対応を行ったもの

ディストリビューション: Kondara MNU/Linux 1.1

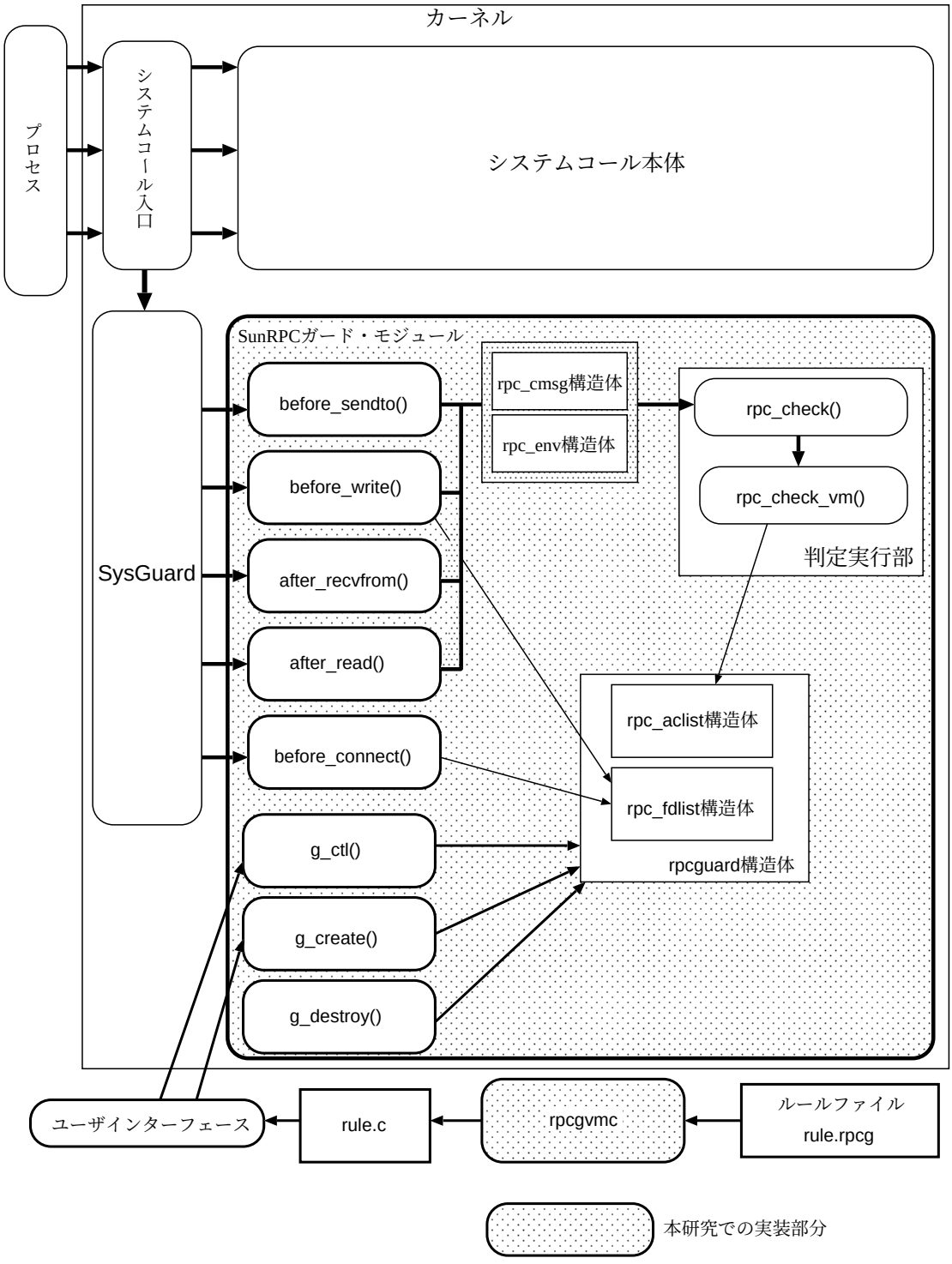

図 5.1: SunRPC ガードの構成

```
struct rpcguard{
  struct rpc_aclist *aclist;
  struct rpc_fdlist *fdlist;
};
```

図 5.2: rpcguard 構造体

なお，VMware を動作させた実機の環境は以下の通りである．

OS: Linux

CPU: Intel Pentium

カーネル: バージョン 2.2.16

ディストリビューション: Red Hat Linux 6.2

5.3　データ構造

この節では，本研究での実装にあたり使用された各データ構造と，それを操作するために実装した単純な関数について述べる．

5.3.1　rpcguard 構造体

rpcguard 構造体の定義を，図 5.2 に示す．rpcguard 構造体は，SunRPC ガードが動作するために必要な情報を含むデータ構造である．通常，rpcguard 構造体へのポインタは，guardid 構造体 idp の work 要素に収められる．

SunRPC ガード内の手続きが rpcguard 構造体に含まれる情報を必要とした場合，idp->work に保持されている rpcguard 構造体へのポインタから情報を引き出す．

rpcguard 構造体は，次の 2 つの関数により，割り当てと開放が行われる．

(1) initial_rpcguard()

initial_guardid() は，次に示す引数を取る．

idp guardid 構造体へのポインタ

initial_rpcguard() は，新たな rpcguard 構造体のための領域を確保し，内容を初期化し，そのポインタを idp->work へ収める．

```
struct rpc_cmsg{
  u_int mtype;
  u_int rpcvers;
  u_int prog;
  u_int vers;
  u_int proc;
};
```

図 5.3: rpc_cmsg 構造体

(2) free_rpcguard()

free_guardid() は，次に示す引数を取る．

idp guardid 構造体へのポインタ

free_rpcguard() は，idp->work が指す rpcguard 構造体として割り当てられたメモリ領域を解放する．またその際に，rpcguard 構造体が含む aclist，fdlist について，その領域を解放するため free_aclist()，free_fdlist() の各手続きを呼び出す．

5.3.2　rpc_cmsg 構造体

rpc_cmsg 構造体の定義を図 5.3 に示す．

rpc_cmsg 構造体は，アクセス制御の対象となったメッセージについて，その SunRPC 呼び出しメッセージのヘッダ部分に相当する領域の内容を収めるためのものである．

rpc_cmsg 構造体を操作するための関数として，以下のものを実装した．

(1) decode_rpc_cmsg()

decode_rpc_cmsg() は，次に示す引数を取る．

cmsg rpc_cmsg 構造体へのポインタ

msg SunRPC 呼び出しメッセージの先頭部分へのポインタ

decode_rpc_cmsg() は，msg の指す SunRPC 呼び出しメッセージから，そのヘッダ部分の rpc_cmsg 構造体の要素に対応するフィールドを，cmsg の指す領域にコピーする．コピーの際には，XDR 形式 (= ネットワークバイトオーダー) からホストバイトオーダーへの変換も行う．

```
struct rpc_env{
  unsigned int addr;
  unsigned int port;
};
```

図 5.4: rpc_env 構造体

```
struct rpc_aclist{
  struct rpc_aclist *next;
  int op;
};
```

図 5.5: rpc_aclist 構造体

5.3.3　rpc_env 構造体

rpc_env 構造体の定義を図 5.4 に示す．rpc_env 構造体は，アクセス制御の対象となった通信について，その環境に関する情報を収めるための構造体である．

rpc_env 構造体を操作するための関数として，以下のものを実装した．

(1) pack_rpc_env()

pack_rpc_env() は，次に示す引数を取る．

env rpc_env 構造体へのポインタ

addr sockaddr_in 構造体へのポインタ

tolen addr の指す内容の長さ

pack_rpc_env() は，addr の指す sockaddr_in 構造体から，IP アドレスとポート番号を取り出し，env の指す領域にコピーする．コピーの際には，ネットワークバイトオーダーからホストバイトオーダーへの変換も行う．

5.3.4　rpc_aclist 構造体

rpc_aclist 構造体の定義を図 5.5 に示す．rpc_aclist 構造体は，アクセス制御リストを収めておくためのリストである．

rpc_aclist 構造体を操作するための関数として，以下のものを実装した．

```
struct rpc_fdlist{
  struct rpc_fdlist *next;
  int fd;
};
```

図 5.6: rpc_fdlist 構造体

(1) add_aclist()

add_aclist() は，次に示す引数を取る．

idp guardid 構造体へのポインタ

code アクセス制御リストに追加する仮想機械コード

leng code の指す命令コードの長さ

add_aclist() は，guardid 構造体が含むアクセス制御リストの末尾に新たなリストを追加し，追加されたリストに code の指す内容をコピーする．

(2) free_aclist()

free_aclist() は，次に示す引数を取る．

idp guardid 構造体へのポインタ

free_aclist() は，guardid 構造体が含むアクセス制御リストについて，その全要素のメモリ領域を解放する．

5.3.5　rpc_fdlist 構造体

rpc_fdlist 構造体の定義を図 5.6 に示す．rpc_fdlist 構造体は，SunRPC ガードのチェック対象となるソケットについてのファイルディスクリプタのリストである．

rpc_fdlist 構造体を操作するための関数として，以下のものを実装した．

(1) add_fdlist()

add_fdlist() は，次に示す引数を取る．

idp guardid 構造体へのポインタ

fd ファイルディスクリプタ

add_fdlist() はguardid 構造体が含む対象ファイルディスクリプタのリストに，fd を追加する．

(2) `del_fdlist()`

`del_fdlist()` は，次に示す引数を取る．

idp `guardid` 構造体へのポインタ

fd ファイルディスクリプタ

`del_fdlist()` は `guardid` 構造体が含む対象ファイルディスクリプタのリストから，`fd` を削除する．

(3) `find_fdlist()`

`find_fdlist()` は，次に示す引数を取る．

idp `guardid` 構造体へのポインタ

fd ファイルディスクリプタ

`find_fdlist()` は `guardid` 構造体が含む対象ファイルディスクリプタのリストに，`fd` が含まれているかどうかを調べる．
真であった場合には 0 を，偽であった場合はそれ以外を返す．

(4) `free_fdlist()`

`free_fdlist()` は，次に示す引数を取る．

idp `guardid` 構造体へのポインタ

`free_aclist()` は，`guardid` 構造体が含む対象ファイルディスクリプタのリストについて，その全要素のメモリ領域を解放する．

5.4　関連システムコールの捕捉

関連システムコールの捕捉は，*SysGuard* の機能によって実現する．SunRPC ガードでは，図 5.7 に示したように `guardbody` 構造体を定義した．
これにより，SunRPC ガードが機能している場合には，`before_sendto()`，`after_recvfrom()`，`before_write()`，`after_read()`，`before_connect()` の各関数が，それぞれ対応するシステムコールの実行直前に呼び出される．

5.4.1　`before_sendto()`

プログラムリストを図 5.8 に示す．`before_sendto()` は，`sendto()` システムコールの引数から，必要な情報を取り出し，`rpc_check()` に渡す．

```
static struct guardbody bodies [] =
{
  {
    __NR_SENDTO  | GUARD_BEFORE,
    before_sendto
  },
  {
    __NR_RECVFROM  | GUARD_AFTER,
    after_recvfrom
  },
  {
    __NR_write   | GUARD_BEFORE,
    before_write
  },
  {
    __NR_read   | GUARD_AFTER,
    after_read
  },
  {
    __NR_CONNECT | GUARD_BEFORE,
    before_connect
  },
  {
    0
  }
};
```

図 5.7: SunRPC ガードにおける guardbody 構造体

```
int
before_sendto (struct guardid *idp,
       int fd, const void *msg, int len, unsigned int flags,
       const struct sockaddr *to, int tolen)
{
  struct rpcg_env env;

  pack_rpcg_env(&env,to,tolen);
  if(rpc_check(idp,msg,len,env) == DENY)
    return -EACCES;
  else
    return GUARD_ALLOW;
}
```

図 5.8: プログラムリスト：before_sendto()

5.4.2　after_recvfrom()

プログラムリストを図5.9に示す．after_recvfrom() は，recvfrom() システムコールの引数から，必要な情報を取り出し，rpc_check() に渡す．

5.4.3　before_write()

プログラムリストを図 5.10 に示す．before_write() はまず，対象のファイルディスクリプタがSunRPC ガードによるチェックの対象となっているかどうかを，find_fdlist() により判定する．

チェックの対象となっていない場合は実行許可を返す．チェックの対象となっている場合 before_write_body() を実行して，チェックを継続する．

before_write_body() では，sendto() システムコールの引数から必要な情報を取り出し，rpc_check() に渡す．

その後，rpc_check() に返戻値に応じた結果を返すが，rpc_check() の返戻値が NOT_RPCCALL だった場合にはそのファイルディスクリプタをチェックの対象外とするために del_fdlist() を実行する．

ただし，write() の場合，次のような注意点がある．TCP/IP で SunRPC メッセージ送信を行う場合，メッセージの先頭に，メッセージ長を記録した 4 バイトのフィールドが付加される．従って，write() によるチェックの場合には，メッセージのヘッダ部分としては，この先頭部分のフィールドをスキップした，データ列の 4byte 目を起点とした 24bytes の領域をコピーする必要がある．

```
int
after_recvfrom (struct guardid *idp,int result,
        int fd, const void *msg, int len, unsigned int flags,
        const struct sockaddr *from, int fromlen)
{
  struct rpc_env env;

  if(result < 0)
    return result;

  pack_rpc_env(&env,(struct sockaddr_in *) from,fromlen);
  if(rpc_check(idp,msg,len,env) == DENY){
    return -EACCES;
  }else{
    return GUARD_ALLOW;
  }
}
```

図 5.9: プログラムリスト：after_recvfrom()

5.4.4　after_read()

プログラムリストを図5.11に示す．after_read()は，read()システムコールの引数から，必要な情報を取り出し，rpc_check()に渡す．

5.4.5　before_connect()

プログラムリストを図5.12に示す．新たにTCP/IPの接続先とされたソケットを，チェックの対象とするために，add_fdlist()を実行する．

5.4.6　通信先の取得

通信先のIPアドレスとポート番号を取得する方法はUDP/IPの場合とTCP/IPの場合とで異なる．

UDP/IPの場合，すなわちsendto()もしくはrecvfrom()を捕捉した場合には，第4引数としてsockaddr構造体へのポインタが渡されている．このポインタはUDP/IPの場合，sockaddr_in構造体へのポインタである．sockaddr_in構造体の定義を図5.13に示しておく．

ポインタが示す先はユーザーメモリ空間であるため，メッセージからの情報取り

```
int
before_write (struct guardid *idp,
      int fd, const void *buf, size_t count)
{
  if(find_fdlist(idp,fd) < 0)
    return GUARD_ALLOW;
  else
    return before_write_body(idp,fd,buf,count);
}

inline int
before_write_body(struct guardid *idp,
  int fd, const void *buf, size_t count)
{
  int ret,tolen;
  struct sockaddr to;
  struct rpcg_env env;

  tolen = sizeof(struct sockaddr_in);
  if(fdtopeername(fd, &to, &tolen) < 0)
    return GUARD_ALLOW;
  pack_rpcg_env(&env,&to,tolen);

  ret = rpc_check(idp,buf + 4,count-4,env);
  switch(ret){
  case DENY:
    return -EACCES;
  case NOT_RPCCALL:
    del_fdlist(idp,fd);
    return GUARD_ALLOW;
  }
  return GUARD_ALLOW;
}
```

図 5.10: プログラムリスト：before_write()

```
int
after_read (struct guardid *idp,int result,
      int fd, const void *buf, size_t count)
{
  int ret,fromlen;
  struct sockaddr from;
  struct rpc_env env;

  if(result < 0)
    return result;

  fromlen = sizeof(struct sockaddr_in);
  if(fdtopeername(fd, &from, &fromlen) < 0){
    return GUARD_ALLOW;
  }
  pack_rpc_env(&env,(struct sockaddr_in *) &from,fromlen);
  ret = rpc_check(idp,buf + 4,count-4,env);
  switch(ret){
  case DENY:
    return -EACCES;
  case NOT_RPCCALL:
    return GUARD_ALLOW;
  }
  return GUARD_ALLOW;
}
```

図 5.11: プログラムリスト：after_read()

```
int
before_connect (struct guardid *idp,
int sockfd, struct sockaddr *serv, int addrlen)
{
  add_fdlist(idp,sockfd,serv,addrlen);
  return GUARD_ALLOW;
}
```

図 5.12: プログラムリスト：before_connect()

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    u_int16_t      sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;  /* internet address */
};

/* Internet address. */
struct in_addr {
    u_int32_t      s_addr;     /* address in network byte order */
};
```

図 5.13: sockaddr_in 構造体の定義

出しのときと同様に，copy_from_user でメモリをコピーしてから，pack_rpc_env() により，デコードして rpc_env 構造体に収める．得られた rpc_env 構造体は，判定実行部に渡す．

TCP/IP の場合，すなわち write() または read() を捕捉した場合には，第1引数であるファイルディスクリプタから，対象のソケットがどこに接続されているかを調べる．

ガード・モジュール内でファイルディスクリプタからソケットの接続先を調べるには，fdtopeername() を用いる．TCP/IP での通信であるので，fdtopeername() からは sockaddr_in 構造体が返り値として戻ってくる．得られた sockaddr_in 構造体から，sendto() の場合と同様にして IP アドレスとポート番号を取得し，rpc_env 構造体をアクセス制御判定部に渡す．

5.5　判定実行部

5.5.1　rpc_check()

rpc_check() は，引数として得た rpc_cmsg 構造体と rpc_env 構造体に収められた情報を元に，アクセス制御判定を行う手続きである．プログラムリストを図5.14に示す．

rpc_check() では，大別して以下の二つの処理を行っている．

(1) SunRPC メッセージの真偽判定

判定対象となっているメッセージが SunRPC のメッセージであるかどうかを判定する．

```
int
rpc_check(struct guardid *idp,
  const void *msg, int len,
  struct rpcg_env env)
{
  int    result;
  struct rpc_cmsg cmsg;
  struct rpc_aclist *list = ((struct rpcguard *) idp->work)->aclist;

  decode_rpc_cmsg(&cmsg,msg);

  /* check non-RPC message */
  if( cmsg.mtype != 0 || cmsg.rpcvers < 1 || 2 < cmsg.rpcvers ){
    return NOT_RPCCALL;
  }
  /* kick rpc_check_vm() */
  while(list != NULL){
    if((result = rpc_check_vm((int *) &list->op, &cmsg, &env)) != 0)
      return result;
    list = list->next;
  }
  return PASS; /* default return value */
}
```

図 5.14: プログラムリスト：rpc_check()

具体的には，rpc_cmsg 構造体に含まれる情報に対し，以下の判定を行う．

- メッセージ種別が 0 (= RPC 呼び出しメッセージ）でない
- RPC プロトコルバージョンが 1 – 2 の範囲にない

いずれかでもが真であった場合，対象のメッセージは SunRPC のメッセージでないとして，NOT_RPCCALL を返り値として戻す．
どちらとも偽であった場合，対象のメッセージを SunRPC のメッセージと判別し，次の処理（SunRPC メッセージに対するアクセス制御判定）に移る．

(2) SunRPC メッセージに対するアクセス制御判定

rpc_check() は，判定の実行のために，仮想機械部である rpc_check_vm() を呼び出す．実際の判定（命令列の評価）は rpc_check_vm() で行なわれ，rpc_check() では，その返り値に応じて適切な処理を行う．
rpc_check() では，リスト構造になっているアクセス制御リストから順に，その内容である命令語へのポインタを取り出す．このポインタを，判定用情報である rpc_cmsg 構造体と，rpcg_env 構造体と共に rpc_check_vm() へと渡す．
rpc_check_vm() での判定結果が返戻値として戻ってくると，rpc_check() はその値に応じて，

- PASS (許可) を返す．
- DENY (不許可) を返す．
- リストの次の要素に移り，繰り返し．

のいずれかの動作を行う．

5.5.2　rpc_check_vm()

プログラムリストを図 5.15 に示す．ただし，プログラム中に登場する VM_TEST_FIELD() 及び PROG,VERS 等の大文字のフィールド名はマクロであり，同値，非同値，範囲の各判定へと展開される（プログラムカウンタを進める処理も，マクロ内で行われている）．
この関数の引数は以下のようになっている．

- pc

  命令語列へのポインタ．rpc_check_vm() 内部では，プログラムカウンタとして機能する．

```
int
rpc_check_vm(int *pc, struct rpc_cmsg *cmsg, struct rpcg_env *env)
{
  while(1){
    switch(*pc){
    case PASS:
      return PASS;
    case DENY:
      return DENY;

      VM_TEST_FIELD(PROG);
      VM_TEST_FIELD(VERS);
      VM_TEST_FIELD(PROC);
      VM_TEST_FIELD(IPADDR);
      VM_TEST_FIELD(IPPORT);

    default:
      return 0;
    }
  }
}
```

図 5.15: アクセス制御判定用仮想機械部 rpc_check_vm()

- cmsg

  捕捉した SunRPC メッセージのヘッダ部分の情報を収めた構造体.

- env

  接続先 IP アドレス及びポート番号を収めた構造体.

rpc_check_vm() は，pc で与えられた命令語列を，cmsg，env を参照しながら評価し，その評価の結果を返す.

5.6　ユーザインタフェース

ユーザインタフェースの作成補助となるコマンドとして，rpcgvmc を作成した.

rpcgvmc は，150 行程度の Perl のスクリプトとして作成した．仮想機械コードの解析には，Perl の正規表現によるマッチングの機能を利用した．これにより，rpcgvmc 自体はごく短い時間で開発を終えることができた.

同様に，機能追加の際も簡単に対応を終える事ができるものと考えられる.

また，出力ファイルにはテスト用の main() 関数を含むようにした．これによって，以下のようにしてコンパイルすることで，テスト用のガードを登録する実行ファイルが生成できるようになった.

```
gcc -DTEST -o rpcgvmcOut rpcgvmcOut.c
```

第6章　実験と考察

6.1　実行例

この節では，本研究で実装した機構の実行例として，NIS の YPPROC_ALL 手続きを止めてみる．この手続きは不正アクセスでは，passwd マップを盗み出すために用いられる．また，通常では *~username* の補完に用いられている．

なお，この節では実行例に以下の表記を用いる．

- % は，一般ユーザのコマンドラインのプロンプトを表す．
- #は，スーパーユーザ (root) のコマンドラインのプロンプトを表す．
- **太字**はユーザによるタイプ部分を表す．

動作を確認するために，以下の yp-all コマンドを用意し，実行する．

```
% ./yp-all
usage:  ./yp-all host domainname mapname
%
```

このコマンドに，適切な引数を与えて実行すると，NIS サーバである host から，NIS ドメイン domainname の mapname というマップの全内容を，YPPROC_ALL 手続きによって取得して，表示する．

たとえば，マップに passwd.byname を指定した場合，以下に示すような実行結果が得られる．

```
% ./yp-all nishost nis.domain.name passwd.byname
muramatu:*:1046:40:Ryosuke MURAMATSU,[HLLA],lab,5161:/home/hlla
/muramatu:/usr/local/bin/tcsh
（中略）white:iGHyGpadvSwzZ:40083:40:Yoshinori NAKATA, [HLLA], lab, 516
1:/home/hlla/white:/usr/local/bin/tcsh
maki:PXqwf/JRgxEQ4:40487:40:Maki HIDUME, [HLLA], lab, 5161:/hom
e/hlla/maki:/usr/local/bin/tcsh
yas:HLlAdMCHccd6N:1231:40:Yasushi SHINJO,[HLLA],5511:/home/hlla
/yas:/usr/local/bin/tcsh
%
```

ただし，上記の実行結果では，セキュリティの問題により暗号化済みパスワードの部分を実際の結果とは異なるものに変更してある．また，ホスト名とドメイン名も実際のものとは異なる．

yp-all コマンドで用いられている NIS の YPPROC_ALL 手続きは，以下の番号にて識別されるものである．

プログラム番号 100004 (NIS)

バージョン番号 2

プロシージャ番号 8 (YPPROC_ALL)

これに対する呼び出し禁止を実現するため，以下のようなルールファイルを記述した．

```
% cat deny_ypall.rpcg
clientrule: deny_ypall
progeq 100004
proceq 8
deny
%
```

これを rpcgvmc で変換すると，出力として deny_ypall.c が得られる．

```
% rpcgvmc deny_ypall.rpcg
Successful made : addrule_deny_ypall()
output file is 'deny_ypall.c'
%
```

次に，deny_ypall.c をコンパイルし，テスト用の SunRPC ガードを登録するためのコマンドとして，deny_ypall を作った．

```
% gcc -DTEST -o deny_ypall deny_ypall.c
%
```

これを以下のようにして実行し，SunRPC ガードを有効にした．

```
% su
Password:
# ./deny_ypall
# exit
exit
%
```

アクセス制御がかけられた状態で，yp-all コマンドで passwd.byname マップの取得を試みる．

```
% ./yp-all nishost nis.domain.name passwd.byname
yp_all: clnt_call: RPC: Unable to send; errno = Permission denied
No such map passwd.byname. Reason: -3
%
```

RPC 呼び出しは拒否され，実行が中断された．

6.2　実行性能

性能を評価するため，以下の条件で Linux を起動し，実行時間を測定した．

カーネル *SysGuard* 対応版 Linix カーネル 2.2.16

CPU Intel Celeron 533 MHz

メモリ 256 M バイト

クライアントの実行時間を，1000 回測定し，最小値を測定結果とする．

アクセス制御が，以下の各条件それぞれの場合について測定．ただし，ルールはいずれも実行するプログラムに適合しないものとする．

- SunRPC ガード未登録
- ルール 0 個で SunRPC ガードを登録
- ルール 100 個で SunRPC ガードを登録
- ルール 200 個で SunRPC ガードを登録
- ルール 300 個で SunRPC ガードを登録

6.2.1　アクセス制御リストに適合しない RPC 呼び出し

echo プロトコルと同様の動作を行う SunRPC のサーバとクライアントを作成し，実際の通信に UDP/IP を用いた場合と TCP/IP を用いた場合とについて，実行時間を測定した．ただし、6.2 節で挙げた条件に、以下に示す条件が加えられる。

- ネットワークの影響を除くため，サーバ / クライアント共に同一のローカルホストで実行．

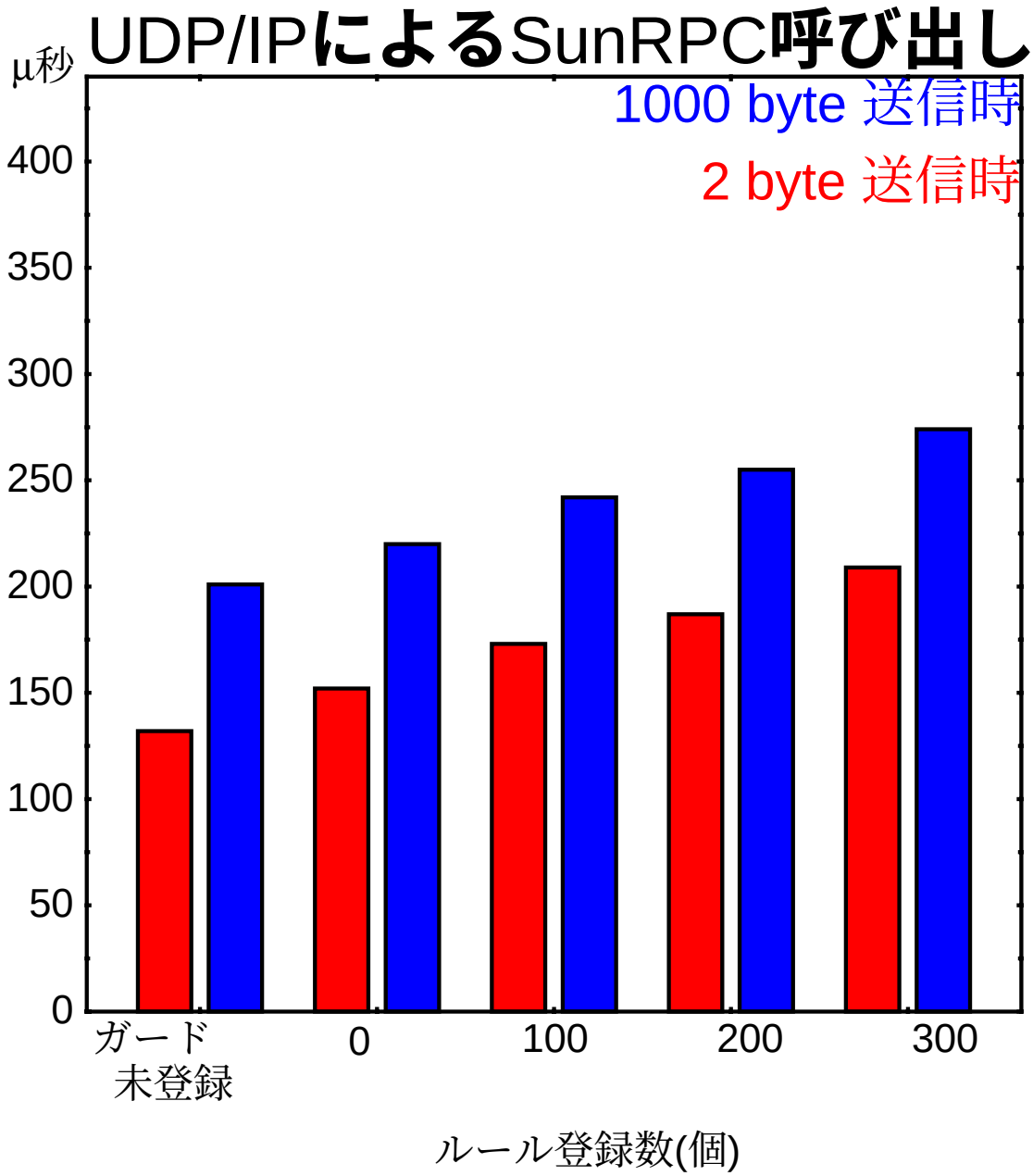

図 6.1: SunRPC による echo クライアント (UDP/IP 版) の実行時間

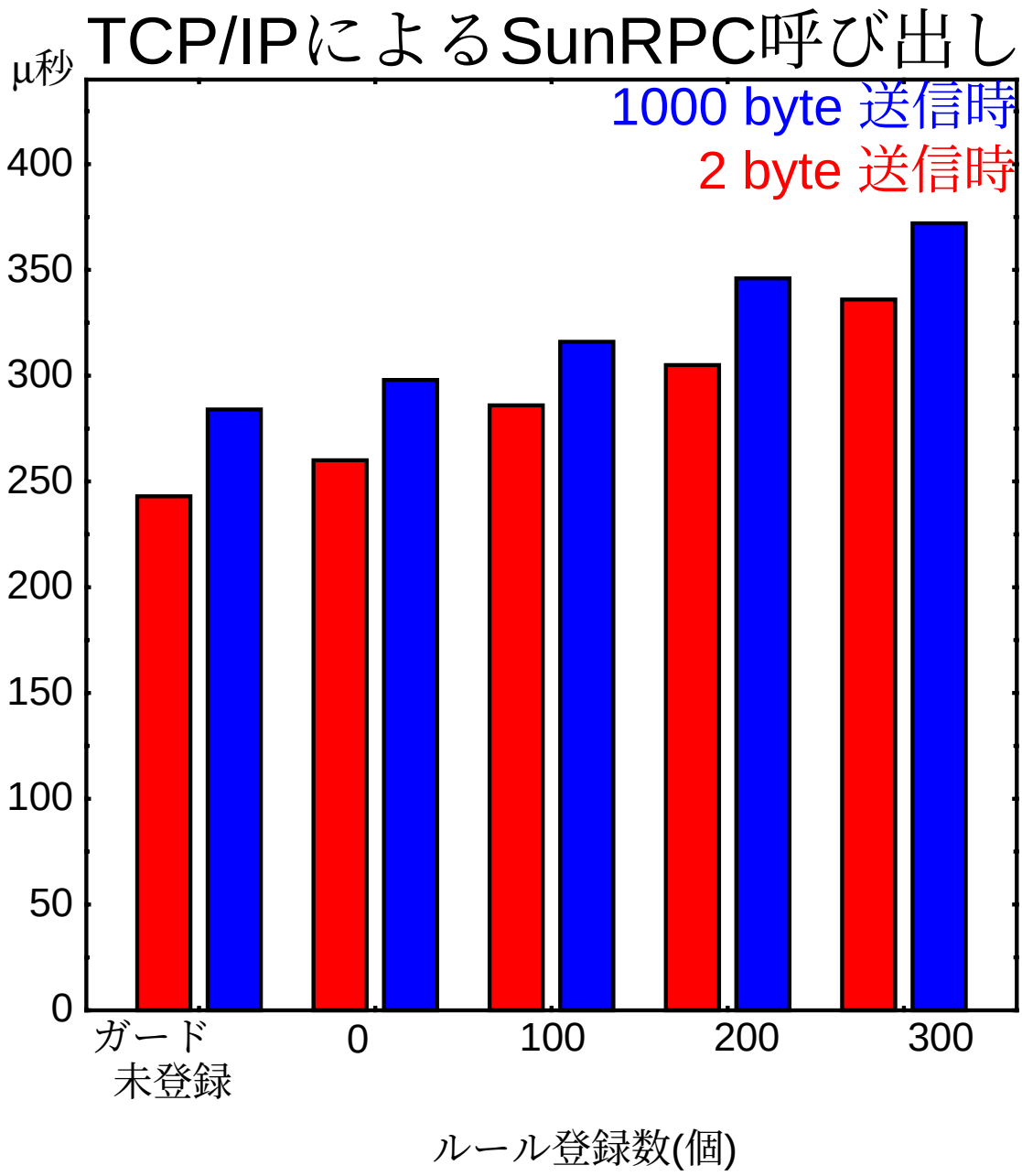

図 6.2: SunRPC による echo クライアント (TCP/IP 版) の実行時間

- クライアントプログラム内の RPC の呼び出し命令の前後にて，gettimeofday() 関数を用いて μ 秒単位で時刻を測定し，実行前と実行語の時刻の差を実行時間とする．

測定結果を図 6.1 と図 6.3 に示す．

同一ホストでの実行の結果であるため，数値にはクライアント側だけでなく，サーバ側のオーバヘッドも含まれている．

ただし，サーバ側でかかるオーバヘッドは，サーバから送信されるメッセージが，SunRPC 呼び出しメッセージでないことを判定する分のみである．

未登録の場合と登録した場合との差は，システムコール実行前のガード・モジュール呼び出しのためのオーバーヘッド分である．

アクセス制御リストの個数を増やした場合については，比例するような形で実行時間が延びているのがわかる．

6.2.2　write() でのファイルへの書き込み

ローカルに用意されたファイルに対し、write() システムコールにより書き込みを行う際の実行時間について測定した。

いずれも場合も、10μ 秒程度のオーバヘッドが加えられているのがわかる。行った書き込みが、SunRPC によるメッセージ送信でないため、アクセス制御リスト数による影響は見られない。

6.3　既知の問題

現時点での実装では，いくつか問題となる点が判明している．この節ではそのような点について述べ，またその対策について論じる．

6.3.1　ライブラリ関数を利用しない RPC 呼び出しへの対応

本研究における実装では，標準的なライブラリ関数内で使用されているシステムコールに対してチェックを行っている．したがって，クライアントがライブラリ関数を利用して RPC 呼び出しを行っている限りは，問題なく呼び出しに対するチェックが行なわれる．

しかし，ライブラリ関数を使用せずに RPC 呼び出しメッセージを生成し，これを sendto() ないしは write() によらずに送信すれば，本研究の実装をかいくぐって RPC 呼び出しを実行することが可能となる．これに対応するためには，本研究と同様の実装を send()，sendmsg()，writev() あるいはなどに対しても施す必要がある．また，ソケットの型として SOCK_RAW を指定したソケットを利用された場合についても，同様に対策が求められる．

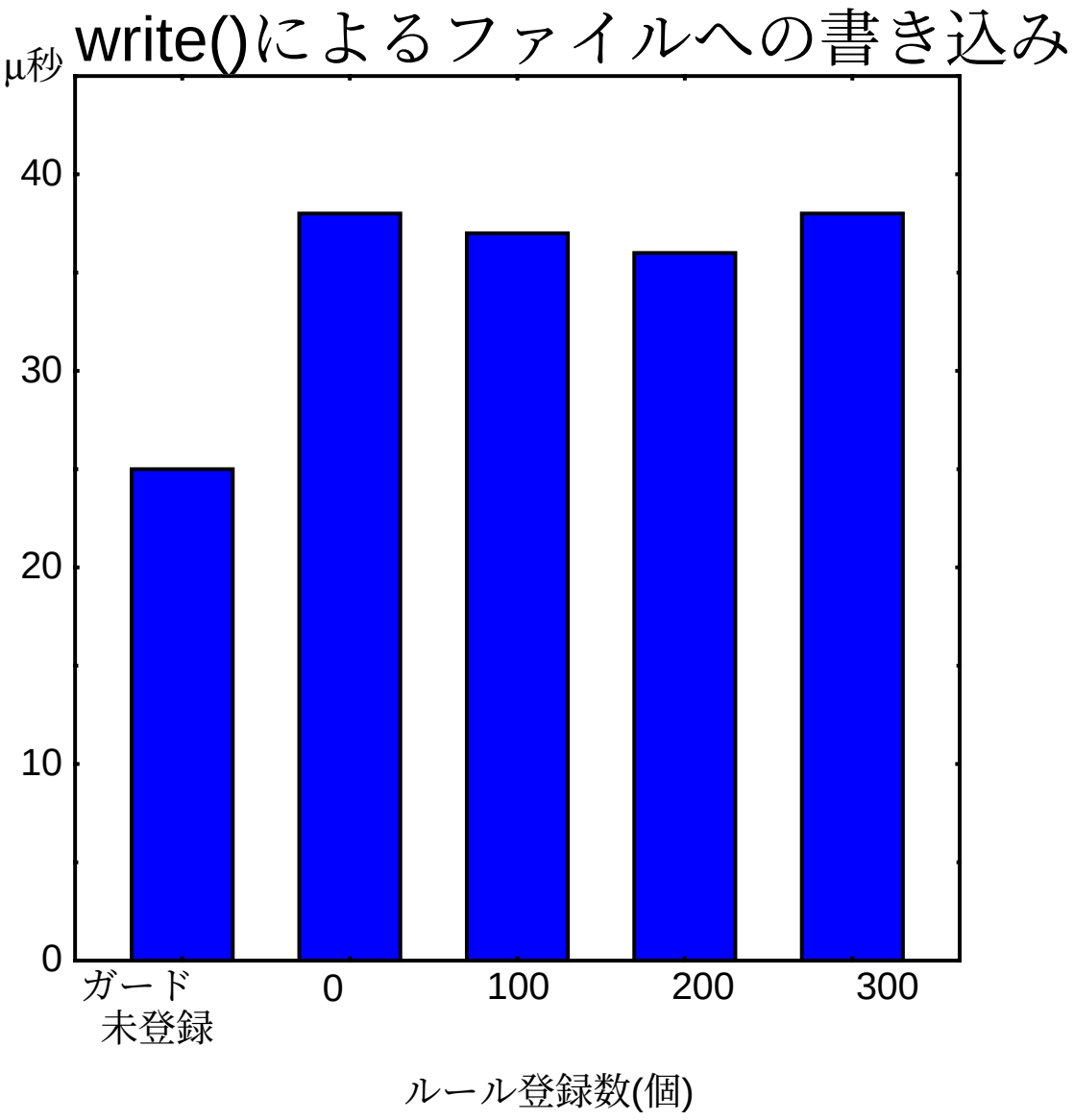

図 6.3: `write()` によるファイル書き込みの実行時間

6.3.2　機能拡張

本研究の実装では，いくつかの点で不足している機能がある．以下では，何点かの望まれる機能の拡張について述べる．

(1) 引数の査察

本研究では，RPC 呼び出しのアクセス制御のために，RPC 呼び出しのヘッダ情報を取得している．しかし実際には，呼び出し時の引数に応じて処理を行いたい場合が想定される．

UDP/IP の場合，1 回のシステムコールでメッセージの全体が送信されるため，必要な情報にアクセスすることが保証されるが，TCP/IP の場合には，1 回のシステムコールでメッセージの全体が送信されないことがある．このような場合にどのように処理をすればいいかは，未検討の課題である．

(2) 単機能ガードとの併用

6.1 節で例に挙げた yp-all に対するアクセス制御方針として，アクセス自体は許可しながら，返信メッセージのうち暗号化済みパスワードのような致命的となりうるフィールドのみを変更したい適当な記号で置換する，という方式が考えられる．ただし，yp-all のパスワードを置換するような機構を実現する際に，本研究で実装した汎用的なアクセス制御機構に組み込むという形で実装するべきかどうかについては十分，検討すべきである．

第 3 章でも述べたが，本研究が基礎としている *SysGuard* は，個々のガード・モジュールが単純化される点に重点を置いている．この考えに基づくならば，本研究の単純な機構に組み込んで複雑化を招くよりも，専用の単機能なガード・モジュールを作成した方がよい．たとえば本研究で実現したアクセス拒否と，yp-all の返り値を書き換える専用ガードを併用する方が，アクセス制御機構全体の単純化が図れる点で，有利であると考える．

6.3.3　誤動作の可能性について

本方式では，SunRPC メッセージの選別に，RPC メッセージ種別と RPC プロトコルバージョンの 2 つのフィールドを使っている．このフィールドによる判定方式では，SunRPC メッセージは確実に捕捉できるが，同時に，非 SunRPC メッセージが捕捉される恐れもある．ただし，前述のフィールドの取りうる値は，SunRPC の仕様より，前者は 0 または 1，後者は 1 または 2 のみである．（将来的に拡張される可能性があるが，本研究ではそれを考慮していない）これらの値 (0,1,2) を，XDR での int 値の表現形式である 4bytes の 2 進値で表現すると，必ず全てのビッ

トが 0 となる byte が出現する．一方，HTTP や SMTP などの，ASCII 文字ベースのプロトコルの場合，そのメッセージ中にはでヌル文字が登場しない．従って，そのようなプロトコルでの誤動作の発生はないことは保証される．

従って，問題は非 ASCII 文字ベースのプロトコルの場合に限られる．だが，誤動作の頻度について論じるのは，容易なことではない．頻度を概算するだけでも，誤動作の可能性がある全てのプロトコルの一覧，およびその利用頻度と誤動作の対象となりうるメッセージの発生頻度についての統計が必要である．

ただし，実際にはアクセス制御リストで拒否されない限りは，見かけ上は問題なく通信が行われる．よって，アクセス制御リストの取り扱いについて，安全側に倒す，という方針である以下のような方針を採れば，誤って SunRPC メッセージであると判定された場合にも，実行が拒否される可能性を減らすことが可能である．

- デフォルトの返り値を許可とする
- 制限条項を少なくするため，必要のあるプログラム/プロシージャだけを制限するようにする．

第7章　まとめ

本論文では，SunRPC に対する付加的なアクセス制御を，システムコールレベルで実現する方法について述べた．

研究の背景として 1 章では，SunRPC の置かれた現在の状況を述べた．SunRPC は現在広く用いられているが，セキュリティに関する機能が弱い．この問題点を解決するために，本研究の目的として SunRPC に対する付加的なアクセス制御を実現することを設定した．

2 章では，SunRPC の技術的背景について述べた．加えて，SunRPC に対して適用可能な既存のアクセス制御技術を紹介し，またその問題点を指摘した．IP ルータによるアクセス制御では，ルータによって隔離された外部ネットワークからの利用は制限できるが，内部ネットワークからの利用は制限できない．portmapper の置換によるアクセス制御では，ポートスキャンを行う SunRPC クライアントに対しては有効に機能しない．SecureRPC では，サーバも含めたプログラムの書き換えが必要である．

3 章では，本研究で利用したシステムコールに対するラッパ，*SysGuard* について説明した．*SysGuard* でアクセス制御を実現するには，カーネル内のモジュールであるガード・モジュールを構成し，システムに組み込んだ上で動作させればよい．

4 章では，3 章で紹介した *SysGuard* を利用して，SunRPC に対するアクセス制御を行うためのガード・モジュールである，SunRPC ガード・モジュールの設計について，その方針と仕様を述べた．SunRPC ガード・モジュールでは `sendto()` 及び `write()` システムコールの実行前に，そのメッセージの内容を調べることでアクセス制御を行う．また，アクセス制御リストを仮想機械用のコードとして保持する方式を採ることとした．

5 章では，4 章で述べた仕様に沿って，どのように SunRPC ガード・モジュールを実現したかについて述べた．

6 章では，実装したガード・モジュールを用いて本研究で実現できたアクセス制御を示し，さらに実行性能の測定も行った．また，本研究の実装についての既知の問題をいくつか取り上げて論じた．

今後の課題としては，6 章でも取り上げたように，ライブラリ関数を利用しない RPC 呼び出しへの対応，RPC 呼び出し時の引数の査察，あるいは本研究の実装と併用して使用する単機能ガードの開発などが挙げられる．

付 録A　実行時間測定用プログラムのプログラムリスト

6.2節で用いた実行時間測定用プログラムのプログラムリストを以下に示す。

これは、送られてきたint値をそのまま送り返すSunRPCによる手続きecho_1()について、その実行直前から実行直後までの時間を測定するものである。

実際にはこのプログラムをPerlによるスクリプトで所定の回数だけ実行させ、データを取得した。

プログラムリスト：SunRPCによるecho / クライアント側プログラム

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>

#include <rpc/rpc.h>
#include "echo.h"

main( int argc, char *argv[] )
{
  struct timeval start,stop;
  CLIENT *cl;
  int leng;
  int times;
  echoarg argument ;
  echores *result ;
  char *server_name;
  char proto[4] = "udp";
  static char optproto[4] = "tcp";
```

```
  if( argc == 4 ) {
    if(!strcmp(&optproto,argv[3]) ){
      strcpy(proto,"tcp");
    }
  }
  else if( argc != 3 ) {
    fprintf(stderr, "usage: %s server number\n", argv[0]);
    exit(1);
  }
  server_name = argv[1];
  cl = clnt_create( server_name, ECHOPROG, ECHOVERSION, proto );
  cl->cl_auth = authunix_create_default();
  if( cl == NULL ) {
    clnt_pcreateerror( server_name );
    exit( 1 );
  }
  argument.a = argv[2];

  gettimeofday(&start,NULL);
  result = echo_1( &argument, cl );
  gettimeofday(&stop,NULL);

  if( result == NULL ) {
    clnt_perror( cl, server_name );
    exit( 1 );
  }
  auth_destroy(cl->cl_auth);
  times =
      (stop.tv_sec  - start.tv_sec ) * 1000000
      + (stop.tv_usec - start.tv_usec);
  printf("%d\n",times);
  xdr_free( xdr_echores, (void *) result );
}
```

プログラムリスト：SunRPC による echo / サーバ側プログラム

```
#include <rpc/rpc.h>
```

```
#include "echo.h"

#ifdef OS_solaris
echores * echo_1(echoarg * arg, struct svc_req * svcreq){
#endif
#ifdef OS_linux
echores * echo_1_svc(echoarg * arg, struct svc_req * svcreq){
#endif
  static echores res;
  struct authunix_parms *unix_cred;
  int uid;
  res.echo = arg->a;
  return(&res);
}
```

プログラムリスト：SunRPCによるecho / インタフェース定義

```
struct echoarg{
  string a<8192>;
};

struct echores{
  string echo<8192>;
};

program ECHOPROG{
  version ECHOVERSION{
    echores ECHO(echoarg) = 1;
  } = 1;
} = 609000001;
```

プログラムリスト：writeによる書き込み速度測定

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main( int argc, char *argv[] )
{
  int fd,leng;
  int times;
  struct timeval start,stop;

  if((fd = open("/tmp/TESTFILE",O_WRONLY | O_CREAT)) < 0){
    fprintf(stderr,"open failed\n");
    exit(1);
  }

  leng = strlen(argv[1]);
  gettimeofday(&start,NULL);
  write(fd,argv[1],leng);
  gettimeofday(&stop,NULL);
  times =
      (stop.tv_sec  - start.tv_sec ) * 1000000
      + (stop.tv_usec - start.tv_usec);
  printf("%d\n",times);
}
```

謝辞

本研究を行うにあたり，多くのご指導とご助言を下さいました板野肯三先生，新城靖先生，千葉滋先生，坂元英紀先生に，心より深く感謝致します．また、*SysGuard* 開発者の榮樂恒太郎さんにも，多くのご助言とご協力を下さったことを、感謝致します。ならびに，日頃研究室でお世話になりました，立堀道昭さん，横田大輔さん，光来健一さん，西尾克己さん，和田慎也さん，福地広之さん，石井孝衛さん，佐々木俊幸さん，渥美 茂樹さん，大塚 紀子さん，海外 浩平さん，久保 聡之さん，鈴木 真一さん，種村 昌之さん，樋爪 真紀さんにも，この場を借りて感謝を述べさせていただきます．

参考文献

[1] Simson Garfinkel and Gane Spafford, 山口英 監訳, 谷口功 訳: “UNIX & インターネットセキュリティ”, オライリー・ジャパン,1998 年

[2] The Open Gorup : “CAE Specification DCE 1.1: Remote Procedure Call Document Number: C706”, 1997

[3] The Object Management Group : “The Common Object Request Broker : Architecture and Secification”, (Minor Editorial Revision:CORBA 2.4.1:November 2000)

[4] Sun Microsystems :
“JAVA REMOTE METHOD INVOCATION – DISTRIBUTED COMPUTING FOR JAVA”, http://java.sun.com/marketing/collateral/javarmi.html

[5] Wietse Venema :
4th enhanced portmapper release (May 1996),
ftp://ftp.porcupine.org/pub/security/index.html

[6] R. Srinivasan : “RPC: Remote Procedure Call Protocol Specification Version 2”, RFC1831, Sun Microsystems(August 1995)

[7] R. Srinivasan : “XDR: External Data Representation Standard”, RFC1832, Sun Microsystems(August 1995)

[8] R. Srinivasan : “Binding Protocols for ONC RPC Version 2”, RFC1833, Sun Microsystems(August 1995)