システム・コールに対するラッパ/リファレンス・モニタ SysGuard の設計と実現

榮樂 恒太郎[†,☆] 新城 靖[††] 板野 肯三[††]

著者らは，Unix においてシステム・コールのレベルでアクセス制御の機能を強化する仕組み SysGuard を実現した．SysGuard では，ガードと呼ばれる，デバイス・ドライバと同様にカーネル内に組み込むモジュールを利用する．各ガードは，システム・コール処理の実行の前後で，付加的にアクセス権をチェックする．SysGuard の設計の特徴は，ガードが適用されるスコープを柔軟に設定できる点にある．SysGuard の実現の特徴は，カーネルの修正個所が少ないこと，および，ポータビリティが高いガードの開発を支援していることである．また，ガード開発キットを用いることにより，ユーザ空間内で簡単に開発やデバッグを行うことができる．SysGuard は，現在，Intel x86 プロセッサ版，および，DEC Alpha プロセッサ版の Linux カーネル 2.2 で利用可能になっている．

The Design and Implementation of SysGuard, a Wrapper/Reference Monitor for System Calls

Kotaro Eiraku,[†,☆] Yasushi Shinjo[††] and Kozo Itano[††]

The authors have implemented a mechanism called SysGuard which improves the access control facility of Unix at the system call level. SysGuard uses modules called guards that work in a kernel like device drivers. Guards are called before or after the processing of system calls, and provide additional access control. One of the main features of the SysGuard design is flexible setting of guard scope. The features of the SysGuard implementation are few modifications against the kernel, and the support for developing portable guards. Moreover, by using the guard development kit, the development and debugging of guards can be easily performed in the user space, and the developed guard can be registered easily to the kernel. SysGuard has been available in the Linux kernel 2.2 on an Intel x86 processor and a DEC Alpha processor.

1. はじめに

プログラムから完全にバグを除去することは不可能である．そのようなバグの一部は，セキュリティ的な弱点となる．近年，この弱点を利用した不正アクセスが増加している．この不正アクセスを未然に防ぐには，セキュリティの技術が重要である．情報生存能力（Information Survivability）[5] とは，セキュリティに加えて，それが破られたとしても，重要なサービスを提供し続ける能力である．さらに，従来の弱点を含んだソフトウェアに外付けの要素を追加し弱点を補うという意味もある．

本研究では，不正アクセスを防止し，高い情報生存能力を実現することを目的として，システム・コールに対するラッパ/リファレンス・モニタ SysGuard を開発している[6),7]．SysGuard では，ガードと呼ばれる，ユーザ定義のモジュールを用いる．ガードは，デバイス・ドライバのようにカーネル内で動作するモジュールであり，システム・コール処理の実行の前後で，付加的にアクセス権をチェックする．一般に，不正アクセスを防止するため，アプリケーション・ソフトウェアのレベルをはじめとする様々なレイヤでアクセス制御を強化する必要があるが，SysGuard では，最後の砦であるシステム・コールのレベルでアクセス制御を強化する．

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

† 筑波大学大学院修士課程理工学研究科
Master's Program in Science and Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
☆ 現在，株式会社アルファシステムズ
Presently with Alpha Systems Inc.

を強化する．この方法の利点は，各ガードの実現が単純化される点にある．SysGuard では，ガードが適用されるスコープを柔軟に設定することができる．ガードは，グローバル，ユーザ，グループ，および，プロセスに加えて，さらに安全度ごとに設定可能である．たとえば，安全度をユーザの位置に対応させ，リモートからのユーザに対して資源へのアクセスを制限することで，万一侵入されたときにも，その被害を最小限にとどめることができる[7]．

この論文では，SysGuard の設計と実現について述べる．ここで設計とは，インタフェースの仕様，実現とは，その仕様に沿って具体的にプログラムとして記述することを意味するものとする．SysGuard の設計の特徴は，ガードが適用されるスコープを柔軟に設定できる点にある．SysGuard の実現の特徴は，カーネルの修正個所が少ないこと，および，ポータビリティが高いガードの開発を支援していることである．実際に，Intel x86 プロセッサ版，および，DEC Alpha プロセッサ版の Linux カーネル 2.2 で SysGuard が動作している．また，FreeBSD への移植を進めている．さらに，ユーザ空間内で簡単に開発やデバッグを行い，開発したガードをカーネルに簡単に組み込むことができるよう，ガード開発キットを用意している．

SysGuard は，SysGuard コア，ガード開発キット，および，個々のガードから構成される．この論文では，まず 2 章で，関連した研究について述べる．3 章では，SysGuard 全体の設計を示し，4 章では，SysGuard コアの実現について述べる．5 章では，ガードの開発と利用に触れる．6 章では，ガード開発キットについて述べる．7 章では，実験の結果を示す．最後に 8 章で以上についてまとめ，結論を述べる．

2. 関連研究

既存のモジュールに対して，外付けで機能を拡張する仕組みは，ラッパ[8),13)] と呼ばれている．ラッパは，一般に，アクセス制御を強化する目的以外にも利用可能である．また，保護したいジュールに対する手続き呼び出しを捕捉し，手続き呼び出しの前後で付加的にアクセス制御を強化する仕組みは，リファレンス・モニタ[2),3)] と呼ばれている．SysGuard は，システム・コールに対するラッパ/リファレンス・モニタの一種である．システム・コールに対するラッパ/リファレンス・モニタは，アクセス制御を判定するモジュールがどこに置かれるかにより，カーネル・レベルとユーザ・レベルに分類される．

カーネル・レベルでアクセス制御を強化する方法として，まず文献 3) で提案されているリファレンス・モニタがあげられる．このシステムでは，バッファ・オーバフロー攻撃を防ぐことを目的として，`execve()` や `setuid()` などのシステム・コールの実行の途中に付加的なチェックを加えている．この方法では，付加的なアクセス制御を行うためのルールを記述しなくても，すべてのプロセスで `execve()` と `setuid()` のシステム・コールに空のルールをチェックするためのオーバヘッドが生じてしまうという問題がある．文献 9) で提案されている Hypervisor は，文献 3) のシステムと異なり，カーネルを修正せずにカーネル・ローダブル・モジュールのみで構成されている．この方法では，何のモジュールも追加していない状態ではそのようなチェックのオーバヘッドは生じない．しかしながら，システム全体で 1 つしかないシステム・コールのジャンプ表のエントリを書き換えることにより実現されているので，1 つでもモジュールを追加すると，そのモジュールが監視しているシステム・コールについて，すべてのプロセスでチェックのためのオーバヘッドが生じてしまう．たとえば，`execve()` を監視するモジュールを追加してしまうと，すべてのプロセスについて，チェックのためのオーバヘッドが生じる．これらの研究と比較して SysGuard では，柔軟なスコープの記述能力があり，オーバヘッドをそのスコープの範囲内に閉じ込めることができる．監視の対象になっているプロセスについて，その監視対象の特定のシステム・コールについてのみオーバヘッドが生じ，その他については，いっさいオーバヘッドが生じない．

ユーザ・レベルでアクセス制御を強化する方法として，カーネル空間からのコールバックを用いる方法[11)] や，Unix System V が提供するシステム・コールのトレース機能を用いる方法[1),12)] がある．本研究の SysGuard では，カーネル内のモジュールによりシステム・コールの実行の可否を判断する．したがって，より高速な実現が可能であり，プロセス構造体などカーネル内のデータ構造を操作することも可能である．

SPIN[4)] は，タイプセーフな言語（Modula-3）で記述されており，同じくタイプセーフな言語で書かれた拡張モジュールをカーネルに組み込むことを許している．これを利用して，システム・コール処理の実行の前後に付加的なアクセス権のチェックを行うことも可能である．この研究と比較して SysGuard の特徴は，Linux や FreeBSD といった，C 言語で記述された，既存の広く普及しているシステムに対して，システム・コール処理の実行の前後で付加的なアクセス権のチェックを行う仕組みを実現している点にある．SysGuard

では，既存の数多くのソフトウェアを修正したり再コンパイルしたりすることなく，アクセス権の付加的なチェックを行い，情報生存能力を高めることができる．

Generic Software Wrappers[8] では，システム管理者が WDL（Wrapper Description Language）と呼ばれる高級言語を用いて，アクセス制御モジュールを記述することができる．そこには，アクセス制御の方針，仕組み，および，活性化の基準を記述することができる．このようにして WDL で記述されたラッパは，WDL コンパイラによって C プログラムに翻訳され，カーネル・ローダブル・モジュールとして組み込まれる．本研究の SysGuard では，方針と仕組みは異なるグループの人々，すなわち，カーネル・プログラマとシステム管理者により記述される．カーネル・プログラマは，デバイス・ドライバのように，C 言語でアクセス制御モジュールを記述する．システム管理者は，アクセス制御モジュールを選択し，スコープと引数を与えることによって，方針を記述する．Generic Software Wrappers では，WDL という言語で記述可能な範囲でしかラッパを記述することができない．たとえば，3.2 節で述べる `GUARD_SUIDEXEC` というラッパは，カーネル内で `stat()` システム・コールと同等の処理を行い，Set-User-ID ビットが立っているかどうかを調べている．Generic Software Wrappers では，`stat()` のような機能を Generic Software Wrappers の設計者が提供していなければ，たとえカーネルで提供されている機能であったとしても，利用することはできない．これに対し，SysGuard では，各ラッパを，システム・コール本体を記述している言語である C 言語により記述する．したがって，カールで提供されている機能を簡単に利用することができる．

3. SysGuard の設計

ここでは，SysGuard におけるガードの基本的な考え方について述べる．

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・コールが発行されたときに，カーネル内のシステム・コールの入口，または，出口から呼び出される．**図 1** に，ガードの呼び出しの様子を示す．ガードの呼び出しを実現している部分を SysGuard コアと呼ぶ．ユーザ・プロセスがシステム・コールを発行すると，トラップによりカーネル内のシステム・コールの入口に制御が移る．従来のシステムでは，単にシステム・コールの番号によりジャンプ表が引かれ，システム・コール本体が呼び出される．

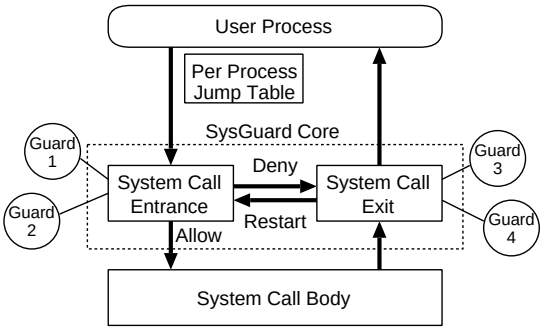

図 1 システム・コールの入口と出口におけるガードの呼び出し
Fig. 1 Invocation of guards at the entrance and exit of system calls.

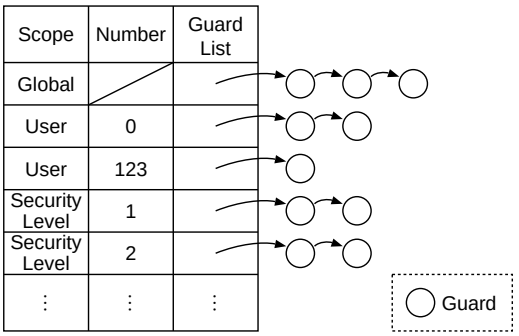

図 2 ガード表
Fig. 2 Guard table.

SysGuard では，システム・コール本体のジャンプ表が引かれる前に，そのシステム・コールに設定されているガードが呼び出される．すべてのガードが許可すると，システム・コールの本体が実行される．どれか 1 つでもガードが拒否を返した場合は，ユーザ・プロセスにエラーが返される．

ガードには，システム・コールの本体から復帰した後，システム・コールの出口において呼び出されるものもある．この場合，設定されたすべてのガードが許可を返した場合，ユーザ・プロセスにそのままシステム・コールの結果が返される．あるガードが拒否を返した場合，そのシステム・コールの影響がキャンセルされる．そして，そのシステム・コールの種類やガードの種類によって，ユーザ・プロセスにエラーが返されるか，あるいは，そのシステム・コールが再実行される．このようなガードの例としては，`accept()` システム・コールで接続先を参照して許可や拒否を行うものがあげられる．

ガードは，概念的には**図 2** に示したような**ガード表**に登録される．実際の方法は 4.1 節で述べる．ガード表の第 1 列はスコープ，第 2 列はスコープに応じた

番号，第 3 列はガードのリストである．たとえば，スコープがユーザの場合，番号はユーザ ID である．スコープには，次のようなものがある．

グローバル システムで 1 つ存在する．すべてのプロセスにより参照される．

ユーザ，グループ ユーザ ID，グループ ID ごとに設置される．

安全度 安全度[7] ごとに設置される．

プロセス プロセス ID ごとに設置される．ただし，`fork()` システム・コールでは，親プロセスから子プロセスに引き継がれる．

各ガードが対象とするシステム・コール，および，働く場所（入口または出口）は，個々のガードを保持するガード表には登録されていない．それらは，ガードの種類（クラス）ごとに保持されている．

ガード，および，ガード表を操作するためのシステム・コールを，付録 A に示す．

3.2 ガードを利用した不正アクセスの防止

現在までに，次のようなガードを開発した．これらのガードは，様々なスコープで，自由に組み合わせて利用することができる．

(1) **DoS 攻撃プログラムの実行を妨害するガード**[6]
このガードでは，通信を行うシステム・コールの発行パターンを監視し，特定のパターンを検出すると，その後のシステム・コールの実行を拒否する．

(2) **SunRPC のアクセス制御を強化するガード**[6]
このガードでは，クライアントやサーバの IP アドレスに加えて，RPC のヘッダに含まれているプログラム番号や手続き番号によりアクセス制限を行う．

(3) **ガードの利用を制限するガード**
このガードでは，ガードを操作するシステム・コールの実行をすべて拒否する．

(4) プログラムの実行を制限するガード（5 章）

(5) Set-UID ビットが立ったプログラムの利用を制限するガード

(6) デバイスの利用を制限するガード

(7) 特権の利用を制限するガード

(8) UDP による受信を送信相手からのパケットのみに制限するガード

(9) `connect()` のアクセス制御を強化するガード

(10) DoS 攻撃による資源の消費をパケット・フィルタリングの制御により抑止するガード

たとえば，CGI の利用を許可している WWW サーバを安全に運用したい場合は，次のようなガードを組み合わせて利用する．(5) のガードにより，Set-UID ビットが立ったプログラムの悪用を防ぐ．(6) のガードにより，デバイスの利用を `/dev/null` などの基本的なものを除き禁止する．(9) のガードにより，外部との通信を防ぐ．これらのガードと (3) のガードを，WWW サーバのプロセスを実行するためのユーザやグループに対して設定することにより，CGI プログラムを安全に実行させることができる．

SysGuard の適用範囲は，システム・コールの引数や発行パターンを観測しただけで正当な利用方法か否かを判定できるものである．SysGuard は，特に，次のような応用に適している．

- ネットワークに対するアクセス制御の強化．上記のガード (1)，(2)，(8)，(9)，(10) は，これに該当する．
- 特権を保持しているプロセスの保護．上記のガード (4)，(5)，(7) は，これに該当する．
- システム・コールのレベルの細かいアカウンティング．上記のガード (10) は，これに該当する．そのほかに，`fork()` システム・コールの乱用を防ぐものが考えられる．

SysGuard の適用範囲は，従来のユーザ・レベルでシステム・コールを監視する研究[1),11),12)]，制限された言語を使う研究[4),8)]，監視対象のシステム・コールが限定されている研究[8)] よりも広く，本研究と同様に追加可能なカーネル内モジュールを用いる研究[9)] と同じである．これらの研究と比較して SysGuard の特徴は，柔軟で分かりやすいスコープが設定可能なこと，および，4.1 節で述べるプロセスごとのジャンプ表を用いた高速なスコープの実現にある．

逆に，システム・コールの引数や発行パターンを観測しただけで正当な利用方法か否かを判定できないものは，SysGuard を使ってアクセス制御を強化することはできない．たとえば，Apache では，ページ単位でユーザ名とパスワードによるアクセス制御を行うことができる．これは，システム・コールの引数や発行パターンを監視しただけでは正当な利用方法か否かを判定できないので，SysGuard では実現できない．

システム・コールを監視するものでも，引数を書き換えてしまうようなものは，SysGuard には不向きである．その理由は，ガードの呼び出し順序を制御できないので，引数を書き換えてしまうと，適用されるガードの順序によって結果が違うことがあるからである．たとえば，あるプロセスに対して，そのプロセスをスコープとするガード A とユーザ ID をスコープとするガード B が設定されていたとする．これら 2 つのガードが両方とも引数を書き換えないものならば，結果（全体として許可するか拒否するか）は適用順序に

依存しない．しかしながら，たとえばガード A が引数を書き換えたうえで許可を返すものであった場合，ガード B が，書き換えられる前の引数を見るか，書き換えられた後の引数を見るかで，結果が違う可能性がある．

SysGuard を用いてシステム・コールのレベルでアクセス制御を強化することが可能であっても，手間の面でアプリケーション・レベルでアクセス制御を強化した方がよい場合がある．たとえば，TCP Wrapper[13] の機能を，すべて SysGuard のガードとして実現することも可能である．TCP Wrapper は，`accept()` システム・コールに対するアクセス制御をアプリケーション・レベルで強化する機能を持っている．たとえば，接続先の IP アドレスに加えて，それを逆引きしたドメイン形式のホスト名を用いてアクセス制御を行うことができる．このような機能を，カーネル内で実現することも可能ではあるが，それには非常に大きな手間がかかる．こういう場合は，カーネルの外のアプリケーション・レベルでアクセス制御を行った方がよい．

3.3 ガードを実現しているモジュールの登録

これまでに述べたように，SysGuard では単機能のガードを組み合わせて用いる．この方法の利点は，ガードが単純化され，個々のガードの開発が容易になる点にある．単純なガードはバグが混入しにくいという利点もある．また，単機能のガードは，Unix におけるフィルタ型コマンドのように，組み合わせて利用することも容易である．各ガードは，独立したモジュールで実現される．実現したモジュールをシステムに容易に組み込むことを可能にするために，Linux の他のモジュール（デバイス，実行形式など）の登録方法に倣い，次のような手続きを提供している．

`int register_guardmod (struct guardmod *module)`

`module` は，ガードの情報を含んだ構造体へのポインタである．この構造体は，**図 3** のようになっている．各ガードは，`guardtype` で示される整数によって区別される．`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．詳しくは，次の節で述べる．`guardbody` 構造体は，このガードが対象とするシステム・コールの番号と，働く場所（入口または出口），および，各々のシステム・コールに 1 対 1 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を含んでいる．

```
struct guardmod
{
  struct guardmod *next;
  int guardtype;
  char *name;
  struct guardbody *bodies;     /* array */
  int (*g_create) (struct guardhandle *gh,
                   char *argval, size_t arglen);
  int (*g_destroy) (struct guardhandle *gh);
  int (*g_ctl) (struct guardhandle *gh,
                char *argval, size_t arglen);
  int (*g_fork) (struct guardhandle *oldgh,
                 struct guardhandle *newgh,
                 struct task_struct *newtask);
  int (*g_exit) (struct guardhandle *gh);
};
```

図 3 ガードのモジュールを定義する構造体
Fig. 3 The structure that defines a guard module.

3.4 プロセスの複製と終了にともなうガードのインスタンスの処理

プロセスごとに働くガードのインスタンスを `fork()` システム・コールの実行時に複製したいという要求がある．`fork()` システム・コールの入口のガードは，親プロセスのコンテキストで実行されるため，子プロセスのプロセス ID やガード ID を得ることができない．そこで，ガードには，`fork()` システム・コールの中から呼び出される `g_fork()` という手続きを用意する．

また，ガードのインスタンスがプロセスごとの情報を保持している場合，プロセスの終了時にその情報を破棄したいことがある．このような要求に対応するため，ガードに，`g_exit()` という手続きを用意する．この手続きは，プロセスの自発的な終了時（`exit()` システム・コール）に加え，強制終了時（`kill()` システム・コール）にも呼び出される．

4. SysGuard コアの実現

現在，Linux カーネル 2.2.18 を修正することにより，SysGuard を実現している．また，FreeBSD への移植を進めている．ここでは，SysGuard のコア部分の実現について述べる．

4.1 柔軟なスコープの実現

ガードは，概念的には，図 2 に示したようなガード表に登録される．実際には，プロセス ID に関するものはプロセス構造体へ登録され，その他はガード表へ登録される．

適用されるガードの集合は，プロセスごとに異なる．すべてのシステム・コールをとらえて実現する方法では，ガードが付加されていないプロセスについても

```
int
schook (....)
{
  ret = GUARD_ALLOW;
  no = getsyscallno (....);
  ret = gchk (ret, no | GUARD_BEFORE, ....);
  if (ret < 0)
    return ret;
  ret = ((sysfun_p) sys_call_table[no]) (....);
  return gchk (ret, no | GUARD_AFTER, ....);
}
```

図 4 システム・コールをフックする関数
Fig. 4 The system call hook function.

表 1 プロセスごとのジャンプ表を更新するシステム・コール
Table 1 System calls that update the per process jump table.

Category	System call
Standard	setuid(), seteuid(), setreuid(), setresuid(), setgid(), setegid(), setregid(), setresgid(), setgroups(), execve()
SysGuard	guardtbl_set(), setsecuritylevel()

オーバヘッドが生じてしまう．そこで，従来カーネル内に 1 つしか存在していなかった，システム・コールのジャンプ表を，プロセスごとに用意した．この表のエントリのうち，ガードの対象となるシステム・コールについてのみジャンプ先を変更する．

変更されたジャンプ先には，**図 4** のようなフック用の関数を置く．この関数では，まずシステム・コールの番号を得て，`gchk()` という手続きを呼び出す．`gchk()` では，指定されたシステム・コールに対応するガードのインスタンスをガード表から検索し，それらのモジュールを呼び出してチェックを行う．次に，本来のシステム・コールのジャンプ表である `sys_call_table` を参照してシステム・コールの本体を実行する．最後に再び `gchk()` を呼び出し，システム・コールの出口で働くガードによるチェックを行う．

プロセスごとにジャンプ表を持たせることで，ガードの対象とならないプロセスについては，1 回間接アクセスをするオーバヘッドだけで済む．ただし，`fork()` システム・コールが発行されるたびに，ジャンプ表をコピーするオーバヘッドが生じる．

プロセスごとのジャンプ表が更新されるタイミングは，ガード表を操作するシステム・コールによりガード表の内容が変化したときのほか，ガードを登録できるスコープであるユーザ ID，グループ ID，および，安全度が変化したときである．具体的に対象となるシステム・コールを**表 1** に示す．SysGuard で提供するシステム・コール `guardtbl_set()` および

```
int
sys_setuid (uid_t uid)
{
  if (uid == current->uid)
    current->euid = uid;
  else if (....)
    ....
  else
    return -EPERM;    /* Permission denied */
  update_jumptable (current);   /*  ADDED  */
  return 0;                     /* Success */
}
```

図 5 システム・コール本体（`sys_setuid()`）からのジャンプ表更新手続きの呼び出し
Fig. 5 Calling the update procedure of a jump table from a system call body (`sys_setuid()`).

`setsecuritylevel()`[7] を除く 10 個のシステム・コールについては，ジャンプ表を更新する手続きを呼び出すよう，システム・コール本体に 1 行ずつ書き加えた．その一例として，`setuid()` システム・コールの本体である `sys_setuid()` の概略を**図 5** に示す．ユーザ ID の変更が正常に終了し，return を行う直前（図 5 の `Success` と記した行の直前）で，SysGuard コア内にあるジャンプ表を更新する手続き `update_jumptable()` を呼び出している．なお，`fork()` システム・コールによりプロセスが複製されたときには，プロセス ID に関するガード表のエントリも複製されるため，ジャンプ表の内容は変化しない（単にコピーを行う）．

4.2 ガードのポータビリティ

SysGuard は，現在のところ Linux 版が動作しており，FreeBSD への移植を行っている．将来的には Solaris や MacOS X など他のシステムでの実現も目標としている．そこで，ガードのソース・コードを変更せずに利用できるようにするため，各 OS 固有のプロセス構造体やメモリ管理などを抽象化し，相違点を吸収するようにしている．たとえば，ソケットの操作を行うシステム・コールは，UNIX98[10] で標準化されている．そのような標準に従ったアプリケーション・プログラムには，ポータビリティがある．同様に，その標準のシステム・コールに対するガードにもポータビリティがあることが期待される．

DEC Alpha プロセッサ版以外の Linux では，ソケットの操作を行う `connect()` や `accept()` などはライブラリ関数になっており，カーネルのレベルでは `socketcall()` システム・コールという 1 つのカーネル・エントリ・ポイントにまとめられている．そのような Linux で `connect()` などを扱うために `socketcall()` 用のガードを作成すると，そのガードのポータビリティが非常に低くなってしまう．そこで，`socketcall()` が

存在する Linux 版の SysGuard では，socketcall() 専用の呼び出し関数を用意し，その中から各ソケットの操作に対応するガードを呼び出すようにした．これにより，ガード開発者はこのような Linux 特有の socketcall() の存在を意識することなく，通常のシステム・コールの場合と同じように connect() や accept() などに対するガードが開発できるようになった．

4.3 SysGuard コアのポータビリティ

ここでは，OS が同じで CPU アーキテクチャが異なる場合の SysGuard コアのポータビリティについて述べる．

同じ Linux でも，CPU アーキテクチャの違いにより，システム・コールを呼び出すアセンブリ・コードの部分に相違点がある．SysGuard の実現にあたって，CPU アーキテクチャに依存するアセンブリ・コードにより記述された部分は 2 カ所あり，一カ所はプロセスごとのジャンプ表を参照する部分，もう一カ所は図 4 で示したフック用の関数である．その他の部分はすべて C 言語で記述される．このことから，他の CPU アーキテクチャへの対応は比較的容易であると考えられる．

また，現在の Linux 版 SysGuard は，システム・コールのジャンプ先を変更して実現している．しかし，この方法で SysGuard を実現すると，fork()，execve()，sigreturn() のシステム・コールが問題となった．これらのシステム・コールは，Linux ではスタック・フレームを直接操作する．現在の SysGuard の実現方法では，フック用の関数の中からシステム・コールの本体を呼び出しているため，スタック・フレームの状態が本来とは異なっている．そこで，スタック・フレームを参照する特殊なシステム・コールについては，入口のガードを呼び出した後，スタック・フレームを元の状態に戻してからシステム・コールの本体を呼び出すようにした．これらのシステム・コールの実行後にはフック用の関数に制御が戻らないため，これらのシステム・コールの出口にはガードが設定できないという制限が生じている．

FreeBSD 版 SysGuard の実現では，アセンブリ・コードは使用していない．しかし，システム・コールの本体を呼び出す方法が CPU アーキテクチャに依存しているため，システム・コールのフック用の関数は CPU アーキテクチャごとに用意する必要がある．なお，Linux 版 SysGuard と異なり，スタック・フレームの問題は生じていない．

4.4 エラーの扱い

ガードは，様々なスコープに対して複数設定することが可能である．このため，1 つのシステム・コールに対して複数のガードが働くことがある．1 つのガードが拒否を返した場合は，その戻り値であるエラー・コードをシステム・コールのエラー・コード（errno）として採用する．複数のガードが異なる戻り値を返した場合は，EACCES（Permission denied）を返す．

システム・コールの出口で働くガードが拒否を返した際には，そのシステム・コールの影響をキャンセルする．現在のところ，ファイル記述子を作成するシステム・コールであった場合に，そのファイル記述子を閉じる（close する）ようになっている．

5. ガードの開発と利用

ガードの開発と利用は，一般的には次のように行われる．

(1) 特定の問題（たとえば，FTP サーバの保護）を解決する方法を考える．
(2) 再利用性を考慮してガードの機能を設計する．
(3) ガード本体（カーネルに組み込まれるモジュール）を開発する．
(4) インスタンスを生成するプログラム（カーネル外で動作）を開発する．

ここでは，ガード GUARD_EXEC を例にとり，ガードの開発から活性化させて利用を行うまでの手順を示す．5.1 節では，このガードの利用例として FTP サーバの保護を取り上げる．なお，このガードは，他のサーバの保護や他の目的，たとえば，バイナリ形式で入手したプログラムを安全に実行する目的などにも利用することができる．

5.1 FTP サーバの保護とガード GUARD_EXEC の設計

FTP サーバは，コール・バックのために特権を維持して動作する必要がある．FTP サーバにバグが含まれていると，リモートから特権ユーザの権限で /bin/sh などの任意のプログラムが実行される可能性がある．このような攻撃を防ぐには，FTP サーバ ftpd に対して，/bin/sh などが実行されないよう，プログラムの実行を制限する方法が考えられる．しかし，全面的に禁止すると，ディレクトリの中身をリスト表示する ls コマンドも使えなくなってしまう．そこで，一部のプログラムについては実行を許可する，といったような細かい制御を行いたい．一方，このように実行可能なプログラムを制限する機能は，FTP サーバを保護する目的以外にも利用可能であることは容易に類推され

```
int
g_create (struct guardhandle *gh, char *argv,
          size_t argl)
{
  gh->work = makehash (argv, argl);
  return 0;
}

int
before_execve (struct guardhandle *gh,
    char *filename, char *argv[], char *envp[])
{
  p = searchhash (gh->work, filename);
  if (p)
    return GUARD_ALLOW;
  return GUARD_DENY;
}
```

図 6 ガード GUARD_EXEC の主要部分
Fig. 6 The main part of the guard GUARD_EXEC.

```
int
main (int argc, char *argv[], char *envp[])
{
  guardid_t id[2];
  guard_execve_arg garg =
    { "/usr/sbin/in.ftpd", "/bin/ls", 0 };
  id[0] = guard_create (GUARDID_ANY,
            GUARD_EXEC, GUARDATTR_AUTODESTROY,
            &garg, sizeof (garg));
  id[1] = guard_create (GUARDID_ANY,
            GUARD_GUARD, GUARDATTR_AUTODESTROY,
            NULL, 0);
  guardtbl_set (GUARDTBL_PID, getpid (), id, 2);
  execve ("/usr/sbin/in.ftpd", argv, envp);
  perror ("in.ftpd");
  return 1;
}
```

図 7 ガード GUARD_EXEC のインスタンスの生成と活性化
Fig. 7 Creation and activation of the guard GUARD_EXEC.

る．ガードの再利用性を高めるために，実行可能なプログラムは，インスタンスの生成時にガードの引数として指定できることが望ましい．

このような，実行可能なプログラムを制限するためのガード GUARD_EXEC を設計する．このガードは，execve() システム・コールによるプログラムの実行を拒否するものである．ただし，あらかじめ登録されているプログラムについては，実行を許可する．

5.2 ガード GUARD_EXEC の本体

ガード GUARD_EXEC は，**図 6** に示したような手続きを含む．g_create() は，ガードのインスタンスを生成するシステム・コール guard_create() が発行されたときに呼び出される．第 1 引数は，ガードのインスタンスに固有の情報を持つハンドル gh であり，argv と argl は，guard_create() システム・コールに指定されたガードへの引数である．このガードへの引数として指定されるのは，実行を許可するプログラムである．そのリストを，makehash() により生成している．

このガードが働くプロセスで execve() システム・コールが発行されると，図 6 の手続き before_execve() がシステム・コールの入口から呼び出される．この手続きの第 1 引数は，ガードのインスタンスに固有の情報を持つ gh であり，第 2 引数以降は execve() システム・コールの引数と同じである．この引数からファイル名を得て，searchhash() を呼び出す．これによって，ガードのインスタンスの生成時に guard_create() システム・コールを通じて g_create() により登録された，実行を許可するプログラムであるかどうかをチェックする．登録されている場合は，許可を返す．そうでなければ，拒否を返す．あらかじめ登録しておくプログラムを変えることにより，様々な用途に利用することができる．

5.3 ガード GUARD_EXEC のインスタンスの生成と設定

ガードを利用するには，カーネル外でガードのインスタンスを生成し設定するプログラムが必要である．**図 7** は，5.1 節で述べたアクセス制御を実現するために 5.2 節で示したガードのインスタンスを生成し設定するプログラムの例である．このプログラムでは，まず guard_create() システム・コールによりガードのインスタンスを生成している．GUARDID_ANY は，システムに利用可能な任意のガード ID を割り当ててもらうことを意味する．GUARD_EXEC は，プログラムの実行を制限するガードを示している．GUARDATTR_AUTODESTROY は，生成したインスタンスがどこからも参照されなくなったときに自動的にそのインスタンスを破棄することを意味する．garg はガードへの引数であり，この場合は実行を許可するプログラムとして FTP サーバである /usr/sbin/in.ftpd と /bin/ls を指定している．また，GUARD_GUARD で示される，3.2 節で述べたガードを操作するシステム・コールの利用を拒否するガードのインスタンスも生成している．これにより，ガードが解除され攻撃されることを防ぐことができる．

次に，guardtbl_set() システム・コールを使って，生成したガード ID をガード表に登録している．GUARDTBL_PID と getpid() で，自分自身のプロセスに対して登録することを示している．続いて，execve() システム・コールにより，プログラム /usr/sbin/in.ftpd を実行している．最後の perror() は，プログラムの実行に失敗した場合にエラー・メッセージを表示する．

このように自分自身のプロセスにガードを設定し他

のプログラムを実行するラッパ・プログラムは，TCP Wrapper[13] と同様にして用いることができる．たとえば，inetd の設定ファイル inetd.conf に，次のように登録されていたとする．

```
/usr/sbin/in.ftpd -l -a
```

この行を，次のように修正する（作成したラッパ・プログラムを guard_ftpd とする）．

```
/usr/sbin/guard_ftpd /usr/sbin/in.ftpd -l -a
```

この状態で FTP サービスを提供すると，たとえ特権が奪われたとしても，ls コマンド以外の実行を防ぐことができる．ガードの利用を制限するガードも設定してあるため，ガードの変更や解除も防ぐことができる．

6. ガード開発キット

ガードの開発を容易にするための開発キットを用意した[6]．ガード開発者は，ガードを作成し，最終的に SysGuard のモジュールとしてカーネルに組み込む．開発キットにより，ユーザ空間内で簡単に開発やデバッグが行えるようになる．

6.1 2 段階の開発

ガード開発キットを利用した開発は，次の 2 段階となる．

(1) **ユーザ空間における開発キットによる開発**

ガード開発者は，まず，ユーザ空間内で動作するガードのモジュールを開発する．この段階では，特定の対象となるプログラムを 1 つ選び，開発中のモジュール，および，システム・コールのスタブとともにリンクして実行する．スタブは，実際にシステム・コールを発行する前後で，ユーザ空間内にある開発中のガードを呼び出す．

(2) **カーネルへの組込み**

ユーザ空間内でデバッグが完了したガードのモジュールを，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，行単位のステップ実行を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．

6.2 支援関数

開発キットでは，**表 2** のような手続きを提供している．これにより，本来カーネル空間でのみ利用できる手続きを，ユーザ空間で利用することができる．さらに，**表 3** のようなよく使われる手続きを提供している．これにより，本来ユーザ空間でのみ利用できる手続きを，カーネル空間で利用することができる．

ガードでは，この表に示されていない，カーネル内の手続きを直接呼び出して利用することもできる．ただし，利用する手続きによっては，そのガードのポータビリティが下がってしまう．たとえば，3.2 節で述べた (10) のガードは，Linux のカーネル内にあるパケット・フィルタリングを制御する手続きを直接呼び出している．このガードを，たとえば FreeBSD 版 SysGuard にそのまま組み込むことはできない．

表 2 ガード開発用にユーザ空間で利用できる手続き
Table 2 Kernel-level functions usable in user-space for guard development.

Function	Description
kmalloc(), kfree()	Allocate and free memory
copy_from_user()	Copy from user-space to kernel-space with checking access rights
copy_to_user()	Copy to user-space

表 3 ガード開発用にカーネル空間で利用できる手続き
Table 3 Usable functions in kernel-space for guard development.

Function	Group
stat()	File
getsockname(), getpeername()	Network
getuid(), getgid(), geteuid(), getegid(), getpid()	Process
gettime()	Timer

7. 実　　験

SysGuard を Intel x86 プロセッサ版，および，DEC Alpha プロセッサ版の Linux カーネル 2.2.18 で実現し，ガードを呼び出すオーバヘッドを Intel x86 プロセッサ版で測定した．実験に用いたハードウェアは，CPU が Pentium III（Coppermine）1 GHz（FSB 133MHz），メモリが 384 M バイト（PC133 CL2）のパーソナル・コンピュータである．実験対象として，以下に示す 3 種類のガードを用意した．

(1) getpid() システム・コールの入口で働く，何もしないガード（つねに許可を返すガード）

(2) execve() システム・コールの入口で働く，Set-UID プログラムの利用を制限するガード（/usr/lib/sendmail のみ実行を許可するようハッシュ表に登録したもの☆）．内部に stat() システム・コールに相当する処理を含む．

☆ 登録するプログラムの数を 31 個（Red Hat Linux 7.0 を標準でセットアップしたときに存在するすべての Set-UID プログラムの数）に増やした場合についても，類似の実験を行った．その結果，実行時間は，登録するプログラムの数が 1 個のときと同じであった．その理由は，登録されているプログラムのハッシュ表が引かれるのが，stat() の結果 Set-UID ビットが立っていたときだけであり，この実験では，Set-UID ビットが立ったプログラムが 1 度も実行されていなかったからである．

表 4 システム・コールの実行時間（最小時間）
Table 4 Results from system call micro-benchmark test (minimum).

Guard type and system call	Normal kernel	SysGuard kernel	
		no Guard	with Guard
(1) `getpid()`	0.31 μs	0.31 μs	0.97 μs
(2) `execve()`	68.9 μs	69.1 μs	72.0 μs
(3) `execve()`	68.9 μs	69.1 μs	71.6 μs
`fork()`	69.1 μs	69.6 μs	—

(3) `execve()` システム・コールの入口で働く，特定のプログラムのみ利用を許可するガード `GUARD_EXEC`（実験に用いるプログラムのみ実行を許可するようハッシュ表に登録したもの）

まず，それぞれのシステム・コールと `fork()` システム・コールの実行時間を測定した☆．**表 4** に，測定の結果を示す．この実験では，対象となるシステム・コールを数秒間繰り返し実行し，全体の実行時間を測定した．その実行時間を繰返し回数で割ることで，1 回あたりの実行時間を求めた．同じ実験を 100 セット繰り返し，その最も短い時間を採用した．ガードのない状態では，`getpid()` は同時間である．これは，システム・コールのジャンプ表を参照する際の 1 回の間接アクセスによるオーバヘッドが CPU のパイプラインに吸収されたためと考えられる．一方，`execve()` は標準のカーネルに比べて遅くなっている．これは，ガードがない状態であっても，プロセスごとのシステム・コールのジャンプ表の初期化を行うためである．次に，ガードがない状態とある状態を比べてみると，`getpid()` で約 0.7 μ 秒，`execve()` で約 3 μ 秒のオーバヘッドがみられる．これは，ガード内部でのハッシュ表の検索や，`stat()` に相当する処理が原因である．また，`fork()` については，標準のカーネルに比べて SysGuard のカーネルで約 0.5 μ 秒のオーバヘッドがみられる．これは，プロセスごとのシステム・コールのジャンプ表（1 K バイト）を複製するためである．

続いて，コンパイル作業（GNU patch 2.5.4）に要する時間を測定した．実験には (2) のガードを用いた．このガードは，Set-UID ビットの立ったプログラムの実行を制限するものである．コンパイル中に Set-UID プログラムは実行されないため，このガードは `stat()` に相当する処理により Set-UID ビットが立っていないことを確認し，許可を返すという動作を行う．コンパイル作業中に発行された `fork()` および `execve()` の回数は 288 回であった．

☆ `fork()` システム・コールの実行時間は，子プロセス側の処理（`exit()` システム・コール），および，`wait()` システム・コールの実行時間を含む．

表 5 `patch` のコンパイルの所要時間
Table 5 Results from compiling `patch` benchmark test.

	Normal kernel	SysGuard kernel	
		no Guard	with Guard
Avg.	5337 ms	5338 ms (+0.0%)	5342 ms (+0.1%)
Min.	5309 ms	5312 ms (+0.1%)	5312 ms (+0.1%)
Max.	5364 ms	5365 ms (+0.0%)	5372 ms (+0.1%)

表 5 に，測定の結果を示す．括弧内の数値は，標準のカーネルに対するオーバヘッドを示している．平均時間をみると，ガードを実装したカーネルでは標準のカーネルに比べてオーバヘッドはほぼ存在しない．最小時間や最大時間を比べてみると，この値は誤差の範囲であるとみることができる．次に，ガードがない状態とある状態を比べてみると，`execve()` にガードが設定されたことによるオーバヘッドは約 0.1%であることが分かる．このことから，Set-UID プログラムの利用を制限するガードを設定しても，実際のアプリケーション・ソフトウェアの実行時間はほぼ変わらないといえる．

8. おわりに

この論文では，Unix においてシステム・コールのレベルでアクセス制御の機能を強化する仕組み SysGuard について述べた．SysGuard では，ガードと呼ばれる，デバイス・ドライバと同様にカーネル内に組み込むモジュールを利用する．各ガードは，システム・コール処理の実行の前後で，付加的にアクセス権をチェックする．SysGuard の設計の特徴は，ガードが適用されるスコープを柔軟に設定できる点にある．SysGuard の実現の特徴は，カーネルの修正個所が少ないこと，および，ポータビリティが高いガードの開発を支援していることである．また，ガード開発キットを用いることにより，ユーザ空間内で簡単に開発やデバッグを行うことができる．さらに，Intel x86 プロセッサ版の Linux カーネル 2.2.18 で性能の測定を行った．その結果，`execve()` システム・コールのガードについて，実際のアプリケーション・ソフトウェア（コンパイル作業）の実行時間には約 0.1%のオーバヘッドしか存在しないことを示した．

現段階では，Intel x86 プロセッサ版，および，DEC Alpha プロセッサ版の Linux カーネル 2.2 で SysGuard が利用可能になっている．今後は，さらに有用なガードの種類を増やすとともに，FreeBSD への SysGuard の移植を完了させたい．

参 考 文 献

1) Acharya, A. and Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications, *Proc. 9th USENIX Security Symposium* (2000).
2) Ames, S.R., Gasser, M. and Schell, R.R.: Security kernel design and implementation: An introduction, *IEEE Computer*, Vol.16, No.7, pp.14–22 (1983).
3) Bernaschi, M., Gabrielli, E. and Mancini, L. V.: Operating System Enhancements to Prevent the Misuse of System Calls, *Proc. 7th ACM Conference on Computer and Communication Security* (*CCS*), pp.174–183 (2000).
4) Bershad, B., Savage, S., Pardyak, P., Sirer, E.G., Becker, D., Fiuczynski, M.E., Chambers, C. and Eggers, S.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating System Principles* (*SOSP*), pp.267–284 (1995).
5) Cowan, C. and Pu, C.: Death, Taxes and Imperfect Software: Surviving the Inevitable, the New Security Paradigms Workshop (1998).
6) 榮樂恒太郎，新城　靖，樋爪真紀，中田吉法，板野肯三：高い情報生存能力を実現するラッパ SysGuard におけるガード・モジュールの開発環境，日本ソフトウェア科学会第 4 回プログラミングおよび応用のシステムに関するワークショップ（SPA2001）(2001).
7) 榮樂恒太郎，新城　靖，板野肯三：位置情報を利用したシステム・コール・レベルのアクセス制御，情報処理学会研究報告 2000-OS-84-29, Vol.2000, No.43, pp.213–220 (2000).
8) Fraser, T., Badger, L. and Feldman, M.: Hardening COTS Software with Generic Software Wrappers, *Proc. 1999 IEEE Symposium on Security and Privacy*, pp.2–16 (1999).
9) Mitchem, T., Lu, R. and O'Brien, R.: Using Kernel Hypervisors to Secure Applications, *Proc. 13th Annual Computer Security Applications Conference* (*ACSAC '97*), pp.175–181 (1997).
10) The Open Group: *The Single UNIX Specification Version 2* (*UNIX98*) (1997).
11) 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).
12) 東村邦彦，松原克弥，相河　亨，加藤和彦：モーバイルオブジェクトシステム Planet のプロテクション機構，日本ソフトウェア科学会第 1 回インターネットテクノロジーワークショップ（WIT '98）(1998).
13) Venema, W.: TCP Wrapper: Network Monitoring, Access Control, and Booby Traps, *Proc. 3rd Usenix UNIX Security Symposium*, pp.85–92 (1992).

付録　SysGuard 関連のシステム・コール

A.1　ガードの操作

ガードは，次の 3 つのシステム・コールにより操作される．

（1） `guardid_t guard_create (guardid_t id, int guardtype, int guardattr, char *argval, size_t arglen)`

これは，ガードのインスタンスを生成するシステム・コールである．`id` は，要求するガード ID である．`GUARDID_ANY` を指定することにより，システムに割り当ててもらうことも可能である．`guardtype` は，ガードの型（種類）である．カーネル内にあるガードを実現するモジュールを選択するために用いる．`guardattr` は，ガードの属性である．生成したインスタンスがどこからも参照されなくなった際（参照カウンタがゼロになった際）に自動的にインスタンスを破棄するかどうかを指定する．`argval` はガードへの引数，`arglen` はその長さである．`argval` の内容は，ガードの型に依存する．このシステム・コールは，生成されたガードの ID を返す．

（2） `int guard_destroy (guardid_t id)`

これは，ガードを破棄するシステム・コールである．`id` は，破棄したいガードの ID である．

（3） `int guard_ctl (guardid_t id, char *argval, size_t arglen)`

これは，すでに存在するガードを制御するシステム・コールである．`id` は，制御したいガードの ID である．`argval` はガードへの引数，`arglen` はその長さである．

A.2　ガード表の操作

生成されたガードのインスタンスは，ガード表に登録されて初めて活性化される．ガード表は，概念的には 3 つの列，すなわち，スコープ，スコープに応じた番号，および，ガードのリストから構成される．この表は，次の 2 つのシステム・コールにより操作される．

（1） `int guardtbl_set (int scope, int number, guardid_t *list, int listlen)`

これは，ガード表にガードを登録するシステム・コールである．`scope` と `number` は，ガード表中で登録の対象となるエントリを指すスコープと番号である．`list` と `listlen` は，登録するガードの ID のリストとその

長さである．スコープとしてはグローバル，安全度，ユーザ，グループ，プロセスがある．

（2） `int guardtbl_get (int scope, int number, guardid_t *list, int listlen)`

これは，ガード表のガードのリストを取得するシステム・コールである．`scope` と `number` でエントリを指定する．`list` に ID のリストが，返り値としてリストの長さが返される．

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

榮樂恒太郎（学生会員）

1978 年生．2000 年筑波大学第三学群情報学類卒業．2002 年同大学大学院修士課程理工学研究科理工学専攻修了．同年株式会社アルファシステムズ入社．在学中は OS および情報セキュリティに関する研究に従事．修士（工学）．

新城　靖（正会員）

1965 年生．1993 年筑波大学博士課程工学研究科電子・情報工学専攻修了．同年琉球大学工学部情報工学科助手．1995 年筑波大学電子・情報工学系講師．分散型オペレーティング・システム，オブジェクト指向，分散処理，並列処理に興味を持つ．1990 年情報処理学会第 40 回全国大会学術奨励賞，1995 年情報処理学会山下記念研究賞受賞．博士（工学）．日本ソフトウェア科学会，ACM，IEEE CS 各会員．

板野　肯三（正会員）

1948 年生．1977 年東京大学大学院理学系研究科物理学専門課程単位取得後退学．1993 年より筑波大学電子・情報工学系教授．計算機アーキテクチャ，分散システム，プログラミングシステム等に関する研究に従事．理学博士．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEE CS 各会員．