社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＣＳＥＣ－22　(17)
2003／7／17

Cryptographic Memory System

Get high with a little help from my kernel

稲村　雄†

† 株式会社 NTT ドコモ マルチメディア研究所　〒239–8536 神奈川県横須賀市光の丘 3–5
E-mail: †jane@mml.yrp.nttdocomo.co.jp

あらまし　SSL，SSH，IPSec 等の現実的な保護機構が整備されつつあるネットワーク環境と比較して，計算機内部に関する保護機構はまだ十分とはほど遠い状況に留まっている．本稿ではそのような計算機内部の実行環境を改善するための一方式として“暗号化メモリシステム”なる手法を提案する．

キーワード　暗号技術，メモリ保護，マルチタスク OS

Cryptographic Memory System

Get high with a little help from my kernel

INAMURA YU†

† Multimedia Laboratory, NTT DoCoMo, Inc.　3–5, Hikarinooka, Yokosuka, Kanagawa, 239–8536 Japan
E-mail: †jane@mml.yrp.nttdocomo.co.jp

Abstract　Currently many practical protection mechanisms such as SSL, SSH, and IPSec are actively being deployed in the Internet. However, there're far less such practical mechanisms for the individual computer's internal execution environments. This paper proposes a “Cryptographic Memory System”, which aims at achieving the improvements over such internal execution environments.

Key words　Cryptography, Memory Protection, Multi Task OS

1. はじめに

近年のインターネットの爆発的な普及およびその上で価値ある情報を流通することへの欲求は，各種のいわゆる“セキュア・プロトコル”の考案そして普及へと繋がってきた．主として Web トランザクションを保護するために用いられる SSL，安全な遠隔ログインを提供する SSH，IP レベルでのセキュリティ機構である IPSec 等，ネットワーク通信に対する保護機構は多彩な品揃えを誇っている．これらのセキュアプロトコルはいずれも暗号技術を用いることで (1) 通信経路上でのデータ秘匿性の確保，(2) 通信相手の認証，(3) データ完全性保証といった機能を実用的なレベルで提供することが可能だ．

しかし，その通信経路の終端となる計算機内部実行環境に対する保護機構は残念ながらまったく不十分なものと言える．たとえば，前述した安全な通信を行なうためには暗号化/復号鍵等の秘密情報を計算機のメモリに保存しなければならないが，後で説明するように実在するほとんどすべての計算機/OS では，そのような秘密情報を他の同時に実行されている悪意ある（かもしれない）プロセスから効果的に隠蔽する手段を備えていないのである．

いかに強力な保護が通信経路上のデータに対して適用されていたとしても，そのようなデータに対して送信もしくは受信機器上で不正にアクセスできてしまうのならば，その保護は意味をなさないだろう．そのような行為を可能とするマルチタスク環境は，現状のモバイル通信機器としてはまだ一般的ではないが，システム高性能化シフトの傾向が今後も続くとすれば，将来的に，たとえば 10 年後の移動通信の世界では当然考慮されるべき危険だと考えられる．

本稿ではそのような計算機内部での安全な実行環境実現の方策に関する議論を行ない，暗号技術を用いた安全なマルチタスク環境に関して提案を行なう．

2. 問　題

計算機の内部環境を安全にできない主たる理由は，大半の既存計算機オペレーティングシステム（OS）が仮想的に複数のプロセスを同時実行可能とする，いわゆるマルチタスク環境をサ

ポートしている点にある．そのため，あるプロセスが同時に実行されている他のプロセスに対してなんらかの干渉を加える可能性が避け難くなっているのである．

もちろん，現代的なOSにはハードウェアメモリ管理機構を利用した仮想記憶空間機能が備わっており，一つのプロセスの持つメモリ空間は当該プロセスが意図的にそう指示しない限り，通常の操作では当該プロセスのみしかアクセスできないように図られてはいる．しかし，最も現代的かつ一般に普及したOSであるUNIX系OSもしくはWindows(注1)であっても，以下に述べるような問題点を抱えているため，計算機そのものへの侵入を許した場合に実行中のプロセスが持つ秘密情報を確実に保護することは非常に難しい．

2.1 Core file

Core fileとは典型的には実行中になんらかの不具合を発生したプロセスが，検死解剖(注2)型デバッグによる原因究明を可能にするために生成するファイルであり，当該プロセスが不具合発生時点で持っていたメモリ内容を含む実行環境すべてが書き出されたものだ．そのため，あるプロセスが生成したCore fileにアクセスできれば当該プロセスが保持していたすべての情報を精査することが可能である．

シグナルと呼ばれる非同期プロセス間通信機構を用いれば，大半のプロセスに任意のタイミングでCore fileを生成させるように仕向けることが可能なため，攻撃者は自分がシグナルを送ることが許されているプロセス(注3)に関してはほぼ自由に内部状態を探ることができる．

2.2 Swap/paging

Swap/pagingというのは，実装メモリ容量を越える仮想メモリ空間の利用を可能とするために提供される仕組である．このような仕組を備えたOSでは，実メモリが足りなくなった際に利用頻度の低いメモリ領域の内容を外部記憶装置に待避し，当該待避済み領域を利用可能とする，という処置が取られる．

実メモリ上に置かれている状態では，メモリ管理機構により他プロセスからのアクセスは効果的に制限されるが，それが外部記憶装置に書き出されてしまえば単純にファイル/ディスクに対するアクセス管理機構で保護されるのみであり，障壁はずっと低いものとなる．

2.3 デバッガインターフェース

大半のOSには実行途中でのプロセスデバッグを可能にするための仕組が用意されている．たとえば，UNIX系OSではptraceシステムコールにより実行中のプロセスに接続し，以降の当該プロセスの実行を観察/制御することが可能である．この仕組が計算機内部に侵入した攻撃者に悪用されると，当該計算機上で実行される任意のプロセスに接続し，その内部情報に対して自在にアクセスする道が開かれることになる．

一般にptraceを用いたプロセス制御は対象プロセスが実行されているのと同じユーザもしくは特権ユーザのみが行なえるようになっているため，計算機上で特権ユーザ権限を奪取されることは，イコール当該計算機上で実行中のすべてのプロセスが保持する秘密情報が危機に瀕することに等しい．

3. 従来の解決策

前章で見てきたように，現在利用されている一般的な計算機およびOSでは，内部に攻撃者が存在する場合，実行中プロセスが保持する機密情報を保護するための仕組は決して十分なものとは言えない．

これに対抗する手段としては現在大別して以下の二つの方向が検討されているように見受けられる．

権限細分化 UNIXにおけるrootのようにシステムに対してほぼ全能の権限を持つ特権ユーザを可能な限り排除することで，システムに侵入を受けた場合の影響を最低限に抑えようとするもの．各種セキュアOS[1]〜[3]で用いられているのがこの方向性

耐タンパ性ハードウェア利用 システムに耐タンパ性を持ったハードウェアを導入し，機密情報をそのハードウェア的な耐タンパ性によって保護しようというもの．TCPA（Trusted Computer Platform Alliance）等[4],[5]が用いているのがこの方向性

しかし，これらの方向性に関しては

- 前者では

（1） 細分化された権限に応じて適切なアクセス制御設定を行なう必要性

（2） 全能プロセスが存在しないことに伴うシステム管理処理等の繁雑さ

といった点でシステムの使い勝手に対する悪影響が出る他，侵入者が奪取に成功したユーザ権限で動作するプロセスに関しては，十分な保護が与えられない．

- 後者は特殊なハードウェアを必要とするため，現在一般的に普及している大半のシステムには適用不能．かつ，すべての機密情報を耐タンパ性ハードウェアに格納することは，ある程度以上に大規模なシステムでは現実的ではないため，どこかで機密情報をメモリに置くタイミングは持たざるを得ず，そのタイミングで攻撃される可能性は否定できない．

といった問題点が挙げられる．そのため，計算機の内部環境における保護策は現状では不十分なままと言えるだろう．

また，特に2.2で挙げたswap/pagingのみに関しては，ディスクに書き出されるデータを暗号化するという方式がOpenBSD等のOSに導入されているが（[6]），この機能のみでは他の脅威には対抗できない．

4. 暗号化メモリシステム

これまで述べてきた問題，すなわち，計算機内部に侵入した攻撃者から，同計算機上で動作するプロセスが持つ機密情報をいかに安全に保護するか，という点に関しては，なんらかの形で暗号等の情報秘匿技術を利用した対抗策を考案する必要があると考える．

(注1)：ただし，ここではNT/2000/XP等の厳格なプロセス管理機構を備えたバージョンを指す

(注2)：postmortem

(注3)：通常は同じユーザIDの下で実行されるプロセスだが，特権ユーザの場合は任意のプロセスに対して送信可能

たとえば，1993年に提案された“A Cryptographic File System for UNIX”（[7]）は，暗号技術によりハードディスク等に保存されるファイルに対して保護を提供する，という最初期の試みの一つであるが，本稿において提案しようとしているのは，同様の暗号技術による保護を，ファイルに対してではなく，メモリ上のデータに対して適用可能なシステム＝“暗号化メモリシステム”である．

このようなシステムを実現するための障害と考えられるのは主として以下の二点であろう．

処理速度　暗号処理を行なう対象がメモリということは，ネットワークトラフィックやHDDよりも桁が違う速度が要求されるが，それは可能なのか?

鍵の保管方法　暗号化によりメモリ上のデータを保護すると言っても，そのために用いる鍵はどのように保護するのか?

これらに関して以下に考察し，現実的な範囲で実現可能な暗号化メモリシステムの構成について説明する．

4.1 性能への影響

最初は暗号処理性能がシステム全体の性能に対して及ぼす影響に関してである．上にも述べた通り，メモリに対するアクセス速度はネットワークもしくはHDD等と比較すると一桁程度上回るものである．そのため，ネットワークトラフィックもしくはファイルシステムのようにすべてのデータを包括的に処理するということには，たとえ高速な共通鍵暗号であっても現実的ではないだろう．

内外のいくつかの特許として，CPU内部に暗号プロセッサを実装し，CPU内部キャッシュと外部キャッシュ/メモリとの間のデータ転送時に暗号化（Cache write-back/through）/復号（Cache fill）を行なう，というシステムが提案されている（たとえば，[8]）．しかし，十分な性能を得るためにはCPUのコストが高くなり過ぎるためか，実際にそのようなシステムが製造されたという話は聞かないし，汎用CPUと暗号専用プロセッサの性能/コストバランスに関するトレンドを鑑みれば，将来的にも実用的なシステムが量産機に搭載されるということも考え難そうだ．

ただし，現在のCPUが到達した性能レベルを考慮すると，従来型の性能追求を一義としたシステム設計が，今後はある程度性能を犠牲にしてもセキュリティに力を入れるべきとのシステム設計哲学へとシフトする可能性もある．たとえばBruce Schneierらの近著[9]は，『CPU能力の90%をセキュリティに捧げたとしても問題ない』と論じているが，それは極論としても，より多くの能力を純粋性能以外に割り振る余地が生まれてきているのは間違いなかろう．その結果として，将来的には完全にリアルタイムでメモリの暗号化/復号を行なうシステムが実用化されることもあり得る．

ただし，本稿ではあくまでも現時点での技術レベルで実用的な暗号化メモリシステムを実現することを目指す．そのため，後で説明するように暗号技術を適用するのはメモリのごく一部，すなわち，セキュアプロトコルで必要とされる復号鍵のごとき機密情報を格納する部分のみ，とすることで全体的な性能低下を防ぐ，というのが基本的スタンスである．

4.2 鍵の機密性の確保

メモリを暗号化/復号するための鍵自体の機密性に関してだが，当然，通常のユーザメモリ領域中にこれを置いておくわけにはいかないことは明らかだろう．それでは，必要な領域を暗号化して保護したつもりになっていたとしても何の意味もなさない．2.でも述べた通り，攻撃者がターゲットシステム上で十分な権限を得ることに成功しさえすれば，実行中プロセスのメモリから暗号化されていない情報を取り出す手段は複数存在するからである．

かと言って，それ以外のところに，しかも，特権ユーザに属するプロセスであってすら読み出すことができないよう暗号鍵を保存するということは一見できなさそうに見える．しかし，これは単に特権プロセスであればシステム上でどのような操作も可能である，という非常によく見られる誤解に基づく誤認識と言えるだろう．現実には，システム上で真に万能なのはOSカーネルであって，一見万能に見える特権プロセスに可能なのはカーネルが明に許可した操作のみ，なのである．

そこで，メモリを暗号化により保護するための鍵はOSカーネルが自身のみしかアクセスできない領域に安全に保管し，守秘に必要なオペレーションはすべてカーネルが行なう，というモデルが成り立つと考えられる．

4.3 暗号化メモリシステム概要

それでは本システムの概要を以下に述べよう．4.2でも触れた通り，これはOSカーネルの持つ全能性に大きく依拠したシステムと言える．

本システムの基礎となるのは，一般的なマルチタスクOSに関する次のような観察である．

観察　複数のプロセスを同時に実行することができるマルチタスクOSであっても，実際には本当に複数プロセスが同時に実行されるわけではない．各プロセスに対して一定のタイムスロット(注4)が割り当てられ，実行中のプロセスが割り当てられたタイムスロットを使い果たすか，もしくはなんらかの理由で実行がブロックされた場合，CPU使用権はカーネルによって他の実行待ちプロセスに割り当てられる．

このようなプロセス切替えという措置（＝コンテキストスイッチ）は黒子であるカーネルによってプロセスにはまったく見えないような形で執り行われるため，一つのプロセスにとってはあたかも己のみが計算機を独占しているように見えながら，仮想的に複数のプロセスが単一の計算資源を共有することが可能となる．

コンテキストスイッチの際，カーネルは一つのプロセスの実行を止め，当該プロセスが利用している計算コンテキスト（レジスタ等のCPU内部資源）を再開可能なように保存した後，他の実行待ちプロセス群のなかから適当な一つを取り出し，保存されたコンテキストを展開した上で同プロセスの実行を再開させる，という処理を行なう．実際のユーザプロセスが動く舞台裏では，カーネルがこのような処置を忠実に実行し続けているのだ．

(注4)：BSD系UNIXでは一般に100msec

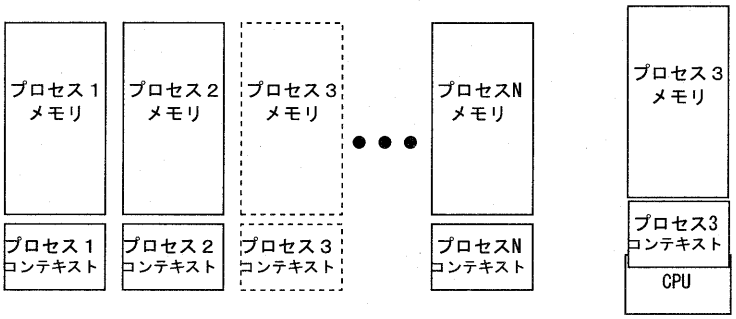

図 1　マルチタスク OS 実行時スナップショット
Fig. 1　Snapshot on multi task OS's execution

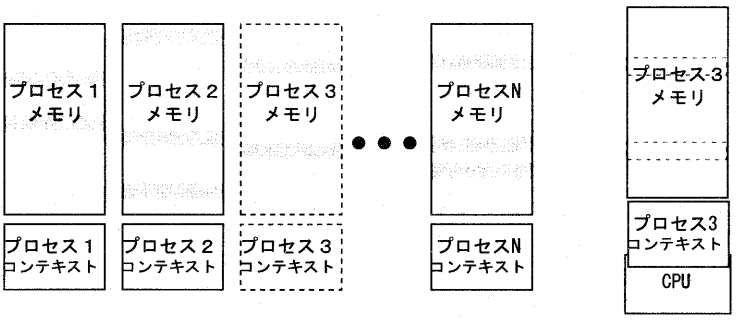

図 2　暗号化/復号機能の追加
Fig. 2　Adding encryption/decryption functionalities

そのため，マルチタスク OS においてある瞬間のスナップショットは，図 1 のようなものとなる．一つのプロセスが計算資源を独占した形で実行を行なっており，それ以外のプロセスは必要な実行コンテキストデータとともに休眠状態に置かれている．カーネルは適切なタイミングで実行中のプロセスと休眠状態のプロセス群のなかの一つを入れ替えることで，すべてのプロセスに対してほぼ公平な計算資源の分配を実現する．

この事情は相手が特権ユーザのもとで動くプロセスだったとしてもまったく同じことで，一見全能と見える特権プロセスであってもこのカーネルによる呪縛からは逃れることはできない．2.3 で述べた通り，特権プロセスはシステムが提供するデバッガインターフェースを利用して同時に実行中のあらゆるプロセスの内部状態を精査することが可能だが，その際に特権プロセスが見るターゲットプロセスは実行状態にあるのではなく，図 1 に示すようにカーネルによって休眠状態とさせられているに過ぎないのである．

以上のような観察は，ただちに次のようなシステムの可能性へと繋がる．すなわち，現状の OS カーネルが日常的に行なっているコンテキストスイッチのタイミングで，休眠状態に移行させられるプロセスの機密情報保持領域を暗号化し，同時に実行を再開させられるプロセスの機密情報保持領域を復号する，というものだ．このシステムのスナップショットは図 2 のようになるだろう．唯一の実行中プロセスに関してはメモリ上のすべてのデータが平文状態であり，休眠中プロセス群に関しては図 1 に加えて機密情報を含むメモリ領域の一部がカーネルによって暗号化された状態で保存させられている．この暗号化/復号のために用いられる鍵はカーネルのみが知っていさえすれば良いため，4.2 でも述べた通り，個々のユーザプロセス空間とは独立したカーネル空間に置いておくことができよう．

最後に，メモリ空間中の機密情報が置かれている位置を如何に判断するか，という点を説明しよう．4.1 で紹介した Bruce Schneier らの議論を是とするならば，いずれはメモリ空間すべてを等しく暗号化により保護するという戦略が現実的となる可能性もあるが，現状ではそれはやり過ぎと考えている．

さて，プログラム中でどの変数が機密情報を含むのかということを把握しているのは，当然プログラムの設計者自身のみだ．そのため，その情報がプログラム中で伝えられることなしにカーネルが適切に判断することは不可能であろう．また，変数のアドレスに関する情報は基本的にはプログラムを実際に走らせる際になって初めて確定するものであるため，そのような動的な情報をカーネルに伝えるには特別なシステムコールが必要である．

当該システムコールの利用法としては以下のような方向性が考えられる．

- 動的メモリ割り当て時（C 言語における malloc 等）の対処．salloc（secure allocation の意）等というライブラリ関数を提供し，その関数のなかで領域の割り当ておよびカーネルへ情報伝達するためのシステムコール呼出を実行する

表 1　概念実証用実装仕様
Table 1　Spec. of Proof-of-concept Implementation

CPU	Intel Pentium III 850MHz
OS	OpenBSD3.2
保護領域指定方法	ライブラリ関数 salloc() による動的割り付けのみ（自動変数等への指定のサポートはなし）
保護領域管理方法	割り付けられた領域中に設けた変数領域による双方向リンクによる
暗号アルゴリズム	Rijndael /w 128bit key. CBC モード
鍵	ブート処理のなかでプロセス毎に疑似乱数鍵を割り当て
IV	128bit all 0 データを Rijndael で暗号化した値

- 変数宣言時の対処．自動/外部/静的変数等として宣言された変数領域のうち必要な部分に対してシステムコールが適用されるように，コンパイラを改造する，もしくは，前処理を行なう．

カーネルは実行中のプロセス毎にシステムコールで通知された要保護領域を管理し，コンテキストスイッチのタイミングでそれらを暗号化／復号することで，個々のプロセスは他のプロセスからの機密情報読み出しを防ぐことができるようになる．

以上が本稿の主題である暗号化メモリシステムの基本的な概念だが，これには以下のような利点がある．

（1） リアルタイムでの暗号化を行なわないため，パフォーマンスに対する影響が小さい

（2） 特別なハードウェアなしに実現可能

（3） キャッシュ上にあるデータをそのまま処理できるため，ライトバック等の措置が不要

（4） Swap/paging に対する保護も自動的に行なえる．

4.4 概念実証用実装

4.3 で説明した概念が実際に動作可能であることを確認するために試験実装を行なったので，ここではこの概念実証用実装を簡単に説明する．まず，テストベッドとした OS 等の仕様は表 1 の通りである．

このシステムで割り付けられた保護領域のサイズが実行性能に与える影響を表 2 に示す．これは 1000byte から 64000byte まで割り付ける領域を 2 倍ずつ増やしながらそれぞれのサイズで実行時間がどのように変化するかを示している．プログラムのなかで行なっている処理は単に割り付けた領域にランダムデータを書き込んでいるだけなのだが，全体の実行処理量がほぼ等しくなるように書き込み回数を加減している．比較対象とするために，保護領域としてではなく通常の malloc を用いて割り付けた場合の実行結果を同表の第 3 カラムに，性能低下率を第 4 カラムに記してある．

表に示した通り，8000byte まではおおむね 5%〜7%の性能低下という程度の範囲だが，これは利用 CPU が Intel Pentium III ということで，L1 データキャッシュの容量内で済むためであろう．測定時には事実上このプログラムだけが走っている状況なので L1 データキャッシュはほぼ占有できていると思われる．割り付け領域が 16000byte を越えると顕著な性能低下を示すため，実用的な範囲と考えられるのは 16000byte 程度までということになるだろうか．このシステムで保護する対象としてもっとも有望と考えられるのは公開鍵暗号で用いる私有鍵ということになるが，私有鍵データのサイズは現在一般的な RSA 1024bit 鍵を用いる場合，CRT による最適化を適用している場合であっても 3584bit （約 450byte） 程度であるため，実際に私有鍵を利用した処理の際に必要とされる一時変数領域分まで考慮したとしても，上記程度のサイズで実用的には十分と考えられるだろう．

表 2　保護領域サイズとシステム性能の関係
Table 2　Performance penalty with the amount of protected region

サイズ	“s” 実行時間 （秒）	“m” 実行時間 （秒）	性能低下率 （%）
1000	33.2	31.5	5.4
2000	33.4	31.5	6.0
4000	33.8	31.5	7.3
8000	34.6	32.2	7.4
16000	36.4	32.4	12.3
32000	40.2	32.0	25.6
64000	53.2	33.0	61.2

また，本システムの実際的な有用性を示すために，実際の私有鍵利用アプリケーションを改造した例を以下に示そう．ここで例題として用いたのは OpenSSH に附属する “ssh-agent” というプログラムであり，このプログラムにユーザの私有鍵を記憶させておくと，SSH を用いて遠隔計算機へのログインを行なう際の認証処理をユーザに代わって行なわせることが可能となる．ただし，オリジナルの ssh-agent の場合には記憶した私有鍵は平文状態でメモリに格納されるため，2. で触れた通りシステム内に潜む攻撃者からは各種の攻撃を受ける可能性があるわけだ．

本システムの機能を用いて私有鍵保管領域が保護されるように改造した ssh-agent プログラムを実行し，gdb で接続して私有鍵（RSA における d）を読み出すという擬似的な攻撃を行なっているのが図 3 である．通常のデバッグ時と同様なんら障害なくデータの読み出しが行なえるわけだが，同私有鍵が保護領域に置かれているために本来の値とはまったく異なるデータとしてしか読み出せない．そのことは同鍵をコマンドラインからダンプさせたところを示した図 4，そして，本システム対応の改造をしていない素の ssh-agent を相手にして gdb で読み出した例を示した図 5 により明らかであろう．

5. 議論および今後の計画

以上，カーネルの手助けを借りることにより，同一計算機内で同時に稼働する他プロセスによる機密情報の盗み出しに関して十分な耐性を備えたシステムの概念を説明し，概念実証用実装を用いたいくつかの結果を提示した．

これらの結果より，この方式で全メモリ空間を保護することは現実的とは言えないことは明らかだが，同時に，一般的な私有鍵程度のサイズの情報を秘匿するために適用した場合には，

図 3　改良 ssh-agent メモリ中の RSA 私有鍵の内容
Fig. 3　RSA private key in the improved ssh-agent

図 4　ファイル中の RSA 私有鍵の内容
Fig. 4　RSA private key stored in file

図 5　オリジナル ssh-agent メモリ中の RSA 私有鍵の内容
Fig. 5　RSA private key in the original ssh-agent

通常実行時とそれほど遜色のない性能でシステムが実現できるということが示せたと考えている．

特にハードウェア的には通常のものとなんら変哲のない PC-AT 互換機で実現できているため，現状の多くの計算機環境に適用することが可能という点は大きいだろう．

また，今後の課題としては以下が挙げられる．

自動変数等への拡張　現状は動的割り付け領域のみのサポートであるが，これを自動/外部/静的変数等にも適用可能とするような拡張を検討する．

SMP システム対策　SMP（Symmetric Multi-Processor）システムでは本稿で説明したような疑似的なマルチプロセス実行環境に加えて，複数の CPU を用いて複数のプロセスを真に同時並列実行することが可能であり，攻撃プロセスが並列動作している他のプロセスに干渉する危険性が生じる．そのため，SMP システムの場合，通常は排他制御に用いられるロック機構を援用する等の対策を導入する必要があるだろう．

GC 機能を持つ言語への対応　Java のように処理系として GC (Garbage Collection) 機能をサポートしている言語の場合，データが置かれているアドレスが暗黙のうちに変わる可能性がある．そのような言語をもサポートするためには，GC の処理期間を通じて保護領域が漏れなくカバーされるような手立てを講じる必要があるが，単純な実装では GC 実行時点で保護領域が 2 倍の容量となり性能に影響を与える可能性があるかもしれない．

カーネル中の鍵保管領域の保護　現状の OS ではカーネルメモリを読み出すインターフェース(注5)を持ったものがあり，そのようなシステムでは単にカーネルメモリ中に鍵を保管したというだけで安全とは言い切れない．CPU アーキテクチャが許すのであれば，当該メモリ領域はカーネルコード実行時以外にはアクセス不可能となるようにハードウェア的な設定を行なうべきであるが，各種 CPU でどうすれば可能となるかは今後の検討事項である．

6. おわりに

以上，ネットワーク環境に対する保護策の充実度と比較すると非常に見劣りする計算機内部環境に対する保護を実現するための一方式を提案した．従来，適切な保護策を講じることが困難と考えられていたためか，この種の危険は効率的な侵入防止措置さえあれば大事ないものとして受忍されてきたと感じるし，また，パフォーマンスを一義とする風潮からもこのような対策は好まれない傾向があっただろう．しかし，近年の各種ワーム事件でも明らかな通り，システム内部への侵入というのは十分現実的な危険であり，そのような事象への対抗手段の確立も急務となりつつある．また，いかにネットワークが安全に使えるようになったとしても，その終端ポイントに明らかな穴があるとすれば，その効果は半減してしまう．その意味で，今後も計算機内部環境の安全を確保する手法の研究を，より活発に行なう必要があると考えている．

文　献

[1] Hewlett Packard, “HP Virtual Vault”, http://h71019.www7.hp.com/HP/render/1,1001,6288-6-100-225-1,00.htm.

[2] Sun Microsystems, “Trusted Solaris Operating System”, http://wwws.sun.com/software/solaris/trustedsolaris/.

[3] National Security Agency, “Security-Enhanced Linux”, http://www.nsa.gov/selinux/index.html.

[4] Trusted Computing Platform Alliance, “TCPA – Trusted Computing Platform Alliance”, http://www.trustedcomputing.org/tcpaasp4/index.asp.

[5] Trusted Computing Group, “Trusted Computing Group: Home”, http://www.trustedcomputinggroup.org/home/.

[6] Niels Provos, “Encrypting Virtual Memory,” 9th USENIX Security Symposium, Colorado, USA, August 2000.

[7] Mat Blaze, “A Cryptographic File System for UNIX,” First ACM Conference on Computer and Communications Security, Fairfax, VA, November 1993.

[8] 橋本 幹生，藤本 謙作，“マイクロプロセッサ、これを用いたマルチタスク実行方法、およびマルチレッド実行方法”，公開特許公報 特許出願公開番号 特開 2001—318787.

[9] Niels Ferguson and Bruce Schneier, “Practical Cryptography”, John Wiley & Sons, April 2003.

(注5)：BSD 系 OS における/dev/kmem 等