Vol. 45 No. 8 情報処理学会論文誌 Aug. 2004

暗号技術によるメモリデータ保護方式の提案

稲 村 雄[†] 本 郷 節 之[††]

SSL，SSH，IPSec 等のいわゆるセキュアプロトコルを利用することにより，インターネット等の公衆ネットワークを介してプライバシ保護の必要な情報を安全にやりとりすることは，現在では十分現実的なオプションとなっている．しかし，そのようなネットワークの末端となる個々の計算機の内部実行環境を見ると，プライバシを要するデータの機密性/完全性を保護する機構はほとんど用意されていない．そのため，今後はこの内部実行環境に存在する脆弱性を突いた形でプライバシ情報を含む機密データを奪取するという形の攻撃に対する防御が重要性を増すと考えられる．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，暗号コプロセッサ等を持たない一般的な計算機ハードウェア（H/W）およびソフトウェア（S/W）で実装可能な“暗号化メモリシステム”なる手法を提案する．

A Proposal for Memory Data Protection Scheme Using Cryptography

Yu Inamura[†] and Sadayuki Hongo[††]

Currently it has become a practical option to use the so-called secure protocols such as SSL, SSH, and IPSec, in order to use public networks like the Internet as exchanging media for privacy-related data securely. However, the terminating points of such protocols, that is, server machines or PCs, have a little or no means in their execution environments, which can be utilized to keep the confidentiality or the integrity of such privacy-related data that will be sent to or have been received from network. This means that it will become more and more important to provide protecting mechanisms for such attacks that will take advantage of the above vulnerabilities in the most of the current computers' execution environments. This paper proposes a "Cryptographic Memory System", a scheme for aiming at achieving the improvements over such internal execution environments, which can be implemented on the most ordinary computer H/Ws and S/Ws.

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つながってきた．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に関する研究/開発は十分な成果を産み，かつ現在でも精力的に続けられている．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であり，重要な価値を持つ情報やプライバシを要する情報を公衆ネットワークを介して低コストで送受信できる環境が整いつつある．

しかし，その安全な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いえる．たとえば，前述した安全な通信を行うためには暗号化/復号鍵等の機密情報を計算機のメモリに保存しなければならない．また，通信相手から受けとったプライバシ関連情報等も同様にいったんはメモリに保存されることになる．しかし，現存するほとんどすべての計算機/OS は，そのような要注意データ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，当該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

† 株式会社 NTT ドコモマルチメディア研究所
Multimedia Laboratories, NTT DoCoMo, Inc.
†† 株式会社 NTT ドコモネットワークマネジメント開発部
Network Management Development Department, NTT DoCoMo, Inc.

ば，プライバシに対する重大な脅威となりうる．

前述した純粋にネットワークレベルの各種セキュアプロトコルとは異なり，電子メールメッセージを対象とするセキュアプロトコルであるS/MIME等の場合，通信経路上のみではなく目的の計算機上に届いた後の時点でも保護を提供することは可能である．しかし，そのように保護されたメッセージであったとしても，当該メッセージを読むためには暗号化等の保護を解除しなければならず，したがってたとえばS/MIMEメーラのような対応アプリケーションがメッセージを読み込み復号したあとは，当該アプリケーションが起動している間，メモリ中の復号済みメッセージに対して不正アクセスを試みられる可能性がある．

このように計算機上で他のプロセスに対する干渉を可能とする環境は，PCやワークステーション，サーバ等のプラットフォームではきわめて一般的である．個人利用のPCであっても，ユーザが直接動かしたプロセス以外に，各種サービスを行う等の目的でバックグラウンドで動くプロセスも多く存在するため，メモリデータに対する効果的な保護策は早急に実現する必要のある重要課題である．また，現状のモバイル通信機器ではそのような環境はまだ一般的ではないものの，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，同様の対策が必要となるものと考えられる．

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い，暗号技術を用いた安全なマルチタスク環境を提案する．

2. 問　　題

現状で計算機の内部実行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存在するのである．

もちろん，現代的なOSにはCPUの持つH/W等を利用した仮想記憶空間機能が備わっており，1つのプロセスが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☆であっても，以下に述べるような問題点が存在するため，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☆☆☆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る．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．

2.2 Swap/paging

Swap/pagingというのは，実装メモリ容量を超える仮想メモリ空間を利用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，その結果空くことになるメモリ領域を再利用する，という処置がとられる．この操作で外部記憶装置に退避されたデータに対する保護は，ディスクに対するアクセス管理機構のみとなるため，実メモリに比べるとずっと弱いものとなる．

2.3 デバッガインタフェース

大半のOSにはプロセス実行途中での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，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. 目標システムおよび従来の対策

3.1 目標システム

2.3節であげたデバッガインタフェースを利用する攻撃は，単にメモリ上のデータを読み出すだけにとどまらず，必要であれば機械語レベルでのステップ実行

☆ 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．

☆☆ サーバアプリケーションに存在するセキュリティホールの利用，ウイルスによる感染等．

☆☆☆ postmortem.

といった解析行為も可能となるため，危険度は最も高い．逆にこの攻撃に耐えられるシステムは安全性のレベルも高いといえるだろう．そのため，本稿では計算機上で管理者権限を持つプロセスが攻撃者の手に落ちたとしても，デバッガインタフェースにより他の実行中のプロセスからプライバシ関連データを含む機密情報を奪取することはできない，というシステムを目標とする．

なお，ここで想定している環境は，各種のプロセスが稼働しているサーバもしくはクライアント PC であり，そこに侵入を果たした攻撃者が現に動作中のプロセスから機密情報を盗み出すことを阻止することが目的となる．それ以外の攻撃，たとえば，実行環境そのものをエミュレートしてその上で 1 からプロセスを実行させることにより内部の機密情報を盗む，といった攻撃は，上記攻撃と比較すると難度が高く，かつ，正規ユーザに露見する可能性も高いものとなるため，検討の対象とはしない．

もちろん，この目標自体はデバッガインタフェースそのものを OS レベルで無効化することでも容易に実現可能なはずだが，それでは正規ユーザ自身も同計算機上でデバッグが行えないことになるため，使い勝手が大きく損なわれてしまうだろう．

3.2 従来の対策

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けっして十分なものとはいえない．

これに対抗する手段としては現在大別して以下の 2 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．

耐タンパ性 H/W 利用　システムに耐タンパ性を持った H/W を導入し，機密情報をその H/W 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．

しかし，これらの対策に関しては

- 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う必要性

(2) 全能プロセスが存在しないことにともな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ほか，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と考えられる．

- 後者は特殊な H/W 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できない．かつ，すべての機密情報を耐タンパ性 H/W 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えず，そのタイミングで攻撃される可能性は否定できない．

といった問題点があげられる．計算機の内部環境における保護策は現状では不十分といえるだろう．

また，2.2 節であげた 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対処方式が 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．

たとえば，1993 年に提案された “A Cryptographic File System for UNIX” [7)] は，ハードディスク等に保存されるファイルに対して暗号技術による保護を提供する，という最初期の試みの 1 つであるが，本稿において提案するのは，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．

ただし，文献 7) はディスク上にあるデータを保護するための機構であり，(1) メモリ⇔ディスク間でのデータ転送に付随して暗号化/復号するという明白な処理タイミングの存在，および (2) 比較的アクセス速度の遅いディスク上のデータに対する処理となるため処理性能がシビアには要求されない，という実装面に関して有利な性格を備えていた．そのためもあって比較的早い時期に現実的な実装も可能だったといえる．

しかし，本システムが目指すのはメモリ上のデータを同時に動作する他プロセスから読めないような形に暗号化するという仕組みであり，文献 7) のように明らかな処理タイミングもなく，かつ，対象とするのもメモリというアクセス速度の速いデバイスである，という難条件を克服しなければならない．

また，暗号化によりメモリ上のデータを保護するといっても，その暗号化/復号処理で用いる鍵をどのように保護するのか，という問題もある．当然，そのような鍵を平文状態でプロセスのメモリ空間上に置くというのは論外である．

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

4.1 処理タイミング

現状のH/WおよびOSに一般的に見られる性質を前提とすれば，同時動作中の他プロセスからのメモリ読み出し攻撃に対抗可能な暗号化メモリシステムを実現するには，暗号処理を

(1) メモリ管理機構に存在する階層構造を利用したデータ転送処理時点で実施する（空間的実現）
(2) 実行プロセス切替え時点で実施する（時間的実現）

という2つの可能性が考えられるだろう．

(1)は基本的には文献7)と同様の考え方といえる．現行計算機システムの大半はCPUとメモリの動作速度差に起因する非効率性を緩和するため，階層的なH/W構造を持っている．CPUから見て遠く遅いメモリ上のデータに直接アクセスするのではなく，CPUに近い，もしくは，CPU内部に存在するキャッシュにいったんデータを読み込んだうえで処理を行う．このメモリ⇔キャッシュ間の転送を文献7)におけるディスク⇔メモリ間の転送と同じアナロジと考えれば，メモリ⇔キャッシュの転送処理時点でデータの暗号化/復号を行うというシステムに行き着くのはごく自然な発想である．

内外のいくつかの特許としてそのような機構の提案が存在するが（たとえば，文献8)），この方式の明らかな欠点はCPUレベルでの特別なH/Wの存在が必須となる点だろう．メモリ⇔キャッシュ間の転送は通常S/Wには透過的な形でH/Wにより実行されるからである．また，キャッシュはOSが管理する単位であるプロセスとは独立な存在であるため，プロセス管理機構との親和性も持たない．この方法で機密データを他プロセスから保護するという所期の目的を実現するためには，プロセスごとにキャッシュを切り替えるといったさらに余分な操作が必要となるだろう．

一方，(2)はOSの提供するプロセス切替えという機構に着目した方式である．単一CPUの計算機で複数のプロセスが同時に実行されているように見せるためには，ユーザに認識できない短い時間間隔で各プロセスの実行を切り替えなければならない．その切替えを実施するタイミングを同時にメモリ上のデータを暗号化/復号するタイミングとしても用いることができるのではないか，というのが発想の原点である．この切替え処理は完全にOSの制御下にあるため，(1)の場合とは異なりこれは特別なH/Wなしで実現できるし，当然プロセス管理機構との親和性も良い．

本稿では現時点での一般的なH/Wを仮定するため，実行タイミングに関しては(2)を選択する．

4.2 処理性能

アクセス速度の速いメモリ上のデータを扱うという困難性克服法としては，メモリ空間全体ではなく，プライバシ関連情報等の機密データを格納する特定部分のみをピンポイントで処理する，という戦略をとる．一般にデータはプログラム上で決められた変数に格納されるものであり，どの変数にどのデータが格納されるのかはプログラマによって完全に制御される．そのため，プログラマにより機密データが置かれる変数に関する指示を受けることにより，メモリ空間中の必要な部分のみを効率的に保護することが可能である．

4.3 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，前述のとおり，通常のユーザメモリ領域中にこれを置くというのは論外である．ユーザ領域に置かれたデータを他プロセスから読み出せるというのがそもそもの前提なのだから，それでは意味がない．

これを解決するために，本稿では，メモリ暗号化保護のための鍵は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をすべてカーネルが行う，というモデルを提案する．4.1節ですでに処理タイミングをOSカーネルによるプロセス切替え処理時とするという戦略をとったため，このモデルも自然に実現できる．

たとえ管理者権限の下で動くプロセスがデバッガインタフェースを駆使したとしても，カーネル空間上のデータにアクセスすることはできない．ゆえにカーネル空間上に暗号化鍵を置くことで，3.1節に記した目標を満たすことができる．

なお，OSカーネル空間に置かれたデータの機密性に関してはデバッガインタフェースのほかにも考慮すべき点があるが，それに関しては後述する．

4.4 暗号化メモリシステム概要

本システムの概要を以下に述べる．まず，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 複数プロセスの同時稼働が可能なマルチタスク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．各プロセスには一定

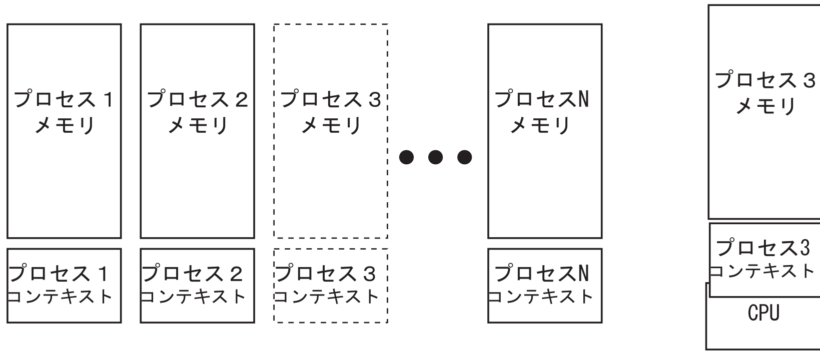

図 1 マルチタスク OS 実行時スナップショット
Fig. 1 Snapshot on multi task OS's execution.

のタイムスライス☆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ライスを使い果たすか，もしくはなんらかの理由で実行継続不能となった場合，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措置（= コンテキストスイッチ）は，動作中のプロセスにはまったく感知不能な形でカーネルによってとり行われるため，1 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，カーネルは実行中のプロセスが利用している計算コンテキスト☆☆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中から適当な 1 つをとり出し，保存されたコンテキストを展開したうえで同プロセスの実行を再開させる，という処理を行う．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である．

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．1 つのプロセスが計算資源を独占した形で実行を行っており，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中の 1 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．

この事情は管理者権限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支配からは逃れられない．2.3 節で述べたとおり，特権プロセスはシステムが提供するデバッガインタフェースを利用することにより，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つながる．すなわち，現状の OS カーネルによ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 に加えてメモリ中の機密情報保持部分がカーネルによって暗号化された状態となっている（図中で網掛けされた部分）．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よいため，4.3 節でも述べたとおり，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（**図 3** を参照）．

最後に，ユーザメモリ中で機密情報が置かれている位置の判断方法を説明する．たとえば Schneier らの近著[9] は，『昨今の性能であれば 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，それを是とするならば特にメモリ消費の少ないプロセスに関してはメモリ空間すべてを対象とすることも可能かもしれない．しかし，OS にとって効率も重要なことを鑑みれば，サイズに比例してコンテキス

☆ BSD 系 UNIX では一般に 100 msec.
☆☆ レジスタ等の CPU 内部資源.
☆☆☆ 復号鍵やプライバシ情報等.

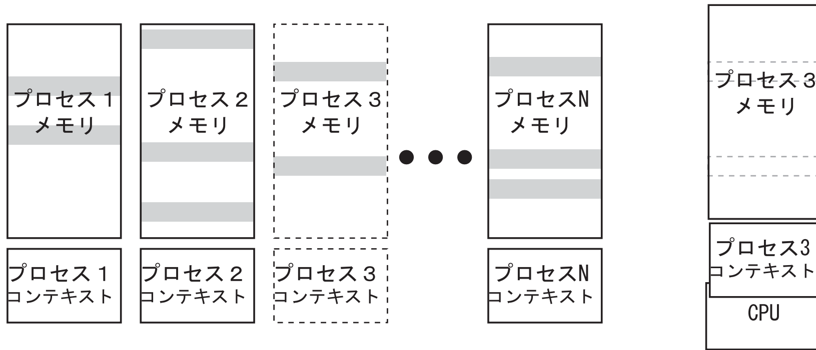

図 2　暗号化/復号機能の追加
Fig. 2　Adding encryption/decryption functionalities.

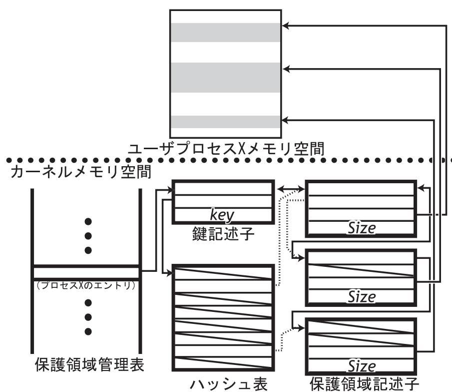

図 3　保護領域管理方式
Fig. 3　Protected region management scheme.

トスイッチのオーバヘッドを大きくする被暗号化/復号処理部分をなるべく小さくしておくにしくはない．

ある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，当該プログラムの設計者本人のみである．そのため，その情報がプログラムを通して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．ある変数がどのアドレスに置かれるのかは基本的にはプログラム実行時になって初めて確定するものであるため，そのような動的に決定される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．このシステムコールは引数としてユーザ空間上のアドレスとサイズをとる．当該システムコールの利用法としては以下のような方法が考えられる．

- 動的メモリ割当て時（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，その関数の中で領域の割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び出しを実行．
- 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，特別な記憶クラス指定子（`secure` 等）を導入．

実際のプログラミング上では両者が適材適所で用いられるため，どちらの方法も提供されるべきであろう．

カーネルは実行中のプロセスごと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．

これは重要な情報が置かれる位置に関してシステムコールにより「目印」を付けていることであり，攻撃者に対して手がかりを与えているかのように感じられるかもしれない．しかし，保護はシステムコール終了直後のコンテキストスイッチ時点から開始され，以降保護領域を平文としてアクセスできるプロセスは同領域保持者のみとなる．そのため，攻撃プロセスが位置を知ることができたとしても，その情報を有効に利用することはできないことになる．

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)　特別な H/W なしに実現可能
(2)　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
(3)　Swap/paging されるのは実行中断中のプロセスのみであるため，その時点ですでに保護済の形であり，core file の作成時にカーネルが保護領域を暗号化することも容易

ただし，保護領域をプログラム中で明示するということは，既存プログラムをそのまま変更なしで保護できるものではないことを意味する．これは本方式の欠点ではあるが，たとえば UNIX 既存の `mlock` システ

ムコールのように，保護対象をプログラム中で明示しなければならない例は他にもあるため，大きな欠点とは考えていない．

5. 試験実装による評価

4.4 節で説明した概念が実現可能であることを確認するために試験実装を行ったので，ここではその試験実装およびそれを用いた評価を説明する．テストベッドとしたのは OpenBSD 3.2 および FreeBSD 4.8 Release であり，いずれも OS カーネルに改修を加えることで実装した．2 種類の BSD 4.4 起源の OS で比較的軽度の修正により実現できたことから，本システムは十分現実的なものと評価している．

OpenBSD 版については文献 10) で報告済だが，これは本概念を実証するための最低限の機能を実装したのみであった．その後，保護領域管理方式の改良，処理の最適化，`fork` 処理への対応等を盛り込んだより本格的な実装である FreeBSD 版を作成したので，ここではその実装について述べる．

5.1 システム仕様

本システムで利用している暗号アルゴリズム等の仕様は**表 1** のとおりである．

`salloc()` 関数は C 言語における動的メモリ割付関数 `malloc()` のセキュア版であり，`malloc()` により必要量のメモリ領域を割り当てたのちに前述のシステムコールを呼び出すことで同領域の保護を開始する．また，不要となった保護領域の回収のためには同じく C 言語の `free()` 相当の `sfree()` を用意してある．`salloc`/`sfree` は C 言語の `malloc`/`free` の API を踏襲している．

鍵は個々のプロセスにたかだか 1 つあれば十分であるため，当該システムコールが各プロセスにおいて最初に呼ばれた時点で暗号学的に十分な強度を持つカーネル内疑似乱数生成ルーチン（`arc4random()`）により生成し，動的割付される鍵記述子に格納する．

また，連続しない複数の保護領域や，不要保護領域の回収を効率的に実現するため，ユーザ空間上の保護領域の管理はカーネル空間上に置く保護領域記述子によって行う．具体的には，システムコールの中で保護領域記述子をカーネル空間に割り当て，保護領域の先頭アドレスとサイズを格納し，鍵記述子からのリンクにつなぐ．このような保護領域記述子が `sfree()` の結果不要となった際には鍵記述子リンクから外す必要があるが，前述のとおり `free()` の API を踏襲したことにより，先頭アドレスのみから対象となる記述子を容易に特定するため，アドレスに基づくハッシュ表への保護領域記述子の登録と鍵記述子リンクの双方向化を行っている（図 3）．

5.2 評　　価

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 1×10^3 byte から $1{,}024 \times 10^3$ byte まで割り付ける領域を増やしたときに実行時間がどのように変化したかを示している．プログラムの中で行っている処理は単に割り付けた領域全体にランダムデータを書き込み続けているだけであり，領域のサイズによらず書き込み回数が一定となるように加減している．比較対象として，通常の malloc を用いた場合の実行結果を同表の第 3 列に，性能比を第 4 列に記した．なお，実行プラットフォームの CPU は，Pentium M 1.60 Ghz/SXGA+，実装メモリ容量は 256 MB である．

表 2 より，保護領域のサイズとしておよそ 128×10^3 byte 程度までは 15%以内の性能減で実行できることが確認できた．なお，数値が文献 10) よりも改善されている理由は FreeBSD 版を実装するにあたって行った処理方式見直しの成果である．

プライバシを要するデータ本体およびそれらを操作するためのワークエリアを含めたすべてをこの仕組で保護したとしても，全体を 100 Kbyte 程度に抑えることができれば十分現実的な性能を実現できるだ

表 1 試験実装仕様
Table 1 Spec. of trial implementation.

OS	FreeBSD 4.8-Release
保護領域指定方法	ライブラリ関数 `salloc()` による動的割付けのみ（記憶クラス指定子のサポートはなし）
暗号アルゴリズム	AES w/ 128 bit key．CBC モード
鍵	各プロセスで最初に保護領域登録システムコールを呼び出したときに疑似乱数鍵を割当て
IV	128 bit all 0 データを AES で暗号化した値
ソース改修量	約 400 行

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（10^3 byte）	“s” 実行時間（秒）	“m” 実行時間（秒）	性能比
1	20.0	20.0	1.0
4	20.1	20.0	0.99
16	20.4	20.0	0.98
64	21.6	20.1	0.93
128	23.9	20.4	0.85
256	27.3	20.7	0.76
512	34.5	20.8	0.60
1024	47.9	20.9	0.43

ろう．また，前述の文献 9) の議論を是とするならば $1{,}024 \times 10^3$ byte 利用時での性能低下さえ容認するべきかもしれない．

なお，前章までで説明した動作原理を理解された読者にはいうまでもないことではあるが，念のため，上の評価プログラムにおける割り付けられた領域へのデータ書き込みは性能に対する影響を与えるものではないという点に注意を促したい．これは本システムが，保護領域を割り付けたということだけでコンテキストスイッチにおける定常的なオーバヘッドを発生するという仕組であるため，自らの関知しないところで暗号化/復号処理を受けるにすぎない保護領域の所持プロセスにとってみれば，当該領域に対する操作は通常の領域に対するものとなんら変わりがない☆1という理由による．所持プロセスが実行の過程で保護領域の内容を変更したところで，それによって余分な暗号化/復号処理が発生したりはしないのである．

逆に，そのような仕組であるということが，この程度の単純な評価プログラムの結果で本システムの有用性を評価できると判断した理由でもあり，この結果は他の OS/アーキテクチャに対しても十分汎用的に敷衍できるものと考えている．

続いて，本システムの機能を用いて私有鍵保管領域を保護するように改造した ssh-agent☆2を実行し，gdb☆3で接続して私有鍵☆4を読み出すという擬似的な攻撃を行ってみたのが**図 4** 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えるのだが，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ていない．そのことは同鍵のコマンドラインからのダンプ操作を示した**図 5** より明らかであろう．

なお，この保護領域の中身は知ることができないという事情は，デバッガを実行するのが正規ユーザであっても変わらない．ただし，それ以外のデバッグ操作は可能であるため，保護領域の内容がプログラムの実行に直接影響しない限りにおいては本システムが有効であってもプログラムのデバッグは十分可能である．その意味で，3.1 節で記したようなデバッガインタフェースの OS レベルでの無効化と比較すると，使い勝手に関する影響は小さいと考えられる．

☆1 もちろん，カーネルによる暗号化/復号処理はキャッシュに対して影響を与えることになるはずだが，事実上無視できるだろう．

☆2 私有鍵を記憶させておくことにより，SSH プロトコルのクライアント認証をユーザに代わって行うプロセス．

☆3 GNU プロジェクト製のデバッガプログラム．OS の提供するデバッガインタフェースを利用して操作を行う．

☆4 RSA アルゴリズムにおける private exponent.

```
claudius.localdomain:/home/jane/CMSdemo
Attaching to program `/home/jane/CMSdemo/ssh-agent.CMS', process 18086
Reading symbols from /usr/libexec/ld.so...done.
Reading symbols from /usr/lib/libutil.so.7.1...done.
Reading symbols from /usr/lib/libz.so.1.4...done.
Reading symbols from /usr/lib/libcrypto.so.8.0...done.
Reading symbols from /usr/lib/libc.so.28.5...done.
0x4017408b in select ()
(gdb) showrsapriv
PrivateExponent:
    00:b6:3f:b1:8c:09:cc:c4:5b:98:bb:16:e6:97:90:
    c9:6a:8b:cc:7d:7b:e5:d3:f1:db:bf:bd:46:90:ea:
    58:18:9d:b9:ea:19:1e:a7:d0:54:bc:00:e8:97:89:
    7c:37:30:ea:85:e6:5e:f5:0c:a1:b3:b1:2f:78:e5:
    98:62:a7:fc:d2:32:c4:f2:ea:e3:f4:b4:b4:da:90:
    62:8b:1e:90:84:d5:b5:47:30:9b:57:79:32:c2:f6:
    fb:88:ca:31:ba:f9:d3:0e:59:7f:e0:6a:32:68:be:
    d2:39:89:e6:a3:d9:3e:d7:4d:b1:ba:02:b0:d0:8e:
    b7:af:5a:10:19:e4:7f:b5:d5:
(gdb)
```

図 4 改良 ssh-agent メモリ中の RSA 私有鍵の内容
Fig. 4 RSA private key in the improved ssh-agent.

```
claudius.localdomain:/home/jane/CMSdemo
claudius[22:48:49]:[228]$ openssl rsa -noout -text < id_rsa | sed -n '/^private/,/^
prime/p'
Enter pass phrase:
privateExponent:
    00:85:09:bb:ab:cf:e6:0b:4b:db:d0:5a:3c:72:ea:
    47:57:80:1f:a6:e8:2f:74:9b:4c:ee:23:25:05:c7:
    de:fb:e8:7b:23:04:2a:0a:65:17:ed:f3:4f:6e:37:
    c0:fd:0c:0f:6a:ae:ff:8e:89:27:8c:6a:74:fc:6a:
    d0:09:c9:0c:b5:02:e0:0a:9b:4f:3b:b2:63:b2:cb:
    fe:c0:15:6c:6e:a8:0a:e0:ff:3f:df:29:0c:69:65:
    67:2f:22:cd:ee:c4:eb:62:ee:b1:98:a7:43:af:65:
    98:2f:02:34:1e:c0:0c:f8:cf:0d:37:6a:77:33:4d:
    36:68:e6:91:6f:66:f3:75:cb
prime1:
claudius[22:49:02]:[229]$
```

図 5 ファイル中の RSA 私有鍵の内容
Fig. 5 RSA private key stored in file.

6. 安全性に関する議論

ここでは暗号化メモリシステムの安全性に関して議論を行う．

まず，このシステムは利用する暗号アルゴリズムの安全性に依存しているのは明らかだろう．安全なシステム構築のためには，十分な計算量的安全性を持つアルゴリズムを用いる必要があるが，本試験実装で選択した 128 bit 鍵 AES はそのようなアルゴリズムの 1 つである．

選択したアルゴリズムが適切であれば，当初の目的であったデバッガインタフェースによる機密情報の盗み出し攻撃に関して耐性を備えたシステムは，ここまで説明した仕組により実現できると考える．

また，これまで明示していなかったが，データ完全性保証に関しては，保護領域中にデータの一方向性ハッシュ値を並べて置いておくだけで，整合性を保った改竄は事実上不可能となるため，暗号化メモリシステムの機能さえあればユーザプログラムレベルで容易に実現可能である．また，その場合，余分に必要となるメモリ容量は 20〜64 byte 程度なので，性能に対する影響もほぼ無視できる．

なお，生存期間が長いプロセスでは多数のコンテキストスイッチが発生するために，攻撃者が大量の暗号化済みデータを入手することで暗号鍵解読の危険性が高まるとの懸念を感じられるかもしれない．しかし，

- 現代的なブロック暗号アルゴリズムは，たとえ攻

撃者に対して適用的選択平文攻撃を許容したとしても容易に破られない☆ことが要請されるため，大量の暗号文入手は鍵の脆弱化にはつながらない，
- 本システムの主たる保護対象は公開鍵暗号における私有鍵のように頻繁に更新されない種類のデータであり，そのようなデータではコンテキストスイッチが繰り返されたとしても攻撃者に新しい暗号文を提供することはない，

という 2 つの理由から，長期生存プロセスの存在は本システムの安全性には影響を及ぼさないだろう．

それ以外にシステムの安全性に影響を与えうる要因としては以下のようなものがある．紙数の関係によりここでは概要のみ紹介するが，いずれも容易に対処できると判断している．

標的プロセスのコンテキスト上での実行

ユーザプロセスデバッガインタフェースの異なる利用法．デバッガから関数等を呼び出すことにより標的プロセスコンテキスト上で処理を実行させるという攻撃．デバッガの管理下で実行されている場合には保護領域を復号しないことで防御可能.

カーネルデバッガ

カーネルデバッガによるカーネル空間メモリへのアクセス．カーネルデバッグ機構の不能化もしくはシステムコンソール等の物理的な保護により防御可能☆☆.

メモリデバイスインタフェース

BSD における `/dev/kmem` 等のカーネル空間アクセスインタフェースについて．ユーザプロセスから直接カーネル空間にアクセスできるということはそもそも危険きわまりない機能であり，今後は制限される方向にある[11]．また，CPU によってはカーネルコード以外からの保護鍵領域へのアクセスを H/W 的に制限することでも対処可能.

また，本方式では OS カーネルは信頼できるものと仮定している．当然，OS カーネルにもバグは存在し，攻撃者がそこを突いてカーネルを自由に操作するというシナリオもありうるわけだが，バグに対する修正の公開と適用がタイムリーに行われるならば，このシナリオが大きな危険につながる可能性は十分低いと考えられる．同様に，Linux 等におけるカーネルモジュールの動的ロード機構は，カーネル空間に対する操作を可能とするため，同様の危険をもたらしうるが，これは本方式にとどまらない汎用的な危険であり，ロードされるモジュールに対するデジタル署名の検証等の技術で対処すべき問題と考える．

☆ 鍵長の全数探索より効率的な攻撃方法が存在しない.

☆☆ カーネルデバッガが必要となるケースはユーザレベルデバッガと比較してはるかに少ないため，通常は最初から無効化されていることが期待される．

7. 結論および今後の課題

以上，同一計算機内の他プロセスによるデバッガインタフェースを用いた機密情報の盗み出し攻撃に関して耐性を備えたシステムの概念を説明し，試験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，メモリ上に置かれたプライバシ関連情報や暗号鍵等の機密データをこの方式で保護することは十分現実的なオプションとなりうると結論付ける．特に通常の PC-AT 互換機で特別な H/W なしで実現できていることから，現状の多くの計算機環境にただちに適用可能と考えられるのは大きな利点といえるだろう.

また，今後の課題としては以下があげられる.

自動変数等への拡張

現状は動的割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

複数の CPU を用いて複数のプロセスの真なる同時並列実行可能な SMP（Symmetric Multi-Processor）システムで，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への対応，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が必要と考えられる.

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時間的に変化しうるため，全実行時間を通じて保護領域が漏れなく保護される必要がある.

複数プロセス共有領域の保護

共有メモリ機能等で複数のプロセスにより共有されるメモリ領域の保護．特定の複数プロセスからは普通にアクセス可能だが，それ以外のプロセスによるアクセスは拒絶するような機構が必要と考えられる.

参 考 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 (Aug. 2000).
7) Blaze, M.: A Cryptographic File System for UNIX, *First ACM Conference on Computer and Communications Security*, Fairfax, VA (Nov. 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，特開 2001-318787.
9) Ferguson, N. and Schneier, B.: *Practical Cryptography*, John Wiley & Sons (Apr. 2003).
10) 稲村　雄：Cryptographic Memory System —Get High with a little help from my kernel, 情報処理学会研究報告 2003-CSEC-22 (July 2003).
11) McKusick, M.K., et al.: *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley (1996).

(平成 15 年 11 月 27 日受付)
(平成 16 年 6 月 8 日採録)

稲村　　雄（正会員）

昭和 35 年生．昭和 61 年東京大学工学部物理工学科卒業．同年日本電気技術情報システム開発（株）入社．同年より平成 5 年まで（財）新世代コンピュータ技術開発機構にて並列推論マシンの研究開発に従事．平成 10 年日本ベリサイン（株）入社．平成 12 年インターナショナル・ネットワーク・セキュリティ（株）入社．平成 14 年より（株）NTT ドコモマルチメディア研究所主任研究員．インターネット・セキュリティ，暗号技術等に関する研究に従事．著書・訳書に『認証技術 パスワードから公開鍵まで』（R. Smith 著，監訳），『暗号のすべて』（辻井重男/岡本栄司共編，6 章を執筆）等がある．

本郷　節之（正会員）

昭和 59 年岩手大学大学院工学研究科修士課程修了．同年日本電信電話公社入社．昭和 62 年 ATR 視聴覚機構研究所へ出向．平成 3 年 NTT ヒューマンインタフェース研究所へ復帰．この間，視覚情報処理モデルの研究に従事．著書『脳・神経システムの数理モデル』（共著）等．工学博士．平成 11 年 NTT ドコモマルチメディア研究所へ転籍．平成 13 年研究室長．情報流通研究，特に，モバイルセキュリティ方式の研究に従事．電子情報通信学会会員．