Vol.2010-CSEC-50 No.35
2010/7/2

仮想マシンモニタを用いたOSコードの秘匿化

佐久間　充[†1]　大山恵弘[†1]

オペレーティングシステム（OS）のコードには多くの技術が含まれている．それらの技術を秘密にしておくことは簡単ではない．技術の本質的部分は，バイナリコードのリバースエンジニアリングによって最終的には解析される可能性があり，ソースコードを隠すことは不十分である．この問題に対する効果的な解決法は，リバースエンジニアリングを行う人からコードそのものを隠すことである．そこで本論文では，OSのコードをOSのユーザ（ルートユーザを含む）から隠しながら実行するシステム，HyperCensorを提案する．HyperCensorは仮想マシンモニタを用いて実装されている．秘密のコード部分は元のファイルから除かれ，仮想マシンモニタ内に保存される．OSのユーザはその部分をアクセスできず，仮想マシンモニタが必要に応じて実行する．我々はHyperCensorを実装し，Linuxカーネルとデバイスドライバのコードをユーザから隠せたことを確認した．

Hiding of OS Code Using a Virtual Machine Monitor

Mitsuru Sakuma[†1] and Yoshihiro Oyama[†1]

The code of operating systems includes a number of valuable technologies. Keeping the technologies secret is not simple. Hiding the source code is insufficient because the essence of the technologies can be analyzed eventually through reverse engineering of binary code. An effective solution to this problem is to hide the code itself from reverse engineers. In this paper, we propose HyperCensor, a system for hiding the code of an operating system from its users (including a root). HyperCensor is implemented using a virtual machine monitor. Secret code parts are removed from the original file and stored in the virtual machine monitor. They are kept inaccessible from the operating system users and executed on demand by the virtual machine monitor. We implemented the system and confirmed that the system could successfully hide the code of the Linux kernel and a device driver.

1. はじめに

リバースエンジニアリング[1),2)]によるプログラムの解析は，良い目的で使われることもあるが悪用される事もある．例えば，プログラムの解析によって脆弱性を発見して攻撃を行うために用いられたり，商用のクローズドソースのバイナリプログラムを解析して内部で用いられている技術を盗み取るといった知的財産の侵害にも用いられる．このように，悪用でのリバースエンジニアリングは，プログラムを商用目的で扱う者やプログラムのユーザに多大な被害を生じさせる可能性がある．

特に，一般的なプログラムと異なりシステムソフトウェアであるオペレーティングシステム（OS）では，悪用の際の被害が顕著に現れると考えられる．その理由はいくつか存在し，例えば，OSには重要な技術が含まれており，盗用の被害が考えられる．また，悪意のものによる脆弱性の発見が，OSの破壊や乗っ取りなどに繋がってしまう．さらに，ライセンス認証に関わるコードを解析することによって正規にライセンスを取得せずにOSが使用されてしまう可能性もある．したがって，OSの解析を困難あるいは不可能にするような手法は，きわめて重要である．

OSコードの解析を困難にするために利用可能な手法として，難読化[3)–6)]がある．難読化とは，プログラムのコードを複雑にすることにより，そのコードの解析をしにくくさせる技術であり，これまでにソースコードの難読化やバイナリコードの難読化を行う手法が提案されている．難読化は一般的にユーザアプリケーションに対して使用するが，OSコードにも適用することはでき，逆アセンブルなどの静的な解析に対しての防衛策としてある程度有効である．しかし，難読化は解析の手間を増加させるのが主眼であって，解析に投入される手間と技術力次第でコードの解析は可能である[7)]．そのため難読化は，十分な防衛策とはならない場合がある．そこで，難読化とは異なるOSコードの解析防御手法が求められる．

本研究では，命令の部分的な秘匿化によりOSコードの解析を困難にするシステムであるHyperCensorの設計と実装を目的とする．ここでOSコードとは，カーネルコードと動的挿入されるデバイスドライバであるローダブル・カーネル・モジュール（LKM）を合わせたものを指す．難読化がコードを複雑にするのに対し，本研究の秘匿化は，コードを2つの部分に分離し，ユーザに全てのコード内容を与えないようにする手法である．

†1 電気通信大学
The University of Electro-Communications

© 2010 Information Processing Society of Japan

HyperCensor は変換機構と拡張 VMM から成る．変換機構は，カーネルコードと LKM コードを秘匿化するプログラムである．拡張 VMM は，秘匿化前の命令の情報を受け取り，秘匿化された OS コードを実行する仮想マシンモニタ（Virtual Machine Monitor : VMM）である．本研究では，既存の VMM を使わず，Linux を起動できる最小限の機能を持つ VMM を新たに実装した．その VMM 上で Linux を動作させ，オーバーヘッドを測定した．Linux はオープンソースであり，秘匿化する意味はほとんど無いが，まずは扱いやすい OS を用いてシステムの実装及び実験を行うのがよいという見地から今回は Linux を利用している．将来的にはクローズドソースの OS にも適用範囲を広げていきたいと考えている．

本論文では，2 章で関連研究を紹介する．3 章ではシステムの設計と実装について述べる．4 章では実験の結果を，5 章では本研究のまとめを述べる．

2. 関 連 研 究

Privtrans[8] は，1 つのアプリケーションプログラムを，特権的な処理を行うプログラムと，そうではないプログラムの 2 つに自動分割するツールである．2 つのプログラムは特権レベルの異なる別々のプロセスで実行される．2 つのプロセスに処理を分割することにより，攻撃者が特権レベルの高いプロセスの制御を奪うことを難しくする．本研究では，Privtrans とは異なり，OS のコードを分割する．また，分割の目的が攻撃の防止ではなく，コードの秘匿である．

Secure program partitioning[9] は，アプリケーションプログラムを複数に分割し，各プログラムを互いに信頼できないコンピュータ上で実行する技術である．ユーザは情報流に関する制約を注釈としてプログラムのソースコード中に記述する．Secure program partitioning を行うシステムはそれに従って自動的にプログラムを分割する．この技術により，元のプログラムの動作を維持しつつ，信頼できない相手から情報を秘匿することができる．本研究はプログラム分割により情報の秘匿を実現する点で secure program partitioning と共通しているが，OS のコードおよびバイナリコードを対象としているという明確な違いを持つ．

Dotfuscator は[5]，.NET 環境で用いられる中間言語 MSIL のコードに対して難読化を行うプログラムである．難読化は主に，タイプ，メソッド，フィールド名などを「a」や「b」といった短縮された文字列に置き換える Overload Induction によって行われる．また Dotfuscator は，難読化だけでなく，不要なコードの除去によるサイズの縮小やソフトウェアの不正なコピーを防ぐウォーターマークの埋め込みなどの機能を持つ．

ProGuard は[6]，Java のクラスファイルに対して難読化を行うプログラムである．クラス名，フィールド名，メソッド名などを短くて意味が無い文字列に変更することによって，クラスファイルの難読化を実現する．また ProGuard は最適化ツールとも呼ばれ，クラスファイルのサイズの縮小やコードの事前検証などの機能を持つ．

Dotfuscator と ProGuard はバイナリコードを難読化することは出来ない．また，それらはあくまで解析のコストを上げるだけであり，解析そのものを防いでいるわけではない．HyperCensor はバイナリコードを対象としており，また，解析対象のコードをユーザに与えないようにする事が出来る．

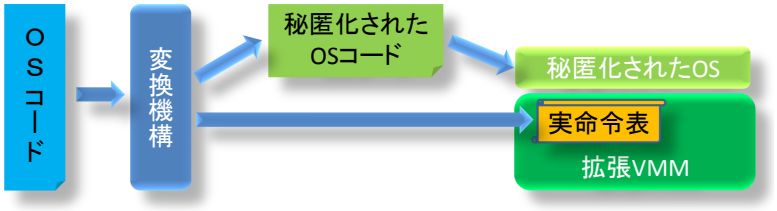

図 1 システム全体図
Fig. 1 Overview of the proposed system

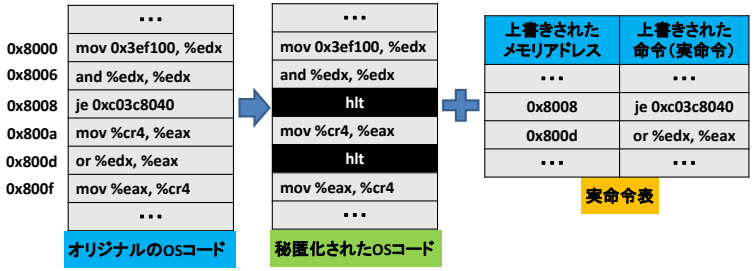

図 2 OS コードの秘匿化
Fig. 2 Hiding of OS code

3. システムの設計

HyperCensor は変換機構と拡張 VMM から成り，全体図は**図 1** となる．

3.1 利 用 法

拡張 VMM は Type II VMM である．つまり，拡張 VMM はホスト OS 上で動作し，拡

© 2010 Information Processing Society of Japan

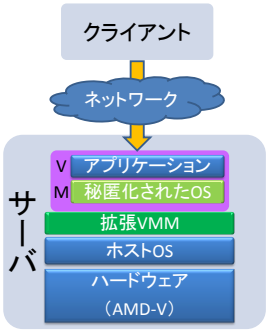

図 3 本システムの利用例
Fig. 3 Usage example

張 VMM 上でゲスト OS が動作する．本システムは，ホスト OS のユーザが，異なるユーザに対して，ゲスト OS のコードを秘匿化した状態でゲスト OS を使わせる目的で利用される．ゲスト OS のユーザは，たとえゲスト OS の管理者権限やカーネルの権限を持っていたとしても，ゲスト OS のコードの秘匿化された部分を読むことは出来ない．秘匿化された部分に元々あったコードは VMM によって適切に実行される．

本システムの利用例の 1 つを**図 3** に示す．この例では，サーバ上で拡張 VMM を動作させ，その上で秘匿化された OS を動作させている．OS コードの変換はサーバの管理者が行う．ゲスト OS のユーザの中には，知的所有権保護の観点から，サーバの管理者が OS のユーザに公開したくない部分が含まれているものとする．その管理者とは異なるユーザがクライアントからリモートログインしてその OS を利用する．クライアントのユーザにはゲスト OS の管理者権限を与える．このような利用形態は，レンタルサーバの運用などにおいてしばしば見られるものである．

3.2 変換機構

変換機構は，秘匿化していないオリジナルの OS コードから「秘匿化された OS コード」と「実命令表」を作成する（**図 2**）．オリジナルの OS コードは，Linux ではカーネルのイメージファイル（vmlinux）である．

変換機構は，OS コードに存在する命令のうち，秘匿する命令を hlt 命令（秘匿命令）で上書きすることによって，秘匿化された OS コードを作成する．hlt 命令を用いて上書きする理由は 2 点存在する．1 つは，hlt 命令を VMM がインターセプトする処理の実装が容易であるためである．もう 1 つは，hlt 命令は 1byte の命令であり，次の命令を上書きすることがないためである．

また変換機構は，hlt 命令によって上書きされた命令（実命令）を実命令表に加える．実命令表の内容は，実命令のバイト列と，その命令が置かれるはずだったメモリアドレスである．メモリアドレスは，どの実命令を実行するかを判断するために使用する．

なお，実命令表は秘匿化された OS のユーザに見られては困る重要なデータである．そのため，秘匿化された OS のユーザがアクセスできない場所で管理する必要がある．そこで HyperCensor では，秘匿化された OS のユーザがアクセスできない場所である拡張 VMM に実命令表を持たせる．

3.3 拡張 VMM

拡張 VMM は，一般的な VMM とは異なり，秘匿化された命令を元の命令に一時的に復元しながら実行できるように拡張した VMM である．拡張 VMM は，仮想化支援機構の AMD-V[10] を利用する Type II VMM として実装した．拡張 VMM の処理は大きく分けて「秘匿命令の特定」，「実命令の取得」，「実命令の実行」の 3 つである．**図 4** に動作の流れを示す．

1. 秘匿命令の特定

拡張 VMM は hlt 命令の実行をインターセプトする．すなわち，秘匿化された OS が hlt 命令を実行すると，拡張 VMM に制御が移動する．秘匿化された OS コードの中には図 2 のように，上書きした hlt 命令（秘匿命令）が存在する．しかし，その OS コード中には，秘匿命令とは関係のない通常の hlt 命令も存在する．したがって，インターセプトした hlt 命令が秘匿命令であるのかどうかを判断する必要がある．拡張 VMM ではその判断に，インターセプトした hlt 命令のアドレスを用いる．実命令表には，秘匿化された命令（実命令）とそのアドレスが存在している．したがって，インターセプトした hlt 命令のアドレスが実命令表にあれば，その hlt 命令は秘匿命令であると判断できる．実命令表にそのアドレスが存在しない場合，その hlt 命令は元々OS コードに存在する命令であると判断する．

2. 実命令の取得

インターセプトした hlt 命令が秘匿命令であった場合，拡張 VMM は，そのアドレスに元々存在した実命令を実命令表から取得する．

3. 実命令の実行

拡張 VMM はその実命令を実行する．ここで，命令の実行方法として 2 種類，VMM によるエミュレーションと CPU 上での直接実行が考えられる．拡張 VMM では，CPU 上で直接実命令を実行する．そのため一旦，実命令を本来あるべきメモリ上に戻す．しかし，メ

© 2010 Information Processing Society of Japan

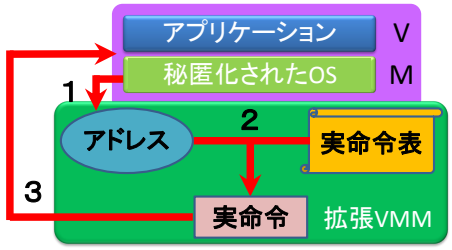

図 4 拡張 VMM の動作の流れ
Fig. 4 Execution flow of the extended VMM

モリ上に実命令を戻したままでは，OS コードの秘匿化も解除してしまうことになる．そこで，戻した実命令をシングルステップ実行し，その後，その命令を hlt 命令（秘匿命令）で再び上書きする．拡張 VMM は，rflags レジスタの Trap Flag（TF）を用いてシングルステップ実行を実現している．

3.4 ローダブル・カーネル・モジュール（LKM）への対応

HyperCensor では，デバイスドライバなどのローダブル・カーネル・モジュール（LKM）のコードの秘匿化を行うことも出来る．OS コードの場合と同様に，LKM のバイナリファイルを変換機構に与えることにより，秘匿化された LKM のバイナリファイルと，実命令表を生成する．秘匿化された OS のユーザは，その LKM をカーネルに動的に挿入することで利用する事が出来る．

LKM の秘匿化にはカーネルの秘匿化と比べて難しい点が 2 つある．1 つは，LKM がロードされるアドレスが動的に決まることである．カーネルコードが配置されるメモリアドレスは通常静的に決定できるのに対し，LKM がどのメモリアドレスに配置されるかは，実際にロードされるまで分からない．そのため，実命令表に存在するアドレスと実際にコードが存在するアドレスが異なってしまう．そこで，LKM のコードの先頭命令をインターセプトすることで，LKM のコードが配置されたメモリアドレスを求める．LKM のコードの先頭命令は，LKM をロードするときに呼ばれる `init_module` 関数の先頭命令である．そこで，`init_module` 関数の先頭命令を AMD-V 特有の命令である VMMCALL 命令で上書きすることでインターセプトを行う．拡張 VMM は，VMMCALL 命令をインターセプトしたら，その命令のアドレスが LKM のコードの先頭命令であるという情報を記憶して実行を続ける．以降で LKM 内の秘匿命令が実行されたときには，その情報を用いてアドレスの変換を行う．なお，LKM の全てのコード領域は連続している必要がある．つまり，プログラムコード中で `__init`，`__exit` などのコードセクションを分割するフラグを使用することは出来ない．

もう一つは，LKM の動的挿入の際に，アドレス解決に伴い，即値のアドレスを含む命令（jmp 命令など）のオペランドの書き換えが発生することである．そのため，実命令表に存在する静的に取得した実命令と実際に実行すべき命令が異なってしまう．そこで，`init_module` 関数の先頭でインターセプトした時に，メモリ上の秘匿命令に続くバイト列を調べ，動的コード書き換えで定まったオペランドを求める．そしてそのデータを元に，実命令表の実命令に変更を加える．これにより，実命令表に存在するデータは，OS コードの秘匿化で作成した実命令表と同等になる．したがって，後は OS コードの秘匿化と同様の処理を行うことによって，本システムは LKM のコードの秘匿化に対応することが出来る．

4. 評　　価

本章では，HyperCensor の実験による評価について述べる．実験環境は，以下の通りである．

- ホスト
  OS: Linux kernel-2.6.28.17
  CPU: AMD Athlon 64, 1.8GHz
  メモリ: 2GB
- ゲスト
  OS: Linux kernel-2.6.23.17
  メモリ: 64MB

拡張 VMM は，ストレージデバイスの仮想化を行っていない．そのため，ゲスト OS の Linux の起動や動作には，通常では一時的にのみ利用される initramfs ファイルシステムを使用した．また，シングルユーザモードで OS を起動させた．

実験は，OS に存在する様々な命令を秘匿化し，その秘匿化された OS の起動および動作を確認した．なお，ゲスト OS として使用した OS に対し，秘匿化以外の変更は加えなかった．

4.1 秘匿化によるオーバーヘッド

OS の起動時間を測定した結果を**表 1** に示す．起動時間とは，シェルプロンプトが表示されるまでの時間 [s] である．秘匿率とは，OS コード内に存在する全命令のうち，どの程度の数の命令を秘匿したのかの割合 [%] である．秘匿命令実行回数とは，プログラム中で実行

© 2010 Information Processing Society of Japan

表 1 OS の起動時間
Table 1 OS boot time

秘匿化した命令	秘匿率 [%]	起動時間 [s]	秘匿命令実行回数
-	0.00	3.67	0
call 命令	4.92	6.82	305,146
pop 命令	5.27	14.96	909,232
rep が付加された命令	0.20	48.89	3,761,429
mov 命令	33.20	798.97	62,154,092

された秘匿命令の回数である．

秘匿率が 4.92%の call 命令の場合はオーバーヘッドは 2 倍程度に収まったものの，秘匿率が 33.20%の mov 命令の場合のオーバーヘッドは非常に増加した．しかし，mov 命令は OS コードの至る所に存在しており，mov 命令の秘匿化により解析を防ぐ効果は非常に大きいと考えられる．また，オーバーヘッドは秘匿率よりも秘匿命令実行回数に強く関係している．そのため，秘匿率が 0.20%の rep が付加された命令の場合のように秘匿率が少なくても，OS の起動時間が 15 倍近くになることもある．つまり，実行回数が少ない命令を多く秘匿することによって，オーバーヘッドは少ないが解析に対して強い耐性を持つ OS を動作させることが可能である．

続いて，ゲスト OS 内で，ls コマンド，フィボナッチ，fork システムコールを実行させたときの実行時間を測定した．実行時間とは動作するプログラムの開始から終了までの時間 [s] である．ls では “ls -laR /” を実行した．フィボナッチではフィボナッチ数列の第 45 項目を求めた．fork システムコールでは fork システムコールを 1000 回発行するプログラムを実行した．それらの結果をそれぞれ**表 2**，**表 3**，**表 4** に示す．

ls プログラム（表 2）では，pop 命令を秘匿した場合の方が rep が付加された命令を秘匿した場合に比べ，実行時間が長くなった．これに対し，fork システムコール（表 4）では，rep が付加された命令を秘匿した場合の方が pop 命令を秘匿した場合に比べ，実行時間が長くなった．このことから，あるプログラムに対しどの命令を秘匿するかによってそのプログラムの実行時間が異なるため，秘匿する命令の選択を最適化することによって，実行時間は抑えつつ秘匿率を上昇できる可能性が高いと考えられる．

4.2 既存の VMM との比較

拡張 VMM と既存の VMM の性能を比較する実験を行った．拡張 VMM における秘匿化命令の実行処理を除いたものと，既存の VMM（VMware Player, QEMU, KVM）および実機を OS の起動時間, ls コマンド, フィボナッチ, fork システムコールの実行時間に関して比較した．その結果を**表 5** に示す．

表 2 “ls -laR /” の実行時間
Table 2 Execution time of “ls -laR /”

秘匿化した命令	秘匿率 [%]	実行時間 [s]	秘匿命令実行回数
-	0.00	9.88	0
call 命令	4.92	25.32	615,515
pop 命令	5.27	49.32	1,948,933
rep が付加された命令	0.20	18.79	479,024
mov 命令	33.20	91.27	6,050,715

表 3 フィボナッチの実行時間
Table 3 Execution time of Fib

秘匿化した命令	秘匿率 [%]	実行時間 [s]	秘匿命令実行回数
-	0.00	34.77	0
call 命令	4.92	40.73	415,388
pop 命令	5.27	69.45	1,875,572
rep が付加された命令	0.20	35.53	41,460
mov 命令	33.20	105.75	6,195,687

表 4 fork システムコールの実行時間
Table 4 Execution time of the fork system call benchmark

秘匿化した命令	秘匿率 [%]	実行時間 [s]	秘匿命令実行回数
-	0.00	119.53	0
call 命令	4.92	128.70	715,319
pop 命令	5.27	154.99	2,307,413
rep が付加された命令	0.20	237.26	9,204,824
mov 命令	33.20	288.89	12,213,704

フィボナッチに関しては，QEMU 以外の VMM や実機と比べてそれほど差はなかった．しかし，ls コマンドは約 3 倍, fork システムコールに至っては 100 倍近い差になった．これは，拡張 VMM でのエミュレートの性能が他の VMM に比べて低いためである．VMM で処理する必要があるセンシティブな命令が数値計算では少ないのに対し，シリアル I/O やメモリの仮想化には非常に多い．このことから，拡張 VMM の秘匿化命令に関する以外の部分の性能を改善することによって，4.1 節の実験で観測されたオーバーヘッドも削減できる可能性が高いと考えられる．

© 2010 Information Processing Society of Japan

表 5 既存の VMM との比較
Table 5 Comparison with existing VMMs

	起動時間 [s]	“ls -laR /”[s]	フィボナッチ [s]	fork[s]
本拡張 VMM	3.67	9.88	34.77	119.53
VMware Player	3.58	5.57	33.88	1.08
QEMU	3.04	2.47	732.18	2.17
KVM	1.29	3.08	33.96	1.45
実機	1.20	0.06	33.40	0.06

5. おわりに

本研究では，秘匿化を用いてカーネルコードおよび動的挿入されるデバイスドライバであるローダブル・カーネル・モジュール（LKM）コードの解析を防ぐシステムである HyperCensor の設計と実装を行った．HyperCensor は，コードを分割することによって OS のユーザに全てのコードデータを与えない秘匿化の技術により，コードの解析を困難にすることが出来る．

HyperCensor の利用にはある程度のオーバーヘッドを伴うが，OS コードの中の小さい部分を隠すことを目的とする場合の利用には非常に有用である．例としては，システムに新しい機能を追加したが脆弱性の検証が十分ではなく，十分な検証を行うまでコードを隠しておきたい場合や，認証システムの部分だけでも絶対に解析されないようにしたい場合などは，秘匿率が数%以内に収まり，オーバーヘッドも少なくて済むと考えられる．また，難読化と本手法による秘匿化は独立しており，これらを組み合わせることによって，より少ないオーバーヘッドで解析を防ぐことができると考えられる．

今後の課題としては，ブロック単位での秘匿化が考えられる．ブロック単位での秘匿化は，命令が順番に実行されるようなブロックに対し秘匿化を行い，ブロックの先頭命令が実行されたらそのブロックに存在する全ての命令を復元して実行する手法である．これにより，命令単位での秘匿化に比べ，命令の実行処理にかかるオーバーヘッドが軽減される．

参 考 文 献

1) Chikofsky, E. J. and Cross II, J. H.: Reverse Engineering and Design Recovery, *IEEE Software*, Vol.7, No.1 (1990).
2) Hex-Rays: *IDA Pro*, http://www.hex-rays.com/idapro/.
3) Popov, I.V., Debray, S.K. and Andrews, G.R.: Binary Obfuscation Using Signals, *Proceedings of the 16th USENIX Security Symposium* (2007).
4) Linn, C. and Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly, *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)* (2003).
5) PreEmptive Solutions: *Dotfuscator*, http://www.preemptive.com/products/dotfuscator/.
6) Lafortune, E.: *ProGuard*, http://proguard.sourceforget.net/.
7) Kruegel, C., Robertson, W., Valeur, F. and Vigna, G.: Static Disassembly of Obfuscated Binaries, *Proceedings of the 13th USENIX Security Symposium* (2004).
8) Brumley, D. and Song, D.: Privtrans: Automatically Partitioning Programs for Privilege Separation, *Proceedings of the 13th USENIX Security Symposium* (2004).
9) Zdancewic, S., Zheng, L., Nystrom, N. and Myers, A.C.: Secure Program Partitioning, *ACM Transactions on Computer Systems (TOCS)*, Vol.20, No.3 (2002).
10) Advanced Micro Devices Corporation: *AMD64 Architecture Programmer's Manual Volume 2: System Programing*, 3.14 edition (2007).

© 2010 Information Processing Society of Japan