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

2005－ARC－164（4）
2005／8／3

プログラムの振る舞い秘匿のための動的アドレス変換

清水一人[†]　高田正法[††]
入江英嗣[†††,†]　坂井修一[†]

暗号化されたプログラムを、プロセッサ内部で復号化して実行することによって、信頼できないオペレーティングシステムやハードウェア上においても安全にプログラムを実行することが可能なセキュアプロセッサが注目されている。しかし、セキュアプロセッサにおいてデータバスは暗号化されているが、アドレスバスは通常通りである。本論文では、アドレスバスを監視することによる情報の取得を防ぐための動的アドレス変換について提案を行う。シミュレータにおいて提案する手法によってプログラムごとの特徴が秘匿でき、また、手法適用による速度や使用メモリ、追加資源のオーバーヘッドは、実現可能な程度に収まることが確認できた。

Dynamic Address Translation for Hiding Behavior of Programs

Kazuto Shimizu,[†] Masanori Takada,[††] Hidetsugu Irie[†††,†]
and Shuichi Sakai [†]

Recently, the secure processor is paid to attention. The secure processor is a processor that can safely execute the program by decrypting and executing the encrypted program in the processor also on the operating system and hardware untrustworthy. The address bus is as traditional though the data bus is encrypted in the secure processor. In this thesis, I propose the dynamic address translation to prevent information being acquired by observing the address bus. The feature at each program was able to be hided secretly by the technique , and, in addition, it was confirmed that the overhead of the speed decrease, the memory usage, and the amount of an additional resource was ranges of a possible reality.

1. はじめに

今日の情報化社会においてプログラムはあらゆる場面で用いられるようになり、そこでは様々なデータを扱っている。その中には他者に知られたくないものも存在し、必要に応じたセキュリティが求められている。一方でプログラムの実行環境は多岐にわたり、信頼できる環境下で実行されるということが保証できない状態にある。

悪意を持ったユーザがプログラムから情報を盗もうとした場合、2 種類の攻撃方法が考えられる。まず 1 つめは、OS を改変することによって通常ではユーザが取得しえない情報を得る、という方法である。これは Linux などでのオープンソースな Operating System(OS) では特に容易であり、改変によって OS は信頼できるものではなくなる。2 つめは、ハードウェアに直接手を加えるという方法である。ハードウェアに手を加えることでソフトウェア的な方法では取得できない情報までもが取得可能になり、メモリなど、プロセッサ以外のハードウェアは信頼できるものではなくなってしまう。

このような OS やハードウェアが信頼できない状況下ではプログラムは解析されやすく、たとえば著作権保護の機能を持ったプログラムやグリッドコンピューティングなどの、プログラムを実行する環境を持つ人間にデータやそのプログラムの挙動を知られたくない場合には、ハードウェアレベルでの対策が必要であると考えられる。

この対策として、セキュアプロセッサというものが研究されている。まず、プロセッサの外側で扱うデータには暗号化を施し、内側はアクセス保護をすることで直接データを読み取ることを不可能にする。さらに、非対称鍵暗号を用いてプロセッサの認証を行うことで実行環境を暗号機能のついた特定のチップに限ることができ、これらを用いて、内容を知られたくないプログラムを暗号化した状態で配布することを可能にする。

† 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 日立製作所 システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.
††† 科学技術振興機構
Japan Science and Technology Agency

しかし、データが暗号化されていてもアドレスバスは通常のアクセスと同じように扱われているため、これらの従来手法だけではアドレスバスをハードウェア的に監視することによってメモリ領域のアクセス頻度などからプログラムの挙動を推測、解析されてしまうという危険性が残されている。

本研究は、プロセッサの内部でプログラムの使用するメモリアドレスを管理し、メモリへの書き込みリクエストのたびに異なる場所へ書き込みを行うことでプログラムの振る舞いが解析されるのを防ぐことを目的とする。

以下、本論文は次のように構成される。2 節で、本研究で前提とするセキュアプロセッサについて述べる。3 節で、アドレスバスが通常である場合に、アドレスバスにどのような特徴が現れるかを示す。4 節では、前節で示したアクセスパターンの特徴を秘匿するための提案手法について説明する。5 節で、提案手法を用いたときの評価を行う。6 節では関連研究について述べ、7 節でまとめを行う。

2. セキュアプロセッサ

ハードウェアによる単純なプログラム保護の対策として、暗号鍵をプロセッサ内部に埋め込む方式が挙げられる。これは、暗号化した状態のプログラムを配布し、プロセッサ内部に埋め込まれた暗号鍵でプログラムを復号化して実行する方式であるが、暗号鍵が埋め込まれていることから不特定のプログラム提供者が存在するマルチベンダ環境ではプログラム保護対策として利用することは難しい。そこで、使用する暗号鍵を対称鍵と非対称鍵の組み合わせにすることによってマルチベンダ環境でも利用可能なプログラム保護機能を持ったプロセッサが提案されている。

動作の様子を図 1 に示す。灰色の部分および黒い太矢印が暗号化されたデータ、通信路である。この方式では、プログラム提供者は共通鍵暗号方式でプログラムを暗号化する。プロセッサは非対称暗号方式を使用しており、内部に秘密鍵を持ち、公開鍵はプログラム提供者たちに公開されている。この公開鍵を用いてプログラムの暗号化時に用いた共通暗号鍵を暗号化し、プログラムとともに配布する。これがプロセッサに渡されると暗号化された共通暗号鍵はプロセッサ内部の秘密鍵によって復号化され、その鍵でさらにプログラムを復号化する。このようにして OS やプロセッサ以外のハードウェアには平文のプログラムを知られることなくプログラムの暗号化配布が可能となる。

しかし、セキュアプロセッサが暗号化するのはデータバスだけであり、アドレスバスは通常のプロセッサと変わりがない。

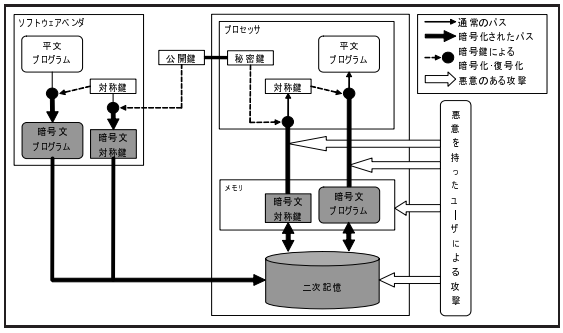

図 1 セキュアプロセッサの動作

3. プログラムの特徴

前節において、プログラムやデータを暗合するためのセキュアプロセッサについて述べたが、アドレスバスは通常のプロセッサと同様であり、アドレスバスを監視することにより、プログラムの特徴を推測できてしまうおそれがある。この特徴を見出すために、Out-of-Order 実行を行うスーパースカラプロセッサのシミュレータである SimpleScalar[1] を用いてメモリアクセスの特徴を計測した。評価パラメータを表 1 に示す。

表 1 評価パラメータ

Simulator	SimpleScalar(sim-outorder)
L1 ICache	16KB, LRU, 1way, 32 Bytes line 1cycle access latency
L1 DCache	16KB, LRU, 4way, 32 Bytes line 1cycle access latency
L2 Cache	256KB, LRU, 4way, 128 Bytes line 6cycles access latency
Memory	48cycles access latency
命令セット	Alpha 21264 ISA
ベンチマーク	SPEC2000int (入力セット:test)
実行命令数	全命令

図 2 は、上記のシミュレータにおいて SPEC2000int の 11 種類のベンチマーク (bzip2, crafty, gap, gcc, gzip, mcf, parser, twolf, vortex, vpr-place, vpr-route) をベンチマーク終了まで全命令実行したときの、同じメモリアドレスに対するアクセス頻度を表すグラフである。横軸は同じメモリアドレスに対する L2 Fill Request があった回数を全命令実行にかかったサイクル数を用いて正規化した値、縦軸はアクセスがあった全ラインに対する百分率の累積を表す。つまり、グラフで真横に線が描かれる部分は、該当する回数のアクセスがあったラインが存在しない部分、縦に線が描かれる部分はその回数のアクセスがあったラインが多い部分、といえる。また、100%に達するのが遅いものは集中的にアクセスされるラインが多いものとなる。

このグラフを見ると、

- mcf のようにほとんどのラインに対して少ない回数のアクセスしかしないもの
- gzip のようにある回数のアクセスをするラインが大多数を占めるもの
- vpr-route のようになかなか 100%に達しない、つまり多くのラインに対して集中的なアクセスがあるもの

のように大きな特徴が現れていることがわかり、このアクセス頻度の割合からプログラムの特徴が推測できる可能性があることがわかる。特に、グラフが 100%近くで右に伸びていればいるほど、少ない範囲に対して非常に偏ってアクセスしていることがわかる。これは、どのデータがよく読まれているかを知るために有効な情報となる。

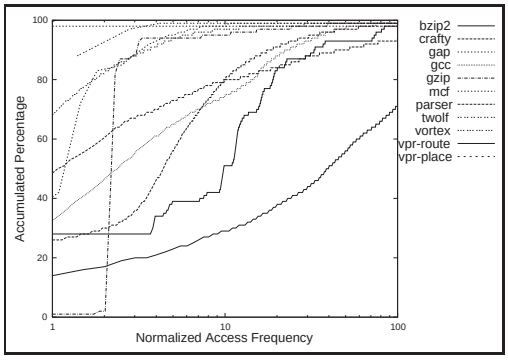

図 2 同一アドレスに対するアクセス頻度

4. 提 案 手 法

本研究ではプロセッサ内部にアドレス管理機能を持ち、キャッシュからデータがメモリに書き込まれる時に動的にアドレスを変換させて読み込まれた場所とは別の場所へ書き込み、同一アドレスへの局所的アクセスを隠蔽することで空間的局所性を秘匿するという方法を提案する。

4.1 提案手法概要

まず、提案手法の動作の概要を示す。通常のプロセッサとの違いは、アドレス変換テーブル、空き領域アドレス群、の 2 つを持つ点である。

キャッシュミスが発生してメモリからキャッシュへの読み込みが発生した場合にはアドレス変換テーブルにしたがって論理アドレスを実際のアドレスに変換し、対象のメモリアドレスにアクセスする。

逆にキャッシュからデータが追い出されメモリに書き込まれるときには、今回アクセスする論理アドレスに対応する実際のアドレスをアドレス変換テーブルより参照するが、このアドレスには書き込まず、プロセッサが管理している空き領域アドレス群に書き込む。これによりアドレス変換テーブルで参照した実際の対応アドレスは使われなくなるのでこのアドレスを空き領域アドレス群に入れる。

このような方法をとることで、メモリアクセスの空間的局所性を秘匿することができる。

4.2 構 造 詳 細

次に、各部分についての詳細について述べる。

4.2.1 アドレス変換テーブル

アドレス変換テーブルは、プログラムからみた通常の論理アドレスと、プロセッサが管理する実際のアドレスの対応テーブルである。

それぞれのラインに対して変換前アドレスと変換後アドレスの 2 つのアドレスが必要であり、実装する対象のプロセッサが 64bit プロセッサである場合には 1 ラインに対して最大で 128bit のデータサイズが必要となる。仮に L2 キャッシュのラインサイズを 128Bytes とすると、128Bytes に対して 128bit のデータを持つために、プログラムが使用するメモリサイズのおおよそ $\frac{1}{8}$ のメモリが必要である。これはプロセッサ内部に配置するにはあまりにも大きいものである。

そこで、このアドレス変換テーブルを階層化したアドレス変換ツリーと、その一部だけをプロセッサ内部に保持するためのアドレス変換キャッシュの 2 つを用いる。

4.2.2 アドレス変換ツリー

アドレス変換ツリーは、アドレス変換テーブルを階層化したもので、図 3 に示される構造をとる。

アドレス変換テーブルを単純にメモリ上に配置し、その中の一部分だけをプロセッサ内に読み込むという方法も考えられるが、これではアドレス変換テーブルへのメモリアクセスの局所性が見えてしまい、結果的に本来のメモリアクセスの局所性も見えることになる。これを回避するために階層構造をとり、書き込みの度に格納先を変化させることで局所性を秘匿する。

ツリーの葉ノードはアドレス変換テーブルの一部に対応し、変換後のアドレスを持つ。ツリーの葉ノードを除くすべてのノードは、それぞれ子ノードへのポインタを持つ。これらはまずメモリ上に配置され、必要に応じてアドレス変換キャッシュに読み込んで使用する。

ラインサイズを 128Bytes とすると、アドレス変換テーブルの時と同様に考えて、ツリーの葉ノードにはプログラムが確保したメモリ領域の $\frac{1}{8}$ のデータサイズが必要である。葉ノード以外のノードは 16 の子ノードへの 64bit のポインタを持つため、その上の階層では $\frac{1}{8}$ の $\frac{1}{16}$ である $\frac{1}{128}$ が必要となり、さらにひとつ上の階層では $\frac{1}{2048}$、と続く。アドレス変換ツリー全体では $\frac{2}{15}$、つまり 13%のデータサイズを使用する。ただし、これはプログラムがページ単位で確保したメ

モリ領域全体を使用する場合である。使われていない部分の変換テーブルは作成しないため、実際のメモリ使用量はこれよりも少なくて済む。

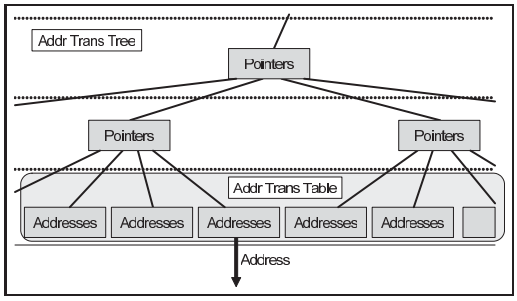

図 3 アドレス変換ツリー

4.2.3 アドレス変換キャッシュ

アドレス変換キャッシュは、アドレス変換ツリーのアクセス遅延を軽減するために用いるもので、図 4 に示されるように階層ごとに分割されている。図の左側はツリーをイメージしたもの、右側が実際のキャッシュの構造である。それぞれの階層ではエントリ数が異なり、階層の上位に近いほどエントリ数は少なく、下位の方がエントリ数は大きくなる。

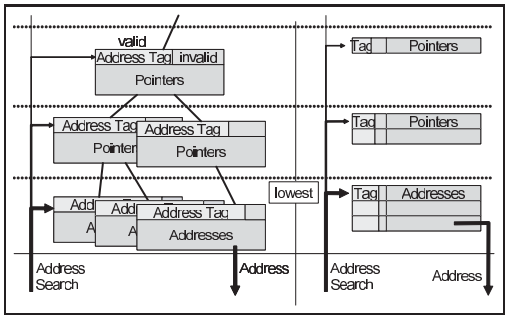

図 4 アドレス変換キャッシュ

各ノードはメモリ上に配置されたアドレス変換ツリーから読み込まれるが、このときのメモリアクセスは L2 キャッシュミスによるメモリアクセスと共通のバスを用いて行う。これは、アドレス変換ツリーのアクセス頻度などから情報が漏れるのを防ぐために、通常のアクセスに紛れ込ませるためである。よってアドレス変換キャッシュのラインサイズを L2 キャッシュのラインサイズにあわせる必要がある。

L2 キャッシュのラインサイズを 128Bytes とした場合、アドレス変換ツリーの各ノードも 128Bytes になり、ツリーの葉ノードは 16 個の変換後 64bit アドレス、それ以外のノードは 16 個の子ノードへの 64bit のポインタを持つこととなる。各ノードが 16 エントリであることからツリーの各階層は 4bit ずつとなり、64bit アドレス空間全体を管理するときには、L2 のラインサイズである 128Bytes を格納するために必要な 7bit を 64bit から差し引いた 57 を 4bit で分割した 15 階層の構造となる。

また、キャッシュの各ラインはそれぞれ対応する変換前アドレスタグを持つ。このタグは階層によって有効ビット数が異なり、階層が上位に近づくにしたがって下位のビットは無効となる。キャッシュ検索を行うときには、全階層に対して同時に変換前アドレスタグとの比較を行い、ヒットしたものの中で最も有効ビット数が大きいもの、つまり階層が一番低いものを有効なヒットとする。

4.2.4 空き領域アドレス

プログラムが用いるデータならびにアドレス変換キャッシュをプロセッサからメモリに書き出す際に、元とは異なる位置に書き込むが、その書き込み先として空き領域を確保している必要がある。そのため、この空き領域のアドレスをまとめて空き領域アドレス群としてプロセッサ内に保持する。図 5 のように、読み込みの際にはアドレス変換キャッシュから得られたアドレスに直接アクセスを行うが、書き込みの時にはアドレス変換キャッシュから得られたアドレスは空き領域アドレス群に格納し、代わりに空き領域アドレス群から一つアドレスをランダムに取り出し、このアドレスをデータ書き込み先として用いる。

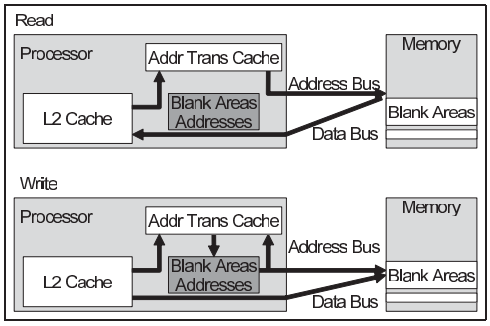

図 5 空き領域アドレス

5. 提案手法の評価

本研究では、以上の提案手法を CPU シミュレータ (SimpleScalar) に組み込んで評価を行った。評価時の基本的なパラメータを表 2 に示す。アドレス変換キャッシュは、L2 キャッシュと同じ速度で動作すると仮定し、テーブルを 1 回参照するのに 6cycles を要するものとした。

5.1 メモリアクセスの局所性

図 6 は提案手法を組み込んだプロセッサにおけるメモリアクセスの局所性を表すグラフである。これは第 3 節で示したプログラムごとの同一アドレスに対するアクセス頻度 (図 2) と同じものをアドレス変換を行っ

表 2　評価基本パラメータ

Simulator	SimpleScalar(sim-outorder) ベース	
L1 ICache	16KB, LRU, 1way, 32 Bytes line 1cycle access latency	
L1 DCache	16KB, LRU, 4way, 32 Bytes line 1cycle access latency	
L2 Cache	256KB, LRU, 4way, 128 Bytes line 6cycles access latency	
Memory	48cycles access latency	
Address Translation Cache Cache	256KB, LRU, 128 Bytes line, 15 階層 6cycles access latency	
	entry 数	最下層から順に 1968,64,4,1, 1,1,1,1,1,1,1,1,1,1
	way 数	最下層から順に 16, 4,4,1, 1,1,1,1,1,1,1,1,1,1
空き領域アドレス	256K エントリ (2MB)	
命令セット	Alpha 21264 ISA	
ベンチマーク	SPEC2000int (入力セット:test)	
実行命令数	全命令	

た場合で作成したものである。アドレス変換を行った結果、同じアドレスに集中してアクセスすることはなくなり、アクセスの空間的局所性は大きく隠蔽されたことがわかる。さらに、一部を除いてはプログラムごとの特徴も薄れ、メモリアクセスからのプログラムの挙動推測は難しくなったと言える。

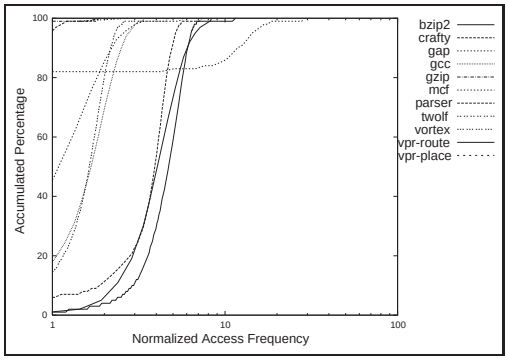

図 6　提案手法を用いた場合の同一アドレスに対するアクセス頻度

5.2 速度低下

次に、提案手法による速度低下について示す。この提案手法では、L2 キャッシュミスが発生したときにアドレス変換キャッシュへのアクセスが起こり、さらにアドレス変換キャッシュミスが発生した場合やアドレス変換キャッシュからのノードの追い出しが発生した場合には、メモリアクセスも増えることになるために速度が低下する。

図 7 が提案手法適用前と適用後の Instructions Per Cycle(IPC) の比較のグラフである。大きな速度低下は見られず、最小で 0.01%から最大で 9.58%、平均で 3.6%の低下となっている。

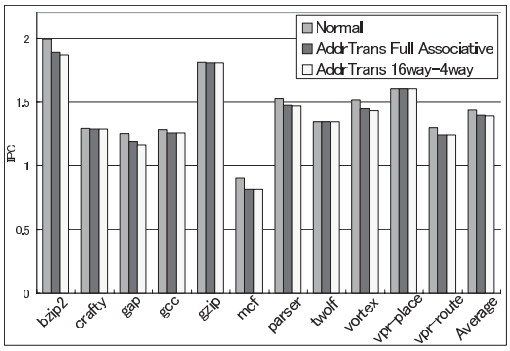

図 7　提案手法を用いる前後の IPC 比較

5.3 メモリ使用量

最後に、アドレス変換ツリーに用いたメモリサイズについて示す。図 8 は、プログラム自身が用いたメモリサイズに対するアドレス変換ツリーが用いたメモリサイズの割合を表したグラフである。第 4 節において 13%の領域が必要だと述べたが、確保した領域に対する実際のメモリ使用率が小さく、アドレス変換ツリーが用いたメモリサイズは最小で 5.2%から最大で 6.6%、平均で 6.3%となった。

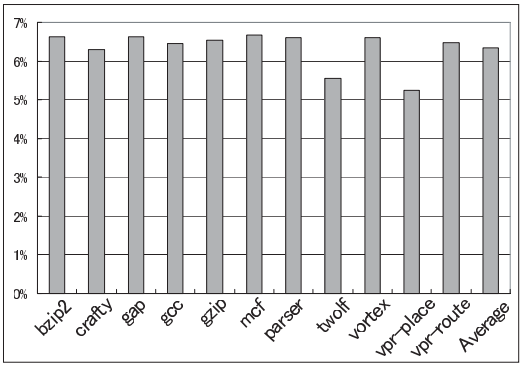

図 8　メモリ使用量の割合

6. 関連研究

David Lie らは、セキュアプロセッサの一種として XOM(eXecute-Only Memory) アーキテクチャを提案している[3]。XOM は、レジスタやキャッシュに専用のタグを追加し、これが一致しないプロセスからの読み込みはできないようにしており、これで特権モードからのレジスタやキャッシュの読み込みも防ぐことが可能となっている。メモリアドレスのアクセスパターンによるプログラム挙動の解析への危惧は指摘されているものの、対策については言及されていない。

L-MSP は橋本、春木らが提案、実装したアーキテクチャで、プロセス管理の機能もプロセッサ内部に持

たせたものである[7),8)]。L-MSP はハードウェアコンテキストスイッチ機能を備え、より高速にプロセスの区別を行う。

AEGIS は Edward Suh らの提案するアーキテクチャで、情報の秘匿に加えて改竄の検出も行うものである[4),5)]。メモリデータのハッシュのツリーを構築し、効率的な検出が可能となっている。

これらのセキュアプロセッサでは、いずれもデータバスは暗号化されているものの、アドレスバスは通常のプロセッサと同様であり、アドレスバスを監視することでプログラムに関する何らかの情報が得られてしまう。

Goldreich らは、メモリとプロセッサの間のやり取りから読み取ることのできるメモリアドレスの参照局所性を厳密に秘匿する手法を Oblivious RAM として提案している。文献[2)] ではメモリとプロセッサをそれぞれチューリングマシンとしてモデル化して、一定期間中に必ず全メモリが一度だけアクセスされることを保証している。しかし、モデルと実際に大きな隔たりがあり、また性能評価などもなされていない。

Xiaotong Zhuang らは、アドレスバスによって得られる情報からプログラムのアルゴリズムを解析できることを示し、暗号鍵生成のプログラムのアドレスバスを監視することで暗号鍵を得ることができるとしている。そしてこの問題への対策として HIDE アーキテクチャを提案している[6)]。

HIDE ではメモリアドレスのランダム入れ替えを行いアクセスの局所性を秘匿しているが、アドレス管理に用いるメモリを全てプロセッサ内部に配置すると仮定しており、プロセッサに膨大な記憶領域が必要となってしまうため、現実的ではない。

7. ま と め

本研究では、プロセッサ以外のハードウェアや OS が信頼できない環境においてプログラムやデータを暗号化しても、メモリアクセスの局所性からプログラムの挙動が解析される可能性があることを指摘した。

そして、それを防ぐためには同一アドレスへの局所的アクセスを隠蔽することが必要であり、メモリから読み込んだデータを異なる領域に書き戻し、そのアドレス変換テーブルを外部から読み取ることのできないプロセッサ内部で管理する手法を提案した。

アドレス変換テーブルを現実的に管理するために、テーブル全体をメモリ上に配置し、必要な部分のみをキャッシュとしてプロセッサ内部に持たせた。これらの読み書きを通常のデータの読み書きと同様にして行うことでメモリアクセスの特徴を見えにくくし、プログラムの挙動が解析されることを防いだ。

また、この手法を用いることで起こる実行速度の低下は数%にとどまること、プロセッサ内部に必要な記憶領域は大きくても数 MB であること、外部メモリに必要な領域はプログラムが使用する領域の数%〜13%程度であることから、この手法が現実的であることを示した。

挙動秘匿の度合いには、資源量や遅延とのトレードオフが存在する。このトレードオフを考慮した最適なパラメータを求めるためには、メモリアクセス局所性を利用した解析方法のモデル化を行う必要がある。これは他の研究でも課題となっており、解析方法のモデル化が本研究の有効性をより正確に示すことにつながると考えられる。

謝辞 本論文の研究は，一部 21 世紀 COE 「情報技術戦略コア」、及び科学技術振興機構 CREST「ディペンダブル情報基盤」による。

参 考 文 献

1) Simplescalar. http://www.simplescalar.com/.
2) Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, Vol. 43, No. 3, pp. 431–473, 1996.
3) David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2003.
4) G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing*, 2003.
5) G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of a single-chip secure processor using physical random functions. In *Proceedings of the International Symposium on Computer Architecture*, 2005.
6) Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: An infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
7) 橋本幹生, 春木洋美. 敵対的な OS からソフトウェアを保護するプロセッサアーキテクチャ. 情報処理学会論文誌 コンピューティングシステム, Vol. 45, No. 3, March 2004.
8) 春木洋美, 橋本幹生, 川端健. 耐タンパプロセッサ (L-MSP) システムの実装とプロセス管理. 暗号と情報セキュリティシンポジウム, 2004.