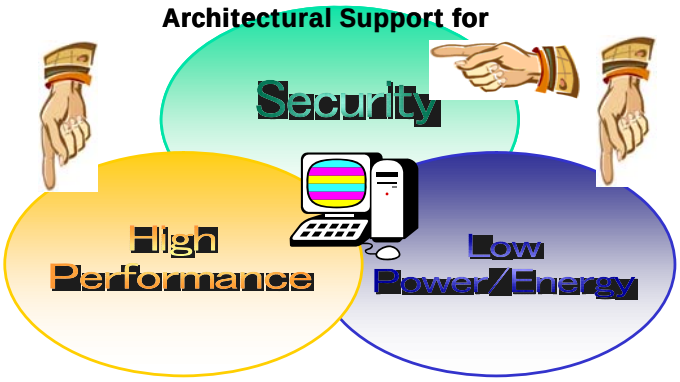
研究課題別評価

1　研究課題名：安全で低消費エネルギーなプロセッサに関する研究

2　研究者氏名：井上　弘士

3　研究の狙い：



現在、我々はインターネットを利用して世界中の様々な情報に用意にアクセスできる。実際、産業界はもちろんのこと、一般市民の生活においてもインターネットは極めて重要かつ重宝な道具として大きな役割を果たしている。しかしながら、これは、我々のコンピュータがインターネットに接続した瞬間から、目に見えない大多数の悪意ある攻撃の対象と成り得ることを意味する。悪意ある攻撃は、我々の日常生活から社会経済にまで多岐に渡り極めて大きな被害をもたらす。例えば、コンピュータ・ウィルスやワームなどの悪質プログラムは深刻な社会問題を引き起こしており、IPA(Information-Technology Promotion Agency, Japan)の「国内・国外におけるコンピュータウイルス被害状況調査(2004 年 4 月)」によれば、世界で発生したウイルス被害額は 2003 年で 135 億ドル(およそ 1.5 兆円)にもなる。

これまでに、データの暗号化やインターネットを介した不正アクセスの検出、ソフトウェアによるウイルス検索など、様々な安全性向上技術に関する研究・開発が進められてきた。そして、これらの多くがアルゴリズム・レベル、ネットワーク・レベル、OS などのソフトウェア・レベルでの議論であった。しかしながら、例えば、コンピュータ・ウイルスと言えども、それを実行するのはハードウェアである、また、重要な秘密情報を記憶し処理を施すのもプロセッサやメモリといったハードウェアである。これに対し、1970 年代初頭に開発されて以来、ハードウェアの代表であるマイクロプロセッサに関する研究開発は高性能化や低コスト化、低消費電力化に焦点が当てられており、安全性の向上を目的とする議論は極めて少ないのが実情であった。

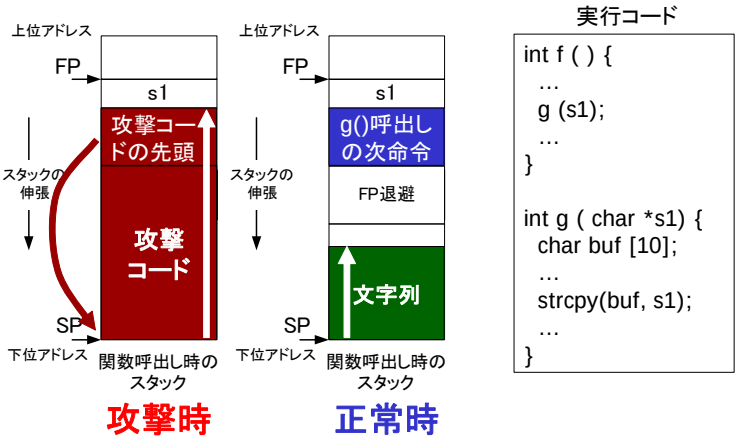
そこで本研究では、コンピュータ・システムにおいて実際に処理を行うハードウェア・レベルでの安全性確保が「最後の砦になる」と考え、それを実現するハードウェア・アーキテクチャの確立を目指している。特に、コンピュータの核となる(実際にプログラムを実行する)プロセッサとメモリに焦点を当て、安全性を向上するメカニズムの考案、ならびに、その際に生じる性能や消費電力オーバヘッドを最小限にするための技術を開発する。システムの安全性を向上するには、様々な攻撃からの防衛策を考案するのはもちろんのこと、性能や消費電力も含めたコストとのトレードオフを考慮することが重要となる。高い安全性を確保できる方式を考案したとしても、その実現において極めて高いコストを要する(例えば、プログラム実行時間や消費電力が数倍になるなど)場合には適用することが難しい。そのため、安全対策を施すこと事態が難しくなり、より危険性を高める結果となりかねない。つまり、「如何に低コストで安全性を高めることができるか？」が極めて重要となる。

4　研究成果:

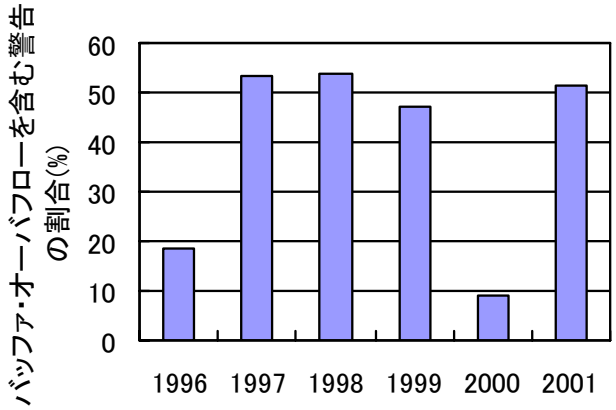
本研究では、1)ウィルス等の悪質プログラムからコンピュータを守る、2)秘密情報の漏洩を防ぐ、の 2 点に着目し、これらを実現もしくはサポートするためのハードウェア・アーキテクチャを開発した。本節ではこれら研究成果の詳細を説明する。

4．1　悪質プログラムの実行を防ぐ～バッファオーバフロー攻撃の動的検出～

近年、バッファ・オーバフローの脆弱性を活用した悪質プログラムによる被害が急増している。例えば、代表的なものとして 2001 年に猛威を振るった Code Red や、2003 年の Blaster などがある。 悪質プログラムは、攻撃対象となるコンピュータが正規アプリケーションを実行している最中にバッファ・オーバフローを引き起こさせ、強制的にプログラムの実行制御を乗っ取る。したがって、特権モードでの実行中にバッファ・オーバフローが発生した場合、悪質プログラムは特権モードで実行されることになる。その結果、ファイルの削除や改ざんが可能となり多大なる被害をもたらす。あるアプリケーションにおいてバッファ・オーバフローに関する脆弱性が既知の場合、ダウンロード等によるアプリケーション・プログラムの更新により解決できる。しかしながら、未知の脆弱性には対応できないため、依然として多くの被害が発生しているのが現状である。例えば、右図に示すように、CERT によって発せられた勧告(1996～2001 年)の内、バッファ・オーバフローに関連するものが高い割合を占めている。

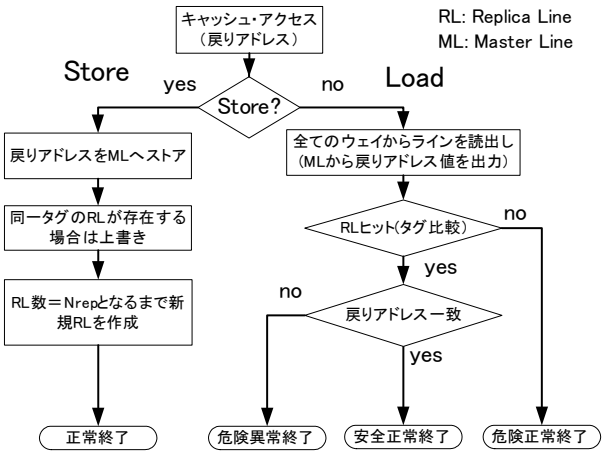


バッファオーバフロー攻撃による実行制御の乗っ取り

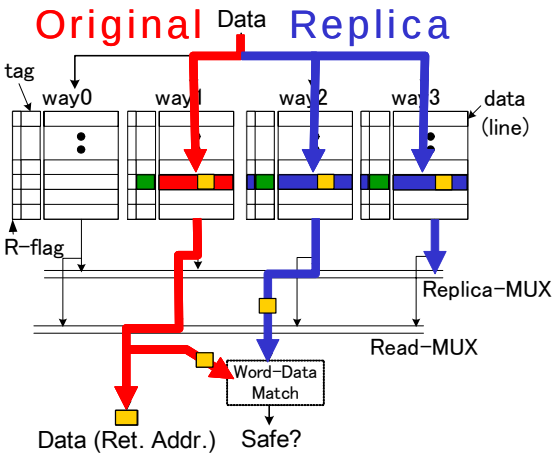


CERT バッファオーバフロー勧告

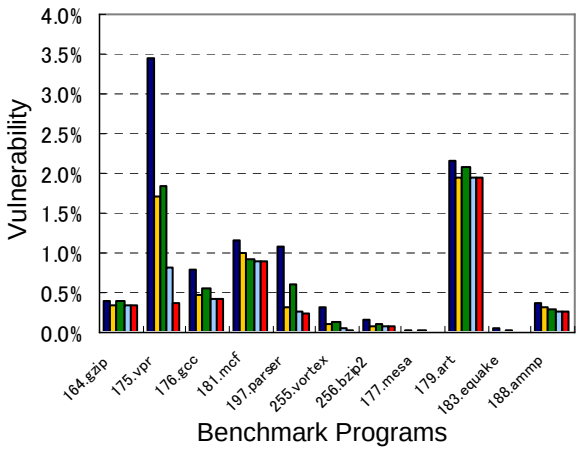
多くの悪質プログラムは、関数呼び出し後にバッファ・オーバフローを引き起こしてスタックを破壊する(スタック・スマッシングと呼ばれる)。そして、関数呼出し側への戻りアドレスを悪質プログラム・コードの先頭アドレスへと改ざんすることで、プログラムの実行制御を乗っ取る。右図を用いてバッファ・オーバフローによる攻撃の様子を簡単に説明する。ここでは、C 標準ライブラリである strcpy を用いて文字列コピーを行う関数 g が、関数 f によって呼出される場合を想定している。通常、関数 g が呼出された際、関数 f への戻りアドレスをスタックに保存する。そして、関数 g での処理終了後、この戻りアドレスを PC に復元する事で関数呼出し側へと実行制御が移る。これに対し、バッファ境界をチェックしない文字列コピーによりスタックの破壊(スタック・スマッシング)が発生した場合、スタック領域に対して悪質プログラム・コードが上書きさ



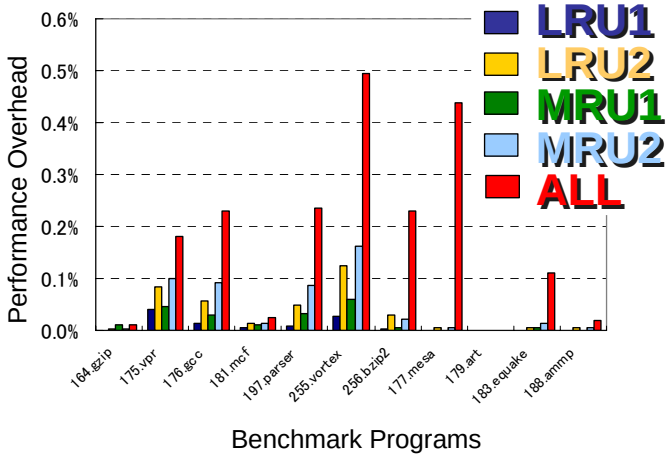
(A) 戻りアドレス・アクセス時の動作



(B) 内部構成

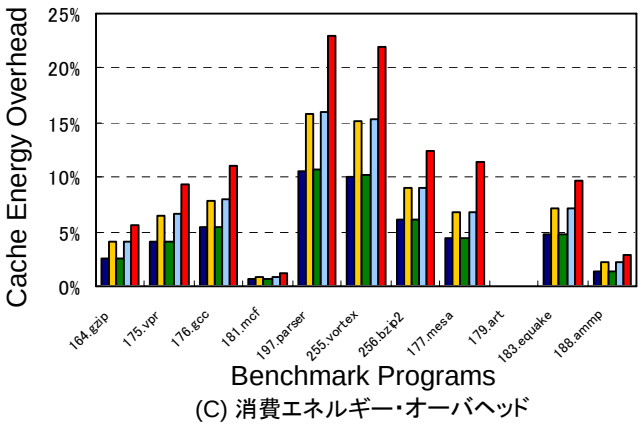


(A) 危険度（保護できない戻りアドレスの割合）



(B) 性能オーバヘッド

れる。また，関数 f への戻りアドレスが悪質コードの先頭アドレスへと改ざんされる。そして、呼出し元関数 f へ復帰する際には改ざんされた戻りアドレスが用いられ、その結果としてスタック内部の悪質プログラム・コードへと実行制御が移る。



(C) 消費エネルギー・オーバヘッド

本研究では、このようなバッファ・オーバフローによる実行制御の乗っ取りを動的に検出するアーキテクチャ・アプローチとして、セキュア・キャッシュ(***SCache***)を提案した。

通常、関数呼出し時にスタック領域へプッシュされる戻りアドレスは、一旦キャッシュにストアされる。また、呼出し元関数へ復帰する際、プロセッサはキャッシュから戻りアドレスをポップする。スタック・スマッシングによるプログラム制御の乗っ取りにおいて、その本質的な問題点は戻りアドレスが改ざんされることにある。そこで SCache では、戻りアドレスがストアされる際、読出し専用の複製ライン(**レプリカ・ライン**と呼ぶ)を同一セット内に作成する。その後、戻りアドレスをロードする時、スタック領域から読出される値と、レプリカ・ラインの値を比較する。もし、比較結果が一致であれば戻りアドレスの安全性が保障され、不一致の場合

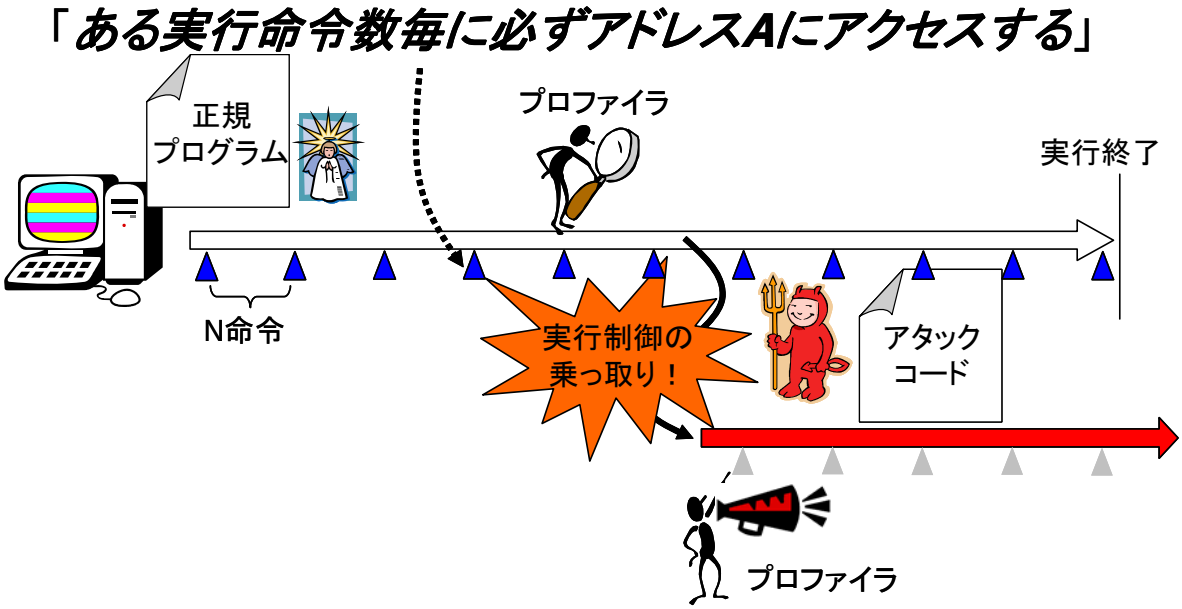
にはスタック・スマッシングの発生を検出する。つまり、キャッシュが本来有する冗長性を有効に活用することで戻りアドレスデータの改ざんを動的に検出する。

SCache のシミュレータ作成、ならびに、キャッシュ構成要素の実設計を行い定量的評価を実施した。その結果、多くのプログラムにおいて、99%以上の戻りアドレス書込みを保護できることが分かった。また、その時の性能ならびに消費エネルギー・オーバヘッドは、それぞれ、0.5%ならびに 20%以下であった。なお、評価結果を示す図において、各評価対象モデル名の最後の数字は生成される複製データの数、LRU ならびに MRU は複製データの配置アルゴリズムを表している。また、ALL に関しては、最大数の複製(実際には 3 個)を作成するモデルである。

2．2 認められたプログラムだけを実行する～動的なプログラム認証～

攻撃対象となるシステムの脆弱性が既知の場合には、前節で説明した SCache のようにハードウェアで対策を施すことができる。しかしながら、システム開発者にとって未知の脆弱性に関しては対応することができない。この問題の解決策として、実行対象となるプログラムが不正コードを含まない事を保障（安全性を保障）し、このような認められたプログラムのみを実行可能とする方法が挙げられる（いわゆる、ホワイトリスト方式）。つまり、システム内部に如何なるセキュリティ・ホールが存在するとしても、実行対象となるプログラムが安全であれば問題は発生しないという考えに基づく。この方式においては、「如何にして対象プログラムがウィルスフリーである事を検査するか？」、また、「如何にしてプロセッサが承認済みプログラムコードであることを認知するか？」が重要となる。前者に関しては、アプリケーション・プログラム発行者が保障する、または、既存のウィルス検索ソフト等によりチェックする方法が考えられる。しかしながら、後者に関しては、現在のプロセッサで実現することは難しい。プロセッサから見たプログラム実行とは単なる「命令シーケンスの処理」にしかすぎないためである。つまり、悪質プログラムの実行においても、安全であると保障された正規プログラムの実行と同様に「ただの命令シーケンスの実行」としてしか認識でいない。

現在、アプリケーション発行時に正しいプログラムである事を証明する鍵（または証明書）を作成し、プログラムを実行する際に OS（オペレーティング・システム）が照合する方法が実用化されている。つまり、OS レベルでのプログラム認証である。しかしながら、最近のウィルスは OS を書き換えるような物も少なくない。

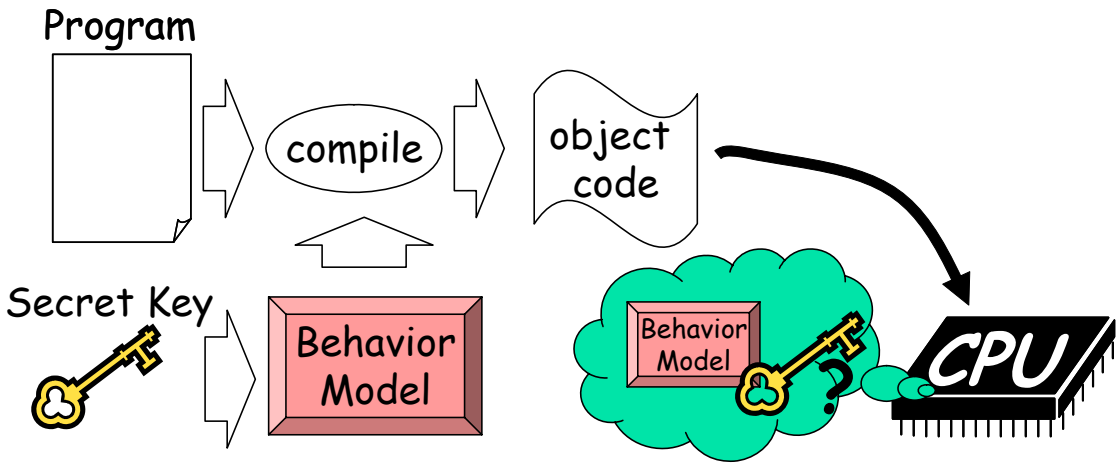


また、プログラム起動時にのみ鍵の照合が行われるため、正規のアプリケーション・プログラムを実行中に突然悪質プログラムが暴走し始めた場合にはそれを停止する手段がない。現在のプロセッサは、いったんプログラムの実行を開始すると、それが悪質プログラムであるか否かは関係なくその実行を完了しようとする。そのため、OS レベルでのプログラム認証を通過した悪質プログラムはコンピュータ・システム内で暴走し多大な被害を与える。

そこで本研究では、実行の振舞いを鍵情報とする動的プログラム認証方式を提案している。提案方式の基本動作を下図に示す。ここでは、利用者側とアプリケーション発行側で共通の秘密鍵を保有していると仮定している。アプリケーション発行側では、この秘密鍵から決定される「プログラム実行の振舞い」を実現するオブジェクト・コードを生成する。一方、利用者側では、秘密鍵から決定される「プログラム実行の振舞い」を検出するための専用プロファイラを生成し、再構成可能ハードウェア上に実装する（複数の秘密鍵を有する場合等にも対応できるよう、専用プロファイラの実装デバイスは再構成可能ハードウェアとしている）。そして、アプリケーション・プログラム実行時、このプロファイラによって常に実行の振舞いを監視する。もし、「鍵としての実行の振舞い」が検出できなかった場合には即座にプロセッサに実行停止割り込みを発行する。なお、ここではプログラム発行者側と利用者側でのデータ転送は暗号化技術等により安全性は保たれていると仮定する。

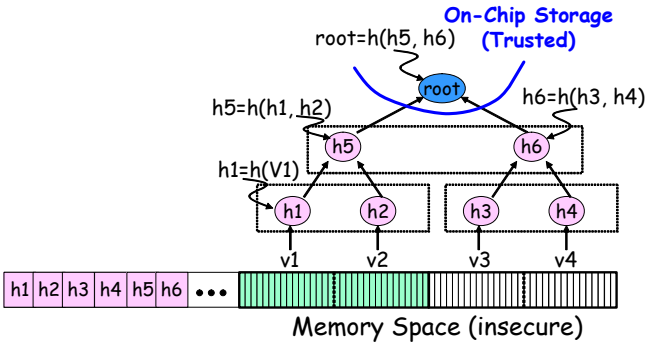
このような動的認証機能を有するプログラム実行方式を実現するには、プログラムのコンパイル時にその実行振舞いを制御する必要がある。しかしながら、一般にプログラム実行の流れは入力データに依存する。また、各基本ブロック・サイズはそれぞれ異なるため、実行振舞いを生成するよう各鍵命令を一定間隔に挿入することが困難となる。この問題を解決するため、本研究では実行振舞いを静的制御可能とするコード生成技術を開発した。プログラムの全基本ブロック・サイズを同一サイズに統一し、各基本ブロックの同一位置に鍵ロード命令を挿入する。これにより、固定命令数毎に鍵ロード命令が実行される。ただし、基本ブロック・サイズを統一するために冗長な NOP 命令を挿入する必要があり、コード・サイズの増加ならびにプログラム実行時間の増大といった欠点がある。

組込み用途で多く使用される StrongARM プロセッサを想定し、提案手法適用における性能ならびにコード・サイズのオーバヘッドを評価した。その結果、コードサイズは約 80%増加した。一方、プロセッサ性能の低下に関しては 9%程度であった。これらの結果より、コード・サイズの増加は比較的大きいが、10%以下の性能低下で動的プログラム認証が可能となり、安全性を向上できることが分かった。

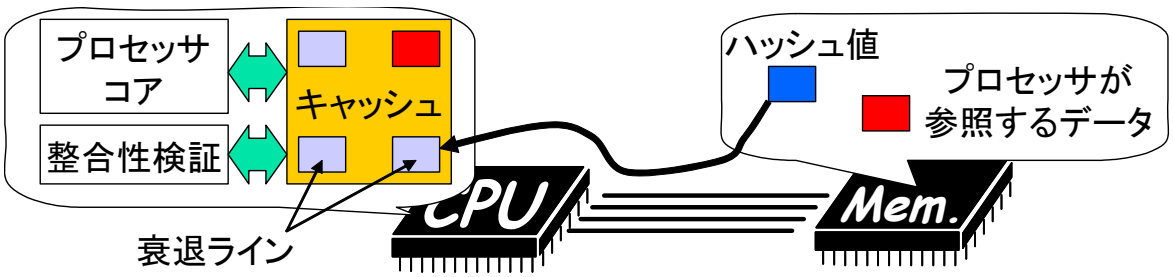


2．3　メモリデータの改ざんを高速に検出する～メモリ整合性検証の高速化～

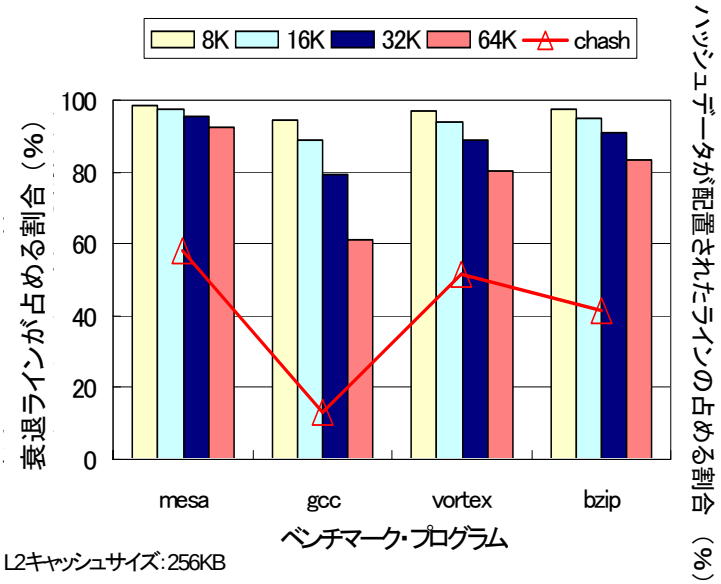
プログラムを正しく実行するためには、その制御フローだけでなく、実行途中に生成される中間結果（データ）も守らなければならない。通常、プログラム実行において必要となる全てのデータをプロセッサ・チップ内部に保存することは難しい。そのため、外部に接続されたメモリ（例えば主記憶）とのデータ通信が発生する。このような状況において、Spoofing アタック、Splicing アタック、Replay アタックなどの攻撃が知られている。このような、メモリデータの改ざんに基づく攻撃の防御策として、ハッシュ木の利用がある。右図で示すように、保護すべきメモリ空間をあるデータ・ブロックに分割（例えば 64 バイト）し、それをリーフとするハッシュ木を構成する。各ノードはデータ・ブロックまたは子ノードのハッシュ値である。主記憶へのライトバック発生時に毎回ハッシュ木を更新するため、ルートは常に保護対象メモリ空間の最新状態を表現することになる。通常，ルートは安全なプロセッサ・チップ内部に保存される。そして、主記憶からキャッシュへデータをロードする際にはルートの値を計算し、チップ内部に保存している値と比較する。もし不一致であれば、以前のメモリ更新から現在までの間に何らかのデータ改竄が発生したことになる。



このハッシュ木を用いたメモリ整合性検証は、オフチップ・メモリアクセスが発生する度に実行される。そのため、単純な実装方法では、ハッシュ値の比較や更新のためにもオフチップ・メモリアクセスが発生し、プロセッサ性能を大幅に低下させる。この解決策として、プログラム実行時に参照されるデータと同様に、ハッシュ木を構成するハッシュ値もオンチップ・キャッシュに格納し、それをハッシュ木のルート値とみなす方法が提案されている（Chash 法と呼ぶ）。しかしながら、この場合にはプログラム実行時に参照されるデータとハッシュ値との間でキャッシュ内競合が発生し、キャッシュ・ヒット率を低下させるといった問題が新たに生じる。その結果、オフチップ・アクセス回数が増加し、引いては、メモリ整合性検証のためのハッシュ木アクセスが新たに生じる。そして、再度キャッシュ内競合が発生し、キャッシュミス率の増加により再びメモリ整合性を繰り返すといった悪循環が生じる。



そこで本研究では、キャッシュ・メモリの衰退ラインを利用したメモリ整合性検証の高速化方式を提案した。一般に、キャッシュ・ラインはミスに伴うデータロードの直後にアクセスが集中し、その後全くアクセスされないという特徴がある。このように、アクセスされなくなったキャッシュ・ラインを衰退ライン呼ぶ。実際には，ある一定の時間（衰退インターバルと呼ぶ）を設定し、非アクセス時間が衰退インターバルを超えた時点で当該ラインを衰退ラインと判定する。本提案手法では、図に示すように、ハッシュ値は衰退ラインにのみ配置可能とする。これにより、ハッシュ値がプログラム実行時に参照されるデータをオンチップ・キャッシュから追い出す確率を低減し、プロセッサ性能を向上する。実際、衰退インターバルを 8K/16K/32K/64K クロック・サイクルと設定した際、256KB の L2 キャッシュにおいては 60%～95%以上の領域が使用済みである衰退ラインとなる（右図の棒グラフ）。これに対し、Chash 法にてハッシュ値をキャッシングするのに必要となる要領は、総衰退ライン要領より少ない（右図の折れ線グラフ）。ベンチマーク・プログラムを用いた定量的評価を行った結果、従来方式と比較して、大きな性能オーバヘッド削減効果を得ることができた。



L2キャッシュサイズ:256KB

5　自己評価:

本研究では、「最後の砦」としてのハードウェア・レベル、特にプロセッサ・レベルでの 1)安全性向上技術の確立、ならびに、2)安全性と性能/消費エネルギー・オーバヘッドのトレードオフを考慮した新アーキテクチャの提案が目標であった。特に、不正プログラム実行の防御を対象とし、プロセッサ構成法に関する研究を進めてきた。

まず、バッファ・オーバフローを動的に検出する SCache アーキテクチャに関しては、キャッシュ・メモリを拡張することで容易に戻りアドレス改ざんの攻撃を検出可能である事をしめした。また、安全性と性能ならびに消費エネルギーのトレードオフを解析し、安全性重視の場合と消費エネルギー重視の場合で調整可能である事を示した。本研究テーマに関しては、当初の目的を十分に達成できたと考える。 次に、プログラム実行の動的認証技術については、実行の振舞いを静的に制御するコード生成技術を考案し、その実現可能性を示した。また、プロセッサ性能の低下は 10%程度である事をしめした。しかしながら、当初の目的であった消費エネルギーとのトレードオフ解析は完了しておらず、今度の課題である。 最後に、メモリ整合性検証の高速化に関しては、本研究スタート時には計画していないテーマであった。さきがけ研究を進める過程において考案したアイデアであり、情報漏えいの防止を考慮したセキュア・ハードウェアへの応用が期待できる。

以上で述べたように、安全性を考慮したプロセッサ構成法に関して一定の成果を達成することができた。しかしながら、真の意味で安全なコンピュータを実現するためには、解決すべき課題は多い。今後、ハードウェアからソフトウェアまでのシステム階層を跨いだ、安全性向上技術の確立が必要である。

6　研究総括の見解：

コンピュータ・システムの安全性向上のために、1）バッファ・オーバフローによる実行制御の乗っ取りを動的に検出するセキュア・キャッシュ(***SCache***)の提案、2）実行の振舞いを鍵情報とする動的プログラム認証方式の提案、3）メモリデータの改ざんを高速に検出するためにキャッシュ・メモリの衰退ラインを利用したメモリ整合性検証の高速化方式の提案をおこない、その有効性を示した。しかしながら、安全性向上のためにはまだ多くの課題が残されている。今後も引続き更なる挑戦を期待している。

7　主な論文等：

論文

（1） K. Inoue, “Return Address Protection on Cache Memories,” IEICE Trans. On Electronics, Dec. 2006.

（2） Koji Inoue, ″Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache,″ *IEEE International Conference on Electronics, Circuits and Systems*, Dec. 2006.

（3） Koji Inoue, ″Supporting A Dynamic Program Signature: An Intrusion Detection Framework for Microprocessors,″ *IEEE International Conference on Electronics, Circuits and Systems*, Dec. 2006.

（4） K. Inoue, ″Secure Cache: Run-Time Detection and Prevention of Buffer Overflow Attacks,″ *Proc. of the Asia and South Pacific International Conference on Embedded SoCs*, July 2005.

（5） K. Inoue, ``Energ y-Security Tradeoff in a Secure Cache Architecture Against Buffer Overflow Attacks,'' ACM Computer Architecture News, vol33, no.1, pp.81-89, Mar. 2005.

特許

（1）発明者：井上弘士，発明の名称：プログラム認証システム，出願人：JST，出願日：　2003 年 12 月 10 日

（2）発明者：井上弘士，発明の名称：キャッシュ・メモリ及びその制御方法，出願人：JST，出願日：2004 年 5 月 24 日

受賞

特になし

招待講演

（1）「10 年後のシステム LSI」、第 10 回システム LSI ワークショップ　イブニングパネル、パネリスト