バッファ・オーバフロー・アタックを動的に検出するセキュア・キャッシュー安全性と消費エネルギーのトレードオフー

井上, 弘士
科学技術振興機構さきがけ | 福岡大学工学部電子情報工学科

出版情報：先進的計算基盤システムシンポジウム（SACSIS2004), pp.315-323, 2004-05
バージョン：
権利関係：

バッファ・オーバフロー・アタックを動的に検出するセキュア・キャッシュ
—安全性と消費エネルギーのトレードオフ—

井上弘士†‡

本稿では，高度情報化社会システムの大きな脅威であるコンピュータ・ウィルス問題に着目し，それを解決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)を提案する．また，このようなキャッシュ・システムを前提とし，安全性と消費エネルギーのトレードオフに関する議論を行う．多くのコンピュータ・ウィルスはバッファ・オーバフローを引き起こし，関数の戻りアドレスを改ざんすることでプログラムの実行制御を乗っ取る．SCache では，本来キャッシュが有する冗長性を活用し，書込みデータの複製を生成することで戻りアドレスを保護する．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，多くのプログラムにおいて99.7%以上の戻りアドレスの安全性を保障することができた．

A Secure Cache Architecture against Buffer-Overflow Attacks
- Considering an Energy - Security Tradeoff -

KOJI INOUE†‡

This paper proposes a novel cache architecture, called Secure Cache (SCache), to protect computer systems from malicious codes, and discuses an energy-security tradeoff. A number of malicious codes attempt to take program-execution flow by causing stack smashing that corrupts a return address. In order to avoid the return address corruption, SCache generates a replica data in the cache area. In our evaluation, for many benchmarks, it is observed that more than 99.7% of return-address loads can be protected.

1. はじめに

近年，LSI 技術やネットワーク技術の発展に伴い，例えばユビキタス・コンピューティングのように便利で快適な社会環境が現実となりつつある．そして，電子財布や電子マネー，さらには，電子投票といった様々なデジタル・サービスの普及に伴い，コンピュータ・システムは今まで以上に重要かつ秘密性の高い情報を処理するようになるであろう．したがって，安心して暮らせる高度情報化社会を実現するためには，コンピュータ・システムの安全性向上が必要不可欠となる．

一方，1970 年代初頭にマイクロプロセッサが開発されて以来，それを核とするコンピュータ・シス

† 福岡大学工学部電子情報工学科 〒814-0133 福岡県福岡市城南区七隈 8-19-1

‡ 科学技術振興機構さきがけ 〒332-0012 埼玉県川口市本町 4 丁目 1 番 8 号

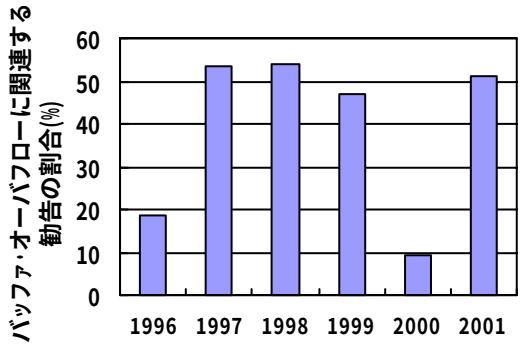

図 1：CERT バッファ・オーバフロー勧告(文献[1])

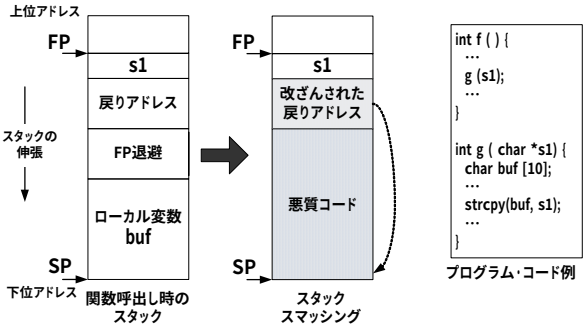

図 2：スタック・スマッシング

テムは目覚しい性能向上を遂げてきた. その中でも特に, 拡大を続けるプロセッサ―主記憶間の性能差を隠蔽するため，キャッシュ・メモリは重要な役割を担っている．そして，高いヒット率と高速アクセスの両立を目指し，これまでに多くの研究/開発が行われた[2][8]．また，1990 年代におけるモバイル・コンピューティングの普及に伴い，高性能化と低消費電力化といった相反する要求を同時に満足する様々なキャッシュ・アーキテクチャが提案されてきた[6][7][9][11]．しかしながら，安全性の向上に関する議論は殆ど行われていないのが現状であり，今後はセキュリティを考慮したメモリ・システムの実現方式を確立する必要がある．

そこで本稿では，コンピュータ・システムの安全性向上を目的とした新しいキャッシュ・アーキテクチャとしてセキュア・キャッシュ(***SCache***)を提案する．具体的には，コンピュータ・ウィルス問題に着目し, 最近特に多くの被害が報告されているバッファ・オーバフロー・アタックの動的検出方式を示す. また，安全性だけでなく，性能や消費エネルギーに関する考慮も必要である．そこで本稿では，これまでには殆ど行われていない安全性と消費エネルギーのトレードオフに関して議論する．

以下, 第 2 節ではバッファ・オーバフローによるスタック・スマッシングの詳細を説明する．また，これまでに提案された安全性向上技術を紹介し, 提案手法との違いを明確にする．次に，第 3 節では SCache アーキテクチャの内部構成ならびに動作の詳細を示し, 第 4 節で安全性と消費エネルギーに関する評価を行う．そして，最後に第 5 節で簡単にまとめる．

2. スタック破壊によるプログラム実行の乗っ取り

2.1 被害状況

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. 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の割合を図 1に示す. 図から分かるように, 多くの悪質プログラムはバッファ・オーバフローの脆弱性を利用しており，その防御策が極めて重要となる．

2.2 スタック・スマッシング

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**)．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．このようなスタック・スマッシングの原因となるバッファ・オーバフローの脆弱性は, strcpy や strcat などの C 標準ライブラリ内に存在する．これらの関数では，文字列をローカル変数に代入する際に領域サイズのチェックを行わない. そのため，ローカル変数で指定したバッファ・サイズより大きな文字列等を代入した場合, 確保されたロ

ーカル変数メモリ領域の境界を越えて書込みを行う．

スタック・スマッシング発生時の様子を図 2に示す．ここでは，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・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．その結果，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．

2.3 関連研究

これまでに，スタック・スマッシングに対する様々な防御方法が提案された．これらは，バッファ・オーバフロー回避のためのコード解析や変換の有無，ならびに，スタック・スマッシングの検出時期によって以下のように分類できる．

- **静的解析かつ静的検出型：**プログラム・コードを静的に解析してバッファ・オーバフロー発生可能場所を検査する．例えば，文字列操作に限定し，「コピー元の文字列サイズ」が「コピー先のローカル変数領域サイズ」を超えていないか検査する方法などがある[13].
- **静的解析/変換かつ動的検出型：**プログラムの静的解析により，スタック・スマッシングを検出するためのコード変換をコンパイル時に行う．文献[4]で提案された SASI では，実行の振舞いを監視するリファレンス・モニタ(RM)をコード中に挿入する．例えば，バッファ・オーバフローの危険性があるコード部分に対し，領域チェックを行うための RM コードを挿入する．また，文献[3]で提案された StackGuard では，canary word と呼ばれる乱数を生成して戻りアドレスと共にスタックへ格納する．canary は戻りアドレスの直後にプッシュされるため，スタック・スマッシングが発生した場合には canary 値も改ざんされる．よって，関数呼出し時と復帰時の canary 値を比較する事で戻りアドレスの改ざんを検査できる．
- **動的解析/変換かつ動的検出型：**プログラム実行中にコードの解析や変換を行う．先に説明した静的解析や変換を行う方式とは異なり，オブジェクト・コードの互換性を保つ事ができる．具体的には，バイナリ変換を行うことで，リファレンス・モニタに相当するバッファ・オーバフロー検査用コードを動的に生成する方法などがある[1][12].
- **解析/変換を必要としない動的検出型：**コード解析や変換を行う事無く，戻りアドレス保護を実現する方式である．ソフトウェア・アプローチとしては，安全性を保障した C ライブラリ(libsafe と呼ばれる)を提供する方法や，OS の介在により動的に canary を追加する方法などが提案されている[1][5]．一方，ハードウェア・アプローチとしては，プロセッサ内部に戻りアドレス専用スタックである SRAS(Secure Return Address Stack)を搭載する方法が提案されている[10]．戻りアドレスをプッシュする際，それと同時に SRAS にも戻りアドレスを書込む．そして，関数から復帰する時，メモリ・スタックならびに SRAS からポップした戻りアドレスを比較する．比較結果が不一致であればスタック・スマッシングが発生していることになる．

本稿で提案する SCache は「解析/変換を必要としない動的検出型」のハードウェア・アプローチに属する．したがって，オブジェクト・コードの互換性を完全に保つ事ができる．また，C ライブラリや OS といったシステム・ソフトウェアの変更は伴わない．SCache の基本アプローチは，SRAS と同様に戻りアドレスをメモリ・スタックとは別領域に保存しておき，関数からの復帰時に比較することでスタック・スマッシングを検出する．しかしながら，SRAS は小容量スタック・メモリとしてプロセッサ内部に実装される．そのため，関数呼出しのネストに伴い容量が不足した場合には，保護された安全な主記憶領域との間でエントリの追出し/リフィルが行われる．SCache も同様の問題を有するが，キャッシュ・メモリという大容量記憶領域を使用するため，より多くの戻りアドレスをプロセッサ・チップ内部で保護できる．また，SRAS とは異なり，LIFO 動作と異なる関数制御(例えば setjmp や longjmp)の場合でもソフトウェアの介在無しに対応できる．さらに，提案手法はプロセッサの内部構造に殆ど影響を与えないため，アウト・オブ・オーダ実行や高

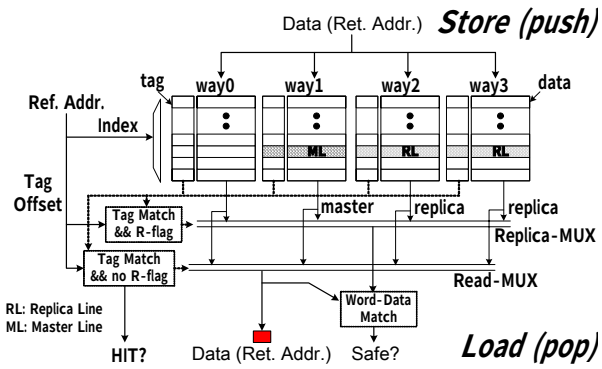

図 3：SCache の内部構成

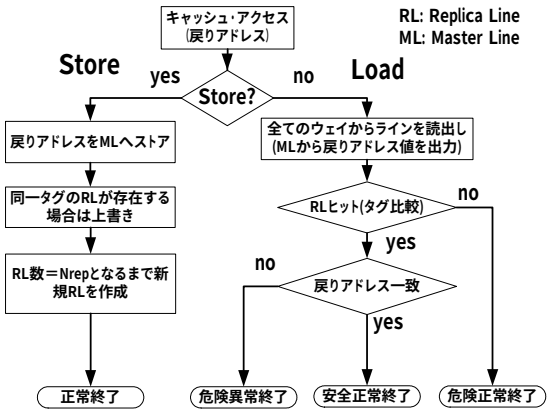

図 4：SCache の動作(ヒット時)

度な分岐予測機構を搭載した複雑なプロセッサに対しても容易に適用可能である．

3. セキュア･キャッシュ･アーキテクチャ

3.1 基本アイデア

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．したがって，キャッシュ上での戻りアドレス保護が可能であれば, プロセッサ構造に影響を与えることなくバッファ・オーバフロー問題を解決できる．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(最大で「連想度－1 個」のレプリカ・ラインを生成可能). 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．

3.2 内部構造と動作

連想度が 4 の SCache 内部構造を図 3に示す. 戻りアドレスの書込み当たりに生成されるレプリカ・ライン数(**Nrep**)は 2 と仮定している．SCache では，全てのタグ･エントリに対して 1 ビットの**レプリカ・フラグ(R フラグ)**を追加する．これは，対応するキャッシュ・エントリがレプリカ・ラインであるか否かを示すフラグである．また，従来の一般的なキャッシュ構造に加え，レプリカ・ライン専用マルチプレクサ(Replica-MUX)と制御信号生成回路，ならびに，32 ビット比較回路(Word-Data Match)が必要となる．SCache のアクセス動作を図 4に示す．ここではキャッシュ・ヒットの場合を想定しているが，ミスの場合でもライン・リプレイスが発生することを除いて基本的に同じである．プログラム実行において戻りアドレスをストアする際, SCache は以下のように動作する．

1. 従来型キャッシュと同様, 参照アドレス中のインデックスを用いてアクセス対象セット(参照セットと呼ぶ)を決定する．そして，タグ比較を行い, ヒットしたラインに戻りアドレスを書込む(このラインを**マスタ・ライン**と呼ぶ).
2. 参照セットにおいて, 同一タグを有するレプリカ･ラインがすでに存在する場合にはストアする戻りアドレスで上書きする．つまり，存在するレプリカ･ラインを更新してコヒーレンシ問題の発生を回避する．
3. 参照セットにおいて，レプリカ・ライン数が Nrep となるまでレプリカ･ラインを生成する．また，対応する R フラグをセットする．

一方, 呼出し元関数への復帰時に戻りアドレスがロードされる際, SCache は以下のように動作する．

1. 参照セットから全ラインとタグを読出す. そして，タグ比較結果が一致しており，かつ，R フラグがリセットされているライン(つまりマスタ・ライン)を選択してプロセッサにデータを転送する．
2. タグ比較結果が一致しており，かつ，R フラグ

がセットされているライン(つまりレプリカ・ライン)が複数存在する場合は何れか 1 つを選択する．もし，レプリカ・ラインが存在しない場合はロードされる戻りアドレスの安全性を保障できないため, その旨をプロセッサに通知してアクセスを終了する．

3. コピー元であるマスタ・ラインの戻りアドレスと，選択したレプリカ・ラインのそれを比較する．もし，比較結果が不一致であればスタック・スマッシングが発生しており，その旨をプロセッサに通知してアクセスを終了する．

実際には, 戻りアドレスの書込みと同時にレプリカ・ラインを生成する．また，戻りアドレスの読出し時，アクセス・データやヒット信号の生成/出力は従来キャッシュと同様の手順となる．さらに，プロセッサは当該ロードが安全である事を示す safe 信号を待つ必要は無い(悪質コードに制御が移る前までに検出されれば良い)．したがって，SCache 方式によるキャッシュ・アクセス時間の増加は殆ど発生しない．一方, 戻りアドレスを操作対象としない通常のロード/ストアは，マスタ・ラインに対してのみ実行される(ヒット条件には R フラグがリセットされていることが含まれる)．よって，レプリカ・ラインに格納した戻りアドレスの複製が通常ストアによって更新されることは無い．なお，SCache を実装する場合，発行されたロード/ストア命令が戻りアドレスを対象とする事を示す情報をプロセッサから入力しなければならない．通常, 戻りアドレスは固定レジスタへ保存される(例えば R31)ため, プロセッサではこのレジスタを対象とするロード/ストアであることを SCache に通知するだけでよく，そのための回路変更は極めてわずかである．

3.3 安全性と消費エネルギーに関する欠点

SCache では，関数呼出し時と復帰時の戻りアドレスそのものを比較し, その結果が一致である場合には戻りアドレスの安全性を保障する．しかしながら, レプリカ・ラインは通常のラインと同様，キャッシュ内競合が発生した場合には置換えの対象となる．そのため，全てのレプリカ・ラインがキャッシュから追出された場合には、戻りアドレスの改ざんを検出することができない。

一方，消費エネルギーに関して，SCache は主に 3 つの欠点を有する．1 つめはキャッシュ・アクセス消費エネルギーの増加である．これまでに提案された多くの低消費電力キャッシュでは, メモリ参照で必要となるデータにのみアクセスする事で低消費エネルギー化を実現する[6][11]．しかしながら, SCache では, 戻りアドレスの読出し/書込みを行うと同時に，レプリカ・ラインの読出しや書込みが必要となる．その結果，本来は必要としないメモリ・アレイの活性化が発生し多くのエネルギーを消費する．第 2 の欠点は，下位階層メモリ・アクセスにおける消費エネルギーの増大である．レプリカ・ライン数の増加に伴い, 有効なキャッシュ容量は小さくなる．その結果，L1 キャッシュのヒット率が低下し，下位メモリ階層へのアクセス回数(例えば L2 キャッシュ・アクセス回数)が増加する．そして第 3 の問題点は, ラインのライトバックに伴う消費エネルギーの増大である．SCache においてレプリカ・ラインを作成する際, 参照セット内に空き領域が無い場合は何れかのラインをキャッシュから追出す必要がある．もし，追出し対象ラインの状態がダーティーである場合にはライトバックが必要となる．

レプリカ・ラインの追出しに関する問題を回避する手段として，複数レプリカ・ラインの作成や，MRU アルゴリズムに基づく配置 (マスタ・ラインを除く MRU ライン) が挙げられる．つまり，レプリカ・ラインのキャッシュ滞在時間をより長くすることで安全性を向上する．また，より安全性を重視する場合，レプリカ・ラインの追出しそのものを禁止する方法も考えられる．しかしその反面，これらはキャッシュ・ヒット率の低下を引き起こし，消費エネルギーに関する欠点をより顕著にする可能性がある．

4. 評価

4.1 実験環境

提案方式の有効性を評価するため，SimpleScalar ツールセット Ver.3.0d を改良して SCache を実装した[14]．また，SPEC2000 ベンチマーク・サイトより 7 つの整数プログラムと 4 つの浮動小数点プログラムを用いて OOO 実行のサイクルレベル・シミュレーションを行った[15]．入力データとしては SPEC より提供される small input を使用している．L1 データ・キャッシュ・サイズは 16KB，ラインサイズは 32B，連想度は 4 と仮定し, その他のプロセッサ構成に関する詳細なパラメータは Simplescalar のデフォルト値を用いた．

SCache では，戻りアドレスがロードされる時，レプリカ・ラインが存在する場合には安全性を保障できる．本稿では，以下の式で安全性を評価する．

$$Vulnerability = (Nv\text{-}rald / Nrald) * 100 \quad (1)$$

ここで，*Nrald* はプログラム実行における IRA ロードの総数である．ここで IRA(Issued Return Address)ロードとは，キャッシュ・メモリに対して発行された戻りアドレス・ロードの事である．また，*Nv-rald* は戻りアドレス改ざんを検出できない(安全性を保障できない)IRA ロード総数を示す．一方，消費エネルギーに関しては以下の式で評価する．

$$Etotal = Erd + Ewt + Ewb + Emp \quad (2)$$

ここで，*Erd* と *Ewt* は，それぞれ，L1 キャッシュの読出し/書込み総消費エネルギーである．また，*Ewb* はレプリカ・ラインの作成に伴うライトバック総消費エネルギーを表す．さらに，*Emp* はキャッシュ・ミス発生において消費されるエネルギーを示す．実際には，0.18 μmCMOS プロセスを用いて 1 ウェイ分(4KB)の SRAM アレイのレイアウト設計ならびに負荷容量抽出を行い，プリチャージ動作も含めた 1 ビット当たりの読出し/書込み消費エネルギーを測定した．また，その結果に基づき，キャッシュ・アクセスにおける消費エネルギーを換算した．なお，*Emp* の値はメモリ階層構造に大きく依存する．そこで，1 回の下位メモリ階層アクセスに要するエネルギーは，従来型キャッシュにおける平均読出し消費エネルギーの 10 倍と仮定した．

4.2 実験結果(安全性)

本評価ではキャッシュの連想度を 4 と仮定しているため，LRU または MRU アルゴリズムに基づき最大 3 個のレプリカ・ラインを生成可能である．そこで，これらの組合せに関して安全性を評価した．実験結果を図 5に表す．また，従来キャッシュ(CONV)におけるミス率と IRA ロード数(*Nrald*)，ならびに，各 SCache モデルにおけるミス率を表 1 に示す．ここで，LRU1R ならびに LRU2R は，それぞれ，LRU 配置アルゴリズムに基づき 1 個もしくは 2 個のレプリカ・ラインを生成するモデルである．同様に，MRU1R と MRU2R は配置アルゴリズムに MRU 方式を用いている．ALL は，参照セット中の全ライン(ただし，マスタ・ラインを除く)にレプリカ・ラインを生成する．

本シミュレーションでは L1 キャッシュ・ミス時のライン置換えアルゴリズムに LRU 方式を採用している．そのため，LRU1R で作成したレプリカ・ラインは他アクセスによって容易にキャッシュから追出される．これに対し，LRU2R ではレプリカ・ラインのキャッシュ生存期間が長くなるため，より高い安全性を実現している．また，同様の理由により，MRU 方式の採用によっても安全性は向上している．全ての SCache モデルを比較した場合，ALL が最も高い安全性を達成しており，多くのプログラムで 99.7%以上の IRA ロードの安全性を保障することができた．一方，表 1で示すように，生成するレプリカ・ライン数の増加，または，MRU 方式の採用に伴い，キャッシュ・ミス率が高くなっている．したがって，性能を重視する場合には，ミス率の低下と安全性の向上を考慮して MRU1R を選択することが適切であると考える．なお，SCache での戻りアドレス書込み時，第3.2節で説明したように，同一タグを有するレプリカ・ラインがすでに存在する場合にはそれらへの上書きを行う．よって，LRU 領域にこのようなレプリカ・ラインが存在する場合，MRU1R は LRU1R と同じ動作となる．そのため，MRU 方式においても，作成されるレプリカ・ライン数を増加することで安全性が向上する．

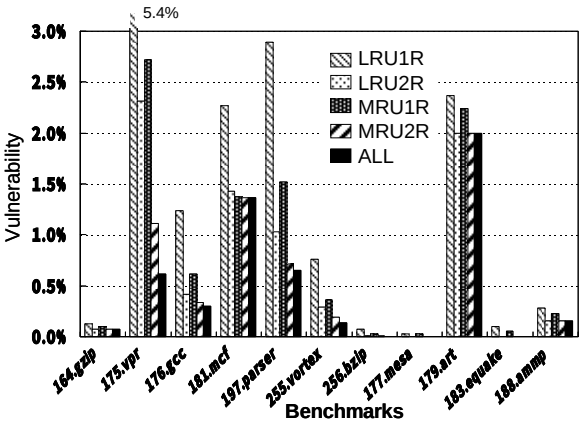

図 5：危険な戻りアドレス・ロードの発生率

4.3 実験結果(消費エネルギー)

第 4.1 節で示した消費エネルギー・モデルに基づき評価した結果を図 6に示す．ここで，図 6は，不必要なウェイ・アクセスを回避するウェイ予測キャッシュ[6]を基準とした際の消費エネルギー・オーバヘッドを表している．この図から，レプリカ・ライン数の増加に伴いオーバヘッドが大きくなっている事が分かる．特に，最も多くのレプリカ・ラインを生成する ALL では，最大で約 23%の消費エネルギー・オーバヘッドが発生した(*197.parser*)．

表 1：キャッシュ・ミス率

Model / Bench.	CONV		LRU1R	LRU2R	MRU1R	MRU2R	ALL
	Miss Rates	#IRA Load(*Nrald*)					
164.gzip	5.22%	4,930,467	5.22%	5.22%	5.22%	5.23%	5.25%
175.vpr	3.53%	5,627,709	3.56%	3.63%	3.59%	3.66%	3.74%
176.gcc	4.26%	37,519,156	4.29%	4.37%	4.33%	4.43%	4.64%
181.mcf	20.02%	992,419	20.02%	20.03%	20.05%	20.06%	20.10%
197.parser	4.13%	45,466,527	4.18%	4.44%	4.23%	4.55%	5.07%
255.vortex	1.75%	22,101,265	1.79%	1.91%	1.82%	1.94%	2.32%
256.bzip	2.31%	18,147,017	2.31%	2.32%	2.31%	2.32%	2.45%
177.mesa	0.14%	4,727,396	0.15%	0.16%	0.15%	0.16%	1.08%
179.art	42.93%	32,466	42.93%	42.93%	42.93%	42.93%	42.93%
183.equake	2.44%	3,580,827	2.44%	2.46%	2.45%	2.47%	2.52%
188.ammp	36.27%	6,307,839	36.28%	36.31%	36.28%	36.30%	36.38%

IRA: Issued Return Address

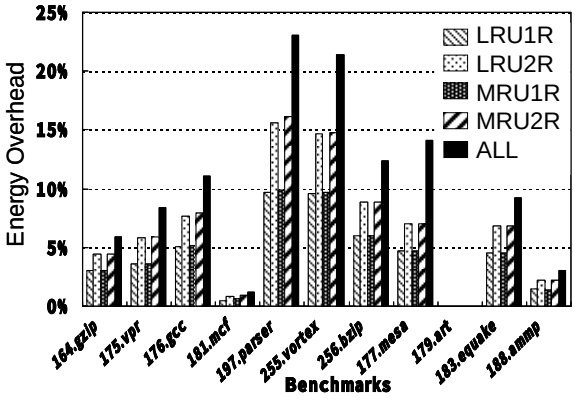

図 6：消費エネルギー・オーバヘッド

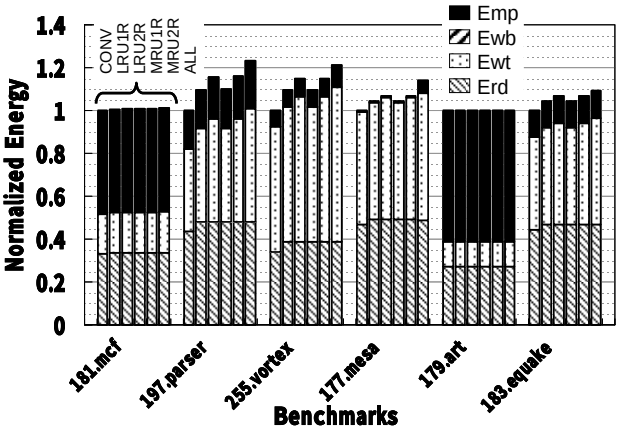

図 7：消費エネルギーの内訳

消費エネルギーを詳細に解析するため, 第4.1節で示した式(2)に関する内訳を測定した．その結果を図 7に示す．ここでは，紙面の都合上，消費エネルギー・オーバヘッドが最も大きな 2 つの整数プログラム(*197.parser*，*255.vortex*)と浮動小数点プログラム(*177.mesa*，*183.equake*)，ならびに，最もオーバヘッドが小さい 2 つのプログラム(*181.mcf*，*179.art*)での結果を示している．従来キャッシュ(CONV)と比較して，読出し消費エネルギー*Erd*は全ての SCache モデルにおいてほぼ同じ増加率である．これは，レプリカ・ライン作成数ならびに配置アルゴリズムに関わらず，戻りアドレス・ロード発行時に全てのウェイが活性化されるためである．反対に，書込み消費エネルギー*Ewt*は作成されるレプリカ・ライン数に比例して増加する．また，*Emp*はミス率の増加に伴い大きくなっている．特に *177.mesa* において, 従来方式と比較した場合, ALL モデルは大幅なヒット率の低下を引き起こしており(表 1を参照)，その影響による *Emp* の増加が顕著に現れている．

オーバヘッドの小さい *181.mcf* ならびに *190.art* とその他を比較した場合, これら 2 つのプログラムではキャッシュ・ミスによる消費エネルギーが多くの割合を占めている．実際，表 1で示すように，これらプログラムのミス率は極めて高い．このように, 従来のミス率と比較して，SCache によるミス率の増加が十分小さい場合，*Emp* に関する消費エネルギー・オーバヘッドが隠蔽される．また，本評価で基準としているウェイ予測方式では, ミス率の増加と共にウェイ予測による消費エネルギー削減効果が低減する．これらの理由により，SCache の採用に伴う消費エネルギー・オーバヘッドは小さくなったものと考える．

一方，全てのプログラムにおいて，LRU1R と MRU1R，LRU2R と MRU2R をそれぞれ比較した

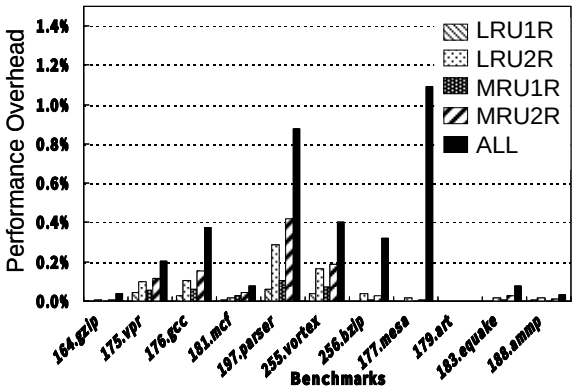

図 8：性能オーバヘッド

場合, 消費エネルギーに関する差は殆ど見られない. 生成するレプリカ・ライン数が同じ場合, 配置アルゴリズムには関係なくキャッシュ消費エネルギーはほぼ同一となる．これに対し，MRU 方式の場合はヒット率の低下を招くが, それによる消費エネルギー・オーバヘッドが比較的小さかった．このような結果を考慮し，レプリカ・ラインの配置アルゴリズムには MRU 方式が適していると考える．

4.4 実験結果 (性能)

SCache ではレプリカ・ラインをキャッシュ内セットに生成するため，キャッシュ・ミス率の増加に伴う性能オーバヘッドが生じる. 各ベンチマークにおける性能オーバヘッドを図 8に示す. 生成するレプリカ・ライン数が最大である ALL モデルにおいて，最悪の場合でも性能低下は 1.1%(*177.mesa*)である．また, その他のモデルに関しては，*197.paser* を除く全てのプログラムにおいて 0.2%以下の性能オーバヘッドである．これは，保護すべき戻りアドレスの数に対し, データキャッシュは十分な容量を有するためである．以上より，提案手法による性能低下は無視できる程度に小さいと考える．

5. おわりに

本稿では，スタック・スマッシングに対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．SCache では，キャッシュの大容量領域を活用して戻りアドレスを保護する．実験を行った結果, 多くのプログラムで 99.7%以上の戻りアドレス・ロードに関して安全性を保障することができた．また，消費エネルギー解析を行った結果, 安全性を重要視する場合には全てのラインに複製を生成する ALL モデルが，一方，ある程度の安全性を維持しつつ低消費エネルギー化が要求される場合には MRU 方式に基づく MRU1R が適切であると分かった．

本稿での提案手法は完全ハードウェアによるバッファ・オーバフロー解決手段の一つである．このような方式においてはハードウェアの追加または変更が必要であるため，今日主流となっている「ソフトウェアの更新」と比較して，既存計算機システムへの適用可能性は低くなる．しかしながら，依然としてバッファ・オーバフローを活用した不正プログラムは猛威を振るっており, 特に重要な情報を処理対象とする次世代電子機器システムではその開発時に安全性を考慮する必要がある．よって，本稿での提案は安全性を考慮した次世代計算機システムの構築における 1 つのアプローチと位置づけることができる．

本実験では, 主に性能測定を目的として使用される SPEC ベンチマークを利用した．今後，実際にバッファ・オーバフローの脆弱性を有するプログラムを用いた評価が必要である．また，様々なキャッシュの構成を前提としたより詳細な性能, 消費エネルギー，ならびに，安全性に関する評価を行い，これらの間に存在するトレードオフの探索ならびに最適化技術を開発する予定である. なお, 現在, 0.18 μm CMOS プロセスを用いた SCache コアの完全版を設計中である．

謝辞

本研究を遂行するにあたり, 多くのご意見を頂いた科学技術振興機構さきがけプロジェクト「情報基盤と利用環境」領域関係者各位に感謝します. また, 様々な議論を共にした福岡大学モシニャガ研究室ならびに九州大学安浦研究室の諸氏に感謝する. なお, 本設計は東京大学大規模集積システム設計教育研究センターを通し, 株式会社日立製作所および大日本印刷株式会社の協力で行われたものである．

参 考 文 献

[1] A.Baratloo, N.Singh, and T.Tsai, “Transparent Run-Time Defense Against Stack Smashing Attacks,” Proc. of 2000 USENIX Annual Technical Conference, June 2000.

[2] J.-L. Baer, "2K papers on caches by Y2K: Do we need more?,” KeyNote Address in the 6th Int. Symp. on High-Performance Computer Architecture, Jan. 2000. http://www.irit.fr/ACTIVITES/EQ_APARA/HPCA6/BaerHpca6.PDF

[3] C.Cowan, C.Pu, D.Maier, H.Hinton, J.Walpole, P.Bakke,

S.Beattie, A.Grier, P.Wagle, and Q.Zhang, “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” Proc. of 7th USENIX Security Symposium, Jan, 1998.

[4] U.Erlingsson and F.B.Schneider, “SASI Enforcement of Security Policies: A Retrospective,” Proc. of the workshop on New security paradigm, 1999.

[5] M.Frantzen and M.Shuey, “StackGhost: Hardware Facilitated Stack Protection,” Proc. of the 10th USENIX Security Symposium, Aug. 2001.

[6] K.Inoue, T.Ishihara, and K.Murakami, “Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption,” Proc. of the Int. Symp.on Low Power Electronics and Design, pp. 273--275, Aug. 1999.

[7] M.B.Kamble and K.Ghose, “Analytical Energy Dissipation Models For Low Power Caches,” Proc. of the Int. Symp. on Low Power Electronics and Design, pp.143--148, Aug. 1997

[8] C.Kim, D.Burger, and S.W.Keckler, “An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches,” Proc. of the 10th Int. Conf. on Architectural support for Programming Languages and Operating Systems, pp.211-222, Oct. 2002.

[9] J.Kin, M.Gupta, and W.H.Mngione-Smith, ”The Filter Cache: An Energy Efficient Memory Structure,” Proc. of the 30th Int. Symp. on Microarchitecture, pp.184--193, Dec. 1997.

[10] R.B.Lee, D.K.Karig, J.P.McGregor, and Z.Shi, “Enlisting Hardware Architecture to Thwart Malicious Code Injection,” Proc. of the Int. Conf. on Security in Pervasive Computing, Mar. 2003.

[11] M.D.Powell, A.Agarwal, T.N.Vijaykumar, B.Falsafi, and K.Roy, “Redicing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping,” Proc. of the 34th Int. Symp. on Microarchitecture, pp.54--65, Dec. 2001.

[12] K.Scott and J.Davidson, “Safe Virtual Execution Using Software Dynamic Translation,” Proc. of the 18th Computer Security Applications Conference, Dec. 2002.

[13] D.Wagner, J.S.Foster, E.A.Brewer, and A.Aiken, “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities,” Proc. of the Network and Distributed System Security Symposium, Feb. 2000.

[14] SimpleScalar Tool Sets, http://www.simplescalar.com/.

[15] SPEC(Standard Performance Evaluation Corporation, http://www.specbench.org/.