不正プログラムの実行防止を目的とするオンチップ・キャッシュ・アーキテクチャ

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：並列/分散/協調処理に関するサマー・ワークショップ（SWoPP04), 情報処理学会研究報告
2004-ARC-159, pp.121-126, 2004-08. 情報処理学会ARC研究会
バージョン：

不正プログラムの実行防止を目的とする オンチップ・キャッシュ・アーキテクチャ

井上弘士†‡

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，それを実現するためのアーキテクチャ・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，その安全性，性能，ならびに，消費エネルギーに関する評価を行う．近年，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，関数戻りアドレスを改ざんする事でプログラム実行制御を乗っ取る．この問題を回避するため，SCache は書き込まれた戻りアドレス値の複製を生成する．ベンチマーク・プログラムを用いて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 Cache Architecture to Prevent Malicious Code Executions

KOJI INOUE†‡

This paper proposes an architectural support to improve computer security, called Secure Cache (SCache), and evaluates its energy/security efficiency. A number of malicious codes attempt to hijack program-execution flow by causing stack smashing that corrupts the return address stored in a stack. 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，バッファ・オーバフローの脆弱性を活用した不正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．不正プログラムは，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，不正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．

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

† 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

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．一方，これまでに，ソースコードの解析等によりバッファ・オーバフローの発生を回避する多くの静的アプローチが提案された[3][4][7]．しかしながら，これらは再コンパイルが必要なため，すでに存在するオブジェクト・コードの実行に関しては適用するのが難しい．

そこで本稿では，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***)を提案する．また，安全性ならびに性能/消費エネルギー・オーバヘッドに関する評価を行い，その有効性を明らかにする．SCache はソフトウェアの介在を必要としないハードウェア・アプローチであるため，オブジェクト・コードの互換性を完全に保つことができる．また，プロセッサとは分離した実現方式のため，プロセッサの内部構造に殆ど影響を与える事無く実装できる．なお，本提案の詳細は文献[1]で示した．これに加え，本稿では複製の追出しを禁止する方式についても評価する．

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．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，消費エネルギー，ならびに，性能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

近年，多くの不正プログラムは，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のがバッファ・オーバフローの脆弱性であり，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 1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．

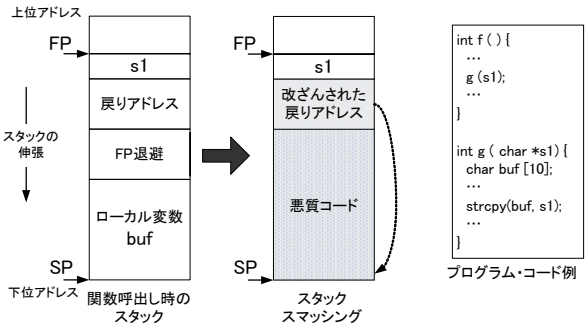

図 1：スタック・スマッシング

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る．一方，不一致

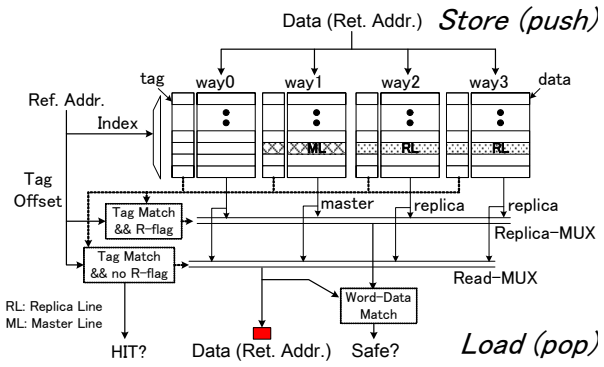

図 2：SCache の内部構成

の場合には，スタック・スマッシングにより戻りアドレスが改ざんされたと判断し，その旨をプロセッサに通知する．

これまでに，戻りアドレスの改ざんを検出するハードウェア・アプローチとして，SRAS(Secure Return Address Stack)が提案された[6]．SRAS はプロセッサ内部に搭載された LIFO メモリであり，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．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，SCache では 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．また，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．

3.2 内部構造と動作

連想度が 4 の SCache 内部構造を図 2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)とその制御回路，ならびに，32 ビット比較回路(Word-Data Match)が必要となる．以下，キャッシュ・ヒットの場合を想定し，SCache の動作を説明する．なお，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．

なお，戻りアドレスの書込み時，マスタ・ラインに含まれる全データの複製を作成する必要は無く，実際には戻りアドレスに対応する 1 語のみが複製の対象となる．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．

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．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 設計選択肢

SCache では，関数呼出し時と復帰時の戻りアドレス値そのものを比較し，一致である場合には安全性を保障する．しかしながら，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。よって，安全性を向上するためには，レプリカ・ラインのキャッシュ滞在時間をより長くする必要がある．レプリカ・ラインの生成アルゴリズムを考えた場合，主に，1)生成するレプリカ・ライン数，2)レプリカ・ライン配置アルゴリズム，3)レプリカ・ラインの追出し許可/禁止，といった 3 つの設計選択肢が存在する．これらを考慮すると，以下のような SCache モデルが実現可能となる（ここではキャッシュの連想度は 4 と仮定）．

- ***LRU1L :*** 戻りアドレス書込み時，マスタ・ラインとレプリカ・ラインを除く通常のキャッシュ・ラインの中から，LRU 配置アルゴリズムに基づき 1 個のレプリカ・ラインを生成する．一度生成されたレプリカ・ラインは，対応する戻りアドレス・ロードが発行されるまでキャッシュ中に滞在し続ける．なお，同一セット内に空き領域が存在しない場合はレプリカ・ラインを作成できない．
- ***LRUn :*** LRU 配置アルゴリズムに基づきレプリカ・ラインを n（ただし，連想度＞n≧1）個生成する．生成されたレプリカ・ラインは通常ラインと同様に置換え対象となる．
- ***MRUn :*** MRU 配置アルゴリズムに基づきレプリカ・ラインを n（ただし，連想度＞n≧1）個生成する．生成されたレプリカ・ラインは通常ラインと同様に置換え対象となる．
- ***ALL:***戻りアドレスが書込まれる同一セットにおいて，最大数(連想度-1)のレプリカ・ラインを生成する．レプリカ・ラインは追出し対象となる．

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[8]を改良して SCache を実装した．また，SPEC2000 ベンチマーク・サイトより 6 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たサイクルレベル・シミュレーションを行った．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，キャッシュ・ポート数は 1 と仮定した．なお，プロセッサ構成を決定するその他のパラメータに関しては文献[2]に示されたデフォルト値を用いている．一方，評価対象は，第3.3節で示した SCache モデル(LRU1L，LRU1，LRU2，MRU1，MRU2，ALL)ならびに従来型キャッシュ（CONV）とする．ここで，全てのキャッシュは，低消費電力化を実現するためにウェイ予測方式を採用していると仮定する[5]．従来型キャッシュの場合，ウェイ予測が誤りであった場合のみ全てのウェイが活性化されるのに対し，SCache では戻りアドレス読出し時にも全ウェイがアクセス対象となる．

4.2 安全性/消費エネルギー・モデル

本評価では，以下の式を用いて安全性を評価する．

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

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 ロード総数を表す．一方，消費エネルギーに関しては以下の式で評価する．

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

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・ミスに伴うライン置換えによって消費されるエネルギーである．実際には，0.18μmCMOS プロセスを用いた 4KB SRAM アレイ（1 ウェイ分）の設計ならびに回路シミュレーションを行い，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

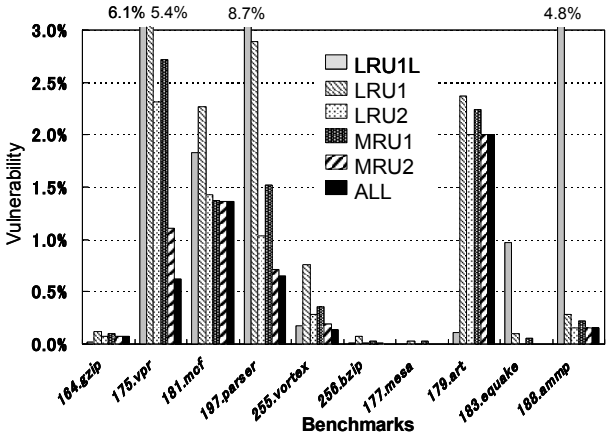

図 3：危険な戻りアドレス・ロードの発生率

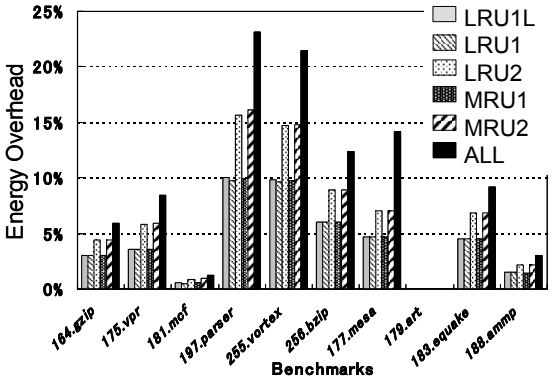

図 4：消費エネルギー・オーバヘッド

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．

4.3 実験結果（安全性）

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，LRU1 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，LRU2 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．これらに対し，レプリカ・ラインの追出しを禁止した LRU1L では，幾つかのプログラム（175.vpr，197.parser，183.equake，188.ammp）において安全性が低くなっている．これは，1)早すぎるレプリカ・ラインの開放，2)レプリカ・ライン生成の失敗，という 2 つの理由に起因する．前者は，アウト・オブ・オーダ実行においてコミットされない戻りアドレス・ロードによって早期にレプリカ・ラインが開放（つまり消滅）される場合に生じる．一方，後者に関しては，同一セット中のラインが全てレプリカによって占領された場合に発生する．

4.4 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 4に示す．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*)．

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 5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*)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，*Emp* の増加が顕著に現れている．これに対し，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，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．

一方，全てのプログラムにおいて，LRU1 と

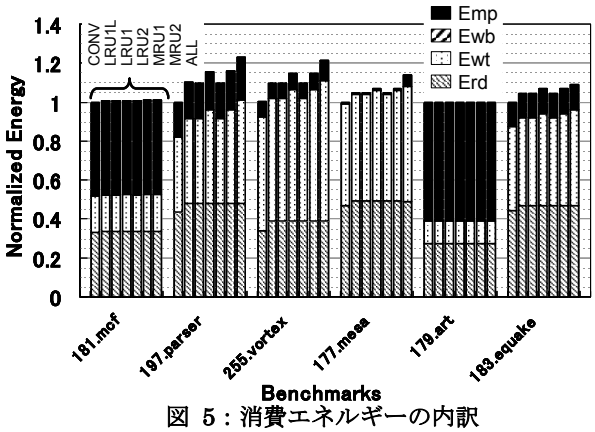

図 5：消費エネルギーの内訳

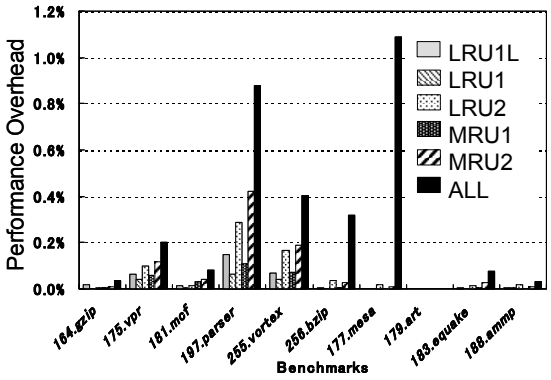

図 6：性能オーバヘッド

MRU1, LRU2 と MRU2 をそれぞれ比較した場合，消費エネルギーに関する差は殆ど見られない．これは，生成するレプリカ・ライン数が同じ場合，配置アルゴリズムの違いはキャッシュ消費エネルギーにさほど影響しない事を意味する．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いためである．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高い安全性を実現する MRU 方式が適していると考える．

4.5 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，ミス率の増加に伴う性能低下が生じる．各ベンチマークにおける性能オーバヘッドを図6 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%以下の性能低下である．これは，保護すべき戻りアドレス数に対してデータキャッシュが十分な容量を有するためであり，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，安全性，性能，ならびに，消費エネルギーに関する評価を行った．その結果，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できる事が分かった．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を行う予定である．

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．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] 井上弘士,”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,” 先進的計算基盤システムシンポジウム SACSIS2004, 2004 年 5 月.

[2] D.Burger and T.M.Austin,”Univ. of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997.

[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] 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.

[6] 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.

[7] 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.

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