RC-001

スタックスマッシング攻撃の正確な検出機構を備えたプロセッサアーキテクチャ
A Processor Architecture Featured with a Precise Detection Mechanism of Stack Smashing Attacks

中務 国男† 山田 徹†* 布目 淳‡ 平田 博章‡ 柴山 潔‡
Kunio Nakatsukasa Toru Yamada Atsushi Nunome Hiroaki Hirata Kiyoshi Shibayama

1. まえがき

本稿では，バッファオーバフローの脆弱性を利用したスタックスマッシング攻撃からの防御を目的として，攻撃の有無を検出するプロセッサアーキテクチャを提案する．我々はすでに，C 言語における *setjmp*()/*longjmp*()を含むシステムライブラリの呼び出しや C++の例外処理を行う実用的なプログラムに対して，攻撃の検出洩れや誤検出を発生させない方式[1]を提案しており，本稿では，その実現方式と性能評価について報告する．

2. 攻撃検出のためのリターンアドレススタック

スタックスマッシング攻撃に対して，手続き呼び出しのたびにそのリターンアドレスを保存するリターンアドレススタック (RAS: Return Address Stack) をプロセッサ内に用意して，攻撃を検出しようとする試み (SplitStack[2], Secure-RAS[3][4], SmashGuard[5], Reliable-RAS[6]) がある．しかし，実際のプログラム実行においては，手続きからのリターンに用いるアドレスと，その時点で RAS のトップに保存されているアドレスとが異なる場合がある．

このような状況は，C 言語における *setjmp*()/*longjmp*()の実行や C++などのオブジェクト指向言語における例外処理など，複数の関数やメソッドを飛び越える非局所分岐が行われる際に発生する．このような状況に対して RAS の処理を適切に行えなければ，正常なプログラム実行であるにもかかわらず，攻撃によりリターンアドレスが改竄されたものと誤検出してしまう事態が生じる．

C 言語における *setjmp*()/*longjmp*()の一般的な実装では，*longjmp*()からのリターン時に実行される分岐命令は，アクティベーションレコード内に保存されたリターンアドレスではなく，以前に実行した *setjmp*()のリターンアドレスを使用する．このリターンアドレスは *setjmp*()の実行時にコンテキスト情報として保存されていたものであり，*longjmp*()を実行する時点では RAS からはすでにポップされている．そのため，*longjmp*()に対する攻撃検出処理を通常通り行った場合，攻撃の誤検出が発生する．

また，オブジェクト指向言語における例外発生時の処理においては，分岐先となる catch 節の先頭アドレスを事前に取得することはそもそも不可能であり，RAS 内に catch 節の先頭アドレスを保存する手段が存在しない．また，処理系で用意された関数は，手続きの巻き戻し処理を行う際に，分岐すべき catch 節の検索処理や呼び出し元手続きへの分岐処理を行うために，スタックフレーム上のリターンアドレスを使用する．そのため，これらの処理の際にもリターンアドレスの改竄の有無を検査する必要がある．

本方式で用いる RAS は，これらの非局所分岐に対応するための機能を付加したものである．以後，この機構を DRAS (Defensive-RAS)と呼ぶことにする．また，マシン命令セットにおいて，手続き呼び出しに使用される分岐命令をコール命令，手続きからのリターンに使用される命令をリターン命令とそれぞれ呼ぶことにする．

3. DRAS とその操作

3.1 DRAS の構造

DRAS は図 1(b)に示すように，リターンアドレスを保存するための RAS と，特に *setjmp*()からのリターンアドレスを保存するための SJS (Set Jump Stack) の 2 つのスタックから構成する．RAS の 1 つのエントリにはリターンアドレスのみを保存し，SJS の 1 つのエントリには，リターンアドレスのほかに，*longjmp*()の実行時にトップとすべき RAS のエントリの位置情報を保存する．

3.2 DRAS 操作のための専用命令

非局所分岐に対応するために，本方式では，DRAS 操作のための専用命令を新たに設ける．

3.2.1 *dcl_setjmp* 命令

dcl_setjmp 命令は，C 言語における *setjmp*()/*longjmp*()の実行に対応するための命令である．オペランドで指定された値 *i* を用いて，RAS のトップから数えて *i* 番目のエントリに保存されている *setjmp*()からのリターンアドレスを SJS のトップエントリにコピーし，コピー先の RAS entry フィールドの値を，RAS のトップから数えて *i*+1 番目のエントリを指すように設定する機能を持つ．*setjmp*()の中で必ずこの命令を実行しなければならないものとする．

プログラミング言語から *dcl_setjmp* 命令を簡易に使用するためには，例えばライブラリとしてラッパ手続きを用意する必要があるが，このラッパ手続きの呼び出しの深さは実装によって異なる．本命令では手続き呼び出しの相対的な深さを表す値 *i* をオペランドで指定することで，実装方式に依存せずに *setjmp*()/*longjmp*()の実行に対応することを可能としている．

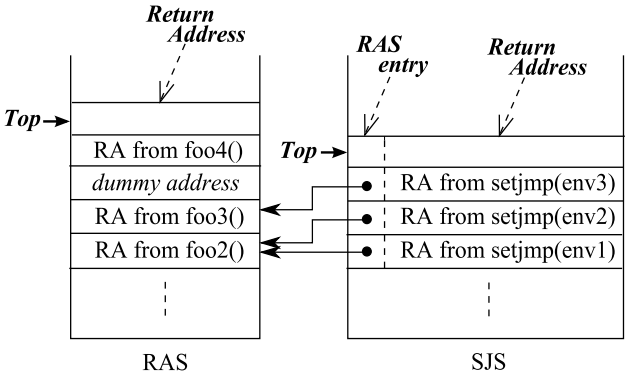
3.2.2 *unwind_test* 命令

unwind_test 命令は，オブジェクト指向言語における例外処理に対応するための命令である．オペランドで指定されたリターンアドレスが RAS 内に存在するか否かを，RAS のトップから下方に向かって検索する機能を持つ．検索対象のリターンアドレスが RAS 内に存在する場合は，その RAS のエントリを変数（記憶装置）*unwind_target* に保存し，RAS 内に存在しない場合は攻撃検出信号を生成

† 京都工芸繊維大学大学院 工芸科学研究科 情報工学専攻
‡ 京都工芸繊維大学大学院 工芸科学研究科 情報工学部門,
Dept. of Information Science, Kyoto Institute of Technology
* 現在，(株) 三菱自動車工業,
Mitsubishi Motors Corporation

```
foo1() { … foo2() ; … }
foo2() { … setjmp(env1) ; setjmp(env2) ; foo3() ; … }
foo3() { … dummycall ; setjmp(env3) ; foo4() ; … }
foo4() { … }
```

(a) プログラム



(b) DRASの内容

図 1　DRAS の構造（例）

する．処理系で用意される例外処理関数の中で必ずこの命令を実行しなければならないものとする．

3.2.3 *trusted_return* 命令

trusted_return 命令は，オブジェクト指向言語における例外処理に対応するための命令であり，*unwind_test* 命令とセットで使用する．*trusted_return* 命令は，通常のリターン命令としての機能のほかに，以前に実行した *unwind_test* 命令によって保存した *unwind_target* の位置まで RAS のエントリをポップする機能を持つ．処理系で用意された例外処理関数において，catch 節へ分岐するためのリターン命令として必ずこの命令をしなければならないものとする．

3.3 コール命令実行時の DRAS 操作

コール命令の実行時には，単に RAS のトップエントリにリターンアドレスをコピーする．実行したコール命令が *setjmp*()を呼び出すものである場合，後に *setjmp*()の中で実行される *dcl_setjmp* 命令により，リターンアドレスをさらに SJS のトップエントリにコピーし，*longjmp*()の実行時にトップとすべき RAS のエントリの位置情報を設定する．例として，図 1 (a) のプログラムにおいて関数 *foo4*()を実行している時点での DRAS の内容を図 1(b)に示す．

なお，システムライブラリ内の関数が自身のリロケーション情報を得るためにコール命令を使用する場合があるが，これはマシン命令レベルでは手続き呼び出しと区別がつかないため，通常のコール命令と同様にその情報を RAS に保存する．図 1 の例では，関数 *foo3*()内でこのような目的でコール命令を実行したものと想定して，これを *dummycall* と記述している．

3.4 リターン命令実行時の DRAS 操作

リターン命令の実行時には，まず RAS のトップから下方に向かって，リターン命令の分岐先アドレスと RAS のすべてのエントリを比較する．比較が一致した時点で，RAS のトップからそのエントリまでをポップして処理を終了する．先に述べた特殊な目的でのコール命令が使用された場合にも正しく攻撃検出を行うためには，このように RAS のトップだけでなく，その下方にあるエントリに対しても比較処理を行わなければならない．ただし，実際には多くの状況において RAS のトップに一致するリターンアドレスが保存されており，最大でも 2 個のエントリに対して比較を行えば十分である．

上記の比較が一致しなかった場合，SJS のトップから下方に向かって，リターンアドレスが一致するエントリが見つかるまで SJS 内を検索する．一致するエントリが見つかった場合，RAS のトップから，SJS の RAS entry フィールドが指す位置までのすべてのエントリをポップする．一致するエントリが見つからなかった場合，その分岐先アドレスは攻撃によって改竄されたものであるとして攻撃検出信号を生成する．

オブジェクト指向言語における例外発生時においては，処理系で用意された関数の中で実行される *unwind_test* 命令により，スタックフレーム上のリターンアドレスの改竄の有無を検査すると同時に，RAS をどこまでポップすべきかを記憶する．これにより，*trusted_return* 命令を用いて catch 節へ分岐する際に，RAS のトップを正しく更新することができる．なお，*trusted_return* 命令のオペランドに指定する分岐先アドレスは，ライブラリ関数内で catch 節の先頭アドレスに自ら書き替えたものであり，改竄の有無を検査する必要はない．

4. DRAS の実現方式

4.1 システム構成の概要

プログラムの実行性能の低下を抑えるためには，DRAS をハードウェアで実装することが望ましい．しかし，DRAS に対する操作は，コール命令，リターン命令，非局所分岐に対応するための専用命令，のそれぞれで異なり，これらに対する処理をハードウェアのみで実現するのは得策とはいえない．また，DRAS を構成する 2 つのスタックの容量が不足する場合やコンテキストスイッチが発生した際に，DRAS 内の情報を失わないためにメモリ上に退避するなどの対策が必要となる．これはプログラム実行の性能低下を招くだけでなく，OS に対しても少なからぬ変更を必要とする．

そこで本方式では，OS やユーザプログラムを実行する通常のプロセッサコアの外部に，DRAS に関する処理を行う専用のプロセッサを設け，その上で実行する組込みソフトウェアによって DRAS の機能を実現する．以後，通常のプロセッサコアをメインプロセッサ (MP: Main Processor)，DRAS 操作を担当する専用のプロセッサをリターンアドレス管理プロセッサ (RAMP: Return Address Management Processor) とそれぞれ呼ぶことにする．

RAMP 上でのソフトウェアを用いた柔軟な処理により，ハードウェアのみでは困難な DRAS 操作を行うことが可能となる．また，MP の外部でリターンアドレスを独立して管理するため，コンテキストスイッチの発生時にスタックのエントリをメモリ上に退避する必要はない．さらに，例えば，メインメモリの数ページを固定的に RAMP の占有領域として割り当て，RAMP 上のソフトウェアで

DRAS 操作を行うことにより，DRAS のサイズについても柔軟に制御することが可能となる．

RAMP は DRAS 操作とそれに関わる攻撃検出処理を行うための必要最低限の機能を持った小規模なプロセッサコアで構成する．RAMP 上で実行するプログラムは DRAS 操作に特化した単純なものであり，MP 上の OS から一切のランタイムサービスを受ける必要がない．したがって，MP 上の OS は RAMP を単なるハードウェア機構として扱うことができる．このようなシステム構成上の観点からは，RAMP 上のソフトウェアには，MP 上の OS で扱うコンテキスト ID に関連づけて DRAS 操作を行う機能が必要となる．

4.2 リターンアドレスの転送経路

MP と RAMP を機能分担して構成したことにより，MP がコール命令やリターン命令を実行した際にリターンアドレスを RAMP に転送する必要が生じる．しかし，リターンアドレスの送信経路として，MP の命令デコーダから RAMP へ専用のデータパスを設けることは，配線及びピン制約の観点から実現が困難である．そこで，図 2 に示すように，MP とデータキャッシュとを繋ぐデータバスに RAMP を接続することにより，実装に要するハードウェア量の増大を回避する．

MP の命令デコーダがコール命令やリターン命令をデコードする際，擬似的なストア命令として機能する内部オペレーションを生成することにより，リターンアドレスをデータバス上に出力する．先に述べた *dcl_setjmp* 命令，*unwind_test* 命令，*trusted_return* 命令のそれぞれの専用命令についても，MP においてはそれ自体を擬似的なストア命令として実行する．内部オペレーションや上記の専用命令の実行によって，データバスに出力される RAMP 用のコマンドやリターンアドレスを RAMP が入力し，DRAS 操作及び攻撃検出の処理を行う．

4.3 システムコールの実行制御

MP はコール命令やリターン命令を実行する際，命令デコーダが生成した内部オペレーションを，ロードストアユニットを経由して RAMP へ転送する．そのため，MP が実行したリターン命令に対して RAMP が攻撃検出処理を開始するまでに時間的な遅延が生じる．

しかし実際には，たとえ攻撃を受けていたとしても，リターン命令の実行時点で即座に攻撃検出を完了する必要はない．プログラムがシステムコールを行うごとに，それまでに攻撃を受けていないことが確認できれば，攻撃の影響をそのプログラムのみに封じることができ，システム全体に被害が及ぶことはない．逆に言えば，それまでに実行されたすべてのリターン命令に対する攻撃検出の処理結果が得られるまで，MP においてシステムコールの実行を遅らせる必要がある．そこで，MP 内に専用のカウンタを用意し，システムコールの実行を制御する．MP がリターンアドレスを RAMP へ送信するごとにカウンタの値を 1 増加させ，RAMP から処理結果を得るごとにカウンタの値を 1 減少させる．MP はシステムコールを実行する際にこのカウンタを参照し，その値が 0 になるまではシステムコールを実行しない．

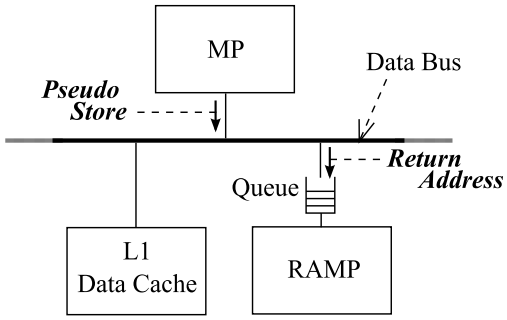


図 2　MP から RAMP へのデータパス

5. 性能評価

5.1 評価環境

PowerPC[7]命令セットを対象とした時間シミュレータを作成し，本稿で提案する攻撃検出機構がプログラムの実行性能に与える影響について，シミュレーションによる評価を行った．同命令セットにおけるすべての種類の分岐命令の中で，命令アドレスの保存を指定するための LK フィールドが 1 のものをコール命令とし，また，bclr 命令をリターン命令とした．評価用プログラムには SPEC CPU ベンチマーク 2000[8]の中の 11 種類を用い，入力データは test データセットを用いた．

5.2 シミュレーション結果

5.2.1 プログラムの実行性能への影響

表 1 に示すパラメータの MP 及び RAMP で構成するプロセッサと，MP のみで構成するプロセッサのそれぞれにおいて，各ベンチマークプログラムの実行サイクル数を測定し，DRAS を適用した場合のプログラムの実行サイクル数の増加率を計測した．また，比較のため，既存の防御方式のひとつである StackGuard[9]を適用したプログラムを MP で実行した場合についても同様の計測を行った．StackGuard は本方式と同じくリターンアドレスの改竄を検出するという観点での防御方式であり，その中でもプログラム実行の性能低下が小さい．計測した実行サイクル数の増加率を図 3 に示す．

図 3 の通り，今回計測したすべてのプログラムにおいて，DRAS を適用した場合の実行サイクル数の増加率は 0.7% 以下であった．一方で StackGuard を適用した場合は，プログラムによって値が大きく異なるが，全体として DRAS よりもはるかに高い値を示しており，最も高い値は gzip の 32.9%であった．両方式の比較からも，DRAS がプログラムの実行性能に与える影響が極めて小さいことが確認できる．

5.2.2 実行サイクル数の増加要因

StackGuard では，手続き呼び出し/リターンのたびにリターンアドレスの改竄を検査するための命令列を実行するため，手続き呼び出しの回数が増えるに従って実行サイクル数が一定の割合で増加する．

これに対して DRAS では，MP がシステムコールを実行する時点で，それまでに実行したすべてのリターン命令

表 1　MP 及び RAMP のパラメータ

Feature	MP	RAMP
Functional units (execution delay)	Int ALU (1cycle)×1 Int MUL (4cycles)×1 Int DIV (23cycles)×1 Fp ALU (5cycles)×1 Fp MUL (5cycles)×1 Fp DIV (35cycles)×1 Ld/St (1cycle)×1 Branch (1cycle)×1	Int ALU (1cycle)×1 Ld/St (1cycle)×1 Branch (1cycle)×1
Pipeline depth	6 stages (Instruction Fetch, Decode, Issue, Execute, Write Back, Retire)	
Branch prediction	static, always not-taken	
Reservation station	50 entries×4	50 entries×3
Fetch/Decode /Issue width	4 instructions/cycle	1 instruction/cycle
Retire width	5 instructions/cycle	
L1 I-cache	size: 8KB, linesize: 16B, direct mapping, miss penalty: 8/17 cycles	
L1 D-cache	size: 8KB, linesize: 16B, direct mapping, miss penalty: 8/17 cycles	
L2 Unified cache	size: 16KB, linesize: 32B, 4-way set associative, miss penalty: 32/65 cycles	

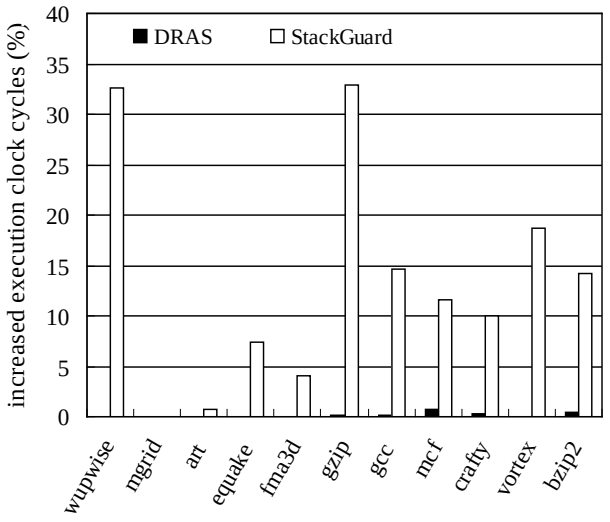


図 3　実行サイクル数の増加率

に対する攻撃検出処理を RAMP が完了していれば，プログラム実行の性能低下は伴わない．ただし，コール命令やストア命令に対して生成する内部オペレーションや非局所分岐に対応するための専用命令は，その命令が実行される際，ストア命令と同じストアパイプラインを利用するため，本来のプログラムに含まれるストア命令の処理効率を阻害する要因となり得る．

また，今回計測を行ったプログラムに関しては，DRAS の適用がプログラムの実行サイクル数に与える影響が極めて小さいが，手続き呼び出しの深さや呼び出し間隔といったプログラムの性質によってはこの影響が増大する可能性がある．したがって，今後より多くのベンチマークプログラムを用いて詳細な性能評価を行う必要がある．

6. むすび

本稿では，スタックスマッシング攻撃の防御方式として，プロセッサコアの外部に専用のプロセッサを設け，そのプロセッサ上で攻撃検出処理を行うソフトウェアを実行することにより，プログラム実行の性能低下を抑えつつ正確な攻撃検出を行う方式を提案した．また，シミュレーションにより，本方式による攻撃検出処理のオーバヘッドが極めて小さいことを確認した．

今後は，より実装に近いレベルでの詳細なパラメータを設定した上で，より多くのベンチマークプログラムを用いて性能評価を行う．例えば，今回のシミュレーションでは分岐予測を静的に予測するものとしているが，これは攻撃検出処理のオーバヘッドに影響を及ぼす要因となる可能性がある．パラメータを再設定した上での性能評価の結果をもとに，必要に応じて，攻撃検出処理に対する最適化などによりさらなる処理効率の向上を図る．

また，プロセッサのコア数を増加させた場合の攻撃検出処理やシステム構成についても現在検討を進めており，これらについては別の機会に報告する予定である．

謝辞

本研究の一部は日本学術振興会科学研究費補助金（基盤研究 (C) 21500053 および同 22500046）の補助による．

参考文献

[1]平田博章, 山田徹, 布目淳, 柴山潔, “マシン命令レベルでのプログラム実行モニタリングによる手続き呼び出し関係の正確な検知方式”, 第 9 回情報科学技術フォーラム講演論文集, RC-006, pp.87-90 (2010).
[2] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, “Architecture Support for Defending Against Buffer Overflow Attacks,” Proc. of 2nd Workshop on Evaluating and Architecting System dependabilitY (EASY) (2002).
[3] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, “A Processor Architecture Defense against Buffer Overflow Attacks,” Proc. of IEEE International Conference on Information Technology: Research and Education, pp.243-250 (2003).
[4] Y. J. Park, Z. Zhang, and G. Lee, “Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks,” IEEE Micro, vol.26, no.4, pp.62-71 (2006).
[5] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote, “Detection and Prevention of Stack Buffer Overflow Attacks,” Communications of the ACM, vol.48, no.11, pp.51-56 (2005).
[6] D. Ye, and D. Kaeli, “A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing,” Proc. of Workshop on Architectural Support for Security and Anti-Virus, pp.73-80 (2005).
[7]IBM Corp., “Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture” (2001).
[8] Standard Performance Evaluation Corp., “SPEC CPU2000,” http://www.spec.org/cpu2000/.
[9] 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 the 7th USENIX Security Symposium (1998).