RC-006

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

Precise Detection of Procedure Calling/Returning through Machine Instruction-Level Monitoring

平田 博章† 山田 徹‡ 布目 淳† 柴山 潔†
Hiroaki Hirata Toru Yamada Atsushi Nunome Kiyoshi Shibayama

1. まえがき

バッファオーバフロー脆弱性の問題について，ハードウェア/ソフトウェアを含む広い方面から様々な研究が行われている．その 1 つに，メモリ上のコールスタックとは別に，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，スタックスマッシュ攻撃の有無を検出しようとする試み（SplitStack[1]，Secure-RAS[2][3]，SmashGuard[4]，Reliable-RAS[5]）がある．しかし，実際のプログラム実行においては，手続きからリターンする時点で RAS のトップに保存されたアドレスが必ずしもリターンアドレスであるとは限らない．

本稿では，このような状況を発生させる非局所分岐とプログラミング言語レベルの手続きの検出に論点を絞って議論することにする．以後，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．

非局所分岐の典型的な例の 1 つは，C 言語における *longjmp*()の使用である．*longjmp*()を呼び出すと，*longjmp*()をコールした命令の次の命令に戻るのではなく，以前に実行した *setjmp*()のリターンアドレスにジャンプする．したがって，RAS を必ずしも後入れ先出し（Last-In First-Out）構造の単純なスタックとして扱うことはできない．

C++などのオブジェクト指向言語における例外処理も非局所分岐の典型例の 1 つである．try 節の実行中に例外が発生すると，呼び出し元のいくつものメソッドを飛び越えて適切な catch 節へと到達しなければならない．ただし，catch 節の先頭アドレスは本来のリターンアドレスではないので，RAS 内にはそのアドレスが記憶されていないのが普通である．

また，コール命令の分岐先アドレスが，必ずしも手続きの先頭アドレスであるとは限らない．例えば，システムライブラリ内の関数では，自分自身がどのアドレスにリロケートされているかを知るための目的で，コール命令を使用する場合がある．プログラミング言語レベルでは，これは手続き呼び出しではないので，実際にそのコール命令の次の命令にジャンプするためのリターン命令は，そのプログラム中には存在しない．

以上の状況に対して適切に処理できなければ，正常なプログラム実行であるにもかかわらず，攻撃があったものと誤検出してしまう事態が生じる．本稿では，スタックスマッシュ攻撃の防御において，上記の課題に起因する誤検出を防止するために，プログラミング言語レベルでの手続き呼び出し関係をマシン命令レベルで正確に把握する方式を提案する．

†京都工芸繊維大学大学院 工芸科学研究科 情報工学部門
‡京都工芸繊維大学大学院 工芸科学研究科 情報工学専攻
Dept. of Information Science, Kyoto Institute of Technology

2. 手続き呼び出し関係の検知方式

2.1 リターンアドレススタックの構造

RAS は従来から分岐予測の機構として用いられてきたが，この場合は命令フェッチステージで機能し，投機実行によって RAS の内容に不整合が生じ得る．また，命令解読前にそれが分岐命令であるかどうかを判別しなければならない．これらが正しく行えない場合には性能上のペナルティを受ける．

しかし，攻撃検出を目的とする場合は，予測が外れた程度の問題では済まされない．攻撃の検出漏れや誤検出につながり，プログラムの実行そのものに重大な問題を生じる．そこで，本稿で議論する RAS は，分岐予測に用いる RAS とは別の独立した機構として備えることにする．命令解読の結果に基づいて，また，命令リタイアのステージで RAS を操作することによって，上記の問題を回避する．

先に述べたように，リターン命令が実行された場合，RAS のトップに保存されたアドレスが実際のリターンアドレスとは限らない．どの関数の中に戻るのかを正確に検知するために，本方式では，アクティベーションレコードを指すフレームポインタの値を用いる．本方式で用いる RAS の構造を図 1(b)に示す．RAS の 1 つのエントリには，リターンアドレスとコール命令実行時のフレームポインタの値のほかに，*setjmp*()/*longjmp*()の処理に対応するための 1 ビットの SJ（Set-Jump)フラグを設ける．

2.2 コール命令実行時の処理

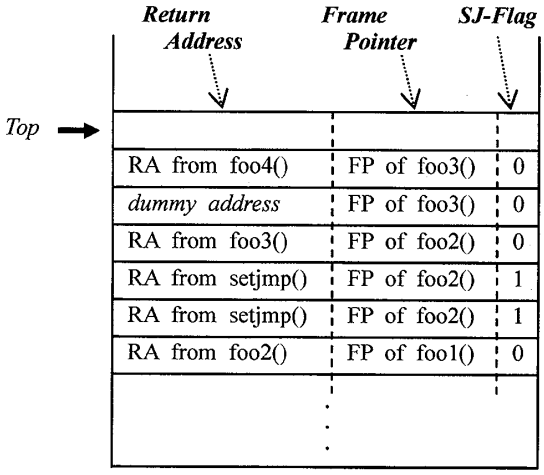
コール命令が実行される場合の RAS における処理アルゴリズムを図 2 に示す．以降のアルゴリズムの記述では，RAS の i 番目のエントリを *RAS*[*i*]で表し，RAS 内に格納されるリターンアドレス，フレームポインタおよび SJ フラグの値をそれぞれ *ret*，*fp*，*sjflag* と表記する．また，RAS のトップの位置を変数（記憶装置）*Top* で指し，定常状態では，*Top* が指すエントリの内容は空とする．

コール命令の実行時は，図 2 に示すように，単に，RAS のトップのエントリにリターンアドレス（*NextPC*）とその時点でのフレームポインタの値（*FP*）をコピーし，同時に，SJ フラグの値を 0 に設定する．

また，本方式では，***dcl_setjmp*** 命令を設け，*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ので，***dcl_setjmp*** 命令のオペランドには手続

```
foo1( ) { ... ; foo2( ); ... }
foo2( ) { ... ; setjmp( ); setjmp( ); foo3( ); ... }
foo3( ) { ... ; dummycall; foo4( ); ... }
foo4( ) { ... }
```

(a) プログラム



(b) リターンアドレススタックの内容

図 1 リターンアドレススタック(RAS)の例

き呼び出しの相対的な深さを指定する．RAS のトップから数えて，オペランドで指定された値だけ下方にあるエントリに対し，その SJ フラグの値を 1 に設定するのが *dcl_setjmp* 命令の機能である．

通常の手続き呼び出しでは，手続きからリターンした後はそのリターンアドレスを RAS 内に保存しておく必要はないが，*setjmp*()の場合は，*setjmp*()から戻った後も，その後の *longjmp*()の実行に備えて保存しておかなければならない．本方式では，*dcl_setjmp* 命令を用いて *setjmp*()からのリターンアドレスが格納されたエントリの SJ フラグを 1 に変更し，リターン命令実行時に，エントリを解放するか否かをこの SJ フラグの値によって判断する．

例として，図 1(a)のプログラムに示すように，関数 foo1()内で関数 foo2()を呼び出し，関数 foo2()内で 2 個の setjmp()を実行した後に関数 foo3()を呼び出し，さらに関数 foo3()内から関数 foo4()を呼び出して，関数 foo4()を実行している時点での RAS の内容を図 1(b)に示す．関数 foo3()では自命令アドレスを取得するためにコール命令を実行したものと想定して *dummycall* と記述している（通常，これはソースプログラムの記述としては現れない）が，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ので，RAS にはその情報を保存する（図 1(b)では *dummy address* としている）．

2.3 リターン命令実行時の処理

リターン命令実行時の処理アルゴリズムを図 3 と図 4 に分割して示す．まず，図 3 に示すように，RAS を *Top*－1 の位置から下方に向かって，順に，フレームポインタの値が等しいエントリを検索する．検索の結果，そのようなエントリが見つからなかった場合は攻撃検出信号を生成する（AttackAlert()）．目的のエントリが見つかった場合は，

```
RAS [Top] . ret ← NextPC ;
RAS [Top] . fp ← FP ;
RAS [Top] . sjflag ← 0 ;
Top ← Top + 1 ;
```

図 2 コール命令検出時の処理アルゴリズム

```
Top ← Top − 1 ;
while ( Top ≧ 0 )
  begin
    if (RAS [ Top ] . fp = FP )
      begin
        ReturnAddressComparison() ;
        Exit ;
      end
    Top ← Top − 1 ;
  end
AttackAlert() ;
Exit ;
```

図 3 リターン命令検出時の処理アルゴリズム

```
if ( RAS [ Top ] . ra = Target )
  begin
    if ( RAS [ Top ] . sjflg = 1 )
      begin
        Top ← Top + 1 ;
      end
  end
else
  begin
    tempTop ← Top − 1 ;
    while ( tempTop ≧ 0 and
                          RAS [ tempTop ] . fp = FP )
      begin
        if ( RAS [ tempTop ] . ra = Target )
          begin
            Exit ;
          end
        tempTop ← tempTop − 1 ;
      end
    AttackAlert() ;
  end
Exit ;
```

図 4 リターンアドレス比較処理のアルゴリズム

Top がそのエントリを指すように更新して，図 4 に示すリターンアドレスの比較処理（ReturnAddressComparison()）を行う．

図 4 のリターンアドレス比較処理においては，まず，図 3 の検索の結果として更新された *Top* が指すエントリに対して，そのエントリ内のリターンアドレスとリターン命令の分岐先アドレス（*Target*）とを比較する．これが一致すれば，正常なリターンとして処理を終了する．ただし，SJ フラグが 1 の場合は，将来の *longjmp*()の実行に備えてこのエントリを解放してはならない．従って，*Top*+1 の値を新たな Top として更新する．

上記のリターンアドレスの比較で一致しなかった場合は，フレームポインタとリターンアドレスの両方が一致するエントリを，さらに下方に向かって検索する．ただし，この検索では *Top* の値は変更しない（つまり，これ以上は RAS

内の情報をポップしない）．目的のエントリが見つかった場合は，正常な手続きからのリターンとして処理を終了する．このようなケースは，1 つの手続き内で複数の *setjmp*() を実行していた場合の *longjmp*()からの戻りや，（本来の手続き呼び出しではなく）自命令のアドレスを取得するためにコール命令を使用していた場合に該当する．一方，目的のエントリが見つからなかった場合は，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処理に対しては，マシン命令の実行をモニタリングして catch 節の先頭アドレスを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記憶する手段が存在しない．そこで，本方式では，専用命令として ***tret*** 命令（trusted-return 命令）を設け，catch 節への分岐はこの ***tret*** 命令を用いるものとする．***tret*** 命令は攻撃検出信号の生成を抑止する以外は，通常のリターン命令と同じ機能をもつ．したがって，図 3 のフレームポインタについての検索処理を行うのみで，図 4 のリターンアドレスに対する比較処理は行わない．以上の方法でどのメソッドの中の catch 節へ制御が渡ったのかを管理できるので，その後も手続き呼び出し関係を正確に追跡することが可能となる．

2.4 攻撃の検出洩れに関する検証

本方式について，スタックスマッシュ攻撃に対して検出洩れが起こらないかどうかを以下に検証する．

(1) 通常のリターン

通常，リターンアドレスはメモリに退避される．また，フレームポインタの値はアーキテクチャによってメモリに退避されるものもあれば，主にレジスタに記憶される（メモリにも退避されるが特殊な場合以外は使用しない）ものもある．いずれにしても，本方式では，手続き呼び出し時にリターンアドレスとフレームポインタの値を RAS 内にコピーし，それらが完全に一致するかどうかをチェックするので，攻撃があった場合には正しく検出することができる．

なお，コンパイラで末端手続き（leaf procedure）に対する最適化を行った場合は，末端手続きの呼び出し自体にコール命令が使用されないので，厳密には手続きの呼び出し関係を正確に検知することはできない．しかし，そのような最適化が行われた場合でも，コール命令実行時とそれに対応するリターン命令の実行時とでフレームポインタの値が一致するので，本方式は，スタックスマッシュ攻撃検出の目的に対しては問題なく機能する．

(2) *longjmp*()

longjmp()の一般的な実装では，アクティベーションレコード内の所定の位置に保存されたリターンアドレスではなく，*setjmp*()によって保存されたコンテクスト情報に含まれるリターンアドレスをターゲットアドレスとして，リターン命令を用いて分岐を行う．攻撃によってコンテクスト情報が改竄さていた場合でも，本方式では，SJ ビットによって，*setjmp*()の呼び出し時のリターンアドレスとフレームポインタの値を RAS 内に保存しておくので，攻撃されたことを正しく検出することができる．

なお，*setjmp*()/*longjmp*()の誤った使用や，結果が未定義の使用法については，本方式では対応しない．ここでの誤った使用とは，例えば，関数 foo()の中で呼び出した *setjmp*()からのリターンアドレスに，関数 foo()から戻った後に *longjmp*()で戻ろうとする場合が挙げられる．このような使用法は，以前には，コルーチンを実現するためのトリックとして故意に用いられたこともあるが，現在は対応する必要性はないと考えられる．

(3) オブジェクト指向言語における例外処理

catch 節への分岐は，メモリに退避してあったフレームポインタを復帰し，また，自らリターンアドレスを catch 節の先頭アドレスに書き換えた後に，リターン命令を実行することにより行う．したがって，アクティベーションレコード上のリターンアドレスが改竄されていたとしても，さらにそれが catch 節の先頭アドレスに書き換えられるため，攻撃は成功しない．このような実装に基づけば，攻撃を検出する必要はないので，***tret*** 命令を用いて，RAS 内に保存されていない命令アドレスへの分岐を許容する．ただし，RAS 内の情報は手続き呼び出し関係を正確に反映しているので，例外処理後も引き続き攻撃の有無を検出可能な状態を保つことができる．

3. 関連研究

SRAS[3]でも，非局所分岐に対応するために，リターンアドレスを RAS にプッシュするための専用命令を設けることを示唆しており，これを用いて *longjmp*()実行時の誤検出を防ぐことができる．しかし，プログラム実行の正しさを保証するための情報をそのプログラム自身で扱うことにすれば，その情報がどのように記憶されるかによってさらに改竄の危険性に対する対応策を講じなければならない．そこで，本方式では RAS 内に SJ フラグを設け，***dcl_setjmp*** 命令のオペランドでリターンアドレスを扱わないように設計している．

また，本方式では，フレームポインタの値を用いて RAS のエントリを（暗示的に）グループ化して管理する（図 3 参照）ので，例えば，1 つの関数内で複数の *setjmp*()を実行し，*longjmp*()からの戻り口を複数個設定する場合にも対応することができる．フレームポインタの値が等しいエントリは，RAS 内でのスタックとしての記憶位置によらず同等に扱うので，実際に *longjmp*()によって 1 つの戻り口に分岐する際に，他の戻り口の情報を RAS から削除してしまうことはない．なお，SJ フラグが 1 のエントリは，*setjmp*()を呼び出した関数から戻る際にすべてポップするので，プログラム終了時まで RAS 内に残ることはない．

オブジェクト指向言語の例外処理に対しても，SRAS では専用命令を用いてあらかじめ catch 節の先頭アドレスを RAS 内にプッシュしておかなければならない．これは，アプリケーションプログラムの再コンパイルを必要とする以外に，その専用命令の実行が性能上のオーバヘッドを生じる可能性もある．例外処理の発生頻度が低いことを勘案すると，明示的または暗示的に catch 節をもつメソッドのすべてについて，そのような専用命令を埋め込んで実行するのは得策とは言えない．これに対して，本方式では，例外処理における非局所分岐の安全性を確認した上で ***tret*** 命令を設け，これを用いて誤検出を回避している．あらかじめ catch 節の先頭アドレスを RAS 内に記憶しておくためのオーバヘッドを伴うことなく，同時に，RAS 内のフレームポインタの値を手がかりにして，RAS 内での巻き戻し（unwinding）を適切に処理することができる．

RRAS[5]では，スタックを用いてリターンアドレスが改竄されていないことを確認しながら，アプリケーションプログラム全体に対してグローバルにリターンアドレスを収集・記憶するが，攻撃の有無の判定方法は明確に示していない．*longjmp*()実行時の分岐先は，それ以前に実行した *setjmp*()のリターンアドレスとして収集されているはずなので，SRAS や本方式のように専用命令を設けることなく，*longjmp*()に対応することができる．しかし，命令アドレスのみを扱うので，オブジェクト指向言語の例外処理に対しては， SRAS と同様に，分岐先の正当性を担保するための何らかの追加処理が必要であるものと推測される．なお，RRAS では，直接再帰呼び出しに対して RAS エントリの消費を軽減する工夫が施されているが，本方式ではフレームポインタの値と組にしてリターンアドレスを記憶するので，このような最適化を施すことはできない．

4. 動作確認

PowerPC[6]を対象とした命令レベルの機能シミュレータを作成し，これに本稿で提案する関数呼び出し関係の検知機能を組み込んだ．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てモニタリングを行った．

例外処理を行う C++のサンプルプログラムを GNU C/C++コンパイラでコンパイルし，シミュレータで実行したときの関数呼び出しに関するトレース出力結果（簡単のため，通常の関数呼び出し関係の出力部分は省略している）を図 5 に示す．図 5 中の call と ret.はそれぞれコール命令とリターン命令の実行を表し，コール命令の場合は呼び出す関数名も併記している．また，縦棒は関数呼び出しのネストの様子を視覚化する目的で表示している．

図 5 の 4 行目で呼び出された関数_ZN7MyStack4pushEi()内で例外が発生し，5 行目において例外が throw されたことが示されている．__cxa_throw()や関数名が_Unwind_で始まる関数は，処理系で用意された関数である．その後，各関数でデストラクトの処理を行いながら関数_Z9testmain1v()内の catch 節へ至るまでの様子が 6～12 行目に示されている．この中で，例えば，7 行目の関数リターンによって，_Unwind_RaiseException()から複数の関数を飛び越して，直接，_Z7test3rdR7MyStacki()内に戻っており，例外処理時の関数呼び出しの関係を正しく検知している様子が示されている．

SPEC CPU ベンチマーク 2006[7]の中の C++ で記述されたプログラムと同ベンチマーク 2000 に含まれるすべてのプログラムについて，本シミュレータで実行した結果，スタックスマッシュ攻撃の誤検出はまったく起こらなかった．*longjmp*()を含むシステムライブラリの呼び出しや C++の例外処理を行うこれらの実用的なプログラムに対して，正しく関数呼び出し関係を検知していることを確認した．

5. むすび

スタックスマッシュ攻撃を防御するための関数呼び出し関係の検知方式を提案し，*longjmp*()や C++の例外処理を含む実用的なプログラムに対して，誤検出が生じないことを確認した．

本稿で示したアルゴリズムは，非局所分岐などに起因する例外的な状況に対応するために RAS 内を検索する．しかし，実際の多くの状況では，RAS のトップにリターンアドレスが記憶されているので，具体的に実装する際には最適化できる可能性がある．また，リターン命令ごとに必ずしも即座に攻撃検知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を確認すれば，たとえ攻撃を受けたとしても，その影響をそのプログラムのみに封じることができ，システム全体に害が及ぶことはない．したがって，システムコールを行うまでに，それまでに実行したすべてのリターン命令に対する攻撃検知を完了できれば，プログラム実行の性能低下は伴わない．

実際の性能および実装の詳細については，RAS のサイズやコンテクストスイッチとの関連も含めて検討中 [8][9] であり，別の機会に報告する予定である．

```
 1:  call    _Z9testmain1v
 2:  call    | _Z7test2ndR7MyStacki
 3:  call    | | _Z7test3rdR7MyStacki
 4:  call    | | | _ZN7MyStack4pushEi
 5:  call    | | | | __cxa_throw
 6:  call    | | | | | | _Unwind_RaiseException
 7:  ret.    | | | | | | |
 8:  call    | | | _Unwind_Resume
 9:  ret.    | | | |
10:  call    | | _Unwind_Resume
11:  ret.    | | |
12:  ret.    | |
13:  ret.
```

図 5 例外処理のトレース例

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，同 22500046 および若手研究(B) 21700058）の補助による．

参考文献

[1] 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).

[2] 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).

[3] 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).

[4] 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).

[5] 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).

[6] IBM Corp., “Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture” (2001).

[7] Standard Performance Evaluation Corp., “SPEC CPU2006,” *http://www.spec.org/cpu2006/*.

[8] 野間翔平, 布目淳, 平田博章, 柴山潔, “バッファオーバフロー検知用付加プロセッサの概要”, 情報処理学会第 71 回全国大会論文集, vol.1, pp.41-42 (2009).

[9] 野間翔平, 布目淳, 平田博章, 柴山潔, “スタックスマッシング攻撃の正確な検出方式とその性能制約条件”, 電子情報通信学会技術研究報告 CPSY2009-47, vol.109, no.319, pp.25-30 (2009).