CPUによるリターンアドレス書換え攻撃検知とソフトウェア支援

久保田 曹嗣[1,a)]　明田 修平[1]　瀧本 栄二[1]　齋藤 彰一[2]　毛利 公一[1]

概要：バッファオーバフローを悪用した攻撃には，リターンアドレスを書き換え，攻撃者の意図した不正な制御フローに処理を移すものがある．我々は，このような攻撃を防止するため，CPU の CALL/RET 命令を拡張し，両者の対称性をチェックすることで攻撃を検知する手法を研究している．しかし，CALL/RET 命令の対称性が損なわれるケースが存在するため，誤検知が発生することがわかった．具体的には，シグナルハンドラ，動的リンカの遅延バインド，zero length call，大域ジャンプ，C++の例外処理の 5 つを確認している．そこで，CPU での CALL/RET 命令の比較処理を変更し，さらに，CPU では対処できない誤検知は Linux カーネルおよび標準 C ライブラリを新たに改変することで誤検知への対処を行った．自作したテストアプリケーションを用いて検証を行った結果，5 つの誤検知に対して対処できたことを確認した．

キーワード：バッファオーバフロー攻撃，ROP，シャドウスタック, QEMU

A Detection Method of Return Address Overwriting Attacks Based on CPU and Support by Software

Soshi Kubota[1,a)]　Shuhei Aketa[1]　Eiji Takimoto[1]　Shoichi Saito[2]　Koichi Mouri[1]

Abstract: Some buffer overflow attacks overwrite return address to execute malicious code. In order to defend from them, we are studying a method detecting these attacks. The method extends CALL/RET instructions of the CPU and checks if the pairs of the instructions are symmetric. However, we have recognized misdetection cases because of the impaired symmetric of the instructions. Specifically, we confirmed five cases of signal handler, lazy bind of dynamic linker, zero length call, global jump and exception handling of C++. Therefore, we have changed the comparison mechanism of the instructions. Furthermore, we have modified Linux kernel and C standard library for misdetection cases which cannot be handled by CPU only. We confirmed that our system can treat all of the misdetections by the test programs.

Keywords: Buffer Overflow Attack, ROP, Shadow Stack, QEMU

1. はじめに

個人情報の流出やシステムの破壊など，企業や個人に対するサイバー攻撃が問題となっている．サイバー攻撃を引き起す要因の 1 つとして，ソフトウェアの脆弱性であるバッファオーバフロー脆弱性がある．バッファオーバフロー脆弱性を悪用した攻撃は，確保したメモリ領域以上のデータを入力することによりオーバフローを発生させ，メモリ内のデータを不正に書き換える．SANS と CWE が発表している，CWE/SANS TOP 25 Most Dangerous Software Errors[1] にも，バッファオーバフローは危険な脆弱性であると記述されている．

バッファオーバフローを悪用した攻撃には，スタック内のリターンアドレスを書き換え，攻撃者の意図した不正な制御フローに処理を移すものが存在する．リターンアドレ

[1] 立命館大学
Ritsumeikan University, Kusatsu,Shiga 525-8577, Japan
[2] 名古屋工業大学
Nagoya Insitute of Technology, Nagoya, Aichi 466-8555, Japan
[a)] skubota@asl.cs.ritsumei.ac.jp

ス書換え攻撃に対する既存の対策手法には，データ実行防止 [2] やアドレス空間のランダマイズ [3] が存在する．データ実行防止は，スタックやヒープなどのデータ領域に格納されたデータを実行不可にする手法である．これにより，コード領域以外に攻撃者による命令コードを挿入し実行するコードインジェクション攻撃を防ぐことができる．アドレス空間のランダマイズ化は，ライブラリやスタック領域などのマップ位置をランダム化する手法である．ライブラリのマップ位置が実行毎に変化するため，ライブラリの命令コードを用いた攻撃を困難にする．しかし，これらの対策手法は，緩和策にすぎず，決定的な解決策ではない [4], [5].

そこで，我々は，リターンアドレス書換え攻撃を完全に検知するアーキテクチャである SAFFRON[6] を開発している．SAFFRON は，CPU により攻撃を検知するため，ユーザ空間で動作する全てのプロセスやスレッドに攻撃検知機構を提供することができる．SAFFRON は，CALL/RET 命令の対称性に着目し，リターンアドレス書換え攻撃を検知する．SAFFRON は，CALL 命令実行時にリターンアドレスを保存し，RET 命令実行時に保存したリターンアドレスとスタック上のリターンアドレスとを比較し，検査を行う．また，コンテキストスイッチによるスタックの切替えに対応するために，OS にも拡張を加えている．なお，現在はプロトタイプ実装として，ハードウェアエミュレータである QEMU[7] を用いて IA-32 アーキテクチャの CALL/RET 命令を拡張している．しかし，CALL/RET 命令の対称性が損なわれるケースが存在し，正常な処理フローに対して誤検知が発生することが明らかとなった．具体的には，シグナルハンドラ，動的リンカの遅延バインド，zero length call，大域ジャンプ，C++の例外処理の 5 つを確認している．そこで，本論文では，SAFFRON での RET 命令実行時の比較処理を変更し，さらに，CPU では対処できない誤検知は Linux カーネルおよび標準 C ライブラリを新たに改変することで誤検知への対処を行った．シグナルハンドラおよび動的リンカの遅延バインドには Linux カーネルと標準 C ライブラリを改変し，zero length call および大域ジャンプにはリターンアドレスの比較処理をスタック全体の比較からスタックのトップのみの比較に変更した．C++の例外処理は，GCC をビルドする際のオプションを追加することで対処した．以下，本論文では，2 章で SAFFRON について述べ，3 章で CALL/RET 命令の対称性が損なわれるケースへの対処について述べる．その後，4 章で評価について述べ，5 章で関連研究について述べる．

2. SAFFRON

我々は，CPU でのリターンアドレス書換え攻撃を検知する SAFFRON[6] を開発している．本章では，SAFFRON の基本設計および課題について述べる．

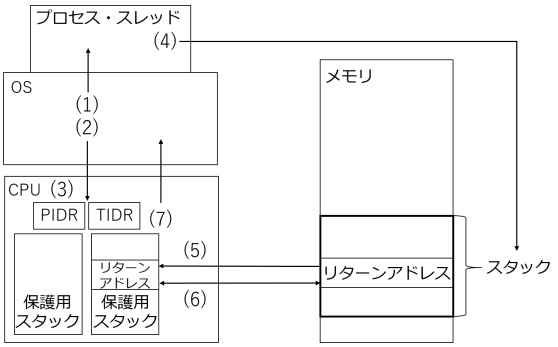

図 1 SAFFRON の基本設計

2.1 基本設計

SAFFRON は基本設計として，CALL 命令と RET 命令を拡張することでリターンアドレス書換え攻撃を検知する．具体的には，CALL 命令実行時にリターンアドレスを保存し，RET 命令実行時に保存したリターンアドレスが書換えられていないか検査する．このように，CPU でリターンアドレスを保存・比較することで，実際に実行される命令を用いるため，リターンアドレスの改ざんを完全に検知することができる．本論文では，リターンアドレスの保存先を保護用スタックと呼ぶ．保護用スタックはプロセスやスレッド毎に用意される．SAFFRON の基本設計を図 1 に示す．なお，図中の番号は処理手順であり，以下にその詳細を述べる．

(1) 新たにプロセスやスレッドが実行される際，それに対して OS はスタックを割り当てる．
(2) OS は，プロセスとスレッドの識別子を格納するレジスタである PIDR(Procese ID Register) と TIDR(Thread ID Register) に，新たに追加した特権命令を用いて値を格納する．これにより，CPU は保護用スタックを一意に決定できる．
(3) PIDR と TIDR で指定されたスレッド用の保護用スタックがない場合は，新たに保護用スタックを生成する．
(4) スレッドによる関数呼び出し時には，これまで通りにスタックにリターンアドレスが保存される．
(5) CPU は，さらに (4) のリターンアドレスとスタックポインタの値を保護用スタックに保存する．
(6) CPU は，RET 命令実行時にスタックと保護用スタックのリターンアドレスとスタックポインタの値を比較する．
(7) CPU は，(6) の比較処理でリターンアドレスが一致しなかった場合，例外を発行する．例外を受けた OS は，スレッドの停止など適切な処理を行うことで，攻撃を防ぐことができる．

SAFFRON が保護用スタックに保存するデータは，リターンアドレスとスタックポインタの値である．リターンアドレスとスタックポインタの値は，現在は実装の簡単化

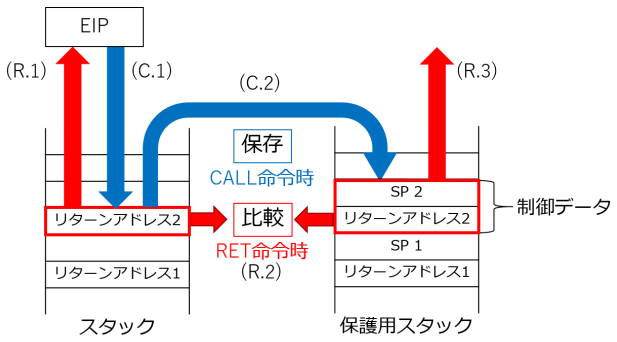

図 2 リターンアドレスの保存・比較処理

のため，QEMU 内で確保したメモリ上に保存している．そのため，ゲスト OS からは参照・操作できない．本論文では，これら 2 つのデータを合わせて制御データと呼ぶ．

2.2 リターンアドレスの保存・比較処理

SAFFRON は，CALL/RET 命令を拡張することでリターンアドレスの保存と比較を行っている．本節では，SAFFRON による CALL/RET 命令の処理手順について図 2 を用いて説明する．図 2 は，SAFFRON での CALL/RET 命令実行時の処理手順を示している．保護用スタックに保存したスタックポインタの値は，3 章で述べる CALL/RET 命令の対称性が損なわれるケースへの対処に使用する．なお，手順 (C.2)(R.2)(R.3) が拡張した処理である．

CALL 命令実行時

(C.1)EIP レジスタの値をリターンアドレスとしてスタックにプッシュする．

(C.2)制御データを保護用スタックにプッシュする．

(C.3)CALL 命令のオペランドで指定された関数へ処理を移す．

RET 命令実行時

(R.1)ESP レジスタが指すスタックのリターンアドレスをポップし，EIP レジスタに格納する．

(R.2)(R.1) のリターンアドレスと保護用スタックの先頭に保存した制御データのリターンアドレスを比較する．値が一致しなかった場合攻撃と判断する．

(R.3)保護用スタックの先頭の制御データをポップする．

(R.4)オペランドがある場合は，オペランドに指定された数だけスタックのデータをポップする．

(R.5)EIP レジスタが指すアドレスに処理を移す．

これらの 3 つの処理を追加することで，スタックのリターンアドレスと保護用スタックのリターンアドレスを比較し，リターンアドレスの書換えを検知する．また，SAFFRON での RET 命令実行時の比較処理は，保護用スタックにプッシュされている全ての制御データについて行う．

2.3 コンテキストスイッチによるスタックの切替え

プロセスやスレッドが用いるスタックは，コンテキストスイッチが発生すると切り換えられる．そのため，保護用スタックも切り換える必要がある．SAFFRON では，保護用スタック切替えのため，プロセス・スレッドの識別子を格納する新たなレジスタである PIDR・TIDR を追加している．プロセス・スレッド識別子は，OS 内で使用されるプロセス ID・スレッド ID をそのまま用いることができる．また，SAFFRON は，PIDR と TIDR に値をロードするための新たな命令として LPID と LTID を追加している (表 1)．LPID と LTID をコンテキストスイッチが発生時に OS が実行することで，CPU が保護用スタックを一意に決定することができる．CPS 命令は，プロセスやスレッドの生成時に OS が実行することで，対応する保護用スタックの作成を SAFFRON に要求する．DPS 命令は，プロセスやスレッドの終了時に OS が実行することで，対応する保護用スタックの破棄を SAFFRON に要求する．RPS 命令は，プロセスを複製する際に OS が実行することで，対応する保護用スタックの複製を SAFFRON に要求する．

新たにレジスタを追加し，そのレジスタにプロセス識別子を格納する特権命令を作成したことで，SAFFRON と OS との協調により保護用スタックの切替えおよび作成・破棄を実現することができる．

2.4 課題

SAFFRON は，CALL/RET 命令の対称性に着目してリターンアドレス書換え攻撃を検知する．しかし，正常なフローで CALL/RET 命令の対称性が損なわれるケースが存在するため，誤検知が発生することがわかった．CALL/RET 命令の対称性が損なわれるケースは，以下の 5 つを確認している．

- シグナルハンドラ
- 動的リンカの遅延バインド
- zero length call
- 大域ジャンプ (setjmp/longjmp)
- C++の例外処理 (DWARF2 EH 方式)

シグナルハンドラと動的リンカの遅延バインドは，CALL 命令以外で呼び出された処理が，RET 命令で戻るという処理フローが存在する．SAFFRON は，処理が CALL 命令以外で呼び出されると，制御データを保護用スタックにプッシュすることができない．そのため，SAFFRON は，RET 命令実行時に，前回の CALL 命令でプッシュされた制御データを用いてリターンアドレスの比較処理を行ってしまう．

zero length call と大域ジャンプは，CALL 命令で呼び出された処理が，RET 命令で戻らないというフローが存在する．SAFFRON は，CALL 命令で呼び出された処理が RET 命令で戻らない場合，CALL 命令実行時にプッシュした制御データをポップすることができない．そのため，SAFFRON は次回の RET 命令実行時に，zero length call

表 1　追加した特権命令

命令	内容
LPID	EAX レジスタの内容を PIDR に格納する
LTID	EAX レジスタの内容を TIDR に格納する
CPS	EAX レジスタの値をプロセス識別子，EDX レジスタの値をスレッド識別子として保護用スタックを作成する
DPS	EAX レジスタの値をプロセス識別子，EDX レジスタの値をスレッド識別子として保護用スタックを破棄する
RPS	EAX レジスタの値を子プロセスのプロセス識別子，EDX レジスタの値を子プロセスのスレッド識別子，EBX レジスタの値を親プロセスのプロセス識別子，ECX レジスタの値を親プロセスのスレッド識別子として，親プロセスの保護用スタックを子プロセスの保護用スタックに複製する

や帯域ジャンプによりプッシュされた不要な制御データを用いてリターンアドレスの比較処理を行ってしまう．

C++の例外処理 (DWARF2 EH 方式) は，正常な処理の中に，リターンアドレスを書換えるというフローが存在する．SAFFRON は，攻撃によるリターンアドレスの書換えか，関数の正常な処理によるリターンアドレスの書換えか判断することができない．そのため，SAFFRON は C++ の例外処理 (DWARF2 EH 方式) によるリターンアドレスの書換えを，攻撃による書換えと誤検知してしまう．

3. CALL/RET 命令の対称性が損なわれるケースへの対処

2.4 節で述べた課題を解決するため，SAFFRON の RET 命令実行時におけるリターンアドレスの比較処理を変更した．さらに，ハードウェアだけでは解決できない誤検知は，Linux カーネルと標準 C ライブラリを改変することで対処した．本章では，5 つの CALL/RET 命令の対称性が損なわれるケースとその対処方法について述べる．本章で述べる Linux カーネルと標準 C ライブラリはそれぞれ，Linux-3.10.5 と uClibc-1.0.9 である．

3.1 シグナルハンドラ

シグナルは，Linux におけるプロセス・スレッド間のイベント通知機構である．シグナルハンドラは，シグナルを受信した際に実行されるハンドラであり，受信プロセスが設定することができる．シグナルハンドラの呼出しは，CALL 命令ではなくカーネルからの IRET 命令で行われるため，誤検知が発生する．

3.1.1 シグナルハンドラの処理

Linux におけるシグナルハンドラの処理を図 3 に示す．kill システムコール等により，ユーザモードからカーネルモードに遷移する (図 3 の (1))．カーネルモードにおける処理が終了し，ユーザモードに戻る際に do_notify_resume 関数が呼び出される．この関数内で，シグナルを受信しているかを判定する．シグナルを受信した場合，シグナルハンドラ呼出しのため，do_signal 関数の処理を行う．do_signal 関数は，受信したシグナルと対処するハンドラを確認する．シグナルハンドラに標準動作が設定されていた場合，do_signal 関数内でシグナルハンドラの実行を行う．標準動作以外が設定されていた場合，handle_signal 関数を呼び出す．handle_signal 関数は，シグナルハンドラ呼出しのため，setup_frame 関数を呼び出す．setup_frame 関数は，シグナルフレームの作成と，カーネルスタックに退避されているユーザプロセスのレジスタ値の書き換えを行う．これにより，IRET 命令でユーザモードに戻った際に，シグナルハンドラを呼び出す (図 3 の (2))．シグナルハンドラは，実行後に sigreturn システムコールを実行するために，__restore 関数を呼び出す (図 3 の (3))．__restore 関数は，sigreturn システムコールを実行する (図 3 の (4))．sigreturn システムコールは，処理の終了後に IRET 命令でユーザーモードのプロセスに戻る (図 3 の (5))．このとき，シグナルハンドラ終了時の RET 命令に対応付けられる CALL 命令が存在しないため，誤検知が発生する．

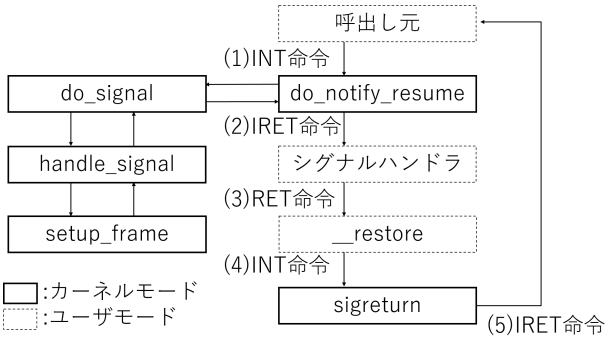

図 3　変更前のシグナルハンドラ実行フロー

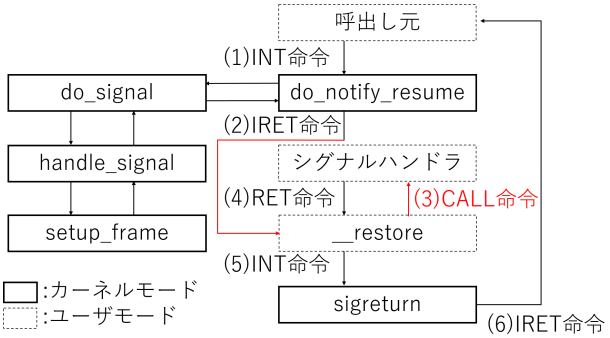

図 4　変更後のシグナルハンドラ実行フロー

3.1.2 シグナルハンドラへの対処

シグナルハンドラによる誤検知に対処するため，シグナルハンドラの実行フローを変更した．変更後のフローを図 4

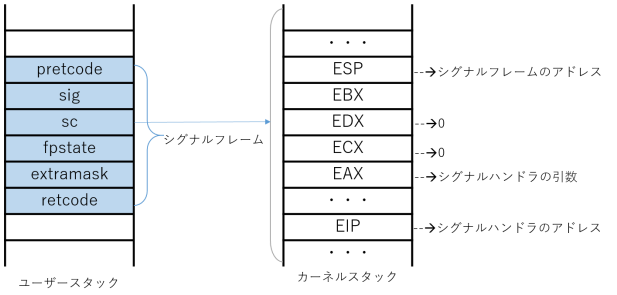

図 5　変更前のシグナルフレームとレジスタ値

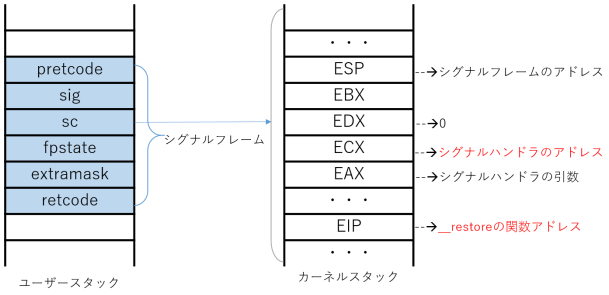

図 6　変更後のシグナルフレームとレジスタ値

に示す．do_notify_resume 関数から IRET 命令で__restore 関数を呼び出す (図 4 の (2))．__restore 関数は CALL 命令でシグナルハンドラを呼び出し (図 4 の (3))，シグナルハンドラは RET 命令で__restore 関数に戻る (図 4 の (4))．これにより，シグナルハンドラを CALL 命令で呼び出すフローに変更したため，誤検知の発生を防ぐことができる．

処理フローの変更は，Linux の setup_frame 関数と uClibc の__restore 関数を改変することで実装した．改変前の setup_frame 関数と__restore 関数を以下で説明する．

setup_frame 関数

setup_frame 関数は，シグナルハンドラを呼び出すためのレジスタ値をセットし (図 5 右)，ハンドラ内で利用するスタックフレームであるシグナルフレームをユーザスタックに作成する (図 5 左)．まず，IRET 命令を用いてシグナルハンドラを呼び出すために，ESP を後述のシグナルフレームのアドレスに書き換える．EAX はシグナルハンドラの引数，EIP はシグナルハンドラのアドレスに書き換える．次に，シグナルフレームを作成する．pretcode には__restore 関数のアドレス，sig にはシグナルハンドラに渡す引数，sc には退避されたレジスタの値を保持しているカーネルスタックのアドレスを格納する．pretcode はシグナルハンドラ実行後の戻り先アドレスにあたるため，シグナルハンドラの次に__restore 関数が呼ばれる．

__restore 関数

__restore 関数は，スタック上の不要なデータを破棄後，sigreturn システムコールを呼び出す．

setup_frame 関数と__restore 関数を改変することで，処理フローの変更を実現した．setup_frame 関数の改変は，退避したユーザプロセスのレジスタ値を変更をした．変更したレジスタ値を図 6 右に示す．ECX はシグナルハンドラのアドレス，EIP は__restore 関数のアドレスを格納するように変更を行った．__restore 関数の改変は，シグナルフレーム上の pretcode を破棄する処理の追加と，シグナルハンドラを呼び出す CALL 命令の追加を行った．これにより，シグナルフレームに新たなリターンアドレスを保存し，シグナルハンドラを呼び出す．シグナルハンドラの実行後，__restore 関数に戻った後は，改変前の__restore 関数と同様の処理を行う．

3.2　動的リンカ

動的リンカによる関数呼出し時のアドレス解決の方法は，事前バインドと遅延バインドという 2 つの方法がある．事前バインド，遅延バインドについて以下で説明する．

事前バインド

事前バインドは，プログラム起動時に動的リンカが関数のアドレスを解決する．事前バインドは，環境変数 LD_BIND_NOW を 1 とすることで用いることができる．事前バインドを用いた場合は，プログラム起動時に関数のアドレスを解決するため，誤検知は発生しない．

遅延バインド

遅延バインドは，関数のアドレス解決を関数の初回呼出し時まで遅延させる処理である．遅延バインドを用いた場合，関数呼出しは CALL 命令ではなく，RET 命令で行われるため，誤検知が発生する．

3.2.1　動的リンカの遅延バインドの処理

動的リンカの遅延バインドによる関数呼出しの処理を図 7 左に示す．関数の呼出し元は，CALL 命令により plt セクションの関数を呼び出す (図 7 左の (1))．この関数は JMP 命令により動的リンカを呼び出す (図 7 左の (2))．動的リンカは，シンボル解決ルーチンから共有ライブラリ内の関数のアドレスを得る．動的リンカは，シンボル解決ルーチンから得たアドレスにリターンアドレスを書き換え，RET 命令により関数を呼び出す (図 7 左の (3))．この関数は，RET 命令により呼出し元に戻る (図 7 左の (4))．動的リンカを用いた関数呼出しは，通常遅延バインドが用いられる．遅延バインドを用いた場合は，RET 命令により共有ライブラリ内の関数を呼び出すため，誤検知が発生する．

3.2.2　動的リンカの遅延バインドへの対処

動的リンカの遅延バインドによる誤検知に対処するため，動的リンカ利用時の関数呼出しフローを変更した．変更後の処理フローを図 7 右に示す．関数呼出しを RET 命令

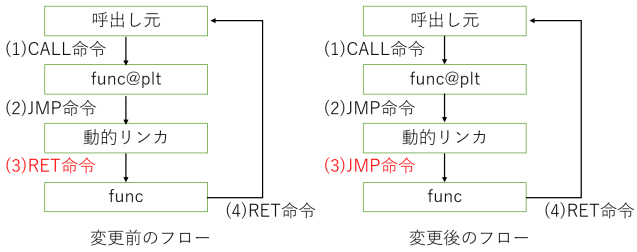

図 7 動的リンカの遅延バインドによる関数呼出しフロー

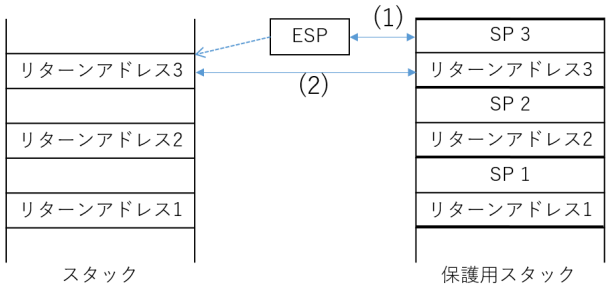

図 8 リターンアドレスの比較 ESP = SP

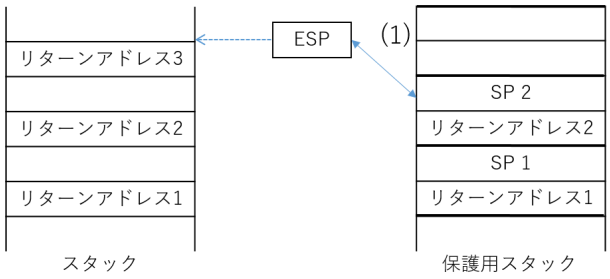

図 9 リターンアドレスの比較 ESP < SP

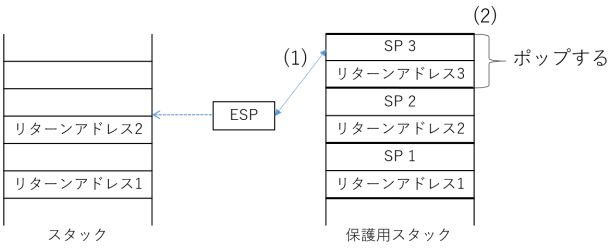

図 10 リターンアドレスの比較 ESP > SP

ではなく JMP 命令で行うように変更した (図 7 右の (3))．これにより，CALL 命令と対応付かない RET 命令が実行されないため，誤検知の発生を防ぐことができる．処理フローの変更は，uClibc の dl_linux_resolve 関数を改変することで実装した．改変の内容は，関数の最後に行う RET 命令を JMP 命令へ変更したことと，元々 RET 命令により破棄されていたスタック上の不要なデータを破棄する処理の追加である．

3.3 zero length call/大域ジャンプ

zero length call は，IA-32 アーキテクチャにおいて次に実行される命令アドレスを取得するための実装技術である．zero length call は，相対アドレッシングで 0 を指定した CALL 命令を実行する．これにより，CALL 命令の次の命令アドレスがリターンアドレスとしてスタックにプッシュされる．その後，POP 命令により命令アドレスを得て処理を終了する．zero length call は，RET 命令ではなく POP 命令により処理が終了するため，次回の RET 命令実行時に誤検知が発生する．

大域ジャンプは，主にエラー処理を目的として利用される．大域ジャンプは，C 言語では setjmp/longjmp 関数により実行される．大域ジャンプは，関数の途中で他の処理に制御を移すことが可能である．また，ジャンプ先の処理が終了した後も，大域ジャンプを実行した関数に処理を戻さない．大域ジャンプを実行した関数は，RET 命令が実行されないため，次回の RET 命令実行時に誤検知が発生する．

3.3.1 zero length call と大域ジャンプへの対処

これら 2 つの処理は，RET 命令を使用しないため，保護用スタックにプッシュされた不要な制御データを破棄することができない．そのため，これらに関しては，RET 命令でのリターンアドレスの比較方法を工夫することで対処した．比較方法の変更について以下で述べる．

RET 命令実行時に ESP と制御データのスタックポインタの値が一致した場合 (図 8 の (1))，リターンアドレスの比較を行う (図 8 の (2))．ESP より制御データのスタックポインタの値が大きい場合 (図 9 の (1))，攻撃を検知したとして例外を発生させる．これは，3.1.2 項と 3.2.2 項の対処により，正常なフローでは制御データの不足は発生しないためである．ESP より制御データのスタックポインタの値が小さい場合 (図 10 の (1))，先頭の制御データをポップする (図 10 の (2))．これにより，zero length call や大域ジャンプが実行された後，次回の RET 命令実行時に不要な制御データをポップした上で，正しくリターンアドレスの比較処理を行うことができる．

3.4 C++の例外処理

C++の例外処理は，主にエラー処理を目的として，try/catch 文により実行される．C++の例外処理は通常，オーバヘッドの少ない DWARF2 EH 方式を用いて行われる．しかし，C++の例外処理に DWARF2 EH 方式を用い

表 2　環境

項目	内容
ゲスト CPU	QEMU-SAFFRON(QEMU-1.0)
OS(カーネル)	Linux-SAFFRON(Linux-3.10.5)
コンパイラ	GCC-4.8(–enable-sjlj-exceptions)
標準 C ライブラリ	uClibc-SAFFRON(uClibc-1.0.9)
BusyBox	Version 1.24.1
Buildroot	Version 2015.11.1

ると，正常な処理としてリターンアドレスが書き換えられる．SAFFRON は，C++の例外処理によるリターンアドレスの書換えか，攻撃によるリターンアドレスの書換えか判断することができないため，誤検知が発生する．

3.4.1　C++の例外処理への対処

C++の例外処理に対処するため，DWARF2 EH 方式ではなく，setjmp/longjmp 関数を使用する方式で C++の例外処理を行う．オプションとして–enable-sjlj-exceptions を指定してコンパイルした GCC を用いることで，DWARF2 EH 方式ではなく setjmp/longjmp 関数を用いた C++の例外処理を行うことができる．setjmp/longjmp 関数は，3.3.1 項で述べた RET 命令実行時の比較処理変更により対処済みのため，誤検知は発生しない．

4.　評価

3 章で述べた CALL/RET 命令の対称性が損なわれるケースに正しく対処できているか確認するため，Linux を拡張し SAFFRON への対処を行った Linux-SAFFRON を用いて，SAFFRON の動作検証を行った．本章では，CALL/RET 命令の対称性が損なわれるケースを発生させる自作のテストプログラムを用いた動作検証の結果について述べる．

4.1　評価環境

評価に用いた環境を表 2 に示す．QEMU-SAFFRON と Linux-SAFFRON, uClibc-SAFFRON はそれぞれ, QEMU-1.0 と Linux-3.10.5，uClibc-1.0.9 に，2・3 章で述べた拡張を加えたものである．Buildroot は，Makefile とパッチから構成され，一括して組込み Linux 環境を構築するシステムである．Buildroot が構築する Linux 環境は，カーネルイメージとルートファイルシステムからなる．ルートファイルシステムは，主な標準 UNIX コマンドの機能をサポートするアプリケーションである BusyBox が含まれる．

4.2　評価内容

自作のテストプログラムを用いた評価の内容を以下に示す．

(1) シグナルハンドラの実行
　シグナルハンドラの実行フローを変更したことにより，正しい制御データがプッシュされ，誤検知が発生しないことを確認する．

(2) 動的リンカの遅延バインドで呼び出した関数の実行
　動的リンカの実行フローを変更したことにより，正しい制御データがプッシュされ，誤検知が発生しないことを確認する．

(3) setjmp/longjmp 関数の実行
　setjmp/longjmp 関数でプッシュされた不要な制御データが正しく破棄され，誤検知が発生しないことを確認する．

(4) zero length call の実行
　zero length call でプッシュされた不要な制御データが正しく破棄され，誤検知が発生しないことを確認する．

(5) SAFFRON の先行研究 [6] での評価内容の実行
　3 章で述べた対処の結果，リターンアドレス書換え攻撃の検知が可能なことを確認する．具体的には，現在のスタックフレームのリターンアドレスの書換え，1 つ前のスタックフレームのリターンアドレスの書換え，コンテキストスイッチのテストを行う．

4.3　結果

(1) を実行した結果，誤検知は発生しなかった．これにより，シグナルハンドラの実行フローを変更したことで，正しい制御データがプッシュされることを確認した．(2) を実行した結果，誤検知は発生しなかった．これにより，動的リンカの実行フローを変更したことで，正しい制御データがプッシュされることを確認した．(3)(4) を実行した結果，誤検知は発生しなかった．これにより，リターンアドレスの比較方法を変更したことで，不要な制御データが正しく破棄されることを確認した．(5) を実行した結果，全ての評価内容において，先行研究と同様に正しく攻撃検知を行えることを確認した．

5.　関連研究

5.1　これまでのバッファオーバフロー攻撃への対策

バッファオーバフロー攻撃への対策として，主にデータ実行防止 [2] やアドレス空間のランダマイズ [3] がある．データ実行防止は，スタックやヒープなどのデータ領域に格納されるデータを実行不可にする手法である．これにより，コード領域以外に命令コードを挿入し実行するコードインジェクション攻撃を防ぐことができる．しかし，この手法は，return to libc 攻撃 [4] や ROP(return-oriented programming) 攻撃 [5] など，コード領域にロードされているコードを用いる攻撃には対処することができない．アドレス空間のランダマイズ化は，ライブラリやスタック領域などのマップ先をランダム化する手法である．ライブラリのマップ先が実行毎に変化するため，return to libc 攻撃や ROP 攻撃を困難にする．しかし，この手法は，攻撃を困難にすることはできるが，完全に防ぐことはできない．

SAFFRON は，CALL/RET 命令を拡張することでリターンアドレスの書換えを検知する．そのため，return to libc 攻撃や ROP 攻撃を完全に防ぐことができる．

5.2 CET(Control-flow Enforcement Technology)

CET[8] は，Intel により開発されている，リターンアドレス書換え攻撃を検知する CPU の拡張機能である．CET は CPU により攻撃を検知するため，CET を有効化した CPU 上で動作する全てのプログラムで，バッファオーバフロー脆弱性を悪用した攻撃を検知することができる．CET は，Intel からプレビュー使用書が公開されたのみであり，考案されているという段階である．CET は，shadow stack と indirect branch tracking と呼ばれる 2 つの攻撃検知機能を提供する．CET による攻撃検知手法は，SAFFRON と類似している．

5.2.1 Shadow Stack

shadow stack は，リターンアドレスの書換えを検知する機能である．shadow stack は，ユーザ用とカーネル用に，検査用のスタックである shadow stack を作成する．CPU は，CALL 命令時に shadow stack にリターンアドレスを保存し，RET 命令時にスタックと shadow stack に保存したリターンアドレスを比較することで，攻撃を検知する．shadow stack は，同一コードセグメント内で行われる Near CALL/RET 命令だけでなく，異なるコードセグメント間で行われる Far CALL/RET 命令にも対応している．

5.2.2 Indirect Branch Tracking

間接 JMP 命令のジャンプ先アドレスは，メモリやレジスタに格納される．そのため，バッファオーバフローにより，ジャンプ先アドレスが書き換えられてしまう可能性がある．indirect branch tracking は，間接 JMP 命令のジャンプ先アドレスが書き換えられていないか検査をする．具体的な方法は，まず，コンパイラにより，ジャンプ先アドレスに ENDBR 命令を埋め込む．間接ジャンプ命令が実行された時に，ジャンプ先が ENDBR 命令でなかった場合は，例外を発生させる．

5.2.3 SAFFRON との比較

CET は，SAFFRON と同様に CALL/RET 命令の対称性を利用してリターンアドレス書換え攻撃を検知する．しかし，CET は，3 章で述べた CALL/RET 命令の対称性が損なわれるケースを考慮していない．SAFFRON は，CALL/RET 命令の対称性が損なわれれるケースに対処済みであるため，正常に動作するソフトウェアの数において優位である．

CET において，CALL/RET 命令の対称性が損なわれるケースへの対処方法を考える場合，シグナルハンドラと動的リンカへの対処は，本論文で述べたライブラリの拡張により対処可能だと考える．zero length call と setjmp/longjmp 関数への対処は，SAFFRON は，保存したスタックポインタの値を用いて行っている．CET は，スタックポインタの値を shadow stack に保存しないため，本論文で述べたものと異なる方法で行わなければならない．C++の例外処理への対処は，DWARF2 EH 方式を用いる場合は，新たな対処方法を考えなければならない．

6. おわりに

本論文では，CALL/RET 命令の対称性が損なわれるケースへの SAFFRON と OS，ライブラリの対処方法について述べた．CALL/RET 命令の対称性が損なわれるケースは，ソフトウェアの支援により対処できることを確認した．また，この対処方法は，関連研究である CET にも適応できると考えられる．

今後は，SAFFRON の開発で得た知見が，他のバッファオーバフロー攻撃への対処手法に適用できないか検討する．特に，CET は，CALL/RET 命令の対称性が損なわれるケースへの対処が必要なため，本論文で述べたソフトウェアによる対処が有効だと考える．

参考文献

[1] CWE/SANS: CWE/SANS TOP 25 Most Dangerous Software Errors, 入手先 ⟨https://www.sans.org/top25-software-errors/⟩ (2016).

[2] Team, T. P.: Homepage of The PaX Team, 入手先 ⟨http://pax.grsecurity.net/⟩ (2016).

[3] Molnar, U.: Index of /mingo/exec-shield, 入手先 ⟨http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf⟩ (2004).

[4] Shacham, H.: The geometry of innocent esh on the bone: Return-into-libc without function calls (on the x86), Proceedings of the 14th ACM conference on Computer and communications security, pp. 552-561 (2007).

[5] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., and Winandy, M.: Return-oriented programming without returns, Proceedings of the 17th ACM conference on Computer and communications security, pp. 559-572 (2010).

[6] 柴田 達也，奥野 航平，大月 勇人，瀧本 栄二，毛利 公一：QEMU を用いた命令拡張によるリターンアドレス書換え攻撃検知手法，情報処理学会研究報告コンピュータセキュリティ（CSEC），Vol.2015-CSEC-68, No.33, pp. 1-8 (2015).

[7] Bellard, F.: QEMU, a fast and portable dynamic translator, Proceedings of the annual conference on USENIX Annual Technical Conference, pp. 41-46 (2005).

[8] Intel: Control-flow Enforcement Technology Preview, 入手先 ⟨https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf⟩ (2016).