Vol.2018-DPS-174 No.30
Vol.2018-CSEC-80 No.30
2018/3/6

スタック領域へのガードページ挿入による戻りアドレス書き換え防止手法

溝内 剛[1]　上川 先之[2]　山内 利宏[2,a)]

概要：多くのオペレーティングシステムでは，メモリ領域での不正なコード実行を防止する機構を実現している．しかし，Return Oriented Programming 攻撃に代表されるコード再利用攻撃は，この仕組みを回避して攻撃コードを実行することができる．そこで，本稿では，この攻撃を防止するために，スタック領域へのガードページの挿入により，関数の戻りアドレス書き換えを防止する手法を提案する．提案手法は，ガードページを関数のローカル変数と戻りアドレスの間に配置する．スタックオーバフローにより，ガードページへのアクセスが起きると，プロセッサ例外が発生しプロセスが終了する．このため，スタックオーバフローによる戻りアドレスの書き換えを防止し，コード再利用攻撃を防止することができる．提案手法を実現する方式として，x86-64 アーキテクチャ上の Linux で動作するプログラムを対象とする方式を示す．また，改変を加えた GCC コンパイラでプログラムのソースコードをコンパイルして提案手法を適用し，評価した結果を報告する．

Tsuyoshi Mizouchi[1]　Hiroyuki Uekawa[2]　Toshihiro Yamauchi[2,a)]

1. はじめに

多くのオペレーティングシステム（以降，OS）は，メモリ領域に書き込み可能と実行可能の属性を同時に持たせないメモリ保護のポリシ (W⊕X) を採用している [1]．これにより，スタックオーバフローなどの脆弱性を用いてメモリに不正なコードを書き込み実行する攻撃の影響を緩和することができる．しかし，return-to-libc 攻撃や Return Oriented Programming (ROP) 攻撃に代表されるコード再利用攻撃によって，この機構を回避でき，攻撃コードをメモリ領域に書き込むことなく任意のコードを実行できる．

既存のスタックオーバフローを用いた攻撃への対策として，関数の戻りアドレスの書き換えを検知する手法が存在する．このような手法として，Stack Smashing Protector (SSP) [2] や Intel Memory Protection Extensions (MPX) [3] がある．SSP は canary と呼ばれる乱数を関数の戻りアドレスの前に配置し，その値の変化を確認することによって戻りアドレスの書き換えを検知する．しかし，攻撃者が canary の値を入手することで，回避されてしまう可能性がある．また，MPX はハードウェアの特別な機能を必要とする．このため，canary のような乱数を必要とせず，既存の多くのプロセッサにおいて実現可能な手法が必要である．

そこで，本稿では，スタック領域へのアクセス不可能なメモリページ挿入により，関数の戻りアドレス書き換えを防止する手法（以降，提案手法）を提案する．提案手法は，プロセスのスタック領域において，関数の戻りアドレスとローカル変数用の領域の間に，読み込み，書き込み，および実行が不可能なメモリページ（以降，ガードページ）を配置する．これにより，スタックオーバフローによる関数の戻りアドレスの書き換えを防止する．スタックオーバフローによってガードページへのアクセスが起こった場合，プロセッサ例外が発生し，プロセスが終了する．このため，canary のような乱数を必要とせず，戻りアドレスの書き換えを防止できる．また，提案手法は，改変した GCC コンパイラによってプログラムをコンパイルすることで，関数フレームの生成処理（以降，関数のプロローグ）内に，スタック領域にガードページを挿入するコードを生成する．提案手法によるコードは，ハードウェアの特別な機能を必要としない．本研究では，x86-64 アーキテクチャ上の

[1] 岡山大学 工学部
Faculty of Engineering, Okayama University
[2] 岡山大学 大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University
[a)] yamauchi@cs.okayama-u.ac.jp

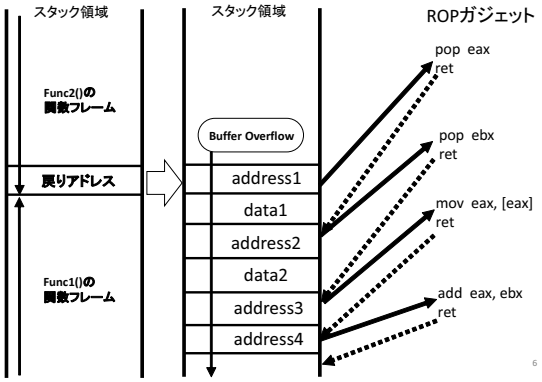

図 1　ROP 攻撃の概要図（文献 [5] より引用）

Linux で動作するプログラムにおいて提案手法を利用できることを確認した．

2. コード再利用攻撃と戻りアドレス書き換え検知手法

2.1 コード再利用攻撃

多くの OS では，メモリ領域での不正なコード実行を防止する機構を実現している．この機構は，Linux では Exec Shield，Windows では Data Execution Prevention (DEP) [4] として実現されている．この機構が有効な場合，一つのメモリ領域は書き込み可能と実行可能の属性を同時に持つことができない．このため，攻撃者はスタック領域やヒープ領域に書き込んだ攻撃コードを実行することができない．

この機能を回避して任意のコードを実行する方法として，コード再利用攻撃が存在する．コード再利用攻撃では，攻撃コードを書き込んで実行するのではなく，プロセスのアドレス空間の実行可能な領域に既に存在するコードを用いて任意のコードを実行する．ここで，利用されるコードの例としては，プログラムのテキストセグメントやそのプログラムがリンクするライブラリがある．

コード再利用攻撃の一つに，ROP 攻撃がある．ROP 攻撃の概要図を**図 1** に示す．ROP 攻撃は，攻撃者がプログラムの脆弱性を用いてスタック上の戻りアドレスを書き換え，リターン時にプログラム中の任意の場所のコードを実行することを基本とする．ROP 攻撃は複数のガジェットを連鎖させて構成される [5]．各ガジェットは，単純な操作を行う数個の命令と分岐命令（`ret` や `jmp`）から成る．この分岐命令により，次のガジェットへ処理をつなげる．攻撃者は，プログラム中からガジェットを見つけ，これらのガジェットをうまく連鎖させることで任意のコードを実行する．

2.2 既存の戻りアドレス書き換え検知手法

2.2.1 Intel MPX (Memory Protection Extensions)

Intel MPX とは x86 アーキテクチャの命令拡張である．各メモリアクセスが上限アドレスから下限アドレスの境界内に収まっているかチェックすることでメモリアクセスの整合性を検知する [6]．この機能を実現するために，メモリアクセスの境界チェックを支援する命令とレジスタを追加している．メモリアクセスが境界レジスタに格納された境界情報のメモリ境界を越えていた場合，例外が発生しメモリアクセスの整合性を検知できる．このコード挿入はプログラムのコンパイル時に行われる．MPX は Intel の Skylake（2015 年出荷）以降のプロセッサ [7] でサポートされている．また，メモリアクセスの境界チェックを支援する命令は GCC コンパイラのバージョン 5.0 からサポートされている [8]．

2.2.2 SSP (Stack Smashing Protector)

SSP は，GCC コンパイラに実装されたメモリ破損攻撃への対策技術の一つである．関数呼び出しの際にスタック領域内の変数とフレームポインタの間に canary と呼ばれるランダムな値を挿入する [2]．関数終了時に canary の値の書き換えの有無をチェックすることでスタックオーバフローを検知することができる．この対策技術は適用対象のプログラムのコンパイル時に関数の先頭に canary を挿入する命令を挿入し，関数の末尾に canary の値が元の値から書き換わっていないかチェックする命令を挿入することで実現される．SSP は GCC コンパイラのバージョン 4.1 からデフォルトで適用されている．

2.3 既存手法の課題

MPX の課題として，それぞれの機能を実現するプロセッサのサポートを必要とすることが挙げられる．MPX は 2.2.1 項で述べたように Intel の Skylake 以降のプロセッサでしかサポートされていない．このため，Intel の Skylake より前のプロセッサを使用するユーザは MPX を利用することができない．

SSP の課題として，canary の値を攻撃者に知られてしまう可能性があることが挙げられる．canary の値を得ることができた場合，SSP のスタックオーバフロー検知の仕組みを回避することができる [9] [10]．まず，攻撃者は正しい canary の値が入ったバッファを用意する．そのバッファを用いてスタックオーバフローを起こす際，canary の格納されたメモリ領域に，バッファの中の canary の値を書き込むようにする．この場合，メモリの中の canary の値は変化しない．このため，SSP はスタックオーバフローが起きても検知することができない．

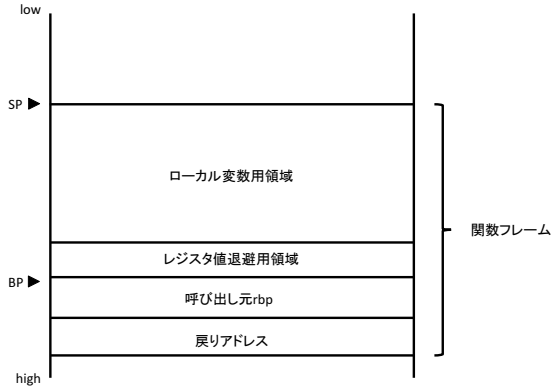

図 2　スタック上の関数フレームの構造

3. スタック領域へのガードページ挿入による戻りアドレス書き換え防止手法の設計

3.1 目的

提案手法は，2.1 節で述べた戻りアドレスの書き換えを必要とするコード再利用攻撃を防止することを目的とする．また，2.3 節で述べた既存手法の課題に対処する．

前述の目的を達成するために，以下の要件が提案手法に求められる．

(要件 1) スタックオーバフローによる戻りアドレスの書き換えを防止可能

(要件 2) 多くの環境で実現するために，プロセッサの特別な機能を必要としない

3.2 考え方

本研究では，デスクトップやサーバ向けに広く普及している x86-64 アーキテクチャ上の Linux バージョン 4.4.0 で動くプログラムを提案手法の適用対象とし，提案手法の適用のために GCC コンパイラのバージョン 7.2.0 を改変する．以降で設計と実現方式を述べる．

コード再利用攻撃は関数フレームのローカル変数を用いてスタックオーバフローを起こし関数の戻りアドレスの書き換えを行う．スタック上の関数フレームの構造を**図 2** に示す．スタック上の関数フレームでは，アドレスの高位から低位にむけて，戻りアドレス，呼び出し元関数のベースポインタ (`rbp`)，保存するレジスタの値，およびローカル変数の順で積まれる．図 2 の構造の関数フレームにおいて，ローカル変数用領域でスタックオーバフローが起きた場合，戻りアドレスを書き換え，攻撃を実行できる．

スタックオーバフローが起きた際，関数の戻りアドレスの書き換えが起こらないようにすることでコード再利用攻撃を防止できる．提案手法では，関数フレーム内のローカル変数用の領域と戻りアドレスの間にアクセス不可能な領域を作ることで，ローカル変数用領域からスタックオーバフローによる戻りアドレスの書き換えを防止する．ここで，アクセス不可能な領域としてガードページを利用する．ガードページに書き込みが行われようとした場合，プロセッサ例外が発生し，プロセスを終了させる．このガードページをスタック領域のローカル変数用領域と戻りアドレスの間に配置することで，ローカル変数用領域からスタックオーバフローによる戻りアドレスの書き換えを防止できる．

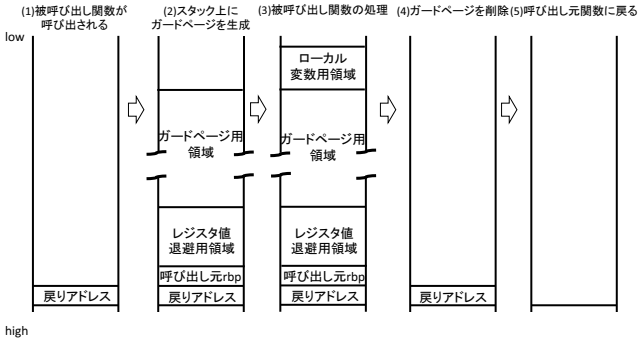

図 3　提案手法の処理流れ

3.3 基本方式

提案手法では，関数フレームの中にガードページを配置する．このため，関数のプロローグの中にガードページを生成する処理を追加する必要がある．また，関数の処理が終わり，関数フレームが解放された際，ガードページを削除していないとプロセスがガードページにアクセスし，プロセッサ例外が発生する可能性がある．このため，関数フレームの解放処理（以降，関数のエピローグ）の中にガードページを削除する処理を追加する必要がある．以上のことから，提案手法では関数のプロローグにガードページの生成処理，関数のエピローグにガードページを削除する処理を追加する．提案手法の処理流れを**図 3** に示し，以下で説明する．

(1) 呼び出し元関数の戻りアドレスがスタックに積まれ，被呼び出し関数が呼び出される
(2) ローカル変数用領域がスタックに確保される前に，スタック上にガードページを生成し，ローカル変数用領域を確保する
(3) 被呼び出し関数の処理が行われる
(4) 被呼び出し関数の処理が終わると，ガードページを削除する
(5) 呼び出し元関数に戻る

4. 実現

4.1 実現方式

提案手法を実現するために，GCC コンパイラのコード生成部にガードページを操作するコードの生成処理を追加する．関数のプロローグと関数のエピローグに追加する処理の概要を以下に示す．

(1) **関数のプロローグに追加する処理**

ガードページを生成する方法として，システムコールの一つである mprotect を用いる．mprotect システムコールは引数で指定された範囲のページの保護属性（書き込み，読み込み，および実行の可否）を変更する [11]．mprotect システムコールを用いてページの保護属性を，読み込み，書き込み，および実行が不可能なものに変更することで，ガードページを生成することができる．また，mprotect システムコールは引数に保護属性を変更するページの先頭アドレスを必要とする．このため，ガードページにするページの先頭アドレスを計算し mprotect システムコールの引数として渡す必要がある．以上のことから，提案手法では関数のプロローグに，ガードページの先頭アドレスの計算，および mprotect システムコールを用いてガードページを生成する処理を追加する．

(2) 関数のエピローグに追加する処理

mprotect システムコールを用い，ガードページの保護属性を，読み込み，書き込み可能に変更することでガードページを削除する．引数として必要なガードページの先頭アドレスは，関数のプロローグで計算したものを使うため再度計算する必要はない．以上のことから，提案手法では，関数のエピローグに，mprotect システムコールを用いてガードページを削除する処理を追加する．

4.2 実現の課題

4.2.1 ローカル変数のアドレス計算

コンパイル時にベースポインタからローカル変数までのオフセットを計算するためには，ガードページ用領域のサイズを考慮して適切に計算する必要がある．これは，提案手法によって関数フレームにガードページを挿入することでベースポインタからローカル変数までのオフセットが変化するためである．

図 3 の（3）のように，レジスタ値退避用領域とローカル変数用領域の間にガードページを挿入する場合を考える．この場合，レジスタ値退避用領域より低位のアドレス側のページのうち，レジスタ値退避用領域に最も近いページをガードページに変更する．ローカル変数用の領域は生成したガードページより，低位のアドレス側に確保される．GCC では，スタックのローカル変数を参照する際，コード生成時に計算したベースポインタからのオフセットを用いて，ローカル変数のアドレスを計算する．このため，関数フレームにガードページ用領域を挿入した場合，ベースポインタからローカル変数までの元のオフセットに，挿入したガードページ用領域のサイズを加えた値をオフセットとし，コードを生成する必要がある．しかし，挿入したガードページ用領域には，ガードページをページ境界に合わせるための可変長のパディングが必要である．このことを考慮してオフセットの計算をする必要がある．

4.2.2 mprotect システムコール呼び出しによって使用するレジスタ値の退避

提案手法でガードページの先頭アドレスを計算，および mprotect システムコールへの引数渡しを行う際，レジスタを使用する．しかし，使用するレジスタには，手続き呼び出し規約により，関数呼び出しの直後と ret 命令の直前で値が保持されている必要があるもの，呼び出し元関数からの引数が入っているため関数のプロローグで書き換えてはいけないものがある．このため，レジスタを使用する前に，レジスタの元の値を退避させ，使い終わった後値を復元する必要がある．

4.2.3 ガードページを挿入する関数

提案手法では関数フレームの中にガードページを挿入する．x86-64 アーキテクチャでのページサイズは最小で 4,096 B であるため，ガードページを挿入した関数フレーム一つの仮想メモリ上のサイズは，最小で 4,096 B となる．このため，提案手法を適用したプロセスは，提案手法を適用しないプロセスに比べ，大きなメモリのオーバヘッドが生じると考えられる．そこで，メモリのオーバヘッドを緩和するために，すべての関数に提案手法を適用するのではなく，ローカル変数のスタックオーバフローの起こりうる関数のみに適用する．

4.2.4 スタック領域へのページ割り当て

プロセス開始時にスタック領域に割り付けられている仮想メモリサイズはそれほど大きくはない（著者らの環境では 132 KB）．関数フレームがスタックに積まれていき，メモリの割り付けられていないスタック領域へのアクセスが発生すると，スタック領域へ新たにページが割り当てられる．新たに割り付けられるページの保護属性は，スタックの最も低位のアドレスに割り付けられているページの保護属性と等しくなる．提案手法によってスタックの最も低位のアドレスにあるページがガードページに替えられた場合，新たに割り当てられるページはガードページと等しい保護属性となる．この場合，本来ガードページにはならないはずのページがガードページとなり，そのページへのアクセスによってプロセッサ例外が発生する可能性がある．このため，スタック領域へ新たに割り当てられるページが読み込み・書き込み可能の保護属性を持つようにする必要がある．

4.3 課題への対処

4.3.1 ローカル変数のアドレス計算

ベースポインタからローカル変数までのオフセットの計算を容易にするため，ガードページ用領域のサイズを固定する．

提案手法を適用した関数でベースポインタからローカル変数のアドレスを計算するためには，ガードページ用領域

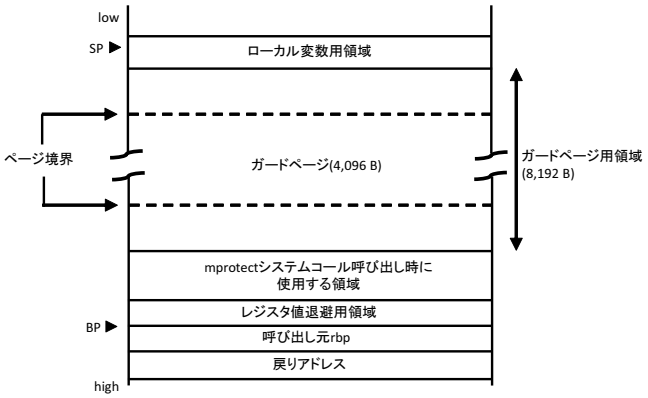

図 4 ガードページ用領域を挿入した関数フレーム

表 1 提案手法で値を退避させる必要のあるレジスタ

レジスタ名	値を退避させる理由
r15	ガードページの先頭アドレスを格納するため
rdi, rsi, rdx	mprotect システムコールの引数渡しに使用するため
rcx, r8, r9	mprotect システムコール呼び出し時に使用される可能性があるため
rax, rdx	mprotect システムコールの返り値が格納されるため

のサイズを含めたオフセットを適切に計算する必要がある．提案手法では，ガードページ用領域として 8,192 B の領域をレジスタ値退避用領域より低位のアドレス側に確保する．これにより，ガードページ用領域の中に，少なくとも 1 ページのガードページを確保できる．関数フレームに 8,192 B のガードページ用領域を確保した様子を**図 4** に示す．また，ベースポインタからローカル変数までのオフセットは，本来のオフセットにガードページ用領域のサイズ (8,192 B) を足したものになる．このオフセットの計算は常に元のオフセットに 8,192 B を足せばよいため，コンパイル時に計算し，コードを生成できる．

4.3.2 mprotect システムコール呼び出しによって使用するレジスタ値の退避

提案手法は System V AMD64 ABI の手続き呼び出し規約 [12] に則ってレジスタ値の退避を行う．提案手法で元の値を退避させるレジスタと，その値を退避させる理由を**表 1** に示す．なお，rax，rdx は呼び出し元関数への返り値を格納する可能性のあるレジスタである．しかし，提案手法は関数のエピローグで mprotect システムコールを呼び出すため，その返り値によって rax，rdx の値が書き換えられてしまう．このため，rax，rdx の元の値も退避させる必要がある．

提案手法において，関数のプロローグで値を退避させる必要があるレジスタは r15, rdi, rsi, rdx, rcx, r8, r9 である．関数のプロローグにおけるスタックの変化の様子を**図 5** に示す．以下に，提案手法を適用した関数のプロローグで行われる処理を示す．

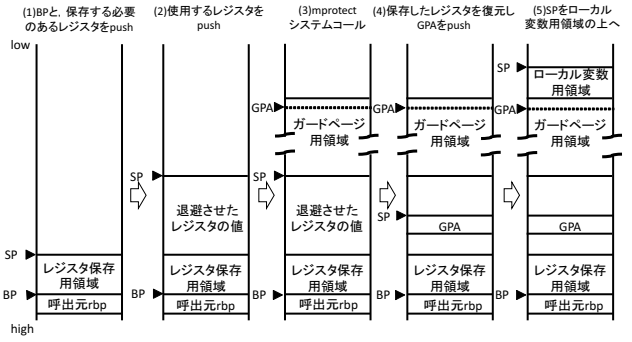

図 5 関数のプロローグにおけるスタックの変化

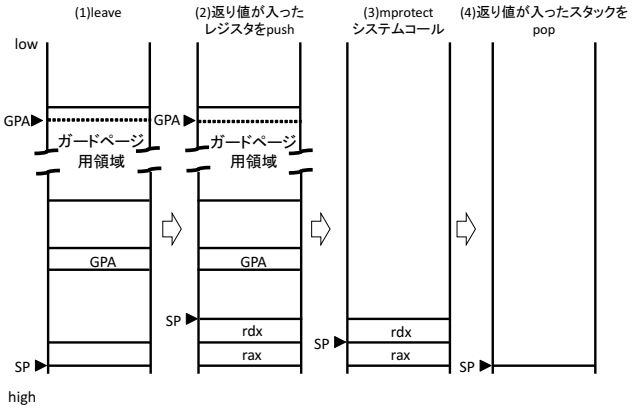

図 6 関数のエピローグにおけるスタックの変化

(1) rbp，および関数内で使用されるレジスタをスタックに push
(2) 提案手法で値を退避させるレジスタをスタックに push（追加の処理）
(3) push したレジスタを用いてガードページの先頭アドレス（以降，GPA）の計算，および引数渡しを行い，mprotect システムコールを呼び出す（追加の処理）
(4) (2) で push したレジスタの値を復元し，GPA をスタックに push（追加の処理）
(5) rsp をローカル変数用の領域の先頭アドレスへ移動

提案手法において，関数のエピローグで値を退避させる必要があるレジスタは rax, rdx である．関数のプロローグにおけるスタックの変化の様子を**図 6** に示す．以下に，提案手法を適用した関数のエピローグで行われる処理を示す．

(1) rbp を rsp にコピーし，スタックに積まれていた呼び出し元関数の rbp を pop
(2) 関数の返り値が入ったレジスタ（rax, rdx）をスタックに push（追加の処理）
(3) スタックに積んであった GPA を引数とし mprotect システムコールを呼び出す（追加の処理）
(4) スタックに積んであった関数の返り値を rax, rdx に pop（追加の処理）
(5) ret で呼び出し元関数に戻る

4.3.3 ガードページを挿入する関数

提案手法を適用する関数をローカル変数のスタックオーバフローが起こる可能性のある関数に限定する．これにより，ガードページを挿入することによるメモリのオーバヘッドを緩和できる．本研究において，スタックオーバフローが起こる可能性のある関数とは，関数フレーム内のローカル変数用の領域に配列を持つ関数とする．スタック領域でのスタックオーバフローは，関数の配列への，配列サイズを越えた書き込みによって起きるためである．GCC コンパイラでコンパイルを行う際，関数が配列を持つか否かの判定を行う．関数が配列を持つ場合，コンパイル時に提案手法を実現するコードを追加して生成する．配列を持たない場合はコードの追加を行わない．

4.3.4 スタック領域へのページ割り当て

スタック領域へ新たに割り当てられるページがガードページと同じ保護属性を持つことが無いようにする必要がある．`mprotect` システムコールによってガードページを生成する前に，ガードページにする領域より 1 ページ分低位のアドレスにアクセスを行う．`mprotect` システムコールによってガードページを生成する前であるため，スタックのトップに割り付けられているページの保護属性は，書き込み，および読み込み可能である．このため，新たに割り付けられるページの保護属性も書き込み，および読み込み可能となる．

4.4 期待される効果

提案手法によって期待される効果を以下に示す．

(効果 1) スタックオーバフローによる戻りアドレスの書き換えを防止可能

戻りアドレスの書き換えを検知してコード再利用攻撃を防止する手法の場合，canary や戻りアドレスの元の値をどこかに保持しておく必要がある．このため，その値が攻撃者に知られ，戻りアドレス書き換えの検知機構を回避されてしまう恐れがある．提案手法はスタックオーバフローによる戻りアドレスの書き換えそのものを防止できる．このため，より安全なシステムを作ることができる．

(効果 2) 既存の環境で使用可能

提案手法を導入する上で x86-64 アーキテクチャに新たな機能を追加する必要はない．このため，既存の x86-64 アーキテクチャ上で動作するプログラムで提案手法を導入できる．

4.5 他の OS やプロセッサでの適用可能性

3.1 節の（要件 2）より，提案手法は Intel 以外のプロセッサでも使用できることが望ましい．また，Linux 以外の OS でも使用可能とすることでより多くのシステムに導入できる．提案手法は，ページ単位でアクセス不可の保護属性を設定できるプロセッサで，かつページ単位の保護属性を変更可能なシステムコールが提供される OS で適用できる．

表 2 防止，検知できる攻撃の種類の比較

	MPX	SSP	提案手法
戻りアドレスの書き換え防止	○	×	○
戻りアドレスの書き換え検知	×	○	○
ローカル変数書き換え防止	○	○	×

5. 評価

5.1 既存の手法との比較

提案手法と 2.2 節で挙げた既存の戻りアドレス書き換え防止手法を定性的に比較し，提案手法と既存の手法との違いを明らかにする．評価項目を以下に示す．

(項目 1) 防止，検知できる攻撃の種類

(項目 2) 導入の要件と容易さ

（項目 1）についての比較結果を**表 2** に示す．提案手法はスタックオーバフローによる戻りアドレスの書き換えの防止，検知ができる．一方，提案手法はスタックオーバフローによるローカル変数の書き換えを防止できない．ローカル変数の書き換えは，SSP で防止できる．しかし，SSP は canary の値を知られることで，戻りアドレス書き換え検知機構を回避されてしまう恐れがある [9]．一方，提案手法は canary の値のチェックを行わず戻りアドレスの書き換え防止，検知ができるため，SSP と提案手法を併用して用いることでよりセキュアなプログラムを作ることができると推察できる．

（項目 2）について，MPX は，Intel の Skylake 以降のプロセッサ，およびサポートしているコンパイラでのソースコードのコンパイルが必要である．SSP と提案手法は，対応する GCC コンパイラでプログラムのソースコードをコンパイルする必要がある．このため，ユーザがソースコードを入手できないプログラムへの提案手法の適用は困難である．しかし，提案手法は，MPX をサポートしていない x86-64 アーキテクチャで利用できる．

5.2 評価内容

提案手法を実現するために，GCC コンパイラのコード生成部の改変を行った．提案手法を実現した GCC コンパイラ（バージョン 7.2.0，以降，GP-gcc）は，プログラムのコンパイル時に，すべての関数のコードにガードページを生成・削除する命令を追加する．この GP-gcc を用いて評価を行う．評価項目と評価の目的を以下に示す．

(評価 1) 戻りアドレス書き換え防止実験

GP-gcc でコンパイルしたプログラムにおいて，スタックオーバフローを起こし，提案手法がスタックオーバフローによる戻りアドレス書き換えを防止できるか否かを評価する．

(評価 2)　性能評価

提案手法によるコードの追加により，プログラムの実行時間やメモリ使用量に影響があると考えられる．そこで，提案手法の導入前と導入後のプログラムの実行時間とメモリ使用量を測定し，比較することにより，提案手法の導入によるプログラムの性能への影響を明らかにする．

5.3　戻りアドレス書き換え防止実験

GP-gcc でコンパイルしたプログラムにおいて，スタックオーバフローを起こし，提案手法がスタックオーバフローによる戻りアドレス書き換えを防止できるか否かを評価する．使用するプログラムの脆弱性は，GNU Binutils 2.28 の Binary File Descriptor ライブラリの bfd/ieee.c におけるスタックオーバフローの脆弱性（CVE-2017-9748 [13]）である．以下に，防止実験の手順を示す．

(1) 脆弱性のあるプログラムを，デフォルトの GCC コンパイラ（バージョン 7.2.0）および GP-gcc でビルドする
(2) ビルドしたプログラムにおいて，スタックオーバフローを発生させる
(3) ガードページへのアクセスによりプロセッサ例外が発生したか否かを確認する

実験の結果，GP-gcc でビルドしたプログラムは，スタックオーバフローによるガードページへのアクセスにより，戻りアドレスにアクセスされる前にプロセッサ例外が発生した．また，デフォルトの GCC コンパイラでビルドしたプログラムは，スタックオーバフローによる戻りアドレスへのアクセスを検知することができなかった．以上により，提案手法は，スタックオーバフローが発生した際，戻りアドレスが書き換えられる前にプロセスを終了し，攻撃を防止できることを示した．

5.4　性能評価

5.4.1　処理時間のオーバヘッド

提案手法によりガードページの生成・削除の処理を追加した関数の処理時間を計測し，オーバヘッドを算出した．本評価における評価環境を**表 3** に示す．ここで，関数の処理時間のオーバヘッドとは，提案手法によりガードページの生成・削除の処理を追加した関数と，処理を追加していない関数の処理時間の差とする．関数の処理時間の計測には，RDTSCP 命令を利用した．提案手法によりガードページの生成・削除の処理を追加した関数と，処理を追加していない関数について，その処理時間を 1,000 回計測し平均値を算出した．

測定結果を**表 4** に示す．表 4 から，提案手法によりガードページの生成・削除の処理を追加した関数呼び出し 1 回あたりのオーバヘッドは，1.07 μs であることがわかる．このことから，提案手法は，ローカル変数のスタックオーバフローが起こる可能性のある関数のみに適用した方がよいことがわかる．

表 3　性能測定における評価環境

ディストリビューション	Ubuntu 16.04.3 LTS
カーネル	Linux version 4.4.0-109-generic
CPU	Intel (R) Core (TM) i5-6500 CPU
クロック周波数	3.2GHz
使用コア数	1 コア
メモリ	4.0 GB

表 4　関数の処理時間のオーバヘッド

	処理の追加前	処理の追加後	オーバヘッド
クロック数	39	3,453	3,414
時間（μs）	0.01	1.08	1.07

表 5　コマンドの処理時間のオーバヘッド（単位: s）

コマンド名	処理の追加前	処理の追加後	オーバヘッド
`tar`	5.184	15.912	10.727
`grep`	0.005	0.351	0.346

また，Linux で使用できるコマンドのソースコードをデフォルトの GCC コンパイラと GP-gcc でビルドし，コマンドの処理時間のオーバヘッドを測定した．ここで，プログラムの呼び出しから終了までにかかった時間を 10 回測定し，その平均値の差をコマンドの処理時間のオーバヘッドとする．`tar` と `grep` コマンドの処理時間を `time` コマンドを用いて測定した．`tar` コマンドは Linux カーネルの圧縮ファイルの解凍，`grep` コマンドは 1.5 MB の C 言語のソースコードで単語検索を行った．測定結果を**表 5** に示す．表 5 から，`tar` コマンドと `grep` コマンドのどちらのコマンドでも大きなコマンドの処理時間のオーバヘッドが生じていることがわかる．これは，すべての関数でガードページの生成・削除の処理が行われるためである．ローカル変数のスタックオーバフローを起こす可能性のある関数のみへの提案手法の適用は残された課題とする．

5.4.2　関数のメモリ使用量のオーバヘッド

提案手法は，ガードページ用領域とレジスタ値の保存用領域を関数フレームの中に確保するため，その分だけ関数フレーム毎のメモリ使用量が増加する．具体的なガードページ用領域とレジスタ値の保存用領域を合わせたサイズは 8,288 B である．提案手法によりガードページの生成・削除の処理を追加した関数の仮想メモリと実メモリ使用量を計測し，オーバヘッドを算出した．ここで，関数のメモリ使用量のオーバヘッドとは，関数あたりのスタック使用量の増加量とする．関数のメモリ使用量のオーバヘッドの算出方法を以下に示す．

(1) 自身を再帰呼び出しする関数を用意し，関数フレームが 1,000 個積まれるように関数を呼び出す
(2) 関数が呼ばれる前のスタックポインタの値と，1,000

個目の関数フレームが積まれた直後のスタックポインタの値の差を求め，その値を関数フレーム 1,000 個分のサイズとする
(3) 関数フレーム 1,000 個分のサイズを 1,000 で除算（関数あたりの仮想メモリ使用量）
(4) スタック領域に割り当てられた実メモリの合計サイズを 1,000 で除算（関数あたりの実メモリ使用量）
(5) (1)，(2)，(3)，(4) を提案手法を適用した関数と適用してない関数について行い，差を算出

計測の結果，提案手法を適用していない関数フレーム 1,000 個分のサイズは 32,242 B，提案手法を適用した場合は 8,319,992 B であった．この結果から，関数の仮想メモリ使用量のオーバヘッドは 8,288 B と求まる．推察した増加量と等しいため，この結果は妥当であると考えられる．また，提案手法を適用していないプロセスの実メモリ使用量は 40 KB であり，提案手法を適用したプロセスの実メモリ使用量は 4,136 KB であった．この結果から，関数の実メモリ使用量のオーバヘッドは 4,194 B と求まる．仮想メモリ使用量のオーバヘッドと実メモリ使用量のオーバヘッドの差は，4,094 B であった．`pmap` コマンドでスタックの中身を確認すると，生成したガードページには実メモリが割り当てられていなかった．このため，関数の仮想メモリ使用量のオーバヘッドより，実メモリ使用量のオーバヘッドの方が約 1 ページ分小さくなったと推察できる．

6. おわりに

スタック領域へのガードページ挿入により，関数の戻りアドレス書き換えを防止する手法を提案し，その方式と評価結果について述べた．提案手法は，関数の戻りアドレスとローカル変数用の領域の間に，ガードページを配置し，スタックオーバフローによる関数の戻りアドレスの書き換えを防止する．

既存の戻りアドレス書き換え検知手法と提案手法の定性的な比較を行った．比較の結果，提案手法は MPX に比べより多くの x86-64 アーキテクチャで利用できることが分かった．また，SSP と併用することでよりセキュアなプログラムを作ることができると推察できる．

スタックオーバフローの脆弱性を持つプログラムに提案手法を適用し，スタックオーバフローによる戻りアドレス書き換え防止実験を行った．実験の結果，提案手法を適用することにより，スタックオーバフローによる戻りアドレス書き換えを防止できることを示した．

性能評価では処理時間のオーバヘッドと関数のメモリ使用量のオーバヘッドを計測した．処理時間のオーバヘッドは関数ごとに 1.07 μs であり，すべての関数に提案手法を適用したプログラムには大きな処理時間のオーバヘッドが生じることが分かった．関数のメモリ使用量のオーバヘッドは，仮想メモリで 8,289 B であるが，ガードページには実メモリが割り当てられないため，実メモリ使用量のオーバヘッドは 4,194 B であった．

残された課題として，提案手法の，ローカル変数のスタックオーバフローを起こす可能性のある関数のみへの適用，および Linux 以外の OS，x86-64 以外のアーキテクチャへの適用がある．

参考文献

[1] Roemer, R., Buchanan, E., Shacham, H. and Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications, ACM Transactions on Information and System Security (TISSEC), vol.15 Issue 1 Article No. 2 (2012).
[2] 菅原捷汰，渡辺亮平，近藤秀太，横山雅展，中村慈愛，須崎有康，齋藤孝道：主要な Linux ディストリビューションおよびバージョンごとのメモリ破損攻撃への対策技術の適用状況の調査と考察，コンピュータセキュリティシンポジウム 2017 論文集，vol.2017, no.2, pp.1309–1316（2017）．
[3] Intel Corporation: Introduction to Intel Memory Protection, available from ⟨`https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`⟩ (accessed 2018-1-5).
[4] Microsoft: Memory Protection Technologies, available from ⟨`https://technet.microsoft.com/en-us/library/bb457155.aspx`⟩ (accessed accessed 2018-1-9).
[5] Nicholas C. and David W.: ROP is Still Dangerous: Breaking Modern Defenses, Proc. 23rd USENIX Security Symposium, pp.385–399 (2014).
[6] 樗林秀晃，毛利公一，齋藤彰一: 動的解析にもとづく Intel MPX 命令挿入による再コンパイル不要のメモリ安全性向上手法, 情報処理学会研究報告，Vol.2017-CSEC-79 No.10, pp.1-8 (2017).
[7] FFRI: Intel Memory Protection Extension とその活用, available from ⟨`http://www.ffri.jp/assets/files/monthly_research/MR201502_Intel_Memory_Protection_Extensions_JPN.pdf`⟩ (accessed 2017-11-16).
[8] GCC Wiki: Intel MPX support in the GCC compiler, available from ⟨`https://gcc.gnu.org/wiki/Intel\%20MPX\%20support\%20in\%20the\%20GCC\%20compiler`⟩ (accessed 2018-1-24).
[9] IPA ISEC セキュア・プログラミング講座: C/C++言語編第 10 章著名な脆弱性対策, available from ⟨`https://www.ipa.go.jp/security/awareness/vendor/programmingv2/contents/c905.html`⟩ (accessed 2018-1-23).
[10] 齋藤孝道，鈴木舞音，上原崇史，金子洋平，角田佳史，馬場隆彰：メモリ破損攻撃への対策技術の調査と分類，コンピュータセキュリティシンポジウム 2014 論文集，vol.2014, no.2, pp.775–782（2014）．
[11] man7.org: mprotect(2)—Linux manual page, available from ⟨`http://man7.org/linux/man-pages/man2/mprotect.2.html`⟩ (accessed 2018-1-5).
[12] Intel: System V Application Binary Interface AMD64 Architecture Processor Supplement, available from ⟨`https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`⟩ (accessed 2017-12-12).
[13] Common Vulnerabilities and Exposures: CVE-2017-9748, available from ⟨`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9748`⟩ (accessed 2018-1-16).