情報処理学会論文誌 Vol.57 No.9 1933–1943 (Sep. 2016)

推薦論文

最新分岐記録によりROPGuard回避を防止する手法の提案

岡本 剛[1,a)] 多羅尾 光宣[1,b)]

受付日 2015年11月24日, 採録日 2016年6月2日

概要：Microsoft の EMET に採用された ROPGuard は，特定の API のリターンアドレスの前の命令が CALL 命令以外のとき，実行を防止する．しかし，CALL 命令の後に続くガジェットを API のリターンアドレスに指定する方法などにより，ROPGuard を回避できる．そこで，本研究は，64 ビット向け Windows から導入された機能を利用して最新分岐記録から API の呼び出し命令を取得し，その命令が CALL 命令（一部の JMP 命令を含む）以外のとき，実行を防止する手法を提案する．最新分岐記録を利用する従来の手法が，Intel の Nehalem 以降のプロセッサを必要とし，ガジェット連鎖が短いとき検知できないという弱点があるのに対して，提案手法は，Intel 64 および AMD64 アーキテクチャのプロセッサに対応し，短いガジェット連鎖でも検知できる点が優位である．

キーワード：ROPGuard，最新分岐記録，ROP，DEP，EMET

ROPGuard Bypass Prevention Method Using Last Branch Recording Facilities

Takeshi Okamoto[1,a)] Mitsunobu Tarao[1,b)]

Received: November 24, 2015, Accepted: June 2, 2016

Abstract: ROPGuard (integrated into Microsoft EMET) was designed to prevent return-oriented programming (ROP) attacks, utilizing evidence that the instruction preceding that at the return address of a critical API is not a CALL instruction. Since CALL-preceded gadgets allow ROP code to elude ROPGuard, we propose a new implementation of retrieving and checking last branch instructions at every critical API call, utilizing a feature of 64-bit Windows. Conventional methods using last branch recording facilities require Intel 64-bit processors later than Nehalem architecture and cannot detect ROP code using short gadget chains, while the proposed method can support both Intel 64 and AMD64 architecture processors and can detect ROP code using short gadget chains.

Keywords: ROPGuard, Last branch recording, ROP, DEP, EMET

1. はじめに

脆弱性の悪用による任意のコードの実行を防ぐために，Microsoft は，Windows XP Service Pack 2 以降に，DEP（Data Execution Prevention）を導入したが，ROP（Return-Oriented Programming）と呼ばれるテクニック [1] により，DEP が有効でも，攻撃者はガジェットと呼ばれるマシン語命令で構成された数バイトの命令列を実行できるようになった．攻撃者は，様々なガジェットを連続して呼び出すことにより，DEP の機能を無効にするか，DEP が機能しないメモリ領域を確保して，任意のコードを実行する．

ROP を検知・防止する様々な手法が提案されている．それらの手法は，アプリケーションを実行する前に行う手法（たとえば，文献 [2], [3], [4], [5]），アプリケーションが実行中に行う手法（たとえば，文献 [6], [7], [8], [9], [10], [11]），実行前と実行中の両方で行う手法 [12], [13] がある．これら

[1] 神奈川工科大学
Kanagawa Institute of Technology, Atsugi, Kanagawa 243–0292, Japan
[a)] take4@nw.kanagawa-it.ac.jp
[b)] s1685011@cce.kanagawa-it.ac.jp

本論文の内容は 2015 年 10 月のコンピュータセキュリティシンポジウム 2015 にて報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

の手法の中で，EMET（Enhanced Mitigation Experience Toolkit）[14] に採用された ROPGuard [7] がよく知られている．ROPGuard は，アプリケーションの実行中に行う手法であることから，前処理の必要がなく，透過性のあるユーザビリティを提供し，そのオーバヘッドもきわめて小さいことが分かっている．ROPGuard は，特定の API のリターンアドレスの前の命令が CALL 命令以外のとき，実行を防止する．しかし，CALL 命令の後に続くガジェットを API のリターンアドレスに指定する方法などにより，ROPGuard を回避できる．

そこで，本研究は，64 ビット向け Windows Server 2003 SP1 から導入された機能を利用して，CPU のデバッグ機能の 1 つである最新分岐記録（LBR：Last Branch Recording）から，API を呼び出した命令を取得し，その命令が CALL 命令（一部の JMP 命令を含む）以外のとき，実行を防止する手法を提案する．LBR による分岐の記録は，攻撃者により書き換えたり偽装したりできないため，ROPGuard 回避と同様の手法により検知を回避することは困難である．

LBR を利用する手法 [12], [13] が，Intel の Nehalem 以降のプロセッサを必要とし，カーネルモードで動作するデバイスドライバやカーネルモジュールを利用するのに対して，提案手法は，Intel 64 アーキテクチャと AMD64 アーキテクチャのプロセッサの両方で動作し，デバイスドライバを必要としない点が新しい．また，これらの手法は，連続して呼び出されるガジェットの個数が少ないとき検知できないという弱点があるが，提案手法は，ガジェットの個数が少なくても検知できる点が優位である．

本論文では，まず，ROPGuard と ROPGuard 回避の仕組みについて述べ，LBR を用いて ROPGuard 回避を防止する実装手法を提案する．次に，提案手法の有効性をプロトタイプシステムにより評価する．最後に，ROPecker など LBR を利用する関連研究と提案手法を比較評価し，結論を述べる．

2. ROPGuard とその回避

ROP は，スタックなどのデータ領域でコードを実行しないため，DEP により検知できない．ROP では，まず，実行したいコードを作成し，それと同じ処理が可能なガジェットを攻撃対象のプロセスから探し出す．ただし，そのガジェットの直後に必ずリターン命令（RET 命令）が必要である．RET 命令がなければ，スタックに制御が戻ってこないためである．つまり，ROP は，攻撃対象のプロセスに含まれるガジェットのアドレスとデータを，スタック（およびヒープ）に積み上げることにより，ガジェットを連続して呼び出す．この連続したガジェット呼び出しをガジェット連鎖と呼ぶ．

しかし，ROP に利用できるガジェットは攻撃対象のプロセス内に限定されるため，シェルコードのように自由度の高いコードを実行することは困難である．そのため，ROP は，DEP の制限を回避してシェルコードを実行できる領域を作る API（以下，DEP 回避の API）を呼び出すことが多い（DEP 回避の API は付録に示す）．たとえば，DEP 回避でよく使われる API が VirtualProtect である．VirtualProtect は，指定されたアドレス領域を実行可能領域に変更できる．ROP により，VirtualProtect を呼び出して，シェルコードを実行するためのサンプルコードを図 1 に示す．この例では，VirtualProtect により，シェルコードのメモリ領域を実行可能領域へ変更してから，シェルコードを実行する ROP コードを示してる．

```
 1: static unsigned char shellcode[] =
 2: "\xfc\xe8\x84\x00\x00\x00\x60\x89・・・
 3:
 4: void ShellcodeExec() {
 5:     int *ret = (int*) &ret + 2;
 6:
 7:     ret[0] = (int) VirtualProtect;
 8:     ret[1] = (int) shellcode;
 9:     ret[2] = (int) shellcode;
10:     ret[3] = sizeof(shellcode);
11:     ret[4] = PAGE_EXECUTE_READ;
12:     ret[5] = (int) &ret[6];
13: }
```

図 1 ROP コードの例
Fig. 1 Example of ROP code.

図 2 関数の呼び出し時のスタックの構成
Fig. 2 Structure of a stack at calling a function.

図 1 の 1 行目は，シェルコードを設定している．実際の攻撃では，シェルコードはスタックやヒープに展開されるが，ここでは分かりやすくするためにデータセクションに配置されるように静的配列として定義している．5 行目で，変数 ret がリターンアドレスを指すように変更している．ここで 2 を加算する理由は，図 2 に示すように，最初のローカル変数 ret がスタックの先頭に積まれており，スタックの先頭から 1 つ下には，ベースポインタアドレス（呼び出し元のスタックフレームの開始アドレス），さらにその下に，呼び出し元の関数に戻るためのリターンアドレスが配置されているためである．このリターンアドレスは，4 行目の ShellcodeExec 関数を呼び出した命令の次の

```
 1: static unsigned char shellcode[] =
 2: "\xff\xd0"  // CALL EAX
 3: "\xfc\xe8\x84\x00\x00\x00\x60\x89・・・
 4:
 5: __declspec(naked) void GadgetPopEax() {
 6:     __asm pop eax
 7:     __asm retn
 8: }
 9:
10: void ShellcodeExec() {
11:     int *ret = (int*) &ret + 2;
12:
13:     ret[0] = (int) GadgetPopEax;
14:     ret[1] = (int) VirtualProtect;
15:     ret[2] = (int) VirtualProtect;
16:     ret[3] = (int) shellcode + 2;
17:     ret[4] = (int) shellcode;
18:     ret[5] = sizeof(shellcode);
19:     ret[6] = PAGE_EXECUTE_READ;
20:     ret[7] = (int) &ret[8];
21: }
```

図 3 ROPGuard 回避の ROP コード
Fig. 3 ROP code for ROPGuard bypass.

命令のアドレスを指している．

7 行目で，このリターンアドレスの値を VirtualProtect のアドレスに置き換えている．これにより，ShellcodeExec 関数が呼び出し元に戻るときに，VirtualProtect が呼び出されるようになる．9 行目から 12 行目は，VirtualProtect の第 1 引数から第 4 引数に対応し，シェルコードのメモリ領域を実行可能領域に変更する値を指定している．VirtualProtect が RET 命令を実行したとき，8 行目に指定されたアドレスへ戻るので，8 行目にシェルコードのアドレスを指定している．これにより，VirtualProtect の処理が終了したら，シェルコードが呼び出される．

ROPGuard は，DEP 回避の API の呼び出し命令とそのオペランドの値をチェックする．通常の場合，API の呼び出し命令は，CALL 命令であり，API のリターンアドレスは，その CALL 命令の次の命令のアドレスであるが，ROP の場合，API の呼び出し命令は，RET 命令（CALL 命令以外）であり，API のリターンアドレスは，次のガジェットまたはシェルコードのアドレスである．これを手がかりにして，ROPGuard は，DEP 回避の API をフックし，フック関数のリターンアドレスの前の命令が CALL 命令であるかを確認する．このほかに，スタックピボットのチェックや，スタックに DEP 回避の API のアドレスが含まれるかどうかのチェック（RET 命令で呼び出された場合，スタックに API のアドレスが含まれる）がある．なお，スタックに API のアドレスが含まれるかどうかのチェックは，EMET には導入されていない[*1]．

ROPGuard は，DEP 回避の API が呼び出されたときに，その API のリターンアドレスの前の命令が CALL 命令であるかをチェックするので，そのリターンアドレスを次のいずれかに設定することにより，ROPGuard を回避できる．

- CALL 命令の直後にあるガジェットのアドレス
- CALL 命令の直後にあるシェルコードのアドレス（**図 3**）

たとえば，図 3 のように，2 行目でシェルコードの前にオペコードが FFD0 の CALL 命令を配置し，16 行目で CALL 命令の直後（2 バイト後）のシェルコードを指定している．13 行目と 14 行目は，CALL 命令のオペランドチェックを回避するために，ガジェットにより，EAX レジスタに VirtualProtect のアドレスをロードしている．

3. LBR により ROPGuard 回避を防止する手法とその実装方法

3.1 手法

Intel 64 アーキテクチャおよび AMD64 アーキテクチャのプロセッサには，デバッグ機能の 1 つに，LBR がある．この機能は，CPU が分岐する命令群（JMP, CALL, RET など）を実行したとき，その命令のアドレスと分岐先のアドレスを MSR（Model-Specific Register）の LastBranchFromIP レジスタと LastBranchToIP レジスタへ記録する．つまり，この機能を使えば，DEP 回避により呼び出された API の呼び出し命令が，CALL 命令であるか RET 命令であるかを正確に識別できる．ただし，KERNELBASE.DLL に含まれる DEP 回避の API は，そのほとんどが JMP 命令で呼び出されるため，JMP 命令も CALL 命令と同様に，ROP 検知をパスさせる必要がある．幸いにも，その JMP 命令は，ROP に使用される JMP 命令と異なる．ROP に使用される JMP 命令は，JMP EAX など，そのオペランドがレジスタであるのに対して（JMP 命令のオペコードが FF25 以外であるのに対して），通常の API 呼び出しに用いられる JMP 命令は，API のインポートアドレスをオペランドに指定する絶対間接 NEAR ジャンプである．具体的には，32 ビットアプリケーションのとき，JMP DWORD PTR DS:[ImportAddress] であり，64 ビットアプリケーションのとき，JMP QWORD PTR CS:[ImportAddress] であり，そのオペコードは FF25 である．したがって，提案手法では，DEP 回避の API の呼び出し命令が CALL 命令以外またはオペコードが FF25 の JMP 命令以外のとき，ROP として検知し，その実行を防止する．ただし，ROPGuard および提案手法の仕様上，DEP 回避の API の呼び出し命令が CALL 命令またはオペコードが FF25 の JMP 命令のとき，ROP コードを検知できない．

LBR を利用するには，MSR の 1 つである DebugCtl レジスタの LBR フラグ（ビット 0）を 1 にセットし，LBR を有効にする必要がある．MSR のレジスタの書き込みや読み込みには，カーネルモード（Ring 0）でのみ実行できる WRMSR 命令と RDMSR 命令を実行する必要がある．これらを実行するには，カーネルモードで動作するデバイスド

*1 これは，F-Secure 社の特許 [8] に抵触することが要因として考えられる．

ライバでの実装が必要になるため，従来の実装では，デバイスドライバやカーネルモジュールを用いるものしかない [12], [13]．デバイスドライバは，バグなどが原因で権限昇格やシステムダウンなどの致命的な脆弱性が生じるおそれがあることから，最小権限の原則 [15] に従って，ユーザモードのプロセスで可能な処理は，デバイスドライバではなくアプリケーションや Windows サービスなどで実装することが望ましい [16]．そこで，本論文では，デバイスドライバではなく，ユーザアプリケーションにより LBR を取得して，ROPGuard 回避をその LBR を用いて防止する手法を提案する．

3.2 実装方法

本研究では，Feryno がリバースエンジニアリングにより明らかにした 64 ビット向け Windows Server 2003 SP1 に導入された LBR 取得の仕組み [17] と，文献 [18] によって示された LBR 取得の方法を参考にして，デバイスドライバを利用しないで，ユーザモードのアプリケーションで LBR から API の呼び出し命令を取得して，ROP を検知する実装方法を提案する．

ここで，LBR を取得する具体的な方法について述べる．文献 [18] によれば，DebugCtl レジスタの LBR フラグと BTF フラグを有効にした状態で，EFLAGS レジスタの TF フラグを有効にして，割込みベクタ 1 のデバッグ例外を発生させれば，例外ハンドラに引き渡される EXCEPTION_POINTERS 構造体内の EXCEPTION_RECORD 構造体の ExceptionInformation[0] と ExceptionInformation[1] に MSR の LastBranchFromIP レジスタと LastBranchToIP レジスタの値がそれぞれロードされる．

64 ビット向け Windows は，DebugCtl レジスタの LBR フラグと BTF フラグを，デバッグレジスタ DR7 の LE フラグ（8 ビット目）と GE フラグ（9 ビット目）に関連づけているため，SetThreadContext によりデバッグレジスタ DR7 の 8 ビット目と 9 ビット目をセットすれば，DebugCtl レジスタの LBR フラグと BTF フラグが有効になる．この処理は，アプリケーション起動時に行うため，DllMain で実行すればよい．

割込みベクタ 1 のデバッグ例外は，ハードウェアブレークポイントまたは ICEBP 命令[*2]により発生させることができる．なお，3 つのフラグ（TF フラグ，LBR フラグ，および BTF フラグ）は，例外が発生するたびにリセットされるため，デバッグ例外を発生させる前に，3 つのフラグを有効にする必要がある．LBR フラグと BTF フラグは，デバッグ例外が発生したときに呼び出される例外ハンドラ内で有効にする．TF フラグは，有効になった直後の次の命令の実行でデバッグ例外が発生するので，例外ハンドラで有効にするのではなく，ICEBP 命令の直前に有効にする必要がある．したがって，DEP 回避の API の呼び出し命令を確かめるために，API のエントリポイントに**図 4** に示す 9 バイトからなるインラインコードを埋め込む．

```
9C      PUSHFD
58      POP     EAX
FEC4    INC     AH
50      PUSH    EAX
9D      POPFD
F1      ICEBP
FFE0    JMP     EAX
```

図 4 ICEBP 命令の埋め込みコード
Fig. 4 Inline code of the ICEBP instruction.

インラインコードの 6 バイト目までが TF フラグを有効にする処理を行っている．文献 [18] では，INC 命令ではなく，OR 命令により TF フラグを有効にしているが，OR 命令はオペランドに 4 バイトのマスクを含むため，INC 命令と比べて，3 バイトだけ長くなる．長いインラインコードは，下位の API や API 内のループ処理を破壊することがある．たとえば，64 ビット向け Windows 7 に含まれる 32 ビット KERNEL32.DLL の VirtualProtect は，API のエントリポイントから 12 バイト目に RegEnumKeyEx のサブルーチンが存在するため，そのサブルーチンを破壊する．そこで，インラインコードを短くするため，OR 命令ではなく，INC 命令を使用する．

インラインコードの 7 バイト目で，割込みベクタ 1 のデバッグ例外を発生させる．このとき，例外ハンドラが呼び出される．例外ハンドラは，LBR による ROP チェックを行い，正常な API 呼び出しであれば，インラインコードを埋め込む前のオリジナルのコードを待避した領域（トランポリン）のアドレスを，例外ハンドラに引き渡された EXCEPTION_POINTERS 構造体内の CONTEXT 構造体の EAX レジスタにロードする．これにより，例外ハンドラから元の処理に復帰したときに，インラインコードの 8 バイト目の JMP 命令により，トランポリンへジャンプできる．

先行の研究 [22] では，JMP 命令のオペランドにトランポリンへの相対アドレスを指定していたが，これには弱点があった．ROP コードが，API のエントリポイントのアドレスではなく，インラインコードの 8 バイト目を呼び出したとき，ICEBP 命令が実行されないため，ROP チェックを行えない．この弱点を修正するために，本論文では，JMP 命令のオペランドをレジスタにすることにより，ICEBP 命令が実行されない限り，トランポリンへのアドレスが EAX レジスタにロードされないため，トランポリンへジャンプできない．トランポリンのアドレスは，ASLR（Address Space Layout Randomization）により，特定することが困難になっている．また，先行のインラインコードでは，

[*2] ICEBP 命令は Intel および AMD のプロセッサマニュアル [19], [20] には記載されていない命令であるが，32 ビットマイクロプロセッサ 80386 から動作することが知られている [21]．

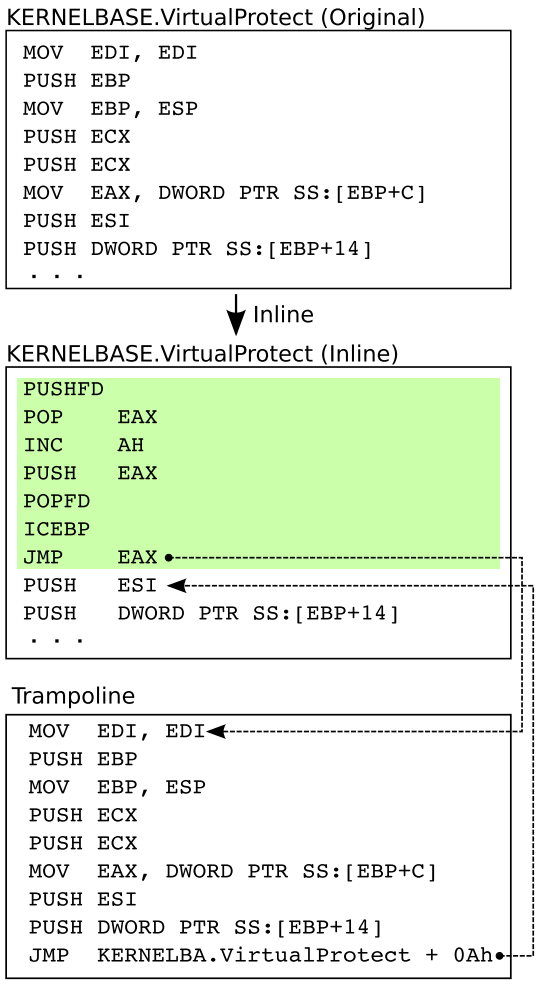

図 5 VirtualProtect の処理の流れ
Fig. 5 Execution flow of VirtualProtect.

JMP 命令が 5 バイトであったのに対して，本論文のインラインコードの JMP 命令は 2 バイトに短縮できるため，上述の例のように，64 ビット向け Windows 7 の 32 ビット KERNEL32.DLL の VirtualProtect にも適用できるようになった．ただし，インラインコードにより隠蔽されたコード（トランポリンのコード）をガジェット連鎖により実行してから，インラインコードの直後の命令にリターンさせる方法（スライディングコール）がある．スライディングコールに対する耐性とその対策は，6.2 節で詳しく考察する．

図 5 に，64 ビット向け Windows 7 の 32 ビット KERNELBASE.DLL の VirtualProtect について，オリジナルのプロローグにインラインコードを埋め込み，API が呼び出されたときの処理の流れの例を示す．

ここまでに述べた方法は 32 ビットアプリケーションの場合である．64 ビットのアプリケーションの場合は，32 ビットアプリケーションと比べて，方法が 3 つ異なる．まず，TF フラグと BTF フラグを有効にしなくても，LBR を記録できるので，上述の埋め込みコードは，ICEBP 命令と JMP 命令の合計 3 バイトになる．次に，LBR フラグは，DllMain で設定しても，システムコール NtContinue が呼び出される前にリセットされるため，NtContinue をフックして，NtContinue で LBR フラグを有効にする．LBR の値は，EXCEPTION_POINTERS 構造体内の CONTEXT 構造体の LastBranchFromRIP メンバと LastBranchToRIP メンバにロードされる[*3]．

上述の ICEBP 命令の埋め込みは，DLL インジェクションにより行う．DllMain が DLL_PROCESS_ATTACH で呼び出されたときに，DEP 回避の API に対して ICEBP 命令の埋め込みを行う．また，ICEBP 命令の実行時に発生するデバッグ例外を処理するために，ベクトル化例外処理を使う．ベクトル化例外ハンドラは，例外の発生場所にかかわらず，構造化例外ハンドラより前に呼び出せるため，確実に，ROP をチェックできる．ただし，プロセス内の他のモジュールが AddVectoredExceptionHandler により，例外発生時に最初に呼び出されるベクトル化例外ハンドラを登録されると，ROP のチェックが行えない可能性があるので，AddVectoredExceptionHandler をフックし，最初に呼び出されるベクトル化例外ハンドラが登録されようとしたとき，強制的に最後に呼び出されるベクトル化例外ハンドラとして登録する．

なお，構造化例外処理でも，最上位の未処理例外フィルタ関数を設定することにより，例外の発生場所にかかわらず，例外ハンドラを呼び出せるが，C ランタイムライブラリなどのスタートアップルーチンが自前の構造化例外ハンドラを登録し，設定した未処理例外フィルタ関数が呼び出されないことがあるため，構造化例外処理は使えない．

3.3 LBR 利用の必須要件

LBR の機能は，Intel の IA-32 アーキテクチャから存在するが，Microsoft が LBR の機能をサポートした OS は Windows 2003 SP1（Windows XP）以降の 64 ビット向け Windows である．AMD64 アーキテクチャは，64 ビット向け Windows 2003 SP1 以降のすべての Windows にサポートされているが，Intel 64 アーキテクチャは，Windows 2008 R2（Windows 7）から部分的にサポートされている．文献 [17] によれば，Windows 2008 R2 は，Core 2 ファミリ（ファミリ 06H，モデル 0FH およびモデル 17H）から初代 Core ファミリ（ファミリ 06H，モデル 1AH）にのみ対応している．これは，AMD64 アーキテクチャにおける LBR のレジスタアドレスが共通しているのに対して，Intel 64 アーキテクチャにおける LBR のレジスタアドレスがプロセッサファミリやプロセッサモデルごとに異なる場合があることが原因と考えられる．なお，Intel Core シリー

[*3] ただし，Intel 64 アーキテクチャと Windows 10（RTM）の組合せの場合は，LastBranchFromRIP の 64 ビット目が 1 になるという不具合があるので，この場合は，64 ビット目を 0 に補正する必要がある．その他の組合せでは，このような不具合はない．

ズから，LBR のレジスタの個数が 16 組と 32 組の違いがあるが，レジスタアドレスの開始アドレスは共通であり，Windows 10 は，第 2 世代の Intel Core シリーズ（Sandy Bridge）をサポートしているので，今後の新しい Intel 64 アーキテクチャでも LBR を利用できると考えられる．

LBR の機能は，いくつかの仮想環境では動作しないことが分かっている [18]．VMware 社の製品群および VirtualBox などが該当する．これらは，MSR の `DebugCtl` レジスタを仮想化していないため，LBR のレジスタにアクセスしてもつねにゼロである．一方，KVM（Kernel-Based Virtual Machine）上で動作する仮想マシンは，LBR を取得できることを確認している．したがって，LBR を利用するには，仮想環境を利用しないか，MSR の `DebugCtl` レジスタおよび LBR の仮想化に対応した仮想環境を利用する必要がある．

4. プロトタイプシステムの構成と評価環境

提案手法の有効性を評価するために，プロトタイプシステムを実装した．プロトタイプシステムは，2 つのコンポーネントから構成される．1 つは，ROPGuard 回避の防止を行う DLL である．もう 1 つは，その DLL をプロセスにインジェクションする EXE である．前者の DLL は，ROPGuard 回避の防止機能以外に，子プロセスに DLL をインジェクションするために，プロセス生成の API（`CreateProcessW` など）をフックする．

DLL インジェクションには，Microsoft Research が無償で提供する Detours Express 3.0 [23] に含まれる `DetourCreateProcessWithDllW` を利用した．Detours は，API フックのためのライブラリであるが，無償版は，IA-32 アーキテクチャのみであるため，API フックは，AMD64 アーキテクチャと IA-32 アーキテクチャの両方に対応し，MIT ライセンスで提供されている，Anka による Mhook [24] を利用した．

プロトタイプシステムによる評価環境を以下に示す．

- PC: Panasonic CF-R7DW6AJR
- CPU: Intel Core 2 Duo U7600 1.20 GHz
- Memory: DDR2-533 3 GB
- Disk: Intel 530 Series/SSDSC2BW120A4K5 120 GB
- OS: Windows 8.1 Enterprise 64 bit

なお，プロトタイプシステムは，Windows 8.1 以外にも，Windows 7 および Windows 10 でも動作することを確認している．

5. プロトタイプシステムの評価

提案手法が正しく動作することを確認するために，プロトタイプシステムで，誤検知のテスト，ROP 検知と ROPGuard 回避の検知のテスト，ベンチマークテストを行った．

5.1 誤検知テスト

提案手法が正常なアプリケーションの実行を妨害しないことを確かめるため，以下の標的にされやすいアプリケーションの動作を調べた．

- Internet Explorer 11（32/64 ビット）
  - Oracle Java 8
  - Adobe Flash Player 19
  - Silverlight 5
- Microsoft Office 2007/2010/2013（32 ビット）
- Adobe Reader XI（32 ビット）

その結果，1 つの共通の不具合が見つかった．その不具合は，メモリが割り当てられていない領域への読み込みによるアクセス違反である．その原因は，分岐命令のアドレスが未使用領域のメモリ空間を指していたためである．このアクセス違反は，特定の API の呼び出しに限定されていないうえに，アクセス違反が発生するタイミングも不定である．様々な評価や計測を行った結果，きわめて希に発生する 64 ビットプロセスにおけるアクセス違反から，分岐命令のアドレスが，カーネル空間のアドレスを指していることが分かった（32 ビットプロセスの場合，上位 32 ビットのアドレスが切り捨てられているため，そのアドレスのほとんどが未使用領域を指すことになる）[*4]．さらに，カーネル空間のアドレスを分析した結果，分岐命令は，`KiDpcInterrupt` や `KiIpiInterrupt` など合計 5 カ所の割込み処理にある `IRETQ` 命令であることが分かった．この命令は，プログラム制御を例外ハンドラまたは割込みハンドラから，例外，外部割込み，またはソフトウェア生成割込みによって中断されていたプログラムに戻す．つまり，API が呼び出されてから，例外ハンドラの `EXCEPTION_POINTERS` 構造体に LBR がロードされるまでの瞬間に，割込みが発生して，割込み処理から復帰するときに，LBR に `IRETQ` 命令のアドレスが記録されたため，API 呼び出しの分岐命令を取得できなかった．これらの一連の処理は，カーネルモードで動作する OS の内部処理のため，この問題を解決することはできない．

そこで，LBR により得られた分岐命令のアドレスがカーネル空間のアドレス（または 0）であれば，従来の ROPGuard により ROP 検知を行うことにする．幸いにも，この問題の発生確率は，おおよそ 0.00058%であることと，攻撃者がこの問題を悪用するには ROP コードにより割込みを制御すると同時に API を呼び出す必要があることから，この問題を悪用することは困難であると考えられる．

ここで，32 ビットプロセスにおいて，LBR により得られた分岐命令のアドレスがカーネル空間の `IRETQ` 命令のアドレスであるかを判断する方法を述べる．32 ビットプロセスの場合，カーネル空間の `IRETQ` 命令のアドレスは，上位 32

[*4] Windows 10 では，割込み発生時の分岐命令のアドレスは 0 にセットされる．

ビットが切り捨てられるので，その切り捨てられたアドレスが，0x0000000 から 0x7FFFFFFF の領域にあれば，分岐命令のアドレスが，カーネル空間のアドレスを指しているかを判断できない．そこで，OS が再起動するまで IRETQ 命令のアドレスは変化しないことと，そのアドレスの個数は最大で 5 つであることから，OS 起動直後に，IRETQ 命令のアドレスを取得することにする．なお，このアドレスの取得は，おおよそ 2 秒から 5 秒以内で完了する．ここで得られた 5 つのアドレスと LBR から取得したアドレスが一致したとき，カーネル空間の IRETQ 命令のアドレスであると判断し，従来の ROPGuard による ROP チェックを行うことにする．この緩和策により，上述のアプリケーションのすべてが，正しく動作することを確認した．

5.2 ROP 検知テスト

提案手法が ROP による API 呼び出しを検知できることを確かめるために，ROP コードにより API 呼び出しを行う 4 つの ROP 検知テストを行った．1 つ目は，通常の ROP コード（図 1）であり，提案手法が検知するとともに実行を阻止できることを確認した．2 つ目は，ROPGuard 回避の ROP コード（図 3）であり，EMET はこの ROP コードを検知できないが，提案手法は検知するとともに実行を阻止できることを確認した．3 つ目は，Metasploit Framework に含まれる ROP コード（java.xml）であり，提案手法がこの ROP コードを検知するとともに実行を阻止できることを確認した．最後に，標的にされやすいアプリケーションの脆弱性に対する攻撃を検知できることを確かめるために，Metasploit Framework のエクスプロイトモジュールにより，以下の脆弱性を攻撃した結果，提案手法がすべての攻撃を検知するとともに実行を阻止できることを確認した．なお，これらのエクスプロイトモジュールは，Windows 7 を対象にしていたため，これらの検知テストのみ，64 ビット向け Windows 7 Enterprise で評価を行った．

- CVE-2013-0634: Adobe Flash Player 11.5
- CVE-2013-1347: Internet Explorer 8
- CVE-2013-2551: Internet Explorer 8
- CVE-2013-3163: Internet Explorer 8
- CVE-2013-3897: Internet Explorer 8
- CVE-2014-0307: Internet Explorer 9
- CVE-2014-0497: Adobe Flash Player 11.5
- CVE-2014-1761: Microsoft Word 2010

5.3 ベンチマークテスト

提案手法のオーバヘッドを評価するために，提案手法を適用した場合と適用しない場合のベンチマークテストを比較した．ベンチマークテストには，PC 全体のパフォーマンスを計測する Futuremark の PCMark 8 Basic Edition を使用した．それぞれの場合について，3 回の計測を行い，

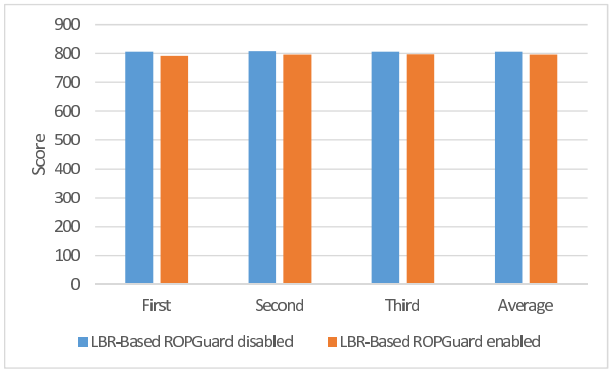

図 6 ベンチマークテストの比較
Fig. 6 Benchmark scores.

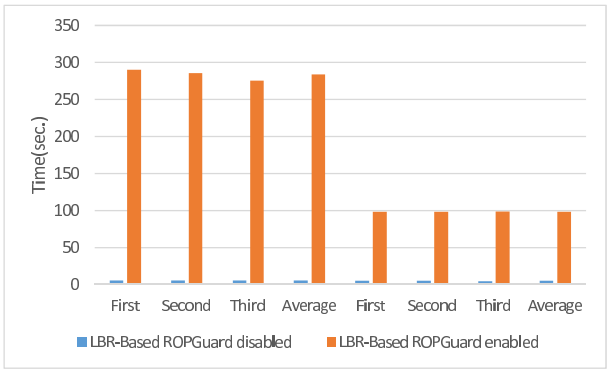

図 7 実行速度の比較
Fig. 7 Runtime speed.

各スコアとその平均値を図 6 に示す．提案手法のオーバヘッドは約 1%程度であり，ユーザエクスペリエンスにはほとんど影響がなかった．

ICEBP 命令を埋め込んだ API に限定して，たとえば，VirtualProtect を 2,000,000 回呼び出すプログラムの実行時間を調べたところ，図 7 のように，32 ビットプロセスで約 50 倍，64 ビットプロセスで約 20 倍の速度低下を計測した．つまり，DEP 回避の API をきわめて短い時間に無数に呼び出すようなアプリケーションでは，速度低下が顕著に現れる．これは，ICEBP 命令による例外処理発生時に，カーネルモードとユーザモード間のコンテキストスイッチが発生することが原因である．なお，64 ビットプロセスより，32 ビットプロセスの方が著しく遅い原因は，WOW64 によるエミュレーションによるものである．しかし，Microsoft Office など一般的によく利用されるアプリケーションで，このように高い頻度で DEP 回避の API を呼び出すことはないので，提案手法によるオーバヘッドがユーザエクスペリエンスに影響することはないと考えられる．

6. 考察

6.1 関連研究との比較

LBR を利用した ROP 検知には，kBouncer [12] と

表 1 関連研究の比較
Table 1 Comparison of related methods.

	ROPGuard	kBouncer	ROPecker	提案手法
AMD64	○	×	×	○
Intel 64	○	△	△	○
トリガ	API	API	システムコールとウインドウ外実行	API
前処理	不要	必要	必要	32 ビットのみ必要
動作モード	ユーザ	ユーザとカーネル	ユーザとカーネル	ユーザ
ROP	○	○	○	○
JOP	×	○	○	○
COP	×	○	○	×
仮想環境	○	△	△	△
オーバヘッド	1%	1%	2%	1%

ROPecker [13] がある．kBouncer と ROPecker は，「ROP により目的のコードを実行するには，長いガジェット連鎖を必要とすることが多い」という経験則に基づいて，LBR に記録された 16 組のアドレス（LBR スタック）に含まれるガジェット連鎖の長さ（連続して呼び出されるガジェットの個数）が閾値を超えたら，ROP コードが実行されていると判断する．本章では，これらの手法と提案手法および提案手法のベースとなった ROPGuard について比較評価する（**表 1**）．表 1 から，提案手法は，ROPGuard と比べて，ROPGuard 回避の ROP コードと一部を除く JOP のコードを検知できる点が優位であり，kBouncer と ROPecker と比べて，サポートする CPU アーキテクチャが多いことと，短いガジェット連鎖であっても，ROP コードと一部を除く JOP のコードを検知できる点が優位である．以下で，表 1 の各項目について詳しく考察する．

6.1.1 サポート可能な CPU

表 1 の CPU の項目（IA-32，AMD64，および Intel 64）について，○は，すべてのアーキテクチャに対応していることを表す．△は，16 組以上のアドレスを記録できる Nehalem 以降のアーキテクチャのみに対応していることを表す．×は，すべてのアーキテクチャに対応していないことを表す．CPU の項目では，提案手法と ROPGuard が優位であることが分かる．

6.1.2 検知のトリガ

トリガの項目について，API は，DEP 回避の API など API の呼び出し時にチェックを行うことを表す．システムコールは，システムコールの呼び出し時にチェックを行うことを表す．ウインドウ外実行は，スライディングウインドウと呼ばれる複数のページの領域以外で命令を実行したときに，チェックを行うことを表す．このスライディングウインドウは，実行するコード領域が変わるたびに，ウインドウをスライドさせる．ROPecker は，スライディングウインドウ領域外に実行の制御が移るたびにチェックするので，他の手法と比べて，チェックの頻度が高い．このトリガにより，API を呼び出さない ROP も検知できる可能性があるが，オーバヘッドが他の手法と比べて大きい．

6.1.3 前処理の有無

前処理は，ROPGuard を除いてすべて必要である．kBouncer と ROPecker は，LBR スタックのアドレスがガジェットであるかどうかを瞬時にチェックできるように，事前に，監視したいアプリケーションが利用するすべてのコードからガジェットを特定する必要がある．提案手法は，5.1 節で述べたとおり，32 ビットプロセスのとき，LBR により得られた分岐命令のアドレスがカーネル空間の `IRETQ` 命令のアドレスであるかを確認するために，OS 起動直後に，`IRETQ` 命令のアドレスを取得する必要がある．なお，このアドレスの取得は，おおよそ 2 秒から 5 秒以内で完了するので，ユーザエクスペリエンスに影響はないと考えられる．

6.1.4 動作モード

動作モードは，各手法が利用する動作モードを表す．提案手法と ROPGuard は，ユーザモードのアプリケーションのみであるが，kBouncer と ROPecker は，カーネルモードのデバイスドライバも必要とする．コードの安全性を考慮するなら，デバイスドライバではなくユーザアプリケーションで実装することが望ましい．

デバイスドライバは，LBR スタックのすべての値を取得できるなど特権レベル 0 の命令が利用できたり，ユーザアプリケーションからアクセスできないカーネル空間のメモリ領域を利用できたりすることから，ユーザモードのアプリケーションと比べて攻撃に耐性がある．しかし，デバイスドライバにメモリ破損の脆弱性があれば，カーネルモードでの実行を許す深刻な脆弱性となる．それゆえ，デバイスドライバでなければならない処理以外は，ユーザモードのアプリケーションで実装することが望ましい．この考え方は，最小権限の原則に基づく [15]．

ただし，ユーザモードのアプリケーションは，デフォルトの状態では，他のプロセスの介入（DLL インジェクショ

ンなど）が可能であるので，Windows のセキュリティ機能であるアクセス制御リストにより他のプロセスからの介入を禁止する必要がある．これにより，管理者権限を奪取されない限り，ユーザモードのアプリケーションであってもセキュリティ上のデメリットはないと考えられる．攻撃者が，何らかの方法により，管理者権限を取得したら，提案手法だけでなく，デバイスドライバであってもアンインストールのオプションがあれば同じく無効にできるので，ユーザアプリケーション特有のデメリットはないと考える．

6.1.5 ROP，JOP，および COP の検知

ROP，JOP（Jump-Oriented Programming）[25]，および COP（Call-Oriented Programming）[26] の項目は，○が検知できる可能性があることを表し，×が検知できる可能性がないことを表す．

kBouncer と ROPecker は，LBR スタックに含まれるガジェット連鎖の長さに基づいて検知するため，ROP だけでなく，JOP や COP も検知できる可能性がある．しかし，図 1 や図 3 のように，ガジェット連鎖が短いとき，ROP に限らず JOP や COP のコードも検知できない．ガジェット連鎖の長さに関する閾値を小さくすれば検知できるが，正常なプロセスにもガジェット連鎖が含まれ，その長さはアプリケーションや OS に依存するため，動作確認がされていないアプリケーションに対して，この閾値をいつ誰が設定するかが難しい課題である．

kBouncer はいくつかのアプリケーションを調査して閾値を 8 に設定しているが，8 以下のガジェット連鎖を用いたエクスプロイトがすでに存在している．攻撃に用いられるガジェット連鎖は，その長さが 8 以上であることが多いが，ガジェット連鎖が長くなる原因は，ASLR によりランダム化されたスタックまたはヒープのアドレス（シェルコードが含まれるアドレス）を特定するためである．つまり，ASLR を回避すれば，スタックを特定できるので，ガジェットを使わずに API を呼び出せる．たとえば，Adobe Reader X 10.1.4 に存在する脆弱性（CVE-2013-2730）に対する Metasploit Framework のエクスプロイトモジュールは，1 つの `RET` ガジェットだけで `VirtualAlloc` を呼び出す．これ以外にも，Metasploit Framework には，ASLR を回避しなくても，長さ 3 のガジェット連鎖で DEP 回避の API を呼び出すコードも存在する．このガジェット連鎖は，Adobe Reader 11.0.2 以前，10.1.6 以前，9.5.4 以前に存在する脆弱性（CVE-2013-3346）をエクスプロイトするモジュールで利用されている．このように，すでに閾値 8 を下回るエクスプロイトが存在するので，ガジェット連鎖に依存しない提案手法に優位性がある．

なお，kBouncer には，ROPGuard と同様の機能が含まれるため，短いガジェット連鎖であっても検知できるが，ROPGuard 回避の弱点を有するため，図 3 の ROP コードは検知できない．kBouncer に提案手法を組み合わせれば，提案手法が検知できない COP とオペコードが `FF25` の JOP のコードを検知できるようになると考えられる．

6.1.6 仮想環境の対応

仮想環境の項目は，○が仮想環境でも検知できることを表し，△が KVM など一部の仮想化技術でしか検知できないことを表す（3.3 節参照）．

6.1.7 オーバヘッド

オーバヘッドの項目は，性能低下の割合を表している．ROPecker はトリガの項目で述べたとおり，他の手法と比べてチェックの頻度が高いため，わずかであるがオーバヘッドが最も大きい．

6.2 スライディングコールの耐性とその対策

スライディングコールの耐性とその対策について考察する．Windows 7 以降の KERNEL32.DLL の DEP 回避 API（`SetProcessDEPPolicy`，`CreateFileMappingA`，`CreateFileMappingNumaA` を除く）は，KERNELBASE.DLL の API へジャンプするスタブしか含まれないため，そのスタブはインラインコードで破壊され，スライディングコールはできない．しかし，それ以外の API はスライディングコールが可能である．

そこで，スライディングコールを困難にするために，API のエントリから RET 命令まで連続したコード領域すべてを，「図 4 のインラインコードの `JMP` 命令の前に多数の `NOP` 命令を挿入したインラインコード」で上書きして，そのすべてのコードをトランポリンに隠蔽する方法が考えられる．たとえば，Windows 10 の場合，DEP 回避の API に含まれるチャンク（API のエントリから `RET` 命令までに含まれない，別の場所にあるコード）はたかだか 1 つであり，モジュール内の各 API は他の API と完全に独立しているので（DEP 回避の API に含まれるサブルーチンを他の API が呼び出すことがないので），チャンクを含む API のすべてのコードをトランポリンへ隠蔽できる．もし，これらのトランポリンを作成する時間がかかるようであれば，事前に，各 API にトランポリンのテンプレートを作成しておけばよい．実行時にテンプレートに対してリロケーション処理を行うだけでよいので，高速にトランポリンを作成できる．ただし，この方法は，セキュリティ更新プログラムやサービスパックの適用により，API のコードが変更されることが予想されるので，そのつど，テンプレートの作成が必要になる．

6.3 ガジェット連鎖によるシステムコール

スライディングコール以外に，命令数が 3 命令から 5 命令しか含まれないシステムコールの場合，ガジェット連鎖のみでシステムコールを実行されるおそれがある．ガジェット連鎖によるシステムコールの実行を防ぐには，Linux であれば，ROPecker と同様にカーネルモードのシステムコー

ルをフックすれば検知可能であるが，マイクロソフト社は，64 ビット版の Windows 7 以降に Kernel Patch Protection（KPP）を導入したので，カーネルモードで動作するシステムコールをフックできない．これは，kBouncer も同様である．kBouncer はカーネルモードでシステムコールをフックする設計であるが，実装は，KPP が原因で，提案手法と同様に，ユーザモードのアプリケーションでシステムコールをフックしている．ガジェット連鎖によるシステムコールを困難にするために，ASLR が回避されたことを想定して，何らかの方法により，システムコールを呼び出す `SYSENTER` 命令や `SYSCALL` 命令とそれに続く `RET` 命令のアドレスを特定できないようにする必要がある．

7. おわりに

本論文では，ROPGuard を回避する手法を，LBR を用いて防止する手法を提案した．LBR による分岐記録は，攻撃者により書き換えたり偽装したりできないため，ROPGuard 回避と同様の手法により検知を回避することは困難である．

プロトタイプシステムの評価により，正常なアプリケーションは正しく動作するとともに，ROP や ROPGuard 回避のコードを検知できることを確認した．また，実在する脆弱性に対する攻撃に使われる ROP コードも検知できることを確認した．ベンチマークテストでは，提案手法のオーバヘッドは 1%程度であることからユーザエクスペリエンスに影響がないことを示した．ただし，DEP 回避の API を短時間に無数に呼び出す場合には，顕著な速度低下を観測したことから，一部のアプリケーションでは，速度が著しく低下する可能性があることが分かった．

最後に，LBR を利用する従来の手法が，Intel の Nehalem 以降のプロセッサを必要とし，ガジェット連鎖が短いとき ROP コードを検知できないという弱点があるのに対して，提案手法は，Intel 64 アーキテクチャと AMD64 アーキテクチャのプロセッサの両方で動作し，ガジェット連鎖が短くても，COP と一部の JOP を除いたすべての ROP コードを検知できる点が優位であることを示した．

参考文献

[1] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), *Proc. 14th ACM Conference on Computer and Communications Security*, ACM (2007).

[2] Li, J. et al.: Defeating return-oriented rootkits with “Return-Less” kernels, *Proc. 5th European Conference on Computer Systems* (2010).

[3] Onarlioglu, K. et al.: G-Free: Defeating return-oriented programming through gadget-less binaries, *Proc. 26th Annual Computer Security Applications Conference* (2010).

[4] Wartell, R. et al.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code, *Proc. 2012 ACM Conference on Computer and Communications Security* (2012).

[5] Pappas, V. et al.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization, *Proc. 2012 IEEE Symposium on Security and Privacy*, pp.601–615 (2012).

[6] Chen, P. et al.: DROP: Detecting return-oriented programming malicious code, *Information Systems Security*, pp.163–177 (2009).

[7] Ivan, F.: Runtime Prevention of Return-Oriented Programming Attacks (2012), available from ⟨https://github.com/ivanfratric/ropguard⟩.

[8] Hentunen, D.: *Detecting a return-oriented programming exploit* (2010).

[9] Yao, F. et al.: JOP-alarm: Detecting jump-oriented programming-based anomalies in applications, *IEEE 31st International Conference on Computer Design* (*ICCD*), IEEE (2013).

[10] Yuan, L. et al.: Security breaches as PMU deviation: Detecting and identifying security attacks using performance counters, *Proc. 2nd Asia-Pacific Workshop on Systems*, Article No.6, ACM (2011).

[11] Wicherski, G.: *Threat Detection for Return Oriented Programming* (2014).

[12] Pappas, V. et al.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, *USENIX Security* (2013).

[13] Cheng, Y. et al.: ROPecker: *A generic and practical approach for defending against ROP attack* (2014).

[14] Microsoft: Enhanced Mitigation Experience Toolkit, available from ⟨http://technet.microsoft.com/ja-jp/security/jj653751⟩.

[15] Saltzer, J.H. and Schroeder, M.D.: The protection of information in computer systems, *Proc. IEEE*, Vol.63, No.9, pp.1278–1308 (1975).

[16] Koret, J.: Breaking Antivirus Software, Symposium on Security for Asia Network (SyScan) (2014), available from ⟨http://joxeankoret.com/download/breaking_av_software_44con.pdf⟩.

[17] Feryno: Enabling Debug Control for newer Intel CPUs at win server 2008 R2 x64 / windows 7 x64 (2013), available from ⟨http://fdbg.x86asm.net/⟩.

[18] nick.p.everdox: Last branch records and branch tracing, CodeProject (2013), available from ⟨http://www.codeproject.com/Articles/517466/Last-branch-records-and-branch-tracing⟩.

[19] Advanced Micro Devices: *AMD64 Architecture Programmer’s Manual*, Volume 2: System Programming (2013).

[20] Intel: *Intel 64 and IA-32 Architectures Software Developer’s Manual*, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C (2015).

[21] Collins, R.: Undocumented OpCodes: ICEBP, available from ⟨http://www.rcollins.org/secrets/opcodes/ICEBP.html⟩.

[22] 多羅尾光宣, 岡本　剛：最新分岐記録を用いた ROPGuard 回避防止手法，コンピュータセキュリティシンポジウム 2015，pp.847–854 (2015).

[23] Hunt, G. et al.: Detours: Binary Interception of Win32 Functions, *Proc. 3rd USENIX Windows NT Symposium*, USENIX (1999).

[24] Anka, M.: MHOOK, AN API HOOKING LIBRARY, V 2.4 (2014), available from ⟨http://codefromthe70s.org/mhook24.aspx⟩.

[25] Checkoway, S. et al.: Return-oriented programming

without returns, Proc. 17th ACM conference on Computer and communications security, ACM (2010).
[26] Goktas, E. et al.: Out of control: Overcoming control-flow integrity, *2014 IEEE Symposium on Security and Privacy* (*SP*), IEEE (2014).

付　　録

DEP 回避に利用される可能性がある API は以下のとおりである．

- KERNEL32.DLL
  - `VirtualAlloc`
  - `VirtualAllocEx`
  - `VirtualAllocExNuma`
  - `VirtualProtect`
  - `VirtualProtectEx`
  - `HeapCreate`
  - `SetProcessDEPPolicy`
  - `CreateFileMappingA`
  - `CreateFileMappingNumaA`
  - `CreateFileMappingW`
  - `CreateFileMappingNumaW`
  - `WriteProcessMemory`
- KERNELBASE.DLL(Windows 7 以降)
  - `VirtualAlloc`
  - `VirtualAllocEx`
  - `VirtualAllocExNuma`
  - `VirtualProtect`
  - `VirtualProtectEx`
  - `HeapCreate`
  - `SetProcessMitigationPolicy`(Windows 8 以降)
  - `CreateFileMappingW`
  - `CreateFileMappingNumaW`
  - `WriteProcessMemory`
- NTDLL.DLL
  - `NtProtectVirtualMemory`
  - `RtlCreateHeap`
  - `NtSetInformationProcess`
  - `NtCreateSection`
  - `NtAllocateVirtualMemory`

推薦文

本論文の取り扱っているテーマはいままさに実際に社会的問題となっており，対策技術研究の必要性が高い．また，本研究は，先行研究による検知を回避する攻撃手法を検知可能な新たな手法を提案しており，その内容の信頼性や有用性が高いため，推薦論文として推薦したい．

（コンピュータセキュリティシンポジウム 2015
プログラム委員長　山内利宏）

岡本 剛　（正会員）

1973 年生．2002 年豊橋技術科学大学大学院博士後期課程修了．博士（工学）．同年神奈川工科大学工学部情報ネットワーク工学科助手．2005 年同大学専任講師．2010 年同大学准教授．情報セキュリティおよび知能情報処理の研究に従事．電子情報通信学会会員．

多羅尾 光宣

1993 年生．2016 年神奈川工科大学情報学部卒業．学士．同年同大学大学院工学研究科情報工学専攻博士前期課程入学．情報セキュリティの研究に従事．