# 最新分岐記録を用いたROPGuard回避防止手法

多羅尾 光宣　　岡本 剛

神奈川工科大学
243-0292 神奈川県厚木市下荻野 1030
s1222002@cce.kanagawa-it.ac.jp　　take4@nw.kanagawa-it.ac.jp

**あらまし** 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Windowsの機能により最新分岐記録を取得し，その最新分岐記録からAPIの呼び出し命令をチェックしROPの実行を防止する実装を提案する．最新分岐記録を利用する従来の手法が，IntelのNehalem以降のプロセッサーとデバイスドライバを必要とするのに対して，提案手法は，Intel 64およびAMD64アーキテクチャのプロセッサーに対応し，デバイスドライバを必要としない．

# ROPGuard Bypass Prevention Method using Last Branch Recording Facilities

Mitsunobu Tarao　　Takeshi Okamoto

Kanagawa Institute of Technology.
1030 Shimo-ogino, Atsugi, Kanagawa 243-0292, JAPAN
s1222002@cce.kanagawa-it.ac.jp　　take4@nw.kanagawa-it.ac.jp

**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 device drivers, while the proposed method supports both Intel 64 and AMD64 architecture processors and requires no device drivers.

## 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 Programing）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，このガジェットを組み合わせることにより，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[2-5], アプリケーションが実行中に行う手法[6-11], 実行前と実行中の両方で行う手法[12-13]がある. これらの手法の中で, 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, 事前の準備の必要がなく, 透過性のあるユーザビリティを提供し, そのオーバーヘッドも極めて小さいことがわかっている. 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[12]. そこで, 本研究は, 64 ビット向け Windows から導入された機能から最新分岐記録(LBR:Last Branch Recording)を取得し, LBR に記録されたアドレスの分岐命令をチェックし ROP の実行を防止する実装手法を提案する. LBR から API の呼び出しに実行された正しい命令を取得できるので, CALL 命令の後に続くガジェットにより ROPGuard を回避しようとしても, ROP を検知できる.

LBR を利用する従来の手法が, Intel の Nehalem 以降のプロセッサーを必要とし, 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ーに対応し, デバイスドライバを必要としない点が新しい.

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方法を提案する. 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にについて考察する.

## 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 の制限を受けない. 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の実行可能領域から探し出す. 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. 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. 例えば, ROP 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のアクセス保護を実行可能状態に変更できる. ROP により, VirtualProtect を呼び出して, シェルコードを実行する ROP コードを図 1 に示す.

```
unsigned char shellcode[] =
"¥xfc¥xe8¥x84¥x00¥x00¥x00¥x60¥x89・・・

void ShellcodeExec() {
        int *ret = (int*) &ret + 2;

        ret[0] = (int) VirtualProtect;
        ret[1] = (int) shellcode;
        ret[2] = (int) shellcode;
        ret[3] = sizeof(shellcode);
        ret[4] = PAGE_EXECUTE_READ;
        ret[5] = (int) &ret[6];
}
```

**図 1 ROP コードの例**

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．この他に，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．

ROPGuard は，DEP 回避の API が呼び出された時に，リターンアドレスの前の命令をチェックするが，攻撃者はリターンアドレスを操作できる．つまり，リターンアドレスに指定するガジェットまたはシェルコードの前に CALL 命令を配置すれば，ROPGuard を回避できる[12]．例えば，図 2 の ROP は，ROPGuard で検知できない．

```
unsigned char shellcode[] =
"¥xff¥xd0"    // CALL EAX
"¥xfc¥xe8¥x84¥x00¥x00¥x00¥x60¥x89・・・

__declspec(naked) void GadgetPopEax() {
        __asm pop eax
        __asm retn
}

void ShellcodeExec() {
        int *ret = (int*) &ret + 2;

        ret[0] = (int) GadgetPopEax;
        ret[1] = (int) VirtualProtect;
        ret[2] = (int) VirtualProtect;
        ret[3] = (int) shellcode + 2;
        ret[4] = (int) shellcode;
        ret[5] = sizeof(shellcode);
        ret[6] = PAGE_EXECUTE_READ;
        ret[7] = (int) &ret[8];
}
```

**図 2 ROPGuard 回避の ROP コード**

## 3 LBR を用いた ROPGuard 回避の防止

### 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ーには，デバッグ機能の 1 つに，LBR がある[15-16]．この機能は，命令ポインタが分岐する命令群（JMP，CALL，RET など）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 回避により呼びだされた 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 や 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，通常，LBR の機能は無効であるので，まず，MSR の 1 つである Debug Ctl 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）でのみ実行できる特権命令 wrmsr と rdmsr 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ライバでの実装が必要になるが，デバイスドライバは，64 ビット向け Windows では，デバイスドライバにデジタル署名が必要になるため，ユーザモードのアプリケーションに比べて，開発と導入のコストが小さくない．また，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].

### 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 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, 64 ビット向け Windows から導入された機能を利用して, ユーザモードのプロセスだけで, LBR を取得する. 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, 割り込みベクタ 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 CONTEXT 構造体の ExceptionInformation[0]と ExceptionInformation[1]に LastBranchFromIP レジスタと LastBranchToIP レジスタの値がそれぞれロードされる. LBR フラグは, デバッグレジスタ DR7 の LE フラグ(8 ビット目)を有効にすれば, 有効になる. 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 命令により, 発生させられる. なお, TF フラグと LBR フラグは, 例外が発生するたびに, リセットされるため, 必ず, デバッグ例外を発生させる前に, TF フラグと LBR フラグを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 3 に示す 12 バイトからなるコードを埋め込む.

```
9C      PUSHFD
58      POP  EAX
FEC4    INC  AH
50      PUSH  EAX
9D      POPFD
F1      ICEBP
E9      JMP  Trampoline
```

**図 3 ICEBP の埋め込みコード**

Trampoline は上述のコードを埋め込む前のコードを退避させたアドレスであり, Trampoline のコードの最後は, API のエントリポイントに埋め込んだ Trampoline への JMP 命令の次の命令へジャンプする JMP 命令で終わる. これらの一連の処理の流れを図 4 に示す. ここでは, 64 ビット向け Windows 8.1 Enterprise における 32 ビット用の KERNELBASE.DLL の VirtualProtect に ICEBP を埋め込んだ例を示している.

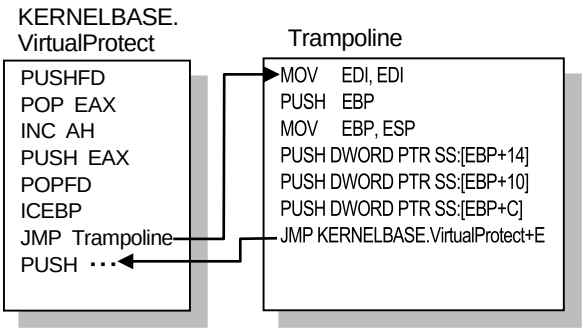


**図 4 ICEBP の埋め込みの例**

LBR フラグは, スレッド生成時に DllMain で SetThreadContext を使って DR7 のビット 8 を 1 に設定するとともに, デバッグ例外発生時に例外ハンドラで DR7 のビット 8 を 1 に設定する.

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 2 つだけ異なる. まず, TF 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 6 バイトになる. 次に, LBR フラグは, DllMain で設定しても, システムコールサービス 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. また, LBR の値は, ExceptionInformation メンバ以外に CONTEXT 構造体の LastBranchFromRIP メンバと LastBranchToRIP メンバにもロードされる.

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 で呼び出された時に, 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 アーキテクチャの時代から存在するが，Microsoft がサポートした OS は Windows 2003 SP1（Windows XP）以降の 64 ビット向け Windows である．AMD 64 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（Window 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ーファミリやプロセッサー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わかっている[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常にゼロである．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，提案手法を利用するには，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実装

LBR による ROPGuard 回避の防止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，LBR による ROPGuard 回避の防止を行う DLL である．もう 1 つは，LBR による ROPGuard 回避の防止を行う 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，LBR による ROPGuard 回避の防止以外に，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（CreateProcess と CreateProcessAsUser）をフックしている．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 [19]を利用した．Detours Express 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た M. Anka による Mhook[20]を利用した．

## 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，次に示す仕様の PC 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.

- PC: Panasonic CF-R7CW5AJR
- CPU: Intel Core 2 Duo U7600 1.20GHz
- Memory:DDR2-533 3GB
- Disk:Intel 530 Series/SSDSC2BW120A4K5 120GB
- OS: Windows 8.1 Enterprise 64bit

### 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, 次のアプリケーションの動作を調べた.

- IE 11(32/64 ビット)
- PCMark 8 Basic Edition(32/64 ビット)
- Microsoft Office 2013(32 ビット)
- Microsoft Office 2007(32 ビット)
- Google Chrome 44(32 ビット)
- Acrobat Reader XI(32 ビット)

その結果, 1 つ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, LastBranchFromIP レジスタが未使用領域のメモリ空間を指していたためである. この不具合の原因は明らかでない. そこで, この不具合を回避するために, 構造化例外処理により, アクセス違反の例外が発生した場合は, LBR による ROP チェックを中止することにした. これにより, 上述のアプリケーションすべてにおいて, 正しく動作することを確認した. この不具合の解決は今後の課題とする.

### 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, 提案手法が検知できることを確認した. 2 つ目は, ROPGuard 回避の ROP コード(図 2)であり, EMET は検知できないが, 提案手法は検知できることを確認した. 3 つ目は, 実際によく使われる ROP コードとして, Metasploit Framework で提供されている ROP コード(java.xml)であり, 提案手法が検知できることを確認した. 最後に, 実際に悪用されている脆弱性(CVE-2014-1761)の攻撃を検知できることを確かめるために, Metasploit Framework のモジュールにより攻撃を行う RTF ファイルを生成し, ROP 検知をテストした結果, 提案手法が検知できることを確認した.

### 5.3 ベンチマークテスト

提案手法のオーバーヘッドを評価するために, 本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 と, ウェブブラウザのパフォーマンスを計測する Peacekeeper(http://peacekeeper.futuremark.com/)でベンチマークテストを行った. それぞれの場合について, 3 回の計測を行い, それらの平均値を表 1 に示す. 表 1 から, 本手法のオーバーヘッドはほとんどないことがわかる.

**表 1 ベンチマークテストの比較**

| | 1 回目 | 2 回目 | 3 回目 | 平均 |
|---|---|---|---|---|
| 提案 | 720 | 730 | 691 | 713.7 |
| 通常 | 715 | 686 | 759 | 720.0 |

ICEBP 命令を埋め込んだ API に限定して, 例えば, 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, 表 2 (単位は秒)のように, 32 ビットで約 50 倍, 64 ビットで約 20 倍の速度低下を計測したことから, DEP 回避の API を無数に呼び出すようなアプリケーションでは, 速度低下が顕著になる. これは, ICEBP 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と考えられる.

表 2 実行速度の比較

| | 1 回目 | 2 回目 | 3 回目 | 平均 |
|---|---|---|---|---|
| 提案 32 | 290.1 | 285.5 | 275.7 | 283.8 |
| 通常 32 | 5.7 | 5.7 | 5.7 | 5.7 |
| 提案 64 | 97.9 | 98.0 | 98.8 | 98.2 |
| 通常 64 | 5.0 | 4.9 | 4.5 | 4.8 |

## 6 関連研究

LBR を利用した ROP 検知には，kBouncer [12]と ROPecker[13]がある．kBouncer と ROPecker は，ROP に限らず JOP[21]や COP[22]を検知できる利点がある．これらの共通の手法は，16 組の LastBranchFromIP レジスタと LastBranchToIP レジスタから構成される LBR スタックにガジェットが含まれる個数を調べ，その個数が閾値を超えたら，ROP として検知する手法である．この手法は，16 組の LBR スタックを前提にしていることから，Intel 64 アーキテクチャの Nehalem 以降のプロセッサーが必須要件であり，一方，AMD64 アーキテクチャは LBR が 1 組しかないため，AMD64 アーキテクチャにこの手法を適用できない．また，この手法は，LBR スタックにガジェットが含まれるかどうかを即時に調べられるようにするために，事前にガジェットのアドレスを調べておく必要がある．そのため，この手法は，バイナリコードの変更を監視するとともに，変更があれば，ガジェットのアドレスを再調査する機能が必要になる．それに対して，提案手法は，ガジェットのアドレスは不要であり，1 組の LBR でよいことから，Intel 64 アーキテクチャと AMD64 アーキテクチャに対応できる点が優れている．ただし，提案手法は，ROPGuard をベースにしているため，CALL 命令により DEP 回避の API が呼び出された場合には，検知できない欠点がある．

実装方法に関して，kBouncer と ROPecker は，LBR を取得するために，カーネルモードのデバイスドライバやカーネルモジュールを使うのに対して，本手法は，64 ビット向け Windows に備わった機能を利用することにより，カーネルモードではなく，ユーザモードのプログラムだけで，実装できる点が新しい．特に，64 ビット向け Windows 7 以降の Windows では，ルートキット対策に導入された KPP（Kernel Patch Protection）[23]がシステムコールサービスのフックを困難にするなど，デバイスドライバによる実装は，いくつかの実装上の課題がある．

## 7 おわりに

本論文では，LBR を用いて ROPGuard 回避を防止する実装方法を提案した．LBR を利用する従来の手法がカーネルモードのデバイスドライバやカーネルモジュールを使うのに対して，提案手法は，64 ビット向け Windows に備わった機能を利用することにより，ユーザモードのプログラムだけで実装できる．また，LBR を用いる従来の手法が，Intel の Nehalem 以降のプロセッサーに依存しているのに対して，提案手法は，Intel 64 アーキテクチャのみならず，AMD64 アーキテクチャのプロセッサーにも対応している．ただし，提案手法は，ROPGuard をベースにしているため，CALL 命令により DEP 回避の API が呼び出された場合には，検知できない．

プロトタイプシステムの評価では，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 回避を検知できることを確認した．また，ベンチマークテストでは，提案手法のオーバーヘッドはほんどなかったが，DEP 回避の API を大量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．

## 参考文献


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

[2] J. Li, et al.: Defeating return-oriented rootkits with “Return-Less” kernels, *Proceedings of the 5th European conference on Computer systems*, (2010).

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

[4] R. Wartell, et al.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code, *Proceedings of the 2012 ACM conference on Computer and communications security*, (2012).

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

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

[7] F. Ivan: Runtime Prevention of Return-Oriented Programming Attacks, (2012) https://code.google.com/p/ropguard/

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

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

[10] L. Yuan, et al.: Security breaches as PMU deviation: detecting and identifying security attacks using performance counters, *Proceedings of the Second Asia-Pacific Workshop on Systems*, 6, ACM, (2011).

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

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

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

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

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

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

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

[18] nick.p.everdox: Last branch records and branch tracing, CodeProject, (2013).

[19] G. Hunt, et al.: Detours: Binary Interception of Win32 Functions, *Proceedings of Third USENIX Windows NT Symposium*, USENIX, (1999).

[20] M. Anka: MHOOK, AN API HOOKING LIBRARY, V2.4, (2014) http://codefromthe70s.org/mhook24.aspx

[21] S. Checkoway, et al.: Return-oriented programming without returns, *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, (2010).

[22] E. Goktas, et al.: Out of control: Overcoming control-flow integrity, *Security and Privacy (SP), 2014 IEEE Symposium on. IEEE*, (2014).

[23] M. E. Russinovich, et al.: インサイド Windows 第6版 上, 日経BP, (2012).