Vol.2017-CSEC-79 No.10
2017/12/5

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

樗林 秀晃[1,a)]　毛利 公一[2,b)]　齋藤 彰一[1]

概要：C/C++のメモリ安全の脆弱性を使用した攻撃が増加している．これを受けて Intel は配列境界のチェック機能を追加したハードウェア Intel Memory Protection Extensions (Intel MPX) を開発した．しかし，MPX はコンパイル時にコード挿入されるため， ソースコードがないバイナリコードでは機能しない．本論文では，バイナリコードの解析と変更による ソースコードの必要ない MPX 実装方法を提案する．バイナリコードを動的解析し，アクセスパターンに注目することで 配列構造を復元する．復元した情報を使い，実行時にバイナリコードを 変更し MPX 命令を挿入することでメモリ安全を提供する．バイナリコード解析及び命令の挿入は Intel Pin を使用する． 現段階では静的及びスタック領域の配列が復元可能である．

Hideaki Kurebayashi[1,a)]　Koichi Mouri[2,b)]　Shoichi Saito[1]

1. はじめに

C/C++のメモリ安全の脆弱性を使用した攻撃が増加している．これは，プログラムに不正入力を行い，メモリアクセスを介してプログラムの予期しない領域へアクセスすることで，情報を抜き出したり攻撃コードを注入したりすることで行う．

メモリ安全性を高めるために，メモリアクセスが正当かどうかチェックする機能を対象のプログラムへ追加する，ソフトウェアベースの手法 [1–4] が提案されている．しかし，ソフトウェアベースの手法はオーバヘッドが大きいという問題点が存在する．これを受けて Intel 社は，メモリアクセス境界のチェック機能を追加したハードウェア，Intel Memory Protection Extensions(MPX) を開発した．MPX はチェック機能をハードウェア命令として実装することで，ソフトウェアベースの手法より，オーバヘッドを削減することが期待できる．しかし，MPX はコンパイル時適用されるため，ソースコードのないバイナリコードには有効に機能しない．

[1] 名古屋工業大学大学院
Nagoya Institute of Technology, Nagoya, Aichi 466-8555, Japan
[2] 立命館大学
Ritsumeikan University, Kusatsu, Shiga, 525-8577, Japan
[a)] h.kurebayashi.732@nitech.jp
[b)] mouri@cs.ritsumei.ac.jp

そこで，本論文では，バイナリコードの解析と変更によるソースコードの必要ない MPX 適用方法を提案する．まずバイナリコードを動的解析し，アクセスパターンに注目することで配列情報を復元する．この配列情報復元は既存手法 Howard [5] の手法に基づき行っている．次に，復元した配列情報に基づいて，実行時にバイナリコードに MPX 命令を挿入することでメモリ安全を提供する．バイナリコードの解析及び変更は，Dynamic Binary Instrumentation(DBI) フレームワークである Intel Pin [6] を使用する．現段階では静的及びスタック領域の配列が復元可能である．

提案手法を Linux 上に実装し，オーバヘッドの測定を行った．静的領域では約 2.5 倍，スタック領域では約 4 倍のオーバヘッドがある．

本論文の貢献は以下の通りである．

1. バイナリコードの動的解析を行うことで再コンパイルなしに，MPX 適用に必要な情報を復元する手法を述べる．
2. MPX 命令は通常コンパイル時に挿入される．本論文ではバイナリコードへ直接 MPX 命令を挿入する手法，復元した配列情報によってメモリ安全を提供する MPX 適用手法を述べる．

本論文のアウトラインは以下のようになる．まず第 2 章で MPX に関連するソフトウェアベースでメモリ安全を提供する手法と，提案手法の配列復元に関連するデータ構

造復元手法についての関連研究を述べる．次に第 3 章で MPX のバードウェア構成と gcc での MPX 適用方法についての概要を述べる．そして第 4 章で提案手法の概要を述べ，第 5 章で提案手法の配列復元と MPX 適用についての詳しい設計とアルゴリズムについて述べる．次に第 6 章で，提案手法を実装する上で使用したソフトウェアとその実装手法について詳しく述べる．第 7 章で，提案手法の正当性と実行時間による評価を行い，第 8 章で提案手法について，メモリエラー検知能力に関する考察を述べる．最後に第 9 章で全体のまとめと今後の課題を述べる．

2. 関連研究

本章では，関連研究について，ソフトウェアベースのメモリ安全手法とデータ構造復元手法を述べる．

2.1 ソフトウェアベースのメモリ安全手法

本節では既存のソフトウェアベースのメモリ安全手法について，コンパイル時にコードを変更する手法と，バイナリコードを変更する手法を述べる．

まずコンパイル時にコードを変更する手法を述べる．AddressSanitizer [1] はメモリの書き換えに注目することでメモリ安全を提供する．オブジェクトの周りに追加領域を挿入することで，オーバフローの発生による追加領域の変更に基づいて，オーバフローを検知する．SoftBound [2] はポインタベースの手法である．コンパイル時にソースコードを解析し，すべてのポインタアクセスとポインタの境界情報を求め，ポインタアクセスの前にチェックコードを挿入することで検知する．Masab らの手法 [3] では SoftBound をマルチコアで実行することでパフォーマンスの改善を図っている．SAFECode [4] はオブジェクトベースの手法である．コンパイル時にオブジェクトの情報を解析し，オブジェクトを指すポインタは，そのオブジェクト内へのアクセスのみに制限するコードを挿入する．MPX はこれらの手法の内 SoftBound と似た手法である．しかし，MPX はチェック機能をハードウェアとして提供しているため，提案手法ではこれら手法よりオーバヘッドの削減が期待できる．また，これら手法はソースコードと再コンパイルが必要であるが，提案手法ではそれらが不要である．

次に，バイナリコードを変更する手法を述べる．Dr.Memory [7] と Memcheck [8] は，バイナリコードのメモリエラーを検知する．バイナリコードを変更し，メモリへのアクセスを追跡することで，未初期化のメモリの読み込みと許可されていない領域へのアクセスを検知する．これらの手法は，ある程度のメモリのエラーを検知するが，各配列構造のメモリエラーのような粒度の細かいエラーは検知できない場合がある．提案手法では，バイナリコードから配列構造を復元することで，これら手法より，粒度の細かいエラー検知が可能である．

2.2 データ構造復元手法

Howard はバイナリファイルのプログラムから，データ構造を復元する手法である．Howard は既存のデータ構造復元手法に加えて，Howard 独自の手法としてメモリへのアクセスパターンを使用してデータ構造を復元する．この独自手法では，プログラムを動的解析し，それぞれのデータ構造特有のメモリアクセスパターンを見つけることでデータ構造を復元する．Howard では静的領域だけでなくスタックとヒープ領域のデータ構造も復元可能である．本論文では Howard の独自手法を参考に配列を復元する．

Laika [9] はメモリを動的解析しポインタらしいデータを集めることでデータ構造の復元を行う．REWARD [10] は動的解析においてシステムコール関数の引数の型など，型情報が事前に分かる構造を用い型情報を復元する．Howard はこれらの手法より精度の高い復元を行うことが可能である．TIE [11] はバイナリファイルを解析し Value Set Analysis(VSA) [12] と似た手法と束縛条件によって変数の型を復元する．これらの手法では配列情報の復元は不可能であるが，Howard では復元が可能である．提案手法は Howard の手法に基づくことで，精度の高い配列の復元が可能である．

3. Intel MPX

MPX は，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エラーを検知する．この機能を実現するために，境界チェックを行う命令と，境界情報を格納するレジスタを追加している．配列の境界情報を境界レジスタに格納し，各メモリアクセスを行う命令の前にアクセスしたアドレスが境界レジスタの境界情報の範囲内に収まっているかチェックする命令を挿入する．アドレスチェックで違反が発生した場合は例外を発生することで，違反を検知する．通常，これらの処理を行う MPX 命令はコンパイル時に挿入される．本論文では，MPX 命令に対応しているコンパイラの内，gcc の MPX 適用手法を参考に設計と実装を行った．本章では MPX のハードウェア構成と gcc における MPX の対応方法について述べる．

3.1 ハードウェア構成

境界情報を格納するために MPX では 4 つのレジスタ bnd0 から bnd3 を追加している．レジスタのサイズは 128bit で，64bit ずつメモリアクセスの上限アドレスと下限アドレスを格納する．アドレスチェックで違反が発生した場合は#BR という例外が発生する．この例外は Linux ではセグメンテーションフォールトに割り与えられている． 表 1 に MPX の主要命令とその説明を示す．

表 1: MPX の主要命令

命令	説明
bndmk (base,disp,1), bnd	境界レジスタ bnd の下限アドレスにレジスタ base の値, 上限アドレスに base+disp の値を格納する
bndmov bnda/mem, bndb	境界レジスタ bnda もしくはメモリ mem から境界レジスタ bndb に値を移す
bndcl reg/mem, bnd	メモリアドレス mem もしくはレジスタ reg 中のメモリアドレスと境界レジスタ bnd の下限アドレスをチェックする．違反した場合例外が発生する
bndcu reg/mem, bnd	上限アドレスをチェックする．そのほかは bndcl と同じ

```
1  int array[10];
2  // (1)
3  // edx=array
4  // eax=sizeof(array)-1
5  // bndmk (%edx,%eax,1),%bnd0
6
7  // (2)
8  // edx=&array[i]
9  // bndcl (%edx),%bnd0
10 // edx=&array[i]+sizeof(int)-1
11 // bndcu (%edx),%bnd0
12 x = array[i];
```

図 1: MPX 適用例

3.2 gcc による MPX 適用

gcc はコマンドラインオプションを指定することによって，MPX を適用したコードを生成する．図 1 に MPX の適用例を，int 型，サイズ 10 の配列 array を参照するコードで示す．このコードに MPX を適用すると，まず配列の定義部分に (1) に示す 3 から 5 行目のコードが挿入される．(1) では 3 行目でレジスタ edx に配列の先頭アドレス，4 行目でレジスタ eax に配列のサイズ-1 を格納し，5 行目の bndmk 命令でレジスタ bnd0 に境界情報を格納する．

次に配列の参照部分に (2) に示す 8 から 11 行目のコードが挿入される．(2) では 8 行目でレジスタ edx に参照アドレスを格納し，9 行目の bndcl 命令で参照アドレスと境界の下限アドレスをチェックする．10 行目でレジスタ edx に参照アドレス+一要素のサイズ-1 を格納し，11 行目の bndcu 命令で edx と 境界の上限アドレスをチェックする．チェックによってアドレスが境界の範囲外であった場合には，例外が発生しアクセス違反を検知する．

3.3 gcc の MPX ライブラリ

gcc は libmpx と呼ばれる MPX ライブラリを提供しており，リンク時に libmpx をリンクする．ライブラリは次の役割を持つ．

1. 境界情報領域の確保
2. MPX コンフィグレジスタの設定
3. 違反検知時の処理

本節では各役割について述べる．

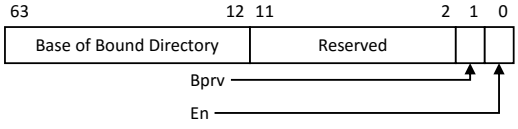

図 2: BNDCFGU レジスタレイアウト

表 2: BNDCFGU ビット機能

Bits	機能	
En	1	MPX 命令を使用可能にする
	0	MPX 命令は NOP になる
Bprv	1	レガシーなコードにおいて境界レジスタを保存する
	0	レガシーなコードにおいて境界レジスタを保存しない
Base of Bound Directory	Bound Directory のベースアドレス	

3.3.1 境界情報領域の確保

MPX は，通常のメモリ領域に加え，Bound Directory と呼ばれる特殊領域にも境界情報を格納することができる．libmpx はロード時に mmap によって Bound Directory 用の領域を確保する．

3.3.2 MPX コンフィグレジスタの設定

MPX は，MPX コンフィグレジスタ BNDCFGU を適切に設定することで使用可能になる．BNDCFGU へのアクセスは xsave と xrstor 命令によって行う．図 2 に BNDCFGU レジスタのレイアウトを，表 2 に各ビットの機能を示す．表の Bprv におけるレガシーなコードとは，MPX が適用されていないコードやライブラリのことである．libmpx はロード時に，En ビットに 1，Base of Bound Directory に「境界情報領域の確保」処理で確保したアドレスをセットする．Bprv ビットは libmpx オプションによって選択可能となっている．

3.3.3 違反検知時の処理

libmpx はロード時にセグメンテーションフォールトに対するハンドラを sigaction によって設定する．ハンドラでは，まず#BR 例外による例外かどうかを調べる．これは，割り込みベクタ番号が格納されたレジスタ trapno の値を調べることで行っている．#BR 例外だった場合はオプションによって，ハンドラをそのまま終了するか，処理を継続するか選択可能である．そのまま終了する場合は exit によって終了する処理で十分だが，継続する場合は追加の処理が必要になる．これは追加処理なしで復帰すると，セグメンテーションフォールトが発生した命令から再実行されるため，再度セグメンテーションフォールトが発生し，無限ループになるためである．そのため libmpx のハンドラではまず，セグメンテーションフォールトが発生した命令の次の命令のアドレスを計算する．これは，現在の命令ポインタのアドレスから 1byte ずつ読み込み，MPX 命令のマシン語と値を比較することで行っている．次に命令ポインタの値を次の命令のアドレスに変更することで，セグ

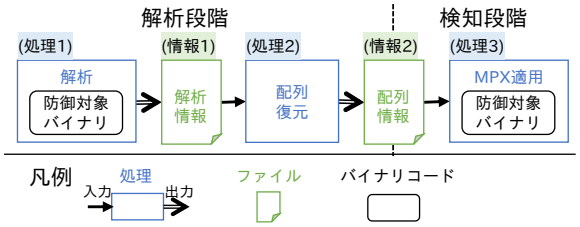

図 3: 提案手法概要図

メンテーションフォールトからの復帰を行っている．

4. 提案手法

本論文では，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を提案する．まずバイナリコードを動的解析し，配列情報を復元する．次に，復元した配列情報を使用し，実行時にバイナリコードを変更し，MPX 命令を挿入することで，メモリ安全を提供する．

4.1 提案手法概要

図 3 に提案手法概要図を示す．提案手法は解析段階と検知段階の 2 つの段階で構成されている．解析段階ではまず図 3 の (処理 1) 解析処理で防御対象バイナリコードの動的解析を行い，図 3 の (情報 1) 解析情報を出力する．次に図 3 の (処理 2) 配列復元処理で解析情報をもとに配列情報の復元を行い，図 3 の (情報 2) 配列情報を出力する．検知段階では配列情報をもとに，(処理 3) 防御対象バイナリコードへの MPX 命令の挿入処理を行うことでメモリエラーの検知を行う．

4.2 提案手法の利点

通常の MPX 適用はコンパイル時に MPX 命令を挿入するため，ソースコードが必ず必要である．それに対し，提案手法はバイナリコードの解析結果をもとに挿入を行うため，ソースコードが必要ない．そのため，ソースコードがないレガシーなコードにも適用可能である．また，提案手法は MPX というハードウェアの機能を使用することで，ソフトウェアベースの手法よりオーバヘッドの削減が期待できる．

5. 設計

本章では提案手法の詳しい設計について，図 3 に沿って述べる．まず処理 1 では，Intel Pin を用いた動的解析を行うため，新しい手法の設計は必要ない．以降，処理 2 と 3 の設計の詳細を，静的領域の配列とスタック領域の配列のそれぞれについて述べる．

5.1 静的領域の配列

本節では静的領域の配列について，復元手法と MPX 適用手法を述べる．

5.1.1 静的領域の配列情報の復元

本項では，図 3 の処理 2 について述べる．配列情報の復元の設計については著者の過去の論文 RINArray [13] に詳しい．本論文では概要のみ述べる．

この配列情報の復元は Howard の手法に基づいて設計を行った．復元する配列情報は次の 3 つで構成される．

1. 配列にアクセスする命令
2. 配列の開始アドレス
3. 配列のサイズ

配列はループ中で参照されることが多いという点に着目し，次に示す手順で配列情報の復元を行う．

1. 関数の検出:バイナリコード上の関数の位置を検出
2. ループの検出:関数中のループの位置を検出
3. 配列の検出:ループ中のメモリアクセスパターンから配列を検出

最初に関数の検出を行う．関数は call 命令で始まり ret 命令で終わるものと定義する．この定義に従って，call 命令を実行してから対応する ret 命令を実行するまでの命令を関数として検出する．

次に関数中のループ部分を検出する．これは後方分岐命令をもとに行う．後方分岐命令を実行するとジャンプ先のアドレスからジャンプ元のアドレスにあるすべての命令を 1 つのループとして検出する．

最後にループ中のメモリ参照部分を解析し「配列らしい」アクセスパターンを検出し配列情報を求める．そのアクセスパターンは次の 2 つである．

1. `elem=*(ptr++)` パターン
2. `elem=array[i]` パターン

`elem=*(ptr++)` パターンはメモリに順番にアクセスするという特徴を持つ．そのため，動的解析の実行において各ループでアクセスするアドレスの差を調べ，その結果アドレスの差がすべて同じならこのパターンとする．この時の一要素のサイズはアドレスの差となる．また，配列の開始アドレスはアクセスした中で最小のアドレスとし，終了アドレスは (アクセスした中で最大のアドレス+一要素のサイズ) とする．配列のサイズは (終了アドレス-開始アドレス) とする．

`elem=array[i]` パターンは 1 つのアドレスからのオフセットを使用しアクセスするという特徴を持つ．そのため，同じベースアドレスを持つパターンを集めてこのパターンとする．この時一要素のサイズはアクセスしたアドレスの差の中で最も小さいものとする．そのほかの配列情報の求め方は `elem=*(ptr++)` パターンと同じである．

5.1.2 静的領域の配列への MPX 適用

本項では，図 3 の処理 3 について述べる．解析段階で復

```
1  // 境界情報
2  typedef struct {
3    uint64_t lowerBound;
4    uint64_t upperBound;
5  } Bnd;
6
7  // bndmk (p, size-1, 1), %bnd
8  // bnd に下限:p, 上限:p+size-1の境界を格納
9  void make_bnd(void *p, size_t size);
10
11 // bndcl p, %bnd
12 // p と bnd の境界の下限をチェック
13 void check_lowerbnd(void *p);
14
15 // bndcu p, %bnd
16 // p と bnd の上限をチェック
17 void check_upperbnd(void *p);
18
19 // bndmov bnd -> mem
20 // 境界レジスタからメモリへ境界情報を移す
21 void mov_bnd_to_mem(Bnd *bnd);
22
23 // bndmov mem -> bnd
24 // メモリから境界レジスタへ境界情報を移す
25 void mov_mem_to_bnd(Bnd *bnd);
```

図 4: MPX インラインコード

```
1  struct StaticInfo{ // 静的領域
2    void *addr;   // 命令のあるアドレス
3    size_t size; // 配列のサイズ
4    void *start; // 配列の開始アドレス
5    Bnd bnd; // 境界情報
6  };
7
8  void init(){
9    for(ary ∈ すべてのStaticInfo)
10     make_bnd(ary.start, ary.size);
11     mov_bnd_to_mem(&ary.bnd);
12   }
13 }
14
15 void check(Bnd *bnd, void *ref, size_t s)
16 {
17   mov_mem_to_bnd(bnd);
18   check_lowerbnd(ref);
19   check_upperbnd(ref+s);
20 }
```

図 5: 静的領域 MPX 適用

元した配列情報を基に検知段階で MPX 適用を行う．gcc の MPX 適用と同様に，まずプログラムが始まる前に，復元した配列情報から MPX 用境界情報を作成するコードを挿入する (図 1(1) 部分)．次に，配列を参照する命令の前に，参照アドレスと境界の下限と上限のアドレスをチェックするコードを挿入する (図 1(2) 部分)．

図 4 に，MPX 適用に使用する関数を示す．これらの関数は実際には MPX 命令をインラインコードで記述したものである．図 5 に静的領域の配列に対する MPX 適用手法を示す．復元した 1 つの静的領域の配列情報は StaticInfo 構造で定義する．静的領域の配列に MPX を適用するために，まず main 関数の前に図 5 の 8 行目の init 関数のコードを挿入する．init 関数では復元したすべての静的領域配列について make_bnd で境界レジスタに配列の境界情報を格納し，mov_bnd_to_mem で境界レジスタからメモリへ値を移す処理を行う．次に，配列を参照する命令の前に，図 5 の 15 行目の check 関数のコードを挿入する．check 関数の引数 bnd にはその命令に対応する境界情報，ref にはその命令で参照するメモリのアドレス，s には参照するメモリのサイズを渡す．参照するメモリのサイズとは，読み込み又は書き込みサイズで，例えば mov 命令で 8bit 書き込みなら s は 1，16bit 書き込みなら s は 2 となる．check 関数では mov_mem_to_bnd でメモリから境界レジスタに境界情報を移し，check_lowerbnd で ref と bnd の下限アドレスチェック，check_upperbnd で ref+s と bnd の上限アドレスチェック処理を行う．

5.2 スタック領域の配列

本節ではスタック領域の配列について，復元手法と MPX 適用手法を述べる．スタック領域へのアクセス方法は `mov $1,-0x4(%ebp)` のようにベースポインタからのオフセットを使用するもののみと仮定し，ほかのアクセス方法を考慮した実装は今後の課題である．この仮定から，スタック領域の配列のアドレスは実行毎に変化するが，ベースポインタからのオフセットは不変となる．

5.2.1 スタック領域の配列情報の復元

スタック領域に対応する図 3 の処理 2 を述べる．解析段階でスタック領域の配列を復元するためには，次の情報が必要となる．

1. 配列にアクセスする命令
2. 配列のサイズ
3. その配列が確保された関数
4. その関数のベースポインタから配列の開始アドレスまでのオフセット

上記 3 と 4 が静的領域の場合と異なる．これらの情報を求めるためにまず，第 5.1.1 項の関数を検出する処理において，追加で，ベースアドレスの値を記憶する．次に，第 5.1.1 項の配列の検出の処理において，配列を検出した時に，各関数のベースアドレスと配列の開始アドレスから，その配列がどの関数に属するかを調べる．その関数のベースポインタから，配列の開始アドレスまでがオフセットとなる．

5.2.2 スタック領域の配列への MPX 適用

スタック領域について図 3 の処理 3 を述べる．スタック領域の配列のアドレスが実行毎に変化することは，関数の開始時に，MPX の境界情報を更新することで対応する．まず関数の開始時に，ベースポインタの値を求める．次に，復元したスタック領域の配列情報におけるベースポインタからのオフセット情報と現在のベースポインタの値から，境界情報を更新する．この処理を行うことで，配列を参照する前のアドレスチェックは静的領域と同じ方法 (図 5 の check 関数参照) で行うことができる．

図 6 にスタック領域の配列への MPX 適用手法を示す．復元した 1 つのスタック領域の配列情報は StackInfo 構造

```
1  struct StackInfo{// スタック領域
2    void *addr;   // 命令のあるアドレス
3    size_t size; // 配列のサイズ
4    void *offset; // ベースポインタからのオフセット
5    void *func;  // 関数開始アドレス
6    Bnd bnd; // 境界情報
7  };
8
9  void update(void *func, void *basePtr){
10   for(ary ∈ func に属する StackInfo){
11     void *start = basePtr+ary.offset;
12     make_bnd(start, ary.size);
13     mov_bnd_to_mem(&ary.bnd);
14   }
15 }
```

図 6: スタック領域 MPX 適用

で定義する．スタック領域の配列に MPX 境界情報を更新するために，すべての関数において，ベースポインタの値が決定した後に，図 6 の 9 行目の update 関数のコードを挿入する．引数の func には関数の開始アドレスを，basePtr にはベースポインタの値を渡す．update では関数 func に属するすべての StackInfo に対し，ベースポインタとオフセットから配列開始アドレスを更新し，make_bnd で，境界情報を境界レジスタに格納し，mov_bnd_to_mem で境界レジスタからメモリへ値を移す処理を行う．配列を参照する命令の前には図 5 の check 関数と同じコードを挿入する．

6. 実装

提案手法を Linux 上で実装した．図 3 の処理 1 と処理 3 は Intel Pin の機能を使用した．図 3 の処理 2 は Java で実装した．

6.1 Intel Pin

Pin は DBI フレームワークであり，動的なバイナリコードの解析及び，編集を行うことができる．Pin を使用することで，実行した命令の取得，各レジスタの値の取得，新規コードの挿入などを行うことができる．この解析及び編集は，関数，命令などの単位で行う．また編集のタイミングは，ロード時編集と実行時編集の 2 つがある．ロード時編集は，オブジェクトファイルやライブラリなどのイメージファイルが読み込まれた時に編集を行う．この編集ではイメージファイル中すべての関数や命令などの静的情報が利用できるが，命令の実行順のような動的情報は利用できない．実行時編集は，関数や命令を実行する直前に編集を行う．この編集では関数や命令の実行順など動的情報が利用できるが，実行されていない関数や命令などの静的情報については利用できない．

図 7 に Pin で解析を行うイメージを示す．ただし，簡略化のために説明に必要なコードのみを示している．19 行目の INS_AddInstrumentFunction 関数は各命令の編集を行うための関数を登録する．登録された 8 行目の event_ins 関数は，各命令の情報を表す構造体 INS が引数に渡され，コールバックされる．10 行目の INS_InsertCall 関数は命令 ins

```
1  void *pointer;
2
3  void at_ins(void *ptr, ADDRINT eaxVal)
4  {
5    ...
6  }
7
8  void event_ins(INS ins, void *v)
9  {
10   INS_InsertCall(ins, IPOINT_BEFORE,
11                  at_ins,
12                  IARG_PTR, pointer,
13                  IARG_REG_VALUE, REG_EAX,
14                  IARG_END);
15 }
16
17 int main()
18 {
19   INS_AddInstrumentFunction(event_ins, 0);
20   return 0;
21 }
```

図 7: Pin のプログラム例

の前に関数を挿入する関数である．IPOINT_BEFORE で命令の前に挿入することを示し，次の引数 at_ins 関数が挿入される．IARG_PTR から IARG_END の前までが at_ins 関数に渡す引数である．IARG_PTR はポインタ pointer の値を引数 ptr に渡すことを示す．IARG_REG_VALUE, REG_EAX はレジスタ eax の値を引数 eaxVal に渡すことを示す．つまり，このプログラムでは，すべての命令の前に at_ins 関数の引数 ptr に pointer の値が，eaxVal にレジスタ eax の値が渡されて，at_ins 関数が挿入される．

6.2 解析及び解析情報の出力

本節では図 3 の処理 1 及び情報 1 について述べる．Pin を使用し動的解析を行うことで，配列復元処理に必要な情報をファイルとして出力する．配列復元処理に必要な情報は，命令があるアドレス，各オペランドの値，各オペランドで使用するレジスタとその値である．そのため，各命令の前に上記ソースオペランドの情報，各命令の後に上記デスティネーションオペランドの情報を出力するコードを挿入する．

6.3 配列復元と配列情報の出力

本節は図 3 の処理 2 及び情報 2 について述べる．解析情報 (情報 1) を読み込み，命令実行順と実行時のレジスタの値の再構築を行うことで配列を復元する．復元が終わると復元した配列情報のファイルを出力する．

6.4 MPX 適用

本節では図 3 の処理 3 について説明する．gcc の libmpx のソースコードを参考に，同じ機能を Pin 上で再現した．コードの挿入は，防御対象バイナリコードのロード時編集で，バイナリコードのオブジェクトファイルに対して行う．

まず MPX の機能を使用可能にするために，libmpx と同様の以下のコードを挿入する．

1. 境界情報領域の確保

表 3: 評価環境

OS	Ubuntu 16.04.3 LTS
kernel	4.4.0-97-generic
CPU	Intel(R) Core(TM) i7-6700, 3.40GHz, 8 core
memory	8GB
gcc	5.4.0
Intel Pin	3.2-81205

2. MPX コンフィグレジスタの設定

3. 違反検知時の処理

libmpx ではライブラリのロード時に上記の処理を行っていたが，Pin では main 関数の前に上記コードを挿入する．また，Bprv ビットには 0 を設定した．

次に MPX 命令を使用したコードを挿入する．MPX 命令は図 4 の関数を，インライン関数でインラインアセンブラを使用し実装した．現在の実装では，4 つの境界レジスタの内，bnd0 のみ使用可能である．Pin を使用し，main 関数，各関数，配列を参照する命令に対して，第 5.1.2 項と第 5.2.2 項で述べた処理を行う関数を挿入した．

7. 評価

提案手法を表 3 の環境に実装し評価を行った．まず配列の復元と MPX 適用結果が正しく処理されていることを確認した．以降，本章では実行時間の評価について述べる．

実行時間は，ループで一定回数配列を参照するプログラムを作成し，ループ回数を変更し比較を行った．図 8 に実行時間の比較，表 4 に各凡例の説明を示す．図 8 は各ループ回数の実行時間を示している．縦軸は各ループ回数の Static の実行時間を 1 として正規化した時間である．最適化オプションは，Static は O2，Stack は O0 である．Stack を O0 とした理由は，O2 とすると配列のアクセスがベースポインタからのオフセットを使用しなくなり，配列の復元を行えないためである．

評価結果から次のことが分かる．まず Pin 上で実行するプログラムではループ回数が多くなるにつれオーバヘッドが減少している．これは Pin では初期化及び最初のコード挿入に時間がかかるためであると考えられる．次に，Static/Stack+Pin(if) と Static/Stack+Pin(MPX) を比べることで，MPX を適用することによりオーバヘッドが減少したことが分かる．さらに，Stack+Pin(MPX) の結果から提案手法におけるスタック領域配列のチェックはオーバヘッドが大きいことが分かる．これは，各関数の最初での配列境界の更新に時間がかかるためであると考えられる．

8. 考察

本章では，提案手法における検知能力についての考察を述べる．

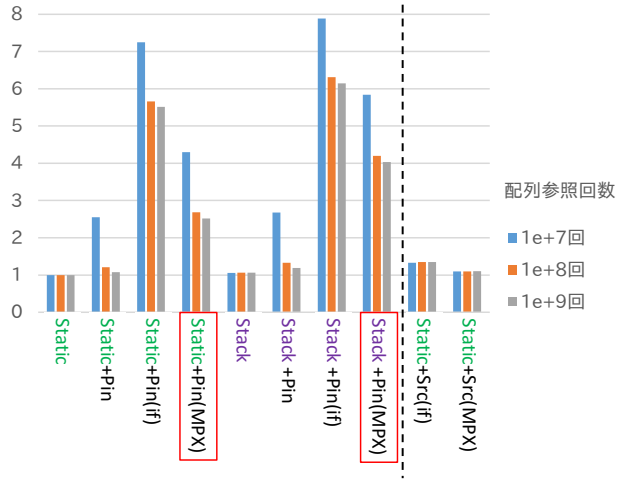

図 8: 実行時間

表 4: 凡例の説明

凡例	説明
Static	静的領域の配列に for 文でアクセスするプログラムを単体で実行
Stack	スタック領域の配列にアクセスする関数を for 文で繰り返すプログラムを単体で実行
Static/Stack+Pin	Static/Stack を Pin 上で実行．Pin では何も行わない
Static/Stack+Pin(if)	Static/Stack を Pin 上で実行．Pin では配列にアクセスする命令の前に if 文でメモリエラーチェックを行うコードを挿入
Static/Stack+Pin(MPX)	Static/Stack を Pin 上で実行．Pin では配列にアクセスする命令の前に MPX でメモリエラーチェックを行うコードを挿入 (提案手法)
Static+Src if	Static のソースコードを変更し，配列にアクセスする命令の前に if 文でメモリエラーチェックを行うコードを挿入
Static+Src MPX	Static のソースコードに gcc で MPX を適用

8.1 配列復元

本節では復元可能な配列と，復元不能な配列について述べる．

8.1.1 ptr++パターン

現段階の実装では ptr++パターンはインクリメントによって，配列の最初から最後まで順にアクセスする配列が復元可能である．デクリメントや，順にアクセスしていない場合は復元不能である．デクリメントはインクリメントと逆の方法とすることで実現可能であると考えている．しかし，順にアクセスしていない場合の検知は難しいと考える．

8.1.2 array[i] パターン

array[i] パターンは，アクセス順によらず復元可能である．ランダムアクセスや，アクセスしない要素があっても復元できる．しかし，配列の最初と最後の要素は必ずアク

セスする必要がある．また，一要素のサイズを求めるために，少なくとも 1 つは隣り合う要素にアクセスした要素が必要である．

8.1.3 配列の存在領域

スタック領域は，関数スコープで宣言された配列である必要がある．if 文や for 文中で宣言された配列など，関数スコープより小さいスコープの配列は復元できない．関数スコープより小さいスコープの配列は，スタックポインタが変更された時をスコープの変更とすることで検知可能であると考えている．ヒープ領域は現段階の実装では検知できない．これは，Howard の手法を使用することで検知可能であると考えている．

8.2 誤検知及び見逃し

本節では提案手法を適用した場合に発生する，誤検知及び見逃しについて述べる．まず誤検知は，解析段階において実際に確保された配列領域よりも小さな領域として配列を復元し，検知段階で，その領域の範囲外にアクセスした場合に発生する．これは，解析段階で配列のすべての要素にアクセスしなかった場合に発生する．次に，見逃しは解析段階でアクセスしなかった配列へのアクセスと，解析段階でアクセスしなかった命令で配列にアクセスした場合に発生する．これらは，配列復元の精度を向上することで削減が可能である．

9. まとめ

本論文では，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を提案した．バイナリコードのアクセスパターンに注目し，配列情報の復元を行い，MPX 命令を挿入する．

次に提案手法を Linux 上に実装し，オーバヘッドを確認した．現段階では `elem=*(ptr++)` パターンと `elem=array[i]` パターン，静的及びスタック領域の配列が復元可能である．

今後の課題は次のようになる．まず復元できる配列の種類を増やすことが課題である．Howard においてはヒープ領域の配列や多次元配列の復元は可能であるが，提案手法の現段階の実装では，それら配列は復元できない．次にいくつかの配列をまとめる処理が課題である．解析段階で違う命令で同じ配列にアクセスした場合，それらの結果を統合することで，より正確な配列情報を復元できる．また，検知段階で，解析段階ではアクセスしなかった命令で解析段階で復元した配列にアクセスした場合，現段階の実装では検知できない．これを検知できるような実装ができれば，誤検知と見逃しを減らすことができる．また，MPX 適用の効率化が課題である．現在の実装では，すべての配列について，境界情報のメモリへの退避を行っている．しかし，レジスタの数が足りる場合はメモリへの退避を行わないことでオーバヘッドの削減が可能である．

参考文献

[1] Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker., *USENIX Annual Technical Conference*, pp. 309–318 (2012).
[2] Nagarakatte, S., Zhao, J., Martin, M. M. and Zdancewic, S.: SoftBound: Highly Compatible and Complete Spatial Memory Safety for C, *SIGPLAN Not.*, Vol. 44, No. 6, pp. 245–258 (2009).
[3] Ahmad, M., Haider, S. K., Hijaz, F., van Dijk, M. and Khan, O.: Exploring the performance implications of memory safety primitives in many-core processors executing multi-threaded workloads, *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy* (2015).
[4] Dhurjati, D., Kowshik, S. and Adve, V.: SAFECode: Enforcing alias analysis for weakly typed languages, *ACM SIGPLAN Notices*, Vol. 41, No. 6, pp. 144–157 (2006).
[5] Slowinska, A., Stancescu, T. and Bos, H.: Howard: A Dynamic Excavator for Reverse Engineering Data Structures., *NDSS* (2011).
[6] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, *ACM SIGPLAN Notices*, Vol. 40, No. 6, pp. 190–200 (2005).
[7] Bruening, D. and Zhao, Q.: Practical memory checking with Dr. Memory, *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, pp. 213–223 (2011).
[8] Seward, J. and Nethercote, N.: Using Valgrind to Detect Undefined Value Errors with Bit-Precision., *USENIX Annual Technical Conference, General Track*, pp. 17–30 (2005).
[9] Cozzie, A., Stratton, F., Xue, H. and King, S. T.: Digging for Data Structures., *OSDI*, Vol. 8, pp. 255–266 (2008).
[10] Lin, Z., Zhang, X. and Xu, D.: Automatic reverse engineering of data structures from binary execution, *Proceedings of the 11th Annual Information Security Symposium*, CERIAS-Purdue University (2010).
[11] Lee, J., Avgerinos, T. and Brumley, D.: TIE: Principled reverse engineering of types in binary programs (2011).
[12] Balakrishnan, G. and Reps, T.: WYSINWYX: What you see is not what you eXecute, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 32, No. 6 (2010).
[13] 樽林秀晃，瀧本栄二，毛利公一，齋藤彰一ほか：RINArray: 配列構造復元による侵入検知システムの精度向上，研究報告コンピュータセキュリティ (CSEC)，Vol. 2017, No. 24, pp. 1–8 (2017).