Computer Security Symposium 2018
22 - 25 October 2018

ヒープ領域に対するソースコード不要の Intel MPX 命令適用手法

加藤 周良[1]　瀧本 栄二[2]　毛利 公一[2]　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 MPX を開発した．しかし，MPX はコンパイル時に実装されるため，ソースコードがないバイナリコードには実装できない．関連研究では，静的領域とスタック領域の配列に対してバイナリコードの動的解析と MPX 命令挿入によるソースコードの必要ない MPX 実装方法を提案した．本論文では，確保したヒープ領域に対しての実装方法を提案する．使用頻度が高い libc ライブラリの malloc 関数に着目し実装を行う．バイナリコード解析及び命令の挿入は Intel Pin を使用する．

キーワード：Intel MPX， Intel Pin，ヒープ領域境界検査，malloc，オーバーフロー

1. はじめに

メモリ安全の脆弱性がある C/C++プログラムが多く存在している．サイバー攻撃の多くはメモリ安全の脆弱性を突き，ネットワークを介してプログラムに不正メモリアクセスを行う．そして，情報の抜き出しや攻撃コードを注入する．

メモリの安全性を高めるための手法 [1] がある．この手法では，Intel が開発したメモリアクセス境界のチェック機能をハードウェア命令として実装している Intel Memory Protection Extensions(MPX) を使用している．MPX を使用することでソフトウェアベースの手法 [2–5] より，オーバヘッドを削減 [6] したメモリ安全を提供している．さらに，コンパイル時に適用する MPX 命令を，バイナリコードへ直接挿入することでバイナリコードのみでメモリ保護を可能にしている．

しかしこの手法 [1] では，静的領域とスタック領域の配列にしか対応しておらず，ヒープ領域のメモリ安全は保障していない．そのため，ヒープ領域に対する攻撃に対処することができない．そこで本論文では，ヒープ領域に対するソースコードの必要ない MPX 適用方法を提案する．

提案手法では，C/C++で使用頻度が高い libc ライブラリの malloc 関数に着目してバイナリコードを動的解析し，malloc により確保したメモリ領域情報を取得する．そして，メモリ領域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．提案手法によって，ソースコードを用いず，バイナリコードのみでヒープ領域のメモリの安全性を高めることが可能である．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 [7] を使用する．

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 のハードウェア構成と gcc での MPX 適用方法についての概要を述べる．第 3 章で静的領域とスタック領域への MPX 適用手法 [1] の詳細を述べる．次に第 4 章で提案手法の概要を述べ，第 5 章で提案手法の確保したヒープ領域の情報の取得と MPX 適用についての詳しい設計を述べる．第 6 章では提案手法の実装方法について述べる．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課題と考察を述べる．最後に第 9 章で全体のまとめと今後の課題を述べる．

2. Intel MPX

MPX は，x86 と x86-64 の命令拡張機能である．メモリアクセスの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．境界情報を格納するレジスタには，有効なアドレス範囲の上限と下限のアドレスをセットする．境界チェックを行う命令は，メモリアクセスを行う命令の直前に挿入して，当該メモリアクセス命令がアクセスするアドレスがレジスタにセットされた境界内に収まっているか否かを確認する．もし，境界の外のアドレス

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University

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．その他は 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

をアクセスしようとしている場合は違反となる．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通知する．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．

Intel Skylake 世代から利用可能であり，対応してない CPU では nop 命令扱いになる．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．

2.1 ハードウェア構成

MPX では境界情報を格納するために 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．境界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 の命令とその説明を示す．

2.2 gcc による MPX 適用

gcc はコマンドラインオプションに “ -mmpx ” 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

```
1 void * __mpx_wrapper_malloc (size_t size)
2 {
3   void *p = (void *)malloc (size);
4   if (!p) return __bnd_null_ptr_bounds (p);
5   return __bnd_set_ptr_bounds (p, size);
6 }
```

図 2: malloc の MPX 適用コード

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が境界の下限より上かどうかチェックする．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 境界の上限より下かどうかをチェックする．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通知する．

2.3 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
4. メモリ操作関数の置き換え

1〜3 については関連研究 [1] に詳しく記載されている．本論文では概要のみ述べる．libmpx では，メモリの境界情報を格納するための Bound Directory と呼ばれる特殊領域をメモリ上に確保する．また，MPX を使用可能にするために MPX コンフィグレジスタを設定する．そして，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．

4 のメモリ操作関数の置き換えについて述べる．libmpx では，malloc 関数，strcat 関数などのメモリ確保や書き込みを行う libc 関数を MPX 適用したラッパ関数に置き換える機能を持つ．図 2 に malloc 関数のラッパ関数である __mpx_wrapper_malloc 関数のコードを示す．

__mpx_wrapper_malloc 関数では malloc 関数を実行し，メモリが確保されなかった場合は境界情報を NULL とする．一方，メモリ確保が行われた場合， __bnd_set_ptr_bounds 関数によって，確保された領域の境界情報を保存する．__bnd_set_ptr_bounds 関数では，図 1 に示す (1) のコードを実行し，境界情報を保存する．境界情報は確保したヒープ領域の先頭アドレス p と引数で渡された size から設定する．そして，保存した境界情報を使用して確保したヒープ領域を参照する命令の前に，図 1 に示す (2) のコードを挿入する．

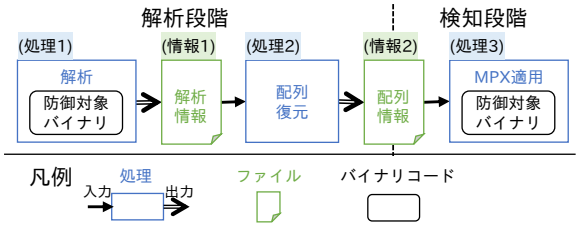

図 3: 配列復元手法概要

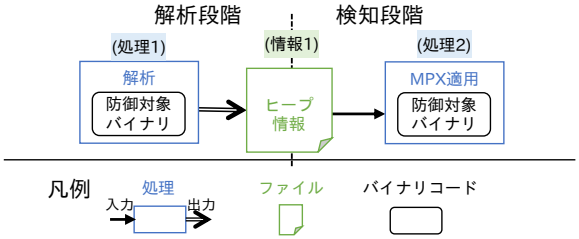

図 4: 提案手法概要

3. 関連研究：静的領域とスタック領域への適用

本章では，文献 [1] で述べられている，ソースコードを用いずに配列情報を復元することで MPX 命令を静的領域とスタック領域に適用する手法について述べる．以後，この手法を配列復元手法と言う．本章では，配列復元手法の概要について述べ，この手法を構成する解析段階と検知段階の詳細を述べる．

3.1 配列復元手法研究概要

配列復元手法の概要図を図 3 に示す．配列復元手法は解析段階と検知段階の 2 つの段階で構成する．解析段階ではまず解析処理 (処理 1) で防御対象バイナリコードの動的解析を行い，解析情報 (情報 1) を出力する．次に配列復元処理 (処理 2) で解析情報を基に配列情報の復元を行い，配列情報 (情報 2) を出力する．検知段階では配列情報を基に，防御対象バイナリコードへの MPX 命令の挿入処理 (処理 3) を行うことで，実行時にメモリエラーの検知が可能となる．

3.2 解析段階

図 3 の処理 1 では， 配列情報を復元するために，動的解析ツールである Intel Pin を用いてバイナリコードの動的解析を行う．動的解析では，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を取得する．そのため，各命令の前にソースオペランドの情報，各命令の後に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．

次に図 3 の処理 2 では，取得した情報から命令実行順と実行時のレジスタの値の再構築を行い，配列情報を復元する．復元する配列情報は配列にアクセスする命令，配列の開始アドレス，サイズで構成される．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，命令実行順からループを検出する．そして，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することで配列情報を復元する．配列情報の復元の設計については RINArray [8] に詳しい．この配列情報復元は既存手法 Howard [9] の手法に基づき行っている．

3.3 検知段階

図 3 の処理 3 では， 解析段階で復元した配列情報を基に MPX 適用を行う．配列復元手法では，gcc の libmpx のソースコードを参考に，同じ機能を Pin 上でバイナリコードのみを用いて再現している．まず復元した配列情報から MPX 用境界情報を作成する MPX 命令を挿入する．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 MPX 命令を挿入する．

4. 提案手法

本章では，静的領域とスタック領域の配列に対して有効な配列復元手法に基づき，malloc 関数によって確保したヒープ領域に対するソースコードの必要ない MPX 適用方法を提案する．まず，提案手法の概要について述べ，提案手法の利点について述べる．

4.1 提案手法の概要

図 4 に提案手法の概要図を示す．配列復元手法同様，解析段階と検知段階の 2 つの段階で構成する．図 4 の解析処理 (処理 1) では，防御対象バイナリコードの動的解析を行い，ヒープ領域のアドレスとサイズに関する情報 (情報 1) を出力する．次に，検知段階では解析段階で取得したヒープ領域の情報をもとに， 防御対象バイナリコードへの MPX 命令の挿入処理 (処理 2) を行うことでメモリエラーの検知を行う．提案手法は，図 3 に示した配列復元手法と違い，解析処理 (処理 1) の段階で確保したヒープ領域の情報を取得することが可能なため，配列復元処理 (図 3 の処理 2) を行う必要がない．提案手法は，配列復元手法の実装に機能を追加することで実現したため，統合が可能である．

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基に挿入を行うため，ソースコードが必要ない．そのため，バイナリコードのみでメモリ保護が可能になる．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．

```
// 境界情報
typedef struct {
  uint64_t lowerBound;
  uint64_t upperBound;
} Bnd;

// bndmk (p, size-1, 1), %bnd
// bnd に下限:p, 上限:p+size-1の境界を格納
void make_bnd(void *p, size_t size);

// bndcl p, %bnd
// p と bnd の境界の下限をチェック
void check_lowerbnd(void *p);

// bndcu p, %bnd
// p と bnd の上限をチェック
void check_upperbnd(void *p);

// bndmov bnd -> mem
// 境界レジスタからメモリへ境界情報を移す
void mov_bnd_to_mem(Bnd *bnd);

// bndmov mem -> bnd
// メモリから境界レジスタへ境界情報を移す
void mov_mem_to_bnd(Bnd *bnd);
```

図 5: MPX インラインコード

```
1  struct HeapInfo{
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 update(void *newAddr){
9    for(data ∈ 対応するHeapInfo){
10     void *start = newAddr;
11     make_bnd(start, data.size);
12     mov_bnd_to_mem(&data.bnd);
13   }
14 }
15
16 void check(Bnd *bnd, void *ref, size_t s)
17 {
18   mov_mem_to_bnd(bnd);
19   check_lowerbnd(ref);
20   check_upperbnd(ref+s);
21 }
```

図 6: ヒープ領域 MPX 適用

5. 設計

本章では提案手法の詳しい設計について述べる．まず，解析段階において確保したヒープ領域を特定する手法について述べ，次に検知段階における MPX 適用について述べる．提案手法では，ヒープ領域は libc ライブラリの malloc 関数により確保されると仮定し，その他の動的メモリ確保を行う関数を考慮した実装は今後の課題である．

5.1 ヒープ領域情報の取得

ヒープ領域に MPX を適用するために解析段階では次の情報を取得する必要がある．

1. 確保したヒープ領域の開始アドレス
2. 確保したヒープ領域のサイズ
3. 確保したヒープ領域にアクセスする命令のアドレス

これらの情報を取得するために libc ライブラリの malloc 関数に着目し，動的解析を行う．

5.2 ヒープ領域への MPX 適用

解析段階で取得したヒープ領域の情報を基に MPX 適用を行う．確保したヒープ領域のアドレスは実行ごとに変化する．そこで，実行時に malloc 関数によって確保されるたびに，ヒープ領域の開始アドレスを更新することで対応する．図 2 で示した malloc 関数のラッパ関数である__mpx_wrapper_malloc 関数を参考に設計を行う．まず，malloc 関数によるメモリ確保時に開始アドレスを取得する．なお，詳細な実装については 6.2 節で述べる．次に，対応するヒープ領域のアドレス情報を更新する．図 5 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 6 にヒープ領域に対する MPX 適用手法を示す．復元した 1 つのヒープ領域の情報は HeapInfo 構造で定義する．ヒープ領域に MPX を適用するために，まず malloc 関数の後に図 6 の 8 行目の update 関数のコードを挿入する．引数の newAddr には malloc 関数によって新たに確保したヒープ領域の開始アドレスの値を渡す．update では対応するヒープ領域の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．

次に，ヒープ領域のメモリを参照する命令の前に，図 6 の 16 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．

6. 実装

提案手法を Linux 上で Intel Pin の機能を使用し実装した．本章では実装の詳細について述べる．

6.1 Intel Pin

Intel 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 の使用を採用したのは，Valgrind [10] より高速に動作し，MPX 命令に対応しているためである．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．解析及び編集は，関数，命令などの単位で行う．

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

```
1
2  void MallocBefore(ADDRINT size)
3  {
4    // 確保したヒープ領域のサイズを記憶
5  }
6
7  void MallocAfter(ADDRINT retValue)
8  {
9    // 確保したヒープ領域の開始アドレスを記憶
10 }
11 void event_img_load(IMG img)
12 {
13   RTN libc_mallocRtn = RTN_FindByName(img, "__libc_malloc");
14   if (RTN_Valid(libc_mallocRtn)){
15     RTN_Open(libc_mallocRtn);
16     RTN_InsertCall(libc_mallocRtn, IPOINT_BEFORE,
17                    (AFUNPTR)Libc_MallocBefore,
18                    IARG_FUNCARG_ENTRYPOINT_VALUE,0,
19                    IARG_END);
20
21     RTN_InsertCall(libc_mallocRtn, IPOINT_AFTER,
22                    (AFUNPTR)Libc_MallocAfter,
23                    IARG_FUNCRET_EXITPOINT_VALUE,
24                    IARG_END);
25     RTN_Close(libc_mallocRtn);
26         }
27 }
28 int main()
29 {
30   IMG_AddInstrumentFunction(event_img_load, NULL);
31   return 0;
32 }
```

図 7: ヒープ領域情報を取得する Pin プログラム

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．

6.2 解析及び解析情報の出力

解析段階では，Pin を使用し動的解析を行うことで，ヒープ領域の情報をファイルとして出力する．図 7 にヒープ領域のアドレスとサイズを取得する Pin のプログラムを示す．ただし，簡略化のために説明に必要なコードのみを示す．

30 行目の IMG_AddInstrumentFunction 関数は各イメージファイルの編集を行うための関数を登録する．登録された 11 行目の event_img_load 関数は，各イメージファイルの情報を表す構造体 IMG が引数に渡され，コールバックされる．13 行目の RTN_FindbyName 関数より libc ライブラリの malloc 関数のオブジェクト名である__libc_malloc を探す．

16 行目の INS_InsertCall は malloc 関数の前に関数を挿入する関数である．IPOINT_BEFORE で命令の後に挿入することを示し，次の引数 MallocBefore 関数が挿入される．IARG_FUNCARG_ENTRYPOINT_VALUE は malloc 関数に渡される第一引数である取得サイズの値を引数 size に渡すことを示す．

21 行目の INS_InsertCall は malloc 関数の後に関数を挿入する関数である．IPOINT_AFTER で関数の後に挿

```
1  void at_ins(ADDRINT address, ADDRINT value)
2  {
3  // 確保したヒープ領域にアクセスしていたら情報を出力
4  // アクセスする領域を確保したmalloc 関数と対応付ける
5  }
6
7  void event_ins(INS ins, void *v)
8  {
9
10   INS_InsertCall(ins, IPOINT_BEFORE,
11                  at_ins,
12                  IARG_ADDRINT, INS_Address(ins),
13                  IARG_MEMORYVALUE_EA,
14                  IARG_END);
15 }
16
17 int main()
18 {
19   INS_AddInstrumentFunction(event_ins, 0);
20   return 0;
21 }
```

図 8: アクセス情報を取得する Pin プログラム

入することを示し，次の引数 MallocAfter 関数が挿入される．IARG_FUNCRET_EXITPOINT_VALUE は malloc 関数の返り値である取得した領域の開始アドレスの値を引数 retValue に渡すことを示す．図 7 のように設定することで malloc 関数の前に MallocBefore 関数，後に MallocAfter 関数が実行される．

次に，ヒープ領域のメモリにアクセスする命令の情報を取得する Pin のプログラムを図 8 に示す．ただし，簡略化のために説明に必要なコードのみを示す．18 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 6 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．

そして，INS_InsertCall 関数によって命令単位で各命令の前に at_ins 関数が挿入される．INS_Address(ins) は命令のアドレスを示し，IARG_MEMORYVALUE_EA はアクセスするメモリのアドレスを示す．これらのアドレスが引数として at_ins 関数に渡される．at_ins 関数では，確保したヒープ領域のメモリをアクセスするかどうか調べ，アクセスする命令の情報を出力する．そして，アクセスする領域を確保した malloc 関数と対応付ける．

6.3 MPX 適用

本節では，検知段階における MPX 適用について述べる．配列復元手法同様，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
2. MPX コンフィグレジスタの設定
3. 違反検知時の処理

表 2: 評価環境

OS	Ubuntu 16.04.3 LTS
kernel	4.4.0-116-generic
CPU	Intel(R) Core(TM) i7-6700, 3.40GHz, 8 core
memory	8GB
gcc	5.4.0
Intel Pin	3.5-97503

```
1  int main(void)
2  {
3    int i,j;
4    unsigned long loop_num;
5    int *p;
6
7    p = (int*)malloc(10*sizeof(int));
8
9    for (j=0; j<loop_num; j++){
10     for(i=0; i<10; i++)
11       p[i] = 1;
12   }
13   free(p);
14   return 0;
15 }
```

図 9: 評価プログラム (最適化なし)

libmpx ではライブラリのロード時に上記 3 つの処理を行っていたが，Pin では main 関数の前に上記 3 処理のコードを挿入する．次に MPX 命令を使用したコードを挿入する．MPX 命令は図 5 の関数を，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．そして，malloc 関数を実行した後とヒープ領域のメモリにアクセスする命令に対して，第 5.2 節で述べた処理を行う関数をそれぞれ挿入した．

7. 評価

提案手法を表 2 の環境に実装し評価を行った．まずヒープ領域の情報の取得と MPX 適用結果が正しく処理されていることかどうか確認を行った．結果は，アクセス違反があった場合，例外が発生し正しく違反を通知できることを確認した．

以降，本章では実行時間の評価について述べる．評価を行うためのプログラムを図 9 に示す．このプログラムは malloc 関数により作成された配列を一定回数参照する．malloc 関数のシンボル情報が削除されるため，最適化は行なっていない．実行時間の比較は，図 9 の loop_num の値を変更することで行った．図 10 に実行時間の比較，表 3 に各凡例の説明を示す．図 10 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ベースの実行時間を 1 として正規化した時間である．

評価結果から次のことが分かる．“ Pin+ベース ”と“ ベース ” を比べることで，Pin によるオーバヘッドが大きい．また，“ Pin+ベース ” と “ Pin(MPX)+ベース ” を比べることで，MPX 命令を挿入する時間と MPX 命令を実行する時間の占める割合は少ないことがわかる．

その他には，Pin 上で実行するプログラムではループ回

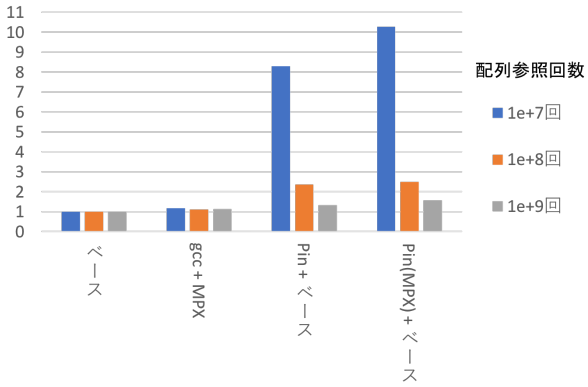

図 10: 実行時間

表 3: 凡例の説明

凡例	説明
ベース	ヒープ領域の配列に for 文でアクセスするプログラムを単体で実行
gcc+MPX	gcc により MPX 命令を適用したベースを単体で実行
Pin + ベース	ベースを Pin 上で実行．Pin では何も行わない
Pin(MPX) + ベース	提案手法 ベースを Pin 上で実行．Pin ではヒープ領域の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

数が多くなるにつれ，ベースのプログラムと比べてオーバヘッドが減少している．そのため，規模が小さいプログラムではオーバヘッドが顕著に現れるが，規模が大きくなるにつれオーバヘッドが約 1.5 倍に収束していく結果となった．

8. 課題と考察

本章では，提案手法におけるメモリエラー検知能力に関する課題と考察を述べる．

8.1 動的メモリ確保を行う関数

提案手法では，libc ライブラリの malloc 関数のみに着目して実装をした．しかし，他にも C/C++にはヒープ領域を確保する関数として calloc 関数や realloc 関数，new 演算子が存在する．これらの関数にも対応することが今後の課題である．calloc 関数と new 演算子に関しては malloc 関数と同様の実装で実現できると考える．realloc 関数に関しては，確保したヒープ領域の開始アドレスの更新だけでなく，サイズを更新する必要がある．

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，配列復元手法で

実装された静的領域とスタック領域では，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する可能性がある．この場合，復元領域外へのアクセスによって誤検知が発生する．提案手法で実装したヒープ領域に関しては，malloc 関数に引数として渡されるサイズを取得するため，すべての要素にアクセスしない場合も誤検知は発生しない．

次に，見逃しは解析段階でアクセスしなかったヒープ領域へのアクセスと，解析段階でアクセスしなかった命令でヒープ領域にアクセスした場合に発生する．これらは，様々なパターンの入力を網羅的に行うなど解析段階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，ヒープ領域に対する動的解析に基づく 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．バイナリコードを動的解析し，確保したヒープ領域のメモリ情報を取得し，MPX 命令を挿入する．本提案により，ソースコードを用いずバイナリコードのみでヒープ領域のメモリ安全を提供することができる．また，配列復元手法と組み合わせることですべての領域のメモリ安全を提供することができる．提案手法は Linux 上に実装し，オーバヘッドを確認した．

今後の課題は次のようになる．まず，検知できるヒープ領域の配列の種類を増やすことが課題である．また，オーバヘッドの削減が課題である．

参考文献

[1] 樽林秀晃, 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，研究報告コンピュータセキュリティ (CSEC)，Vol. 2017, No. 9, pp. 1–8 (2017).

[2] Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker., *USENIX Annual Technical Conference*, pp. 309–318 (2012).

[3] 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).

[4] 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).

[5] 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).

[6] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P. and Fetzer, C.: Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack, *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[7] 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).

[8] 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).

[9] Slowinska, A., Stancescu, T. and Bos, H.: Howard: A Dynamic Excavator for Reverse Engineering Data Structures., *NDSS* (2011).

[10] Nethercote, N. and Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation, *ACM SIGPLAN Notices*, Vol. 42, No. 6, pp. 89–100 (2007).