Vol.2017-DPS-170 No.24
Vol.2017-CSEC-76 No.24
2017/3/3

RINArray:配列構造復元による侵入検知システムの精度向上

樽林 秀晃[1,a)] 瀧本 栄二[2,b)] 毛利 公一[2,c)] 齋藤 彰一[1]

概要：インターネットを介してソフトウェアに不正アクセスする攻撃への対策として，未知の攻撃にも対応することが可能な異常検知型侵入検知システム (IDS) が注目されている．本論文は既存の異常検知型IDS の RIN に注目し，RIN の機能を拡張した RINArray を提案する．RIN はアセンブリ命令レベルで異常を検知することで，細粒度の検知を行うことが可能である．しかし，RIN はコンパイル前のデータ構造を利用した異常検知を行うことができない．そこで RINArray ではバイナリファイルプログラムからコンパイル前の配列を復元し，それを異常検知に用いることで RIN の検知精度を向上する．配列復元は動的解析によって行う．静的およびスタック領域の配列が復元可能である．

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

1. はじめに

インターネットの普及にともない，インターネットを利用しているソフトウェアへ不正アクセスを行う攻撃が増加している．不正アクセスを行う攻撃を検知する一般的な手法として，シグネチャ型侵入検知システム (Intrusion Detection System, IDS) がある．このシステムの検知方法は，過去に行われた攻撃から，攻撃のアクセスパターンを定義し，定義したアクセスパターンを観測すると攻撃として検知するというものである．しかし，この手法では未知の攻撃を防ぐことができないという問題が存在する．

そこで，未知の攻撃にも対抗することが可能な，異常検知型 IDS が注目されている．このシステム は，シグネチャ型とは逆の，防御対象ソフトウェアの正常な動作パターンを定義し，定義したパターンと異なる動作を観測すると攻撃として検知するという方法で攻撃を検知する．

また，ソフトウェアへの不正アクセスは，ソフトウェアのセキュリティホールを突くことで行われる．このセキュリティホールの多くはバッファオーバーフローである．バッファオーバーフローとは，ソフトウェアが確保したバッファ（配列）の領域外にアクセスするバグのことである．バッファオーバーフロー検知は，ソフトウェアをコンパイルする前のソースコードを解析することで，配列構造を検出し，配列の領域外アクセスを禁止するようにソースコードを変更することで可能である．実際，いくつかの研究論文でこのような手法が採用されている [1] [2] [3]．しかし，ソースコードが入手できない場合それらの手法は適用できない．

これらのような背景から，本論文では既存の異常検知型 IDS の RIN [4] に注目し，RIN の機能を拡張した RINArray を提案する．RIN は，ソフトウェアのバイナリファイルを動的解析し，アセンブリ命令のオペランド値を基準とした検知を行う．そのため，他の検知手法より粒度の細かい検知が可能であり，ソースコードが必要ないという利点が存在する．しかし，RIN ではバッファオーバーフローをほとんど検知できないという問題が存在する．そこで，RINArray では RIN を拡張し，配列情報を検知に用いることで，バッファオーバーフロー検知を可能とする．このとき，ソースコードが必要ないという RIN の利点を損なわないように，ソフトウェアのバイナリファイルを動的解析することで配列構造を復元する．配列は，静的領域だけでなく，スタック領域においても復元可能である．

本論文の構成を以下に示す． 2 章で本提案に関連する既存の異常検知型 IDS とデータ構造復元手法について述べ，

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, Kusatusu, Shiga, 525-8577, Japan
a) h.kurebayashi.732@nitech.jp
b) takimoto@asl.cs.ritsumei.ac.jp
c) mouri@cs.ritsumei.ac.jp

© 2017 Information Processing Society of Japan

RIN の概要と RIN の問題点を述べる．3 章で RIN の問題点を解決する RINArray を提案し，4 章で RINArray の設計を述べる．次に，5 章で詳細な実装方法を述べる．さらに 6 章で RINArray の正当性とオーバーヘッドを評価する．そして 7 章で RINArray の問題点と考察及び今後の展開について述べ，最後に 8 章でまとめる．

2. 関連研究

本章では提案手法に関連する研究について述べる．まず，異常検知型 IDS の関連研究とデータ構造を復元する関連研究について述べる．次に RINArray で使用する既存の異常検知型 IDS である RIN の概要とその問題点を述べる．最後に既存のデータ構造を復元する手法である Howard[5] について述べる．

2.1 既存の異常検知型 IDS

Wagner らの手法 [6] では，正常な動作としてシステムコールの発行順を使用する．H.H.Feng らの手法 [7] ではシステムコール発行時のコールスタックの情報を解析し，Wagner らより詳細にソフトウェアの状態を定義する．また，ClearView [8] はアセンブリ命令のオペランド値を元に正常な動作を定義する．ClearView は既存の異常な動作フローを検知する手法である Memory Firewall [9] を用い事前の知識から正常な動作を定義する．RIN は ClearView を改良した手法である．W.Feng ら [10] の手法と Eesa ら [11] の手法はネットワークログを元に異常検知を行う．W.Feng らの手法は Support Vector Machine(SVM) と Clustering based on Self-Organized Ant Colony Network(CSOACN) を組み合わせることでクラス分けを行う．Eesa らの手法は，cuttlefish algorithm(CFA) と decision tree(DT) を用いて特徴選択を行う．RIN はアセンブリ命令を基準としているのでこれらの手法より粒度の細かい手法であり精度の高い検知が可能である．

2.2 既存のデータ構造復元手法

Laika [12] 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 [13] 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 [14] はバイナリファイルを解析し Value Set Analysis(VSA) [15] と似た手法と束縛条件によって変数の型を復元する．しかしこの手法では配列情報を復元することはできない．

2.3 RIN

RIN はアセンブリ命令単位で侵入を検知する異常検知型 IDS である．ClearView では事前の知識から正常な動作を定義していたのに対し，RIN は ClearView を改良し，動的解析によりソフトウェアに合わせて自動で正常な動作を生成する．正常な動作はアセンブリ命令のオペランド値を基準としており．ソフトウェアを実行したときオペランド値が正常でなければ攻撃として検知する．このようなことから，RIN は他の検知手法より粒度が細かく精度の高い検知が可能であり，またソースコードが必要ないという利点が存在する．

図 1 に RIN の動作概要を示す．図 1 に示すように RIN は解析段階と異常検知段階の 2 つの段階で構成される．まず解析段階でソフトウェアの正常な動作であるアセンブリ命令の正常なオペランド値を定義し，次にその定義を元に，異常検知段階でソフトウェアへの攻撃を検知する．2.3.1 項と 2.3.2 項でそれぞれの段階の詳細を述べる．

2.3.1 解析段階

解析段階では防御対象ソフトウェアを動的解析し，解析結果を解析ログとして出力する．これには動的コード操作ツールである DynamoRIO [16] を使用する．解析ではソフトウェアを実行したときのアセンブリ命令のオペランド値を収集する．この解析は対象ソフトウェアへの入力を変えて複数回行う．次に，収集したデータのログから，1 命令ごとのオペランド値が満たす「動作規則」を生成する．これには，複数のデータからそのデータが満たす動作規則を生成するツールである Daikon [17] を使用する．

表 1 に解析ログとそれによって生成される動作規則の例を示す．解析はアセンブリ命令とその実行時のオペランド値を収集する．次に，それらの値を元に「mov 命令のレジスタ eax は常に 3」や「add 命令の eax は常に 3 かつ eax は ebx 以下」といった動作規則が生成される．

2.3.2 異常検知段階

異常検知段階では DynamoRIO 上で防御対象ソフトウェアを実行する．このとき，「解析段階で作成した動作規則を満たしていなければ異常を検知する」というコードを元のコードに挿入することで異常検知を行う．コードの挿入は DynamoRIO を使用する．

2.4 RIN の問題点

RIN の問題点の 1 つとしてバッファオーバーフローをほとんど検知できない点が挙げられる．バッファオーバーフローを検知できない原因については 7.1 節で詳しく述べるが，これは 1 つの命令のみを見て異常検知を行っているためである．そこで RINArray ではこの問題を解決する手法を提案する．

2.5 Howard

Howard [5] はバイナリファイルのプログラムから，コンパイル前のデータ構造を復元する手法である．Howard では既存のデータ構造復元手法を組み合わせることでも復元

© 2017 Information Processing Society of Japan

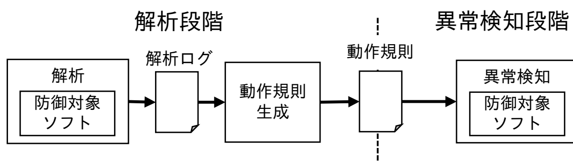

図 1: RIN 動作概要

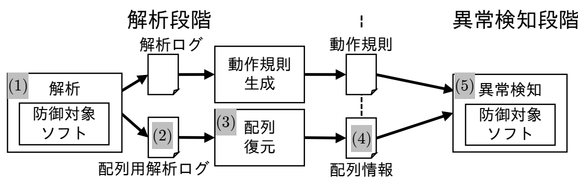

図 2: RINArray 動作概要

表 1: 解析ログと動作規則

アセンブリ命令	mov $3,%eax	add %eax,%ebx
実行 1 回目	eax = 3	eax = 3, ebx = 5
実行 2 回目	eax = 3	eax = 3, ebx = 8
...	...	...
実行 N 回目	eax = 3	eax = 3, ebx = 7
動作規則	eax == 3	eax == 3, eax $\leq$ ebx

を行っているが，Howard 独自の手法として，メモリへのアクセスパターンを元に復元するという方式がある．この手法では，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している．これらの手法により 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．RINArray では Howard の手法を参考に配列を復元する．

3. 提案手法

本論文では，配列構造を復元することによる RIN の精度を向上した RINArray を提案する．バイナリファイルを動的解析することで配列の情報を復元し，その情報を検知に用いることでバッファオーバーフローの検知を可能にする．

図 2 に RINArray の動作概要を示す．(1) 解析部分を拡張し，(2) 配列を復元するために必要な情報を追加でログ出力する．次に (3) そのログを元に配列情報を復元し，(4) 新たに配列情報ログを生成する．最後に (5) 異常検知段階で既存の動作規則と配列情報の両方を用いて異常検知を行う．

3.1 提案手法の利点

提案手法では既存の RIN に配列の情報を加えて異常検知を行うため，バッファオーバーフローを検知することが可能となり，RIN より精度の高い検知を行うことが可能である．また，バイナリファイルのコードから配列の情報を復元することで，ソースコードが必要ないという RIN の利点を保ったまま異常検知が可能である．

3.2 要件定義

ここでは提案手法を実現する上で必要な技術手法について述べる．

3.2.1 配列の復元

バイナリファイルのコードは，配列のサイズや要素数といった配列の情報を失っている．そのため，動的解析で得られるアセンブリ命令の順番やオペランド値などの限られた情報のみを使用し配列を復元しなければならない．

3.2.2 配列情報を用いたバッファオーバーフロー検知

既存の RIN では配列情報は扱うことができないため，専用のバッファオーバーフロー検知手法が必要となる．このときオーバーヘッドを削減するため，解析段階で得られた情報を元に最小限の実装で異常検知を行う必要がある．

3.2.3 スタック領域の配列

静的領域の配列の場合は，配列の存在するアドレスが不変であるが，スタック領域にある配列はアドレスが実行毎に変わるため，実装を工夫する必要がある．

4. 設計

本章では 3.2 節で述べた要件を満たすための設計方針を述べる．まず，静的領域の配列についての設計を述べた後，スタック領域配列の設計を述べる．

4.1 配列の復元

バッファオーバーフローの検知は，配列を参照する命令の前に「配列開始アドレス $\leq$ 参照アドレス $\leq$ 配列終了アドレス」の条件が成立するか否かチェックするコードを挿入する方針で行う．よって検知に必要な情報は次の 3 つである．

1. 配列にアクセスする命令
2. 配列の開始アドレス
3. 配列の終了アドレス

解析段階でこれらの情報を復元する．これは Howard の手法を用いて行う．Howard は，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. ベースアドレスの取得:ポインタ計算が行われる前のポインタの値を取得
4. 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

本節ではまず，上記手順を実現する上で重要な概念である Basic Block について述べた後，上記手順の詳細を述

```
FuncDatabase funcDB;
Stack bbStack, calls;
for(bb ∈ すべてのBasic Block){
  bbStack.push(bb);
  instr = bb の最後の命令;
  if(instr は call 命令) calls.push(instr);
  else if(instr はレット命令){
    call = calls 中から ret と対応する call 命令を検索();
    Function func;
    while(1){ // call の直前までスタックからポップし関数とする
      topBb = bbStack.peek();
      if(topBb は call のある Basic Block)break;
      func.add(bbStack.pop());
    }
    funcDB.add(func);
  }
}
```

図 3: 関数検出

```
LoopDatabase loopDB;
Stack instrStack;
for(bb ∈ 関数中のすべてのBasic Block){
  instrStack に bb の命令をすべてプッシュ();
  instr = bb の最後の命令;
  if(instr が後方分岐でない) continue;
  Loop loop;
  while(instrStack が空でない){
    topInstr = instrStack.peek();
    if(topInstr のアドレス < instr の分岐先のアドレス) break;
    loop.add(instrStack.pop());
  }
  loopDB.add(loop);
}
```

図 4: ループ検出

べる．

4.1.1 Basic Block

Basic Block とは，実行した命令を，call,ret,jmp といったジャンプする命令で区切ったものである．動的解析により実行順に Basic Block を並べ，Basic Block による実行列を作成する．このデータ構造により，命令の検索計算量を削減することができる．

4.1.2 関数の検出

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．

図 3 に関数検出方法を示す．まず，検出した関数のリストを記憶する funcDB を用意する．検出した関数は，関数中で実行した Basic Block のリストで表され，図 3 では Function という名前で定義する．次に Basic Block と call 命令を記憶するスタック bbStack と calls を用意し，実行した Basic Block と call 命令をそれらのスタックに積む．ret 命令を実行したとき，bbStack の先頭から ret 命令に対応する call 命令直前までが 1 つの関数となる．ここで対応とは ret 命令のジャンプ先が call 命令の次の命令となることを指す．また，Basic Block の性質から，call 命令と ret 命令は Basic Block の最後にあるため，Basic Block の最後のみを見て判断することでオーバーヘッドを削減することができる．

4.1.3 ループの検出

関数の検知が終わると，次に関数の中にあるループ部分を検出する．検出は後方分岐ジャンプ命令を元に行う．**図 4** にループ検出方法を示す．まず検出したループのリストを記憶する loopDB を用意する．検出したループは，ループ中で実行した命令のリストで表され，図 4 では Loop という名前で定義する．次に，実行した命令を記憶するスタック instrStack を用意し，実行した命令を順に積む．次に，実行した Basic Block の最後の命令を順に調べる．その結果，最後の命令がジャンプ命令であり，かつジャンプ先アドレスがジャンプ元アドレスより前にあった場合（ジャンプが後方分岐の場合），ループとして検知する．instrStack 中で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 つのループとする．

4.1.4 ベースアドレスの取得

ベースアドレスとは，メモリ参照のアドレス計算が行われる前のアドレスである．**図 5** にループで配列を参照するソースコードとアセンブリコードを示す．**図 5b** の 5 行目では ebx の値をアドレスとしてメモリを参照している．この ebx には ptr2+i の計算結果が格納されている．このときの ptr2 の値がベースアドレスである．配列を検出するには，このようにレジスタの値をメモリアドレスとしての参照（メモリ間接参照）が起こったとき，ベースアドレスを求める必要がある．

メモリ間接参照は一般に次の流れで起こる．まず図 5b の 1 行目で ptr2 の値がレジスタ ebx にロードされているように，ベースアドレスがあるレジスタ reg1 にロードされる．次に ptr2+i といったようなアドレス計算が行われる．ここでアドレス計算はレジスタのみを使用すると仮定する．アドレスは，4 行目で eax(ptr2) と ebx(i) の和を ebx に格納しているように，reg1 に何か計算を施しその結果を reg2 に格納することで計算される．複雑なアドレス計算ではさらに reg2 に何か計算を施しその結果を reg3 に格納というように，$reg1 \rightarrow reg2 \ldots \rightarrow regN$ と計算結果をレジスタからレジスタへ受け渡すことで行われる．最後に 5 行目で ebx の値をメモリアドレスとして参照しているように，最終的なアドレスが格納されたレジスタの値をメモリアドレスとして参照する． 以上によりメモリ間接参照は，(1) レジスタへベースアドレスのロード →(2) アドレス計算 →(3) メモリ参照，という流れで行われる．そこで (1) のベースアドレスを (3) のレジスタ参照が行われるまで記憶しておく．

図 6 にベースアドレスの求め方を，**図 7** にそれを図示した内容を示す．メモリ間接参照が起こった命令のベースアドレスを求めたいので，キーが命令で値がベースアドレスの連想配列 baseAddr を定義する．またアドレス計算中にベースアドレスを記憶するために，キーがレジスタ名で値がベースアドレスの regBase を定義する．そして，まず (1) でレジスタへメモリアドレスがロードされたとき，図 6 のコメント (1) のようにレジスタのベースアドレスを

© 2017 Information Processing Society of Japan

```
int array[10], x, i;
int *ptr, *ptr2;

ptr = array;
ptr2 = array;
i=0;
while(i<10){
  x = *(ptr++); // 1
  x = ptr2[i++]; // 2
}
```

(a) ソースコード

```
1 mov    ptr2, %eax
2 mov    i, %ebx
3 sal    $2, %ebx
4 add    %eax, %ebx
5 mov    (%ebx), %ebx
6 mov    %ebx, x
```

(b) コメント 2 部分のアセンブリコード

図 5: ループで配列を参照するコード

```
struct StaticInfo{ // 静的領域
  addr;   // 命令のあるアドレス
  start;  // 配列の開始アドレス
  end;    // 配列の終了アドレス
};
struct StackInfo{// スタック領域
  addr;
  start; // 配列の開始アドレスからのオフセット
  end;   // 配列の終了アドレスからのオフセット
  func;  // 関数開始アドレス
};
```

図 8: 配列情報

```
Hash regBase, baseAddr;
for(instr ∈ すべての命令){
  if(instr はメモリからレジスタ reg へ値 value をロード){//(1)
    regBase[reg] = value;
  }
  else if(instr はレジスタ reg1 から
         レジスタreg2 への受け渡し){//(2)
    regBase[reg2] = regBase[reg1];
  }
  if(instr でレジスタ reg を参照){//(3)
    baseAddr[instr] = regBase[reg];
  }
}
```

図 6: ベースアドレスの計算

```
struct ArrayRule{
  addr; // 命令のあるアドレス
  start; // 配列開始アドレス
  end; // 配列終了アドレス
};

//struct ArrayRule *rule = その命令で参照する配列の規則
if(!(rule->start ≤ 参照アドレス ≤ rule->end)) 異常検知 ();
```

図 9: 異常検知コード

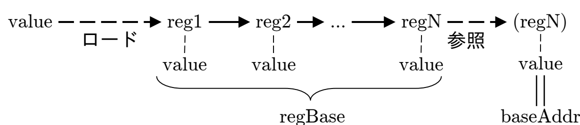

図 7: ベースアドレスの計算図

```
for(info ∈ すべての StaticInfo){
  rule.addr = info.addr;
  rule.start = info.start;
  rule.end = info.end;
  rule をデータベースに登録();
}
```

図 10: 静的領域の配列情報更新

記憶する．次に (2) でレジスタからレジスタへ受け渡しが行われたとき，図 6 のコメント (2) のようにベースアドレスも受け渡していく．ここでは図 7 に示すようにアドレス計算は reg1 からレジスタを受け渡されていき，regN に最終的なアドレスが格納されたとする．この処理によって regBase[reg1] から regBase[regN] はすべて value となる．最後に (3) でレジスタを参照したとき，図 7 のコメント (3) のように regBase[regN] が求めたいベースアドレスとなる．この処理によって図 7 に示すように baseAddr[instr] は最初にロードした value と同じ値になる．

4.1.5 配列の検出

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 (図 **5a** のコメント 1 部分)
2. `elem=array[i]` パターン (同コメント 2 部分)

`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，前回のループと今回のループでアクセスしているベースアドレスの差を調べ，その結果ベースアドレスの差がすべて同じならこのパターンとする．このとき配列の開始アドレスと終了アドレスはアクセスした中で最小と最大のアドレスとする．

`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．配列情報の求め方は `elem=*(ptr++)` パターンと同じである．

4.2 配列情報を用いたバッファオーバーフロー検知

解析段階で復元した配列情報を元に異常検知段階でバッファオーバーフロー検知を行う．この検知は配列を参照する命令の前に配列の範囲内にアクセスしているかチェックする異常検知コードを挿入することで行う．このとき，挿入するコードは最小限にする必要がある．

解析段階で復元した静的領域の配列情報を図 **8** の StaticInfo のように定義する．また，挿入するコードの情報を図 **9** の ArrayRule のように定義する．静的領域の場合はアドレスが不変のため，プログラムの起動時に図 **10** のように，StaticInfo から ArrayRule を決定する．次に異常検知コードを挿入する．挿入手順はまず 1 回目に命令を実行したときその命令で参照する配列の情報を検索する．検索ではすべての ArrayRule の中から命令のアドレスと ArrayRule.addr が一致するものを選択する．次に，その命令の直前に図 9 の最終行に示すように「もし命令で参照したアドレスが配列開始アドレスから配列終了アドレスの範囲内でないならば異常を検知する」というコードを挿入する．2 回目以降は rule が決定しているためこの ArrayRule 検索と挿入処理は必要ない．

© 2017 Information Processing Society of Japan

```
Stack callStack;
if(関数が始まった){
  StackFrame frame;
  frame.ebp = 現在のebp;
  frame.esp = 現在のesp;
  frame.funcAddr =
    始まった関数の開始アドレス;
  callStack.push(frame);
}
else if(関数が終了した){
  callStack.pop();
}
```

(a) コールスタック構築

```
ary = 検出した配列;
detected = false;
for(frame ∈ callStack){
  esp = frame.esp;
  ebp = frame.ebp;
  start = ary の開始アドレス;
  if(esp ≤ start ≤ ebp){
    ary は frame に属する;
    detected = true;
  }
}
if(!detected){
  ary は静的領域;
}
```

(b) 領域検出

図 11: 配列の領域検出

```
for(info ∈ すべてのStackInfo){
  rule = info に対応する ArrayRule を検索(info);
  if(info.func != 現在の関数の開始アドレス) continue;
  ebp = 現在の関数のebp;
  rule.start = epb + info.start;
  rule.end = ebp + info.end;
}
```

図 12: スタック領域の配列情報更新

4.3 スタック領域の配列

スタック領域へのアクセス方法は `mov $1,-0x4(%ebp)` のように ebp レジスタからのオフセットを使用するもののみとする．他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，ebp レジスタからのオフセットは不変となる．

4.3.1 スタック領域の配列の復元

解析段階でスタック領域の配列を復元するためには，4.1 節で述べた配列の情報に加え，次の 2 つの情報が必要となる．

1. その配列が確保された関数
2. その関数の ebp から，配列開始と終了アドレスまでのオフセット

そこで，スタック領域の配列を復元するために，実行中の関数とそのスタックフレームの位置を記憶し，配列を検出したときにその配列のアドレスがどのスタックフレームに属するかを調べる．

図 11 にスタック領域の配列復元方法を示す．まず，**図 11a** のようにスタックフレームの位置を記憶するためのスタック callStack を用意する．このスタックは実際のコールスタックと同じように，関数が始まるときプッシュし関数が終わるときポップする．記憶する内容は関数ごとの ebp と esp の値である．この処理によってスタック領域の配列はこの callStack 中の stackFrame のどれかに属することになる．よって，**図 11b** のように，ある配列を検出したときこの callStack をすべて調べ，配列の開始アドレスが ebp から esp の間にあるか調べる．その結果，配列の開始アドレスが ebp から esp の間にあったスタックフレームがその配列が属するスタックフレームとなる．関数の ebp から配列開始と終了アドレスまでのオフセットは，配列の属するスタックフレームの ebp と配列の開始と終了アドレスの差から求めることができる．

4.3.2 スタック領域の配列の異常検知

スタック領域の配列の異常検知で挿入するコードは静的領域と同じように図 9 のコードを挿入する．しかし，配列の開始と終了アドレスは関数が実行されるたびに変化するため，関数が実行されるたびに配列情報を更新する．まず，解析段階で復元したスタック領域の配列情報を図 8 の StackInfo のように定義する．次に異常検知で関数が実行されたときに，**図 12** のように StackInfo の中から func が現在の関数の開始アドレスと一致するものを検索し，関数の ebp から配列の開始と終了アドレスを更新する．

5. 実装

提案システムを Linux 上で実装した．図 2 の (1) と (5) は，RIN と同様に DynamoRIO の機能で実現した．また，同図の (3) は Java で実装した．

5.1 配列用解析ログの出力

本節は図 2 の (1) と (2) について述べる．配列用解析ログの出力には DynamoRIO のコード解析機能を用いた．DynamoRIO 上で防御対象ソフトウェアを実行し， 4.1 節で述べた必要な情報を出力する．出力する情報は，実行したアセンブリ命令，命令があるアドレス，命令で使用したレジスタ，メモリアドレス，レジスタの値である．また，DynamoRIO は Basic Block を単位に処理が実行されるので，Basic Block の最初の命令の前に，Basic Block の始まりを示す文字を出力する．

5.2 解析ログの読み込みと配列復元

本節は図 2 の (3) と (4) について述べる．(2) のログを読み込み，命令と Basic Block の構造を再構築する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ログを出力する．ログは図 8 に示すデータ構造で出力する．静的領域の配列は，配列を参照する命令のアドレス及び配列の開始と終了アドレスである．スタック領域の配列は上記の情報に加え，配列が属する関数の開始アドレスを出力する．ただし，配列の開始と終了アドレスは関数の ebp からのオフセットを出力する．

5.3 異常検知

本節は図 2 の (5) について述べる．配列情報ログは図 8 に示すデータ構造として読み込まれ，そのログを元に異常検知を行う．異常検知には DynamoRIO のコード編集機能を用い，元のコードに異常検知用コードを挿入する．このとき，元のコードを破壊しないように，挿入するコードの最初に使用するレジスタを保存し，挿入するコードの最後

© 2017 Information Processing Society of Japan

表 2: 評価環境

OS	Ubuntu 14.04.3 LTS
kernel	3.13.0-74-generic
CPU	Core i5, 2.80GHz, 4 core
memory	4GB

```
for(i=0; i<num; i++){
  array[i] = 123;
}
```

図 13: 正当性評価

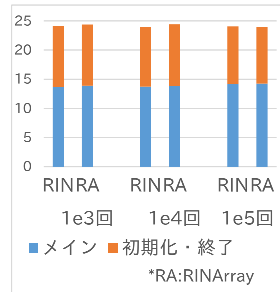

図 14: オーバーヘッド評価

でレジスタを復帰する処理を行う．

6. 評価

本章では実装した異常検知システムの動作の正当性とオーバーヘッドを評価する．**表 2** に評価環境を示す．

6.1 配列復元の正当性

配列情報が正しく復元できていることを確認するために，配列を参照するコードを作成し，動的解析及び配列復元を行った．作成したコードは図 5a のように配列に `elem=*(ptr++)` パターンと `elem=array[i]` パターンでアクセスするもので，配列 array は静的領域のものとスタック領域のもの 2 種類作成した．検証の結果， 4.1 節の配列情報が正しく復元できていることを確認した．

6.2 異常検知の正当性

誤検知及び見逃しが起こらないことを確認するために，配列の範囲内にアクセスする場合と配列の範囲外にアクセスする場合両方について異常検知を行った．作成したコードは**図 13** のようにループ回数を指定し配列にアクセスするコードである．このコードを用い，解析段階ではバッファオーバーフローが起こらないように，異常検知段階ではバッファオーバーフローが起こるようにループ変数 (num) を変更し，評価を行った．検証の結果，誤検知及び見逃し双方とも起こらないことを確認した．

6.3 オーバーヘッド

オーバーヘッドを測定するために，RIN と RINArray をそれぞれ用いたプログラムと，どちらも使用しないプログラム（以下ベースプログラム）の 3 つについて解析及び異常検知時間を計測した．プログラムではループで配列にアクセスし，ループ回数を変更して評価を行った．**図 14** に評価結果を示す．ベースプログラムの時間を 1 に正規化してあり，ファイル読み込みや領域解放などの初期化と終了処理と，それ以外のメインの処理とに分けて計測した．計測時間は 1000 回の平均である．結果からループ回数に関わらず約 25 倍の時間がかかること，また RIN に比べ RINArray のオーバーヘッドはわずかであることが分かる．

7. 課題と考察

本章では RINArray における限界や問題点などの課題とその他考察について述べる．

7.1 RIN と RINArray の精度

本節では RIN と RINArray の精度の違いを述べる．

7.1.1 バッファオーバーフロー見逃し

RIN ではバッファオーバーフローの見逃しが起こる可能性が高い．このことについて次の 2 つの側面から述べる．

1. 配列を参照する命令における動作規則
2. ループ条件分岐部分の cmp における動作規則

まず，配列を参照する命令において，バッファオーバーフローを正確に検知するには「配列の開始アドレス $\leq$ 参照アドレス $\leq$ 配列の終了アドレス」という動作規則が必要である．しかし，RIN では即値を使用した不等式を作成できないためこの動作規則は作成できない．つまり配列を参照する命令部分でバッファオーバーフローを検知することは不可能である．

次にループで配列にアクセスしているとき，`while(i<num)ary[i++]` の i<num 部分のようにループの条件分岐部分の動作規則でバッファオーバーフローが検知できるかを考える．この部分の命令は「cmp src0 src1」のように cmp での比較を行っており，src0 と src1 には i と num の値が入る．このことからバッファオーバーフローを検知するには「src0 $<$ src1」という動作規則が必要となる．この規則は即値ではなくオペランドを使用した不等式なのでこの動作規則を RIN が作成することは可能である．しかし，最後のループ (num+1 回目のループ) の cmp 部分では「i=num+1(src0 $>$ src1)」となるため「src0 <src1」の条件は満たさず，「src0 $<$ src1」という動作規則は生成されない．つまりループ条件分岐部分の cmp における動作規則においてもバッファオーバーフローを検知することは不可能である．

一方 RINArray では配列を復元しているため，配列の範囲内かチェックする規則を生成可能であり，バッファオーバーフローを見逃す確率を減少させることができる．また，RINArray では既存の RIN と同じ検知も行っているため，RIN より確かに見逃しを減少できていると言える．

7.1.2 誤検知

RINArray では正しく配列を復元できない場合もあり，そのときは誤検知が発生する．この誤検知は RIN では起こらない問題であるため，RINArray は RIN より誤検知の数が多くなる場合がある．例えば，要素数 10 の配列に，解

© 2017 Information Processing Society of Japan

析段階は配列の 0 から 8 番目の要素のみにアクセスし，異常検知で配列の 9 番目の要素にアクセスすると RINArray はバッファオーバーフローを誤検知する．

7.2 入力依存の問題

RINArray での配列復元が正しいのは，配列のすべての要素にアクセスした場合に限られる．入力によってはこの条件を満たさない場合もあるため，より復元精度を高める入力を決定する手法を考える必要がある．

7.3 復元可能な配列の種類

現在復元可能な配列の種類は限られている．まず，ヒープ領域にある配列の復元及び異常検知は行うことができない．これは今後の課題である．また，現在は 1 次元配列のみ復元と異常検知が可能であり，多次元配列は扱うことができない．しかし，これは入れ子になったループを検知することで可能であると考えている．

7.4 オーバーヘッド

現段階では，異常検知に大きなオーバーヘッドが存在するため，実際の運用には耐えられず，オーバーヘッドを削減する方法を考える必要がある．オーバーヘッドを削減する 1 つの方法として，現在は異常検知を行うために「異常検知を行う関数」を挿入しているが，関数ではなく必要最低限の命令のみ挿入することを考えている．

8. まとめ

既存の異常検知型 IDS の RIN の精度を向上した RINArray を提案した．既存の RIN ではアセンブリ命令のみを見て動作規則を作成しているため，バッファオーバーフローを検知できない問題があった．そのため RINArray では，提案手法ではバイナリファイルからソースコードの配列の情報を復元し，配列の情報を用いて異常検知を行った．このことにより，既存の RIN より高精度の検知を実現することが可能である．

次に提案手法を Linux 上に実装し，配列の復元と異常検知の正当性とオーバーヘッドを確認した．現段階では，elem=*(ptr++) パターン及び elem=array[i] パターンの配列の復元，静的領域とスタック領域の配列の復元を行うことが可能である．

参考文献

[1] Necula, G. C., Condit, J., Harren, M., McPeak, S. and Weimer, W.: CCured: Type-safe Retrofitting of Legacy Software, *ACM Trans. Program. Lang. Syst.*, Vol. 27, No. 3, pp. 477–526 (2005).

[2] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J. and Wang, Y.: Cyclone: A Safe Dialect of C, *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, Berkeley, CA, USA, USENIX Association, pp. 275–288 (2002).

[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] Shirai, H., Saito, S., Mouri, K. and Matsuo, H.: Monitoring Instruction-based Intrusion Detection and Self-healing System, *Proceedings of the International MultiConference of Engineers and Computer Scientists*, Vol. 1 (2012).

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

[6] Wagner, D. and Dean, D.: Interusion Detection via Static Analysis, *IEEE Symposium on Security and Privacy*, pp. 144–155 (2001).

[7] H.H.Feng, O.M.Kolesnikov, P.Fogla, W.Lee and W.Gong: Anomaly Detection Using Call Stack Information, IEEE Symposium on Security and Privacy，Berkeley，CA，pp. 62–77 (2003).

[8] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S. P., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., fai Wong, W., Zibin, Y., Ernst, M. D. and Rinard, M. C.: Automatically patching errors in deployed software, *Symposium on Operating Systems Principles*, pp. 87–102 (2009).

[9] Kiriansky, V., Bruening, D. and Amarasinghe, S.: Secure execution via program shepherding, *Proceedings of the 11th USENIX security symposium*, pp. 191–206 (2002).

[10] Feng, W., Zhang, Q., Hu, G. and Huang, J. X.: Mining network data for intrusion detection through combining SVMs with ant colony networks, *Future Generation Computer Systems*, Vol. 37, pp. 127–140 (2014).

[11] Eesa, A. S., Orman, Z. and Brifcani, A. M. A.: A novel feature-selection approach based on the cuttlefish optimization algorithm for intrusion detection systems, *Expert Systems with Applications*, Vol. 42, No. 5, pp. 2670–2679 (2015).

[12] Cozzie, A., Stratton, F., Xue, H. and King, S. T.: Digging for Data Structures., *OSDI*, Vol. 8, pp. 255–266 (2008).

[13] 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, p. 5 (2010).

[14] Lee, J., Avgerinos, T. and Brumley, D.: TIE: Principled reverse engineering of types in binary programs (2011).

[15] 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, p. 23 (2010).

[16] Bruening, D.: Efficient, transparent, and comprehensive runtime code manipulation, PhD Thesis, Citeseer (2004).

[17] Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M. and Xiao, C.: The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming*, Vol. 69, No. 1-3, pp. 35–45 (2007).

© 2017 Information Processing Society of Japan