添字変数の制約条件の導出によるバッファオーバーランの理解支援

2019SE032 眞弓遥紀　2019SE034 宮川頌之佑
指導教員：吉田　敦

1　はじめに

プログラムの脆弱性を理解するためには，基礎的なプログラミング学習の段階で脆弱性を学ぶ機会が必要であるが，基礎的なプログラミング学習では考慮しない．これは，プログラミング学習においてプログラムの処理や構造の理解を優先するためである．また，たとえ脆弱性が含まれていても，プログラミング学習では重要なデータベースやシステムを扱うことがなく，情報漏洩などの致命的な影響を受けることはないことも理由として挙げられる．

脆弱性には様々な種類が存在するが，その代表的なものがメモリの領域外を参照することで発生するバッファオーバーランである．プログラミング学習では特に配列の誤用によって発生する．配列に起因するバッファオーバーランの例として，プログラム 1 を示す．このプログラムは文字列 src から文字列 dst に n 文字コピーする関数である．この関数の仕様では，文字列 src を文字列 dst の大きさまでの最大 n 文字をコピーする．ここで n は配列 dst と src の大きさ以下，すなわち，n ≦ min(size(dst),size(src)) となる．なお，ここでは，size() は配列の大きさを表すものとする．しかし実際には，n=size(src) のときに 6 行目の src[i] の参照で，n=size(dst) のときに 9 行目の dst[i] への代入で配列の領域外を参照し，バッファオーバーランが起こる．このような境界に関わる条件は，添字変数が取りうる値の範囲を正確に理解しないと，読み取ることができない．ここでバッファオーバーランを理解するためには，それを引き起こす入力例を考える必要がある．バッファオーバーランが起こらない条件は n ≦ min(size(dst)-1, size(src)-1) であり，これと関数の仕様との差を考えるとバッファオーバーランを起こす条件がわかり，具体的な入力例も想定できる．このように，バッファオーバーランしない添字変数の条件や仮引数の条件が分かれば，脆弱性を起こす箇所や，入力例の理解につながる．

本論文では，バッファオーバーランの理解支援のために，バッファオーバーランが起こらないための添字変数の取りうる範囲と，そこから得られる仮引数の条件を導出する手法を提案する．解析にあたっては、配列の参照式において，添字が満たすべき値の範囲についての制約条件を求め，制御フローグラフ (CFG) 上を辿りながら，それより前に評価される式において，その条件を満足させるための条件を求めていく．これにより，すべての添字変数の出現について制約条件が求まり，出現間の制約条件の変化から添字変数の値の変化を理解できる．さらに，添字変数と仮引数を含む条件式から，仮引数の制約条件が求まり，バッファオーバーランを引き起す入力例の理解を助ける．また，各出現ごとの制約条件を把握しやすくするために，ソースコード内にコメントとして条件式を挿入し提示する．

プログラム 1　脆弱性を含むプログラム

```
1  char *copy1(char dst[], char src[], int n) {
2
3    int i;
4    for (i = 0; src[i] != '\0' && i < n; i++) {
5      dst[i] = src[i];
6    }
7    dst[i] = '\0';
8    return dst;
9  }
```

2　関連研究・技術

Cppcheck[1] は静的解析を行いプログラムに含まれる脆弱性を検出するツールである．このツールは引数次第でバッファオーバーランが発生するか変わるような，潜在的な脆弱性は検出できない．また，脆弱性は検出するがプログラミング学習において，脆弱性を回避するために単にそのコード片を削除すればよいという訳ではない．仕様を維持したまま脆弱性の存在しないプログラムへ修正するには，脆弱性の原因を学習者に気づかせ，学習させることが必要である．

Vinod らは，線形計画法の考え方によるバッファオーバーランの検出方法を提案している [2]．記号実行による静的解析で検出する点で類似しているが，解析アルゴリズムが線形計画法に基づいている点と，バッファオーバーランしない条件ではなく，起こる条件を導出している点で本論文の提案する方法とは異なる．

3　添字変数の制約条件導出方法の提案

3.1　処理の全体の流れ

提案する制約条件の導出方法は次の通りである．図 1 は提案する添字変数の制約条件を導出する流れの全体像であり，これらの処理は図 1 の数字の処理に対応する．

- CFG 上を辿りながら，添字変数が取りうる値の制約条件を導出 (2, 3)
- 仮引数の値の制約条件を導出 (3)
- 制約条件をソースコードに挿入 (4)

CFG 上のすべての実行経路に関して，配列の参照式で与えられる添字変数の制約条件が満たされるために，その参照式より前に評価する式についての制約条件を求めることを再帰的に繰り返すので，CFG の実行経路を逆順に辿る．また，解析では添字変数であるかを判定しないが，配列の参照式で添字である変数のみを対象とするので，添字変数についての制約条件を求める．すべての実行経路について制約の導出を行うために，実行経路上を逆順に移動するオ

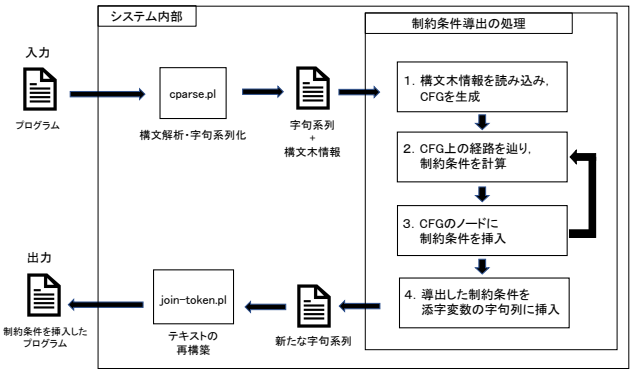

図 1 提案する制約条件導出方法の全体の流れ

ブジェクト (以下, Visitor とする) を用意する [3]．Visitor は変数とその制約条件をまとめた表を持ち，CFG 上を辿りながら各式で制約に関わる影響を受ける変数に対して表を更新していく．CFG の最終ノードから出発し，分岐では複製を作り各経路を進む．Visitor が解析で導出した変数の制約条件を CFG のノードにも記録する．分岐で複製した Visitor が後からそのノードを通過したとき，記録されている制約条件も使用して新たな制約条件を求め，表を更新する．

実行経路数が無限になる場合，Visitor に複製されるので，それを抑制する必要がある．Visitor は各ノードに自身の状態 (表と辿った経路) を記録し，後から通過した Visitor が同じ状態ならばそれ以降の更新結果は変化しないので，解析を終了する．あるノードに記録されている制約条件より狭い条件を持つ Visitor が到達したとき，広い制約条件でバッファオーバーランが起こらないことが得られているので，Visitor の表は広い制約条件に更新する．これにより，単純な繰り返しは Visitor が 2 回ノードを通過することで広い制約条件に収束するので，3 回目の Visitor は同じ状態で通過し，Visitor は停止する．これらから，実行経路数が無限であっても Visitor が無限に複製されず，有限個の経路の計算に抑える．

バッファオーバーランが引き起こす入力例の理解のために，バッファオーバーランが起こらない仮引数の制約条件を導出する．添字条件と仮引数を含む条件式で，仮引数に関わる添字変数の制約条件を使って仮引数の制約条件を導出する．

導出した添字変数の制約条件を把握しやすくするために，ソースコード上に制約条件を提示する．CFG のノードに記録した制約条件をそのノードの添字変数の字句列に埋め込むことで，字句列からソースコードに戻したときにソースコードの添字変数の箇所に生じる制約条件をコメントとして提示する．

3.2 前提条件

問題を簡単にするために，初級の典型的なプログラムを対象とする．

- プログラムは脆弱性が含まれている可能性がある
- 分岐構造は if 文, 繰り返し構造は for 文もしくは while 文が使用されている
- 複数の添字変数が同時に被演算子として出現しない
- 配列の要素を指すポインタを含まない
- 配列の添字式は添字変数のみ
- 変数に代入を行う副作用を持つ関数の呼出しを含まない
- 添字変数の変化はインクリメント，デクリメントのみ

3.3 更新ルール

Visitor が各ノードで行う更新のルールについて説明する．更新ルールは，Visitor が CFG の解析し，添字変数の制約を導出するために，制約の表を更新し辿っている経路における制約を求める計算 (表 1) と，添字変数がすべての経路で発生した制約を満たすために，CFG のノードに記録された制約と Visitor の表の制約の合成による更新 (表 2) を行う．更新ルールは配列の参照式と添字変数のインクリメント，デクリメントを対象に適用する．

Visitor が持つ制約の表は各変数ごとに最大値と最小値を持つ．制約の中では配列の大きさを size() で表現し，制約を更新するときは記号計算を行う．また，制約がないことを表現するために，プラス無限大とマイナス無限大を使用する．

1 本の経路から制約を求めるために，Visitor が持つ制約の表の最大値と最小値を表 1 に基づいて更新を行う．変数が初めて配列の添字としてノードに出現したとき，表にその添字変数名と最大値に size(配列名) - 1 , 最小値に 0 を登録し，それ以降，表の添字変数に対して Visitor の表を更新しながら制約を求める．Visitor の持つ制約条件の表は配列ごとに独立に求め，最終的にはすべての配列でバッファオーバーランが発生しない制約になる．

Visitor が辿りながら導出した制約を CFG の添字変数のノードに記録ながら解析を行う．ある経路における添字変数の導出した制約と他の経路で導出した制約を合成し，すべての経路で発生する制約を満たすために，配列の添字変数の制約は表 2 の計算で合成する．

仮引数の制約条件は初期化式と条件式に含まれる添字変数の制約条件から導出する．条件式に到達したときに辿った経路を True と False にわけ，添字変数と比較する変数の大小関係から表 3 と表 4 のように添字変数の制約を記録する．なお，等号がつく場合も同様である．True と False の経路とは条件式を判定したときにそれぞれ真と偽を返した実行経路を辿ってきた Visitor の経路であり，これらを分けるのは条件式で異なる判定をした経路において添字変数に対し，逆の条件が発生するからである．ここで得た制約条件を使い，添字変数と仮引数の関係から表 5 のように仮引数に対して制約条件を導出する．

初期化式における制約条件は表 2 の更新ルールでノード

表 1　Visitor の持つ添字変数 i の表の最小値 (L)・最大値 (U) の更新

出現パターン	更新後の最小値 i_L′	更新後の最大値 i_U′
`x[i]`	$\min(i_{min}, \max(i\_L , 0))$	$\max(i_{max} , \min(i\_U , \text{size(x) - 1}))$
`i++, ++i`	i_L - 1	i_U - 1
`i--, --i`	i_L + 1	i_U + 1

表 2　添字変数 i に発生する制約の合成

出現パターン	更新後の最小値 i'_{min}	更新後の最大値 i'_{max}
`x[i]`	i_L′	i_U′
`i=N`	$\min(i_{min} , i\_L)$	$\max(i_{max} , i\_U)$

表 3　条件式における添字変数 i の制約合成 (True 経路)

	True の経路	
出現パターン	更新後の最小値 i'_{min}	更新後の最大値 i'_{max}
i < N	/	$\max(i_{max} , i\_U)$
i > N	$\min(i_{min} , i\_L)$	/

表 4　条件式における添字変数 i の制約合成 (False 経路)

	False の経路	
出現パターン	更新後の最小値 i'_{min}	更新後の最大値 i'_{max}
i < N	$\min(i_{min} , i\_L)$	/
i > N	/	$\max(i_{max} , i\_U)$

- size(x) は配列 x の大きさを指す
- i_{max}・i_{min} は添字変数 i のノードに挿入されている最大値・最小値
- i_{max} は初期値に “-∞”，i_{min} は “∞” と仮の値を設定する
- i_U, i_L は Visitor が持つ添字変数 i の最大値・最小値
- i_{max}，i_{min}，i_U, i_L はそれぞれ更新前の最大値・最小値を指す
- i'_{max}, i'_{min}, i_U′, i_L′ は更新を行った後の最大値・最小値を指す
- max(X,Y) は 2 つの引数の大きい方を，min(X,Y) は小さい方を返す関数

表 5　初期化式・条件式に挿入する制約条件

式	制約条件
i = N	$i_{min} \leqq$ N $\leqq i_{max}$
i < N	$i_{min} \leqq$ N-1 $\leqq i_{max}$
i > N	$i_{min} \leqq$ N+1 $\leqq i_{max}$
i <= N , i >= N	$i_{min} \leqq$ N $\leqq i_{max}$

に記録した制約条件を合成する．しかし，初期化式の合成した制約条件を使って，表 5 で添字変数に代入する値に対する制約条件を導出している．

4　実装

3 章で提案した手法を検証するために，解析ツールを Perl で実装した．プログラムから添字変数の制約条件を導出するために，入力したプログラムを抽象構文木 (AST) に変換し，Visitor が辿る CFG を生成，導出した制約条件をプログラムに挿入し提示するための AST からテキストの再構築には TEBA[4] のツールを使用した．制約条件を AST のノードに字句列として挿入することで AST からテキストを再構築したときに制約条件をプログラムのコメントとして提示する．

本論文の提案手法における更新ルールでは，更新パターンによる最大値・最小値と，Visitor が持つデータ (U, L) を比較し，大きい方，もしくは小さい方を新たに制約条件として導出しているが，配列の大きさを暫定的に size() で表現するために値として計算できない．しかし，記号計算ができないと，式の等価性を判定できず，繰り返し文内の解析で最大値・最小値を収束させることができない．これを回避するために，簡約できる数式を簡約し，出力を可能な限り短い数式にする数式簡約化プログラムを実装する．

アルゴリズム 1 CFG 解析アルゴリズム

入力: JSON 形式の AST
出力: JSON 形式の AST の字句列

```
for begin ノードに到達するまで do
    if ラベルが “assign” then
        if インクリメントが出現 && 変数 x が Visitor に登録されている then
            x_U′ = x_U - 1 , x_L′ = x_L - 1 (表 1)
        else if デクリメントが出現 && 変数 x が Visitor に登録されている then
            x_U′ = x_U + 1,x_L′ = x_L + 1 (表 1)
        else if 代入演算子が出現 && 左辺値が Visitor に登録されている添字変数 x then
            x′max = max(xmax , x_U) , x′min = min(xmin , x_L) (表 2)
            x′min ≦ 右辺値 ≦ x′max を制約条件としてノードに追加する.
    else if ラベルが “op” then
        if 不等号が出現 && 変数 x が Visitor に登録されている then
            if x < N || x <= N then
                if True の経路を通ってきた then
                    x′max = max(xmax , x_U) (表 3)
                else if False の経路を通ってきた then
                    x′min = min(xmin , x_L) (表 4)
            else if x > N || x >= N then
                if True の経路を通ってきた then
                    x′min = min(xmin , x_L) (表 3)
                else if False の経路を通ってきた then
                    x′max = max(xmax , x_U) (表 4)

            x′min ≦ 比較する値 ≦ x′max を制約条件としてノードに追加する.
        else if 配列 a が出現 && 添字が変数 x then
            if 初めて添字が x の配列が出現 then
                x_U = size(a) - 1 , x_L = 0
            else if 変数 x が Visitor に登録されている then
                x_U′ = max(xmax , min(x_U , size(a)-1)) , x_L′ = min(xmin , max(x_L , 0)) (表 1)
            x′max = x_U′ , x′min = x_L′ (表 2)
            最大値 x′max と最小値 x′min をノードに追加する
```

表 6　定義する関数

要素	関数名 (引数: 引数の型)	関数が返すもの (オブジェクトの型名)
終端要素	num(x: int)	整数値 x のオブジェクト (num)
	var(v: string)	変数名 v のオブジェクト (var)
	size(a: string)	配列 a の大きさ ($size(a)$)
	p_inf()	正の無限大オブジェクト (p_inf)
	n_inf()	負の無限大オブジェクト (n_inf)
非終端要素	add(l: expr, r: expr)	式 l+r，式オブジェクト $add(l, r)$
	max(l: expr, r: expr)	l と r の大きい方，式オブジェクト $max(l, r)$
	min(l: expr, r: expr)	l と r の小さい方，式オブジェクト $min(l, r)$

- 式オブジェクトはすべて expr 型として扱う
- add, max, min は引数である l, r の大小関係がわからない (簡約できない) 場合にのみ，それぞれ式オブジェクト $add(l, r)$, $max(l, r)$, $min(l, r)$ を返す

3 章で定義した更新ルールに出現する，最大値 max と最小値 min，式同士の加算を表す add，正負の無限大を式オブジェクトとして定義する．式オブジェクトは非終端要素と終端要素に分類され，関数によって生成される．表 6 は数式簡約化プログラムで実現した関数の一覧である．

簡約を行う前に，各オブジェクトは正規化する．正規化では簡約を効率よく行うために非終端要素が持つ 2 つの要素の順序に優先順位をつけて入れ替えたり，すべての関数に共通して言える簡約を行う．正規化を終えた後，非終端要素 min，max，add は簡約化ルールに従って簡約化する．

式オブジェクトはハッシュで実現し，記号計算における値や関数をハッシュオブジェクトで表現する．各関数は式オブジェクトを返すが，すでに存在する等価な式オブジェクトが存在するときはそれを返す工夫をしている．これにより，式の等価判定は，オブジェクトの参照が一致するかどうかで求められる．

5 評価

実装したツールで正しく制約条件が導かれ，研究目的を達成できるかどうかを確認するため，1 節で挙げたプログラム 1 にツールを適用した．出力結果をプログラム 2 に示す．さらに，脆弱性がない場合の出力と比較するために，プログラム 1 と同じ仕様で脆弱性を排除したプログラム 3 を用意した．プログラム 3 に対する出力をプログラム 4 に示す．なお，説明の都合上，制約条件は添字変数の初期化式と制御文の条件式のみに挿入するようにオプションで変更している．

プログラム 2 プログラム 1 に対する出力

```
1 char *copy1(char dst[], char src[], int n) {
2
3   int i;
4   for (i /* 0 <= 0 <= min(size(src) - 1, size(dst) - 1) */ =
          0; src[i] != '\0' && i /* 0 <= n - 1 <= min(size(
         src) - 2, size(dst) - 2) */ < n; i++) {
5     dst[i] = src[i];
6   }
7   dst[i] = '\0';
8   return dst;
9 }
```

プログラム 3 バッファオーバーランが発生しないケース

```
1 char *copy2(char dst[], char src[], int n) {
2
3   int i;
4   for (i = 0; i < n; i++) {
5     dst[i] = src[i];
6     if (src[i] == '\0') break;
7   }
8   return dst;
9 }
```

プログラム 4 プログラム 3 に対する出力

```
1 char *copy2(char dst[], char src[], int n) {
2
3   int i;
4   for (i /* 0 <= 0 <= min(size(src) - 1, size(dst) - 1) */ =
          0; i /* n_inf <= n - 1 <= min(size(dst) - 1, size(
         src) - 1) */ < n; i++) {
5     dst[i] = src[i];
6     if (src[i] == '\0') break;
7   }
8   return dst;
9 }
```

初期化式内の制約条件は右辺の値に対する制約条件である．条件式内の制約条件は条件式の開区間を閉区間に内部で変換したときの右辺に対するものである．`n_inf` は負の無限大を表し，制約がないことを示す．

得られたプログラム 2, 4 を用いてバッファオーバーランが発生する入力のケースと発生しない入力のケースで制約条件がどのように働くのかを評価した．条件式 `i<n` に挿入された制約条件を比較する．どちらのプログラムも文字列 src を最大 n 文字まで文字列 dst にコピーするという仕様であり，n に与えられる条件は配列 dst と配列 src の大きさ以下 ($n \leqq \min(\mathrm{size(dst)}, \mathrm{size(src)})$) である．プログラム 4 の条件式 `i<n` に挿入された min(size(src)-1, size(dst)-1) >= n-1 はこの条件と等価であるが，プログラム 2 の条件式 `i<n` に挿入された min(size(src)-2, size(dst)-2) >= n-1 はこの条件よりも狭い条件である．つまり，同じ条件通りに引数を入力しても，プログラム 2 ではバッファオーバーランしないための制約条件を満たさない場合があることがわかる．評価の結果，ツールはバッファオーバーランが発生する可能性があることを利用者に把握させることができることを確認できた．

6 考察

本論文で提案した手法は初歩的で典型的なプログラムにのみ対応している．実践的なプログラムでは，条件式や添字式に複数の添字変数が同時に含まれることがあり，対応する必要がある．例えば，i < j のように一つのの条件式に複数の添字変数が出現するケースを想定する必要がある．また，理解支援方法を向上するためには，制約条件をプログラムに挿入して明示するだけでなく，実際に具体値を入力してもらい，制約条件が満たされているか考えさせることでより理解に繋げられる．配列の大きさを表す size() や仮引数に具体値を入れることで添字変数や仮引数にかかる制約条件を具体値で考えやすくする．

7 おわりに

本論文では，添字変数の制約条件の導出によるプログラミング学習におけるバッファオーバーランの理解支援方法を提案した．提案した支援方法を実現したツールを作成し，テストケースを用いてバッファオーバーランの理解支援の妥当性を確認した．今後の課題は，前提条件の緩和による解析可能な対象の拡大と，より理解に有効な支援方法への改善である．

参考文献

[1] Cppcheck -A tool for static C/C++ code analysis. `https://cppcheck.sourceforge.io/`

[2] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski and David Vitek. Buffer overrun detection using linear programming and static analysis. CCS '03: Tenth ACM Conference on Computer and Communications Security. pp.345–354 (2003)

[3] 吉田敦，山本晋一郎，阿草清滋：抽象スレッドに基づくソースプログラムの依存解析の枠組みの提案. コンピュータソフトウェア vol.16 No.1 pp.46–56 (1999)

[4] 吉田敦, 蜂巣吉成, 沢田篤史, 張漢明, 野呂昌満：属性付き字句系列に基づくソースコード書き換え支援環境. 情報処理学会論文誌 Vol.53 No.7 pp.1832–1849 (2012)