冗長な検査の削除を行う整数オーバーフロー対策ツールの実装

山崎 悟史† 桑原 寛明† 國枝 義敏†
†立命館大学情報理工学部

1 はじめに

近年，バッファオーバーフロー攻撃などプログラムの脆弱性を悪用した攻撃が増加しており，大きな問題になっている．このような攻撃が成功するとシステムの乗っ取りや情報漏洩が発生する可能性があり，非常に危険なものであると認識されている．

整数オーバーフロー脆弱性は，バッファオーバーフロー攻撃やサービス不能攻撃につながるプログラムの異常動作を発生させる可能性があり，非常に危険であるが，バッファオーバーフロー脆弱性に比べて対策が十分とは言えない．

整数オーバーフロー脆弱性に対処するための既存手法では，整数オーバーフローが起こり得るすべての整数演算や型変換，符号変換に対して検査コードを挿入するので，多くの誤検知が発生してしまう．また，悪用される整数オーバーフロー脆弱性の大半は，メモリ割り当てとバッファアクセスの際に参照される整数変数をターゲットとしており，この2つの用途以外に用いられる整数変数に対する検査は冗長である．本稿では，誤検知を削減するために，検査対象となる整数変数の用途をデータフロー解析により求め，特定の用途の整数変数に対してのみ検査コードを挿入する手法の提案と実装について述べる．

2 関連研究

整数オーバーフローが起こり得るすべての整数演算や型変換，符号変換に対して検査コードを挿入する手法として，David Brumley らの手法 [1] が存在する．David Brumley らの手法は，プログラム中のすべての整数演算と型変換に対して型付けを行い，整数オーバーフローが起こり得るものを検出し，実行時に整数オーバーフローが発生したことを検知するための検査コードを挿入する．この手法は，非常に高い精度で潜在的に危険な整数変数の参照を発見できる．しかし，疑似乱数生成や符号化処理などにおいて，プログラマが故意に発生させた整数オーバーフローを危険なものであると判断してしまう．これに対し，本提案手法では，検査コード挿入対象となる整数演算と型変換を限定することで，誤検知を削減している．

An Implementation of a Defence Tool against Integer Overflow Attack without Redundant Checks
Satoshi Yamasaki†, Hiroaki Kuwabara†and Yoshitoshi Kunieda †
†College of Information Science and Engineering, Ritsumeikan Univ.

また，検査対象となる整数変数の用途に基づいて，検査コードを挿入するかどうかを決定する手法として，Dipanwita Sarkar らの手法 [2] が存在する．Dipanwita Sarkar らの手法は，プログラム中の全ての整数変数に対して，取り得る値の範囲を求め，整数オーバーフローが起こり得る整数演算と型変換に対して検査コードを挿入する．また，検査コード挿入の際，ユーザによる注釈に基づいて検査対象となる整数変数の用途を調べ，メモリ割り当てとバッファアクセスに用いられている整数変数に対してのみ検査コードを挿入する．しかし，各時点での整数変数の取り得る値の範囲や，プログラム中で用いられている関数がメモリ割り当てを行っているかどうかといった情報をユーザがプログラム中に記述する必要があり，ユーザにとって大きな負担となる．これに対し，本提案手法では，検査対象となる整数変数の用途をデータフロー解析により求めて，検査コードを挿入するかどうかを自動的に判断する．

3 整数変数の用途に基づく検査コード挿入手法

ある整数変数が参照されたとき，その参照がメモリ割り当てかバッファアクセスにおけるものである，もしくはそれらが行われる際に参照される整数変数に影響を与える可能性があると判断できる構文要素は，以下に示す5つである．

(a) 代入文

(b) インラインアセンブラのオペランド

(c) 配列の添え字

(d) 関数呼出し時の実引数

(e) return 文

検査対象の整数変数が上記の構文要素において参照された際の対応について，以下で詳細に述べる．なお，2つ以上の構文要素が入れ子状になっているときは，最内側の構文要素として参照されたと判断する．

(a) 代入文 左辺に現れる整数変数が参照される構文要素を調べ，メモリ割り当てとバッファアクセスに用いられる可能性があれば，検査対象となる整数変数が整数オーバーフローを起こしていないか検査するためのコードを挿入する．

```
1  int main(void){
2     unsigned int x,y; char *str;
3     scanf("%u",&x);
4     x += 1;
5     if(x > 0x7fffffff) error(); // 加算結果の検査
6     y = x;
7     str = (char *)malloc(y); // 検査が必要
8     ...
```

図 1: サンプルコード 1

この対応についての具体例を図 1 に示す．サンプルコード 1 の 6 行目において，y への代入文の右辺に x が現われていることから，x は y に影響を与えていると判断する．y がメモリ割り当てを行う関数の引数となっているため，x についても y と同様に整数オーバーフローを起こしていないか検査する必要があると考え，4 行目の加算についての検査を 5 行目で行う．

(b) インラインアセンブラのオペランド　検査対象の整数変数がメモリ割り当てとバッファアクセスに用いられる可能性があるため，検査コードを挿入する．

(c) 配列の添え字　バッファアクセスに用いられていると見なせるので，検査コードを挿入する．

(d) 関数呼出し時の実引数　メモリ割り当てとバッファアクセスに用いられる可能性があるため，基本的には検査コードを挿入する必要がある．しかし，手続き間データフロー解析を行い大域的に検査対象の整数変数の参照される構文要素を調べ，メモリ割り当てとバッファアクセスに用いられていなければ検査コードを挿入する必要はない．

この対応についての具体例を図 2 に示す．サンプルコード 2 の 5，8 行目は共に上位ビット消失の可能性のある危険な代入文であり，検査対象となる整数変数 a，b は共に関数呼び出し時の実引数として用いられている．a については，1 行目で示されているとおり，関数 f 内でメモリ割り当てやバッファアクセスに用いられていないため，検査を行う必要はない．b については，9 行目でメモリ割り当てを行う関数の引数となっているため，8 行目の代入についての検査を 7 行目で行う．

(e)return 文　返り値が配列の添え字や関数呼出し時の実引数として参照される可能性があるため，基本的には検査コードを挿入する必要がある．しかし，(d) と同様に手続き間データフロー解析を行い，返り値がメモリ割り当てとバッファアクセスに用いられていなければ検査コードを挿入する必要はない．

4　実装・評価

提案手法を実装するにあたって，コンパイラインフラストラクチャCOINS[3] を使用した．対象言語は C 言語とし，入力として与えられたソースコードに検査コードを付加したソースコードを出力する．整数オーバーフローの検出には IA–32 アーキテクチャのフローフラグを使用し，整数オーバーフロー検出時にはエラーメッセージ出力後にプログラムの実行を中断させる．また，本ツールでは整数オーバーフロー脆弱性の見逃しを避けるため，David Brumley らの手法で用いられている型付けを行い，整数オーバーフローが起こり得るすべての整数演算と型変換に対して検査コードを挿入後，提案手法により冗長と判断された検査を削除するという処理手順を取っている．代入や四則演算に対して挿入する検査コードは David Brumley らの実装 [4] を参考にした．なお，実装の簡単化のためにポインタを含む演算を対象外としている．

```
1  void f(unsigned int v) { v++; }
2  int main(void){
3     unsigned short a,b; unsigned int i; char *str;
4     scanf("%u",&i);
5     a = i; // 上位ビット消失の可能性
6     f(a);                    // 検査は不要
7     if(i > 0xffff) error(); // bへの代入可能か検査
8     b = i; // 上位ビット消失の可能性
9     str = (char *)malloc(b); // 検査が必要
10    ...
```

図 2: サンプルコード 2

実装したツールに入力として C のソースコードを与え，検査対象となる整数変数がメモリ割り当てとバッファアクセスに用いられていない整数演算と型変換については，検査コードが挿入されないことを確認した．

5　おわりに

本稿では，整数変数の用途に基づく検査コード挿入手法とその実装について述べた．今後は，静的解析結果に基づいた異なる判断基準を提案手法に加え，さらに誤検知を減らす手法を検討する．

参考文献

[1] David Brumley, Dawn Song, and Joseph Slember. Towards automatically eliminating integer-based vulnerabilities. Technical report, CMU-CS-06-136, 2006.

[2] Dipanwita Sarkar, Muthu Jagannathan, Jay Thiagarajan, and Ramanathan Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. Technical report, Microsoft Research, 2006.

[3] coins 開発グループ. COINS コンパイラ・インフラストラクチャ. http://www.coins-project.org/.

[4] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *Symp. on Network and Distributed Systems Security*, 2007.