一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2020-65,CPSY2020-48,RECONF2020-84(2021-01)

脆弱性の原因となるコード最適化のバイナリ比較による検出

東　　優花†　石浦菜岐佐†

† 関西学院大学 理工学部　〒 669–1337 兵庫県三田市学園 2–1

あらまし　本稿では, ソフトウェアの脆弱性の原因となるコード削除がコンパイラの最適化によって行われていることを検出する手法を提案する. 不正なメモリアクセスを防止するためのガードや, 秘密情報の主記憶での残留を防止するための消去コードがソースコードに書かれていても, これがコンパイラの最適化によって削除されてしまうために脆弱性が生じる例が報告されている. 本稿ではコンパイラによるこのようなコード削除をバイナリ比較によって検出する. 与えられたソースコードに対し, 通常の最適化オプションでコンパイルして得られるバイナリコードと, 問題を引き起こす最適化を抑止するオプションで得られるバイナリコードを比較して差分を検出をする. 単純なコード比較では有意な差を得ることができないため, 特定の命令に着目した比較を行う. その後, 元のソースコードを最小化し, 検出された差分とコードの対応を取ることにより問題となるコード部分を特定する. 提案手法に基づくツールを Perl 5 を用いて実装し, 200 行〜 51400 行のプログラム 5 件についてテストを行った結果, 1 件のガード消失と 2 件のデッドストア削除を特定することができた.

キーワード　ソフトウェア, コンパイラ, 最適化, 脆弱性, ガード消失, デッドストア削除

Detection of Vulnerability Inducing Code Elimination by Compiler Optimization Based on Binary Code Comparison

Yuka AZUMA† and Nagisa ISHIURA†

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　In this paper, we propose a method to detect vulnerability inducing code elimination by compiler optimization. It is reported that security codes, which are intended to protect programs against invalid memory accesses or to scrub secret information in memories, can be eliminated by compiler optimization. This paper attempts to detect such code elimination by binary comparison. A pair of binary codes are generated from a given source code and are compared; one is obtained with a typical set of optimization options and the other by suppressing the problematic optimizations. Since the two binaries are often too different to reveal the code elimination, comparison is done focusing on specific instructions. When the code elimination is detected, the source code is minimized to identify the code fragment that caused the difference. A detection tool implemented in Perl 5 has been run on 5 programs consisting of about 200 to 51400 lines, to detect one case of guard elimination and two cases of dead store elimination.

Key words　Software vulnerability, Compiler optimization, Guard elimination, Dead store elimination

1. はじめに

プログラムのセキュリティ脆弱性に起因する計算機システムの不正操作や情報漏洩は社会的に深刻な問題となっている. これを防止するために徹底的なコード監査や形式的手法を用いた検証等様々な対策が取られている.

しかし, ソースコードレベルで完全な対策を行ったとしても, コンパイラの最適化が開発者の意図しないコード変換を行った結果, バイナリコードレベルで脆弱性が発生する例が報告されている [1]. 最適化に起因する脆弱性の例としては, 脆弱性を防止するためのガードの消失, デッドストア削除やコード移動による秘密情報の持続, サイドチャネル攻撃対策コードの削除等が挙げられる. これらの脆弱性は, プログラミング言語の仕様の正確な理解や, コンパイラの最適化に関する知識があれば防げるものである. しかし, 必ずしも全てのプログラマがこれを意識して注意深くコーディングを行っているわけではないため, 脆弱性を引き起こし得るプログラムが作成されてしてしまう.

Wang はコンパイラが未定義動作を利用した最適化により

コードを削除する条件を定式化し, 制約ソルバーによってこれを検出する静的チェッカー STACK を提案している [2]. しかし, この手法は未定義動作に起因するガード消失以外の検出はできない. また, ソルバーのタイムアウトによって必ずしも検出が行われない場合もある.

本稿では, コンパイラの最適化に起因する脆弱性を検出する新しいアプローチとして, バイナリコードの比較に基づく方法を提案する [3]. 通常の最適化を行って得られるコードを, 脆弱性を引き起こす可能性のある最適化を抑止して得られるコードと比較し, その差分から脆弱性の原因となる部分コードを検出する. 本手法は高速に実行でき, 未定義動作に関連するガード削除だけでなく, デッドストア削除を検出することができる.

提案手法に基づくツールを Perl 5 を用いて実装し 200 行〜51400 行のプログラム 5 件についてテストを行った結果, 15 件の差分を検出し, 1 件のガード消失, 2 件のデッドストア削除を検出することができた.

以下, 本稿では 2 章で コンパイラの最適化が引き起こす脆弱性について述べ, 3 章でバイナリ比較に基づく脆弱性の検出について述べる. 4 章で実装および実験について述べた後, 5 章でまとめと今後の課題を述べる.

2. コンパイラの最適化が引き起こす脆弱性

2.1 コンパイラの最適化が引き起こす脆弱性

プログラマがセキュリティを保護するために書いたコードに, コンパイラが最適化を目的とした変換を行った結果, 脆弱性が生じることがある [1]. ガードの消失, デッドストア削除による秘密情報の持続, コード移動による秘密情報の持続, サイドチャネルアタック対策の削除等がその例として挙げられている. これらは, コンパイラの最適化が個々の文の動作ではなく最終的な計算結果だけを保証するようにコード変換を行うこと, および, コンパイラが言語仕様に準拠して強力な最適化行うことによって, プログラマの直感とは異なるバイナリコードが生成されることが一因である.

(1) コンパイラの最適化によるガード消失

本稿ではプログラム中の脆弱性を防止するためのコードを「ガード」と呼ぶ. コンパイラの最適化によりガードが消失するプログラムの例を図 1 (a)(b) に示す.

図 1 (a) の 4 行目は, 3 行目で `size` がオーバーフローした場合のメモリ割り当て防止を意図したガードである. オーバーフロー発生時には `size` の値が小さくなることを期待したコードであるが, C 言語の仕様では符号付き整数型のオーバーフローは未定義動作であり, それ以後の計算結果は保証されない (言語仕様上コンパイラはいかなるコードを生成しても構わない)(注1). 3 行目で オーバーフローが生じなかった場合には, 4 行目の if 文の条件は偽となるため `error()` は呼び出されない. オーバーフローが生じた場合, それは未定義動作であり, 4 行目以降に対してコンパイラはいかなるコードを生成しても構わないので, `error()` は呼び出さなくても良い. この結果, コンパイラは 4 行目のガードを最適化により削除してしまう (注2).

図 1 (b) の 8 行目は, 9 行目で `p` の NULL ポインタ逆参照を防ぐことを意図したガードであるが, 不注意で `p` の逆参照を行なっている 7 行目の後に書かれてしまっている. このガードによって少なくとも 9 行目以降の NULL ポインタ逆参照は防げると期待するかも知れないが, 7 行目の NULL ポインタ逆参照は未定義動作なので, 8 行目以降の動作は言語仕様上保証されない. コンパイラは, `p` が NULL でない場合を想定したコードを生成できるので, 8 行目のガードを削除することがある. このようなガード消失は, 実際に Linux カーネルの脆弱性につながった [4].

```
SOF.c
1: int sub (int n){
2: ...
3:  int size = BASE + n;
4:  if(size < n) error();
5:  char* p = malloc(size);
6:  ...
7: }
```

```
NPD.c
 1: typedef struct{
 2:  int a[SASIZE];
 3:  int x;
 4: } str_t;
 5: ...
 6: int sub(str_t p){
 7:  int y = p -> x;
 8:  if(p == NULL) error();
 9:  int z = p -> a;
10:  ...
11: }
```

(a) SOF: 符号付きオーバーフロー　(b) NPD: NULL ポインタ逆参照

図 1　コンパイラの最適化が削除するガードの例

```
DSE.c
1: char* getPHash (){
2: char pwd[64];
3:  char *shal = (char *)malloc(41);
4:  fgets(pwd, sizeof(pwd), stdin);
5:  ...
6:  memset(pwd, 0, sizeof(pwd));
7:  return shal;
8: }
```

図 2　DSE : デッドストア削除の例

(2) デッドストア削除による秘密情報の持続

パスワード等の秘密情報が主記憶上に残留して不正に参照されるのを防ぐために, 処理後にそれを上書き消去するコードが書かれる. しかし, そのコードがコンパイラの最適化によって削除されることがある. コード例を図 2 に示す. 6 行目は `memset` により文字列 `pwd` の内容を消去することを意図している. しかし, `pwd` の値はそれ以降で使用されないため, コンパイラは不要コード削除の一環であるデッドストア削除によりこの行を削除してしまうことがある.

2.2 脆弱性に対する対策

コンパイラの最適化によるこのようなセキュリティ用コードの消失は, 言語仕様とコンパイラに関する知識があれば防げるものである. 図 1 (a) の例は, 符号付き整数のオーバーフローが未定義動作であることを知って, オーバフローが生じないようにガードの条件を書けば防げる. 図 1 (b) の例は単なる不注意であるが, その結果がもたらす危険性を意識しておく必要がある. 図 2 はコンパイラの最適化に関する知識に基づき, 変数を volatile 宣言することにより防止できる. しかし, 必ずしも全てのプログラマがそのような意識のもとにコーディングを行っているわけではない.

コンパイラにはこのようなコードを検出して警告するオプションを備えているものがある. 例えば, GCC の -Wstrict-overfow

(注1) : 符号なし整数型の場合にはラップアラウンドする仕様になっているので未定義動作は発生しない. また, Java では符号付き整数もラップアラウンドする仕様になっているので未定義動作は発生しない.

(注2) : コンパイラは未定義動作が発生しない前提でコード最適化を行うので, 4 行目の if 文の条件が恒偽と判断されるという考え方もできる.

オプションは符号付きオーバーフローに基づいて最適化を行った際に, -Wnull-dereference は NULL ポインタ逆参照に基づいてコード削除を行った際に警告を表示するものである. しかし, これらのオプションは全てのコード削除を検出するわけではなく(注3), またコンパイラによっては実装されていないことがある(注4).

更にコンパイラの中にはこのような最適化をピンポイントで抑止するオプションを実装しているものもある. GCC の -fwrapv オプションは加算, 減算, および乗算の符号付き算術オーバーフローが生じた場合, 2 の補数表現を想定してラップアラウンドを定義動作とするものである. -fno-delete-null-pointer-checks は NULL ポインタへの逆参照が未定義動作であることを用いた最適化を抑止するオプションである. -fno-dse, -fno-tree-dse は後続の命令によって読み出しが行われない場合でもその変数への書き込みを削除しないようにするオプションである. しかし, これらのオプションは, プログラム中の脆弱性に関連しない大部分の箇所で性能を低下させるという問題点がある(注5).

また上記とは別に, GCC には組込み関数 memset のインライン展開を抑止するオプション -fno-builtin-memset が実装されている. この抑止オプションは, 組み込み関数である memset のインライン展開を抑止することができる. 秘密情報の消去に memset を使用している場合, この抑止オプションによってデッドストア削除が生じるのを防ぐことが期待されるが意図通り機能しない場合がある [5].

Wang はコンパイラが未定義動作を利用した最適化によりコードを削除する条件を定式化し, 制約ソルバーによってこれを検出する静的チェッカー STACK を提案している [2]. STACK は実際に 160 件の脆弱性を検出しているが, この手法が対象とするのは未定義動作に起因するコード削除のみであり, デッドストア削除を検出することはできない.

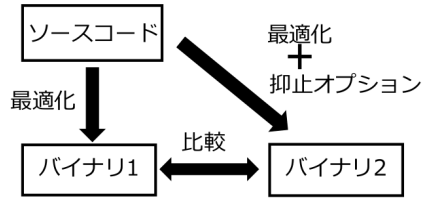
3. バイナリ比較に基づく脆弱性の検出

3.1 概　　要

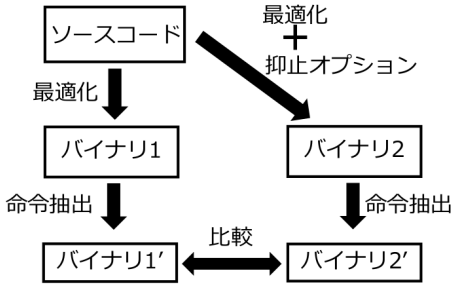
本稿では, コンパイラの最適化が引き起こす脆弱性の原因となるコードを, バイナリ比較によって検出する手法を提案する [3].

本手法の基本的なアイデアを図 3 (a) に示す. 図中の 「バイナリ 1」 は 「ソースコード」 から通常の最適化によって生成されるバイナリコードであり, この中に脆弱性が生じている可能性がある. これに対し 「バイナリ 2」 は, 通常の最適化の中から脆弱性を引き起こす可能性のある最適化のみ抑止したコンパイルにより得られるバイナリコードである. この 2 つのバイナリを比較して有意な差が生じていれば, ソースコードに脆弱性を引き起こす可能性のある部分があると判定できる.

しかし, 近年のコンパイラの最適化は数十のパスからなり, それらが複雑に相互依存しているため, 一つの最適化パスの抑止が他の最適化パスに影響して, コードが大幅に変化することがあるため, 単純なコードの比較では意図したコードの差分を検出することができない. そこで本手法では図 3 (b) のように, コード中の特定の命令を抽出し, その対応をとるような比較を行う.



(a) 基本アイディア



(b) コードの比較法

図 3　バイナリ比較に基づく脆弱性検出手法

また, 二つのバイナリの差分を検出しただけでは, ソースコードのどの部分に問題があるのかわからない. 本手法では有意な差分があると判定した場合にはソースコードの最小化 (差分に関係しない部分を削除してプログラムをできる限り小さくする操作) を行い, ソースコードの問題箇所を特定できるようにする.

3.2 ガード消失の検出

符号付きオーバーフローの最適化抑止は, 例えば GCC ではコンパイラオプション -fwrapv によって行える. また, NULL ポインタ逆参照の最適化抑止はコンパイラオプション -fno-delete-null-pointer-checks によって行える.

これらの最適化を抑止するオプションを使用すると, 最適化の相互依存によってコードが大幅に変わってしまうことがある. 図 4 (a) はガード消失の起きないプログラムであるが, これを抑止オプション -fwapv なしとありでコンパイルした (b) と (c) は, 本質的な部分以外のコードも大きく変わってしまっている.

そこで本手法では, ガードには必ずエラーメッセージを出力したりプログラムを終了させるエラー処理が付随し, それは関数呼び出しによって実装されることに着目し, コード中の関数呼び出し命令 (x86 の場合は call 命令) のみを抽出して比較を行う.

まず, 2 つのアセンブリをそれぞれ関数毎に分割し, 関数呼び出し命令のみを抽出する. 次に, 2 つのバイナリコードにおいて同じ関数の中でオペランドを手がかりに命令の対応を取っていく (図 5 参照). バイナリ 2 において対応のとれない call 命令が残っていて, かつその命令がエラー処理を呼び出している場合, ガードが消失していると判定する.

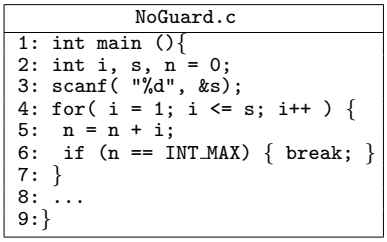
関数呼び出し命令の存在しない関数または, 対応する関数が存在しない場合もある. その場合は次のように判定を行う.

(注3) : GCC-7.5.0 では, 4 章でも示す通り -Wstrict-overflow を指定しても警告が出力されないことがある. また -Wnull-dereference を指定しても図 1 (b) の例に対して警告は出力されない.

(注4) : LLVM-11.0.1 は GCC との互換性のためにコマンドラインでこのオプションを受け付けるが, 実際に警告は出力されない.

(注5) : 例えば, プロセッサの汎用レジスタが 64 ビットの場合には, 演算後に 32 ビット符号付き整数としてラップラウンドさせるための処理を挿入する必要が生じる.

- 同じ関数がどちらにも存在するが, call 命令はバイナリ 1 にしか存在しない場合はガードの消失の可能性はないと判定する.
- 同じ関数がどちらにも存在するが, どちらにも call 命令が存在しない場合はガードの消失の可能性はないと判定する.
- バイナリ 1 にのみ存在する関数がある場合はガードの消失の可能性はないと判定する.
- バイナリ 2 にのみ存在する関数があり, call 命令を含む場合はガード消失の可能性があると判定する.



(a) オーバーフローのないソースコード

(b) -O2 のみ　　(c) -fwrapv -O2

図 4　コンパイル結果

3.3　デッドストア削除の検出

デッドストア削除最適化の抑止は GCC ではコンパイラオプション -fno-dse, -fno-tree-dse によって行える. しかし, これらのオプションを指定した場合も最適化の相互依存によってコードが大幅変わってしまうことがあり, 単純な比較ではデッドストア削除とは無関係の部分で大きな差分が生じてしまう. また, デッドストア削除の場合にはエラー処理が行われるわけではないので, call 命令の対応を取る方法は適用できない.

そこで本手法では, データ転送命令に着目したバイナリコードの比較をおこなう. x86 の場合には, mov 命令, rep 命令, call 命令に着目する. これは, 文字列への書き込みは, バイト数が少ない場合には mov 命令, より多い場合には rep stos 命令, さらに多い場合には call memset によって行われるためである.

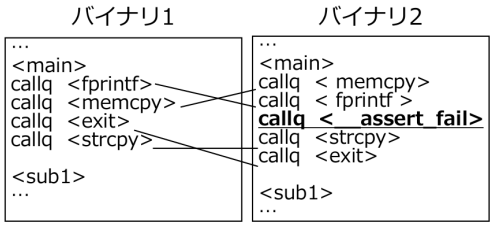


図 5　ガード消失検出のためのコード比較

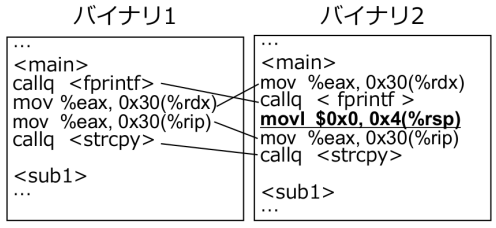


図 6　デッドストア削除検出のためのコード比較

まず, 2 つのバイナリをそれぞれ関数毎に分割し, call, mov, rep 命令のみを抽出する. 次に図 6 のように, 各関数内の call, mov, rep 命令の対応を取っていく. call 命令はオペランドが一致していれば対応が取れたものとする. mov 命令と rep 命令もオペランドが一致していれば対応が取れたものとするが, 最適化抑止オプションによってレジスタの割当が変更されていることがあるため, レジスタの違いは無視する.

デッドストア削除においても, バイナリ 2 において対応のとれない命令が一つ以上残っていて, かつ抽出された命令が文字列の初期化を行っている場合には, 脆弱性の可能性があるとして判定する. call, mov, rep 命令の存在しない関数, 対応する関数が存在しない場合は, ガード消失と同様の手順で判定を行う.

3.4　最　小　化

最適化によるガード消失やデッドストア削除の差分を検出した場合には, ソースコードのどの部分に問題があるかを特定する必要がある. 本手法では, これをソースプログラムの最小化により実現する.

最小化とは, エラー (この場合はバイナリコードの差分) が生じるできる限り小さなソースコードを求めることである. 最小化は Delta Debugging [6] のように, ソースコードを縮約する変換を同じ差分が検出される限り繰り返し適用することにより行える.

具体的な処理の流れを図 7 に示す. ソースコードに対して縮約変換を行い, 縮約されたプログラムに対して, 本手法による比較を行う. その結果, 縮約前のソースコードと同様に差分が検出された場合には, 更に縮約変換を行う. 比較の結果差分が検出されなかった場合は, 今回行った縮約変換を取り消して再度他の縮約変換を行う. この操作をどの縮約変換もそれ以上適用できなくなるまで繰り返す.

本手法でコードを縮約する変換は, 未使用の関数, 変数の削除, 変数名の変更, 関数を void 型に変更等, コンパイルエラーが出ないものなら何でも良い. ただし, 関数レベルでソースコードとバイナリの対応を取るために, 関数名の変更と関数のインライン展開は行わないようにする.

また, 最小化が行われたことにより元のソースコードで生じ

ていた脆弱性と異なる箇所で差分が検出されてしまい, 誤った最小化が行われる可能性がある. そのため, 差分が検出された関数名と call の呼び出し名を保管し, 本手法の比較によって検出される差分がこれらと一致することも確認しながら最小化を進める.

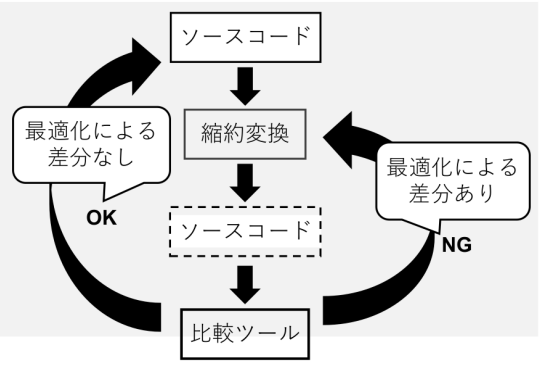


図 7　最小化の流れ

4. 実装と実験結果

4.1 実　　装

本稿の手法に基づくテストシステムを Perl 5 で実装した. 本システムは Ubuntu Linux 上で動作する. 最小化には C-Reduce [7] を用いたが, 関数名の変更と関数のインライン展開は行わないように修正した.

4.2 実　　験

本手法を用いて 5 件のプログラムに対してについて実験を行った. 使用したコンパイラは　GCC　のバージョン 7.3.0 であり, ターゲットは x86 linux, 最適化オプションには　-O2　用いた.

結果を表 1 に示す. 対象プログラムはコンパイラ mcc, PNG リファレンスライブラリ libpng, OpenSSL, OpenSSH, PNG 画像の非可逆圧縮用のライブラリ pngquant, であり, “KLOC” はソースコードの行数 (×千行) である. “SOF”, “NPD”, “DSE” はそれぞれ符号付きオーバーフロー, NULL ポインタ逆参照, デッドストア削除に関する最適化を抑止するオプション-fwrapv, -fno-delete-null-pointer-checks, -fno-dse, -fno-tree-dse を用いた結果である. “差分” は本手法による不一致の検出数, “検出” は目視によって確認したガードの消失またはデッドストア削除の件数である.

pngquant1, pngquant2, pngquant3 は pngquant に対して符号付きオーバーフローによるガード消失, NULL ポインタ逆参照によるガード消失, デッドストア削除をそれぞれ 1 件ずつ挿入したものである. 実験の結果, 挿入した脆弱性を正しく検出することができた.

mcc では, ガード消失を 1 件検出することができた. 図 8 に今回検出した mcc のソースコードの当該部および対応するアセンブリコードを示す. 図 8 (a) の 10 行目は `count` の値がオーバーフローが起きた場合のガードであり, オーバーフローを抑止しているが, コンパイラの最適化によって削除されていることが 図 8(c) の下線部から分かる. なお, GCC は符号付きオーバーフローに関するコード削除を警告するオプション -Wstrict-overflow を実装しているが, 上記ガードの削除は警告

表 1　実 験 結 果

プログラム	KLOC	SOF		NPD		DSE	
		差分	検出	差分	検出	差分	検出
pngquant	0.7	0	0	0	0	0	0
pngquant1	0.7	1	1	0	0	0	0
pngquant2	0.7	0	0	1	1	0	0
pngquant3	0.7	0	0	0	0	1	1
mcc	0.2	1	1	0	0	0	0
OpenSSH	51.4					6	2
OpenSSL	30.1					0	0
libpng	39.8	1	0	0	0	7	0

```
 1:...
 2:if ( x->type == token_STAR) {
 3:   int count = 0;
 4:   while (x -> type == token_STAR ) {
 5:    count ++;
 6:     ...
 7:   }
 8:   if (count == 1) { ...  }
 9:   else if (count > 1) { ...  }
10:   else { assert (0); }
11: ...
```

(a) ソースコード

```
O2
...
jmpq 4021f0 <parse_assign>
mov %r13d,%r9d
mov %r14,%r8
jmp 402246 <parse_assign>
mov $0x40530a,%esi
mov %rbx,%rdi
callq 401230 <syntax_error>
mov $0x40531a,%esi
mov %rbx,%rdi
callq 401230 <syntax_error>
mov $0x405500,%ecx
mov $0x57a,%edx
mov $0x4051e6,%esi
mov $0x405220,%edi
callq 4009a0 <__assert_fail>
nopl (%rax)
...
```

(b) -O2 のみ

```
O2 fwrapv
...
jmpq 402250 <parse_assign>
mov %ebp,%r9d
mov %r14,%r8
jmp 4022b3 <parse_assign>
mov $0x40535a,%esi
mov %rbx,%rdi
callq 401270 <syntax_error>
mov $0x40534a,%esi
mov %rbx,%rdi
callq 401270 <syntax_error>
mov $0x405540,%ecx
mov $0x57a,%edx
mov $0x405226,%esi
mov $0x405260,%edi
callq 4009a0 <__assert_fail>
mov $0x405540,%ecx
mov $0x561,%edx
mov $0x405226,%esi
mov $0x405260,%edi
callq 4009a0 <__assert_fail>
nopl 0x0(%rax,%rax,1)
...
```

(c) -fwrapv -O2

図 8　mcc のコード (一部)

されなかった.

OpenSSH では, デッドストア削除に関する差分が 6 件あり, そのうち, 2 件がデッドストア削除を正しく検出していた. 図 9 にソースコードの当該部および対応するアセンブリコードを示す. 図 9 (a) では 2 行目で `key->type` の値に応じて 3–6 行目で処理を行っている. その後, 12 行目で `*digest_type` の値を 0 にしているがこれがコンパイラの最適化によって削除されている. 4 件の誤検出は -fno-dse, -fno-tree-dse オプションによってレジスタ割当などが変更されたことによって, mov 命令が書き換わったことが原因と考えられる.

OpenSSL においては差分は検出されなかった.

libpng では 1 件のガード消失と 7 件のデッドストア削除に関する差分を検出したが, これらはいずれも誤検出であった. ガード消失検出の誤検出は -fwrapv オプションが関数のインライン展開を抑止し, call 命令が増えたためと考えられる. デッ

```
 1:...
 2: switch (key -> type) {
 3:   case KEY_RSA:
 4:   ...
 5:   break;
 6:   case KEY_DSA:
 7:   ...
 8:   break;
 9:   ...
10:   default:
11:   *algorithm = SSHFP_KEY_RESERVED;
12:   *digest_type = SSHFP_HASH_RESERVED;
13: }
14:...
```

(a) ソースコード

```
O2
...
cmp $0x1,%al
je 68 <dns_read_key>
cmp $0x2,%al
je d8 <dns_read_key>
movb $0x0,(%rsi)
jmp 36 <dns_read_key>
nopw 0x0(%rax,%rax,1)
movb $0x3,(%rdi)
movzbl (%rsi),%eax
test %al,%al
jne 9a <dns_read_key>
jmpq 2a <dns_read_key>
nop
movb $0x4,(%rdi)
movzbl (%rsi),%eax
test %al,%al
je 2a <dns_read_key>
jmp 9a <dns_read_key>
movb $0x0,(%rdi)
jmp a2 <dns_read_key>
nopl (%rax)
...
```

(b) -O2 のみ

```
O2 fno-dse fno-tree-dse
...
cmp $0x1,%al
je 68 <dns_read_key>
cmp $0x2,%al
je e0 <dns_read_key>
movb $0x0,(%rsi)
jmp 36 <dns_read_key>
nopw 0x0(%rax,%rax,1)
movb $0x3,(%rdi)
movzbl (%rsi),%eax
test %al,%al
jne 9a <dns_read_key>
jmpq 2a <dns_read_key>
nop
movb $0x4,(%rdi)
movzbl (%rsi),%eax
test %al,%al
je 2a <dns_read_key>
jmp 9a <dns_read_key>
movb $0x0,(%rdi)
movb $0x0,(%rsi)
jmp a2 <dns_read_key>
nopl 0x0(%rax,%rax,1)
...
```

(c) -fno-dse -fno-tree-dse -O2

図 9　OpenSSH のコード (一部)

ドストア削除の誤検出は, OpenSSH と同様に mov 命令が書き換わったことが原因と考えられる.

5. むすび

本稿では, コンパイラの最適化が引き起こす脆弱性の原因となるコードを, バイナリ比較によって検出する手法を提案した. 実際の脆弱性の検出には至らなかったが, 問題を引き起こす可能性のあるコードを検出することができた. 特に STACK では検出できないデッドストア削除を検出することができた.

今後の計画としては, コード移動による秘密情報の持続やサイドチャネル攻撃対策コードの削除の検出等が挙げられる.

謝　辞

本研究を行うにあたり, 御助言や御協力を頂いた, 関西学院大学理工学部石浦研究室の諸氏に感謝致します.

文　献

[1] Vijay D'Silva, et al.: "The Correctness-Security Gap in Compiler Optimization," in *Proc. IEEE Security and Privacy Workshops*, pp. 73–87 (May 2015).

[2] Xi Wang, et al.: "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior," in *Proc. ACM Symposium on Operating Systems Principles*, pp. 260–275 (Nov. 2013).

[3] Yuka Azuma and Nagisa Ishiura: "Detection of Vulnerability Guard Elimination by Compiler Optimization Based on Binary Code Comparison," in *Proc. Workshop on Synthesis And System Integrations of Mied Information Technologies (SASIMI 2019)*, R3-13, pp. 229–230 (Oct. 2019).

[4] Cve-2009-1897, `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897` (June 2009).

[5] Zhaomo Yang, et al.: "Dead store elimination (still) considered harmful," in *Proc. USENIX Conference on Security Symposium*, pp. 1025–1040 (Aug. 2017).

[6] Holger Cleve and Andreas Zeller: "Locating Causes of Program Failures," in *Proc. International Conference on Software Engineering (ICSE '05)*, pp. 342–351 (May 2005).

[7] John Regehr, et al.: "Test-Case Reduction for C Compiler Bugs," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*, pp. 334–346 (June 2012).