パッチ差分に基づく脆弱性修正箇所の特定支援技術の検討

中島明日香[1] 木村廉[2] 川古谷祐平[1] 岩村誠[1] 針生剛男[1]

概要：近年，既知の脆弱性箇所の特徴に基づいた脆弱性発見手法が注目されており，その手法を利用して少なくはない脆弱性が発見されている．しかし既存手法では，利用する脆弱性箇所の特徴を，主にソースコードから収集しており，実行ファイルに含まれる脆弱性箇所は，その解析・収集の難しさ故に対象外であった．そのため必然的に，プロプライエタリなソフトウェアの脆弱性の特徴は利用できず，発見可能な脆弱性も限られていた．脆弱性修正前後の実行ファイルを比較(パッチ差分解析)して，脆弱性箇所の特定を支援する技術は存在するものの，既存手法では特定可能な脆弱性箇所が限られており，かつ実行ファイル中で脆弱性修正以外の修正が行われていた場合，その判別が難しかった．

そこで本論文では，脆弱性箇所収集を最終目的として，脆弱性修正を含めたプログラム修正が存在する実行ファイル中から，網羅的に脆弱性修正箇所の特定を支援する技術の検討を行った．具体的には，網羅的な脆弱性修正箇所発見を目指して，まずはオープンソースソフトウェアから収集した，既知の脆弱性修正箇所をクラスタ分析手法で分類し，共通する特徴の有無を調べた．そして次に，線形分類器を利用して，収集した脆弱性修正箇所とその他の修正が識別可能か否かを検証した．その結果，脆弱性修正箇所の分類では部分的に共通する特徴は見られたが，脆弱性修正とその他の修正の識別は正解率 56%と高くはなく，識別に利用する入力データの改善が課題として浮かびあがった．

An Investigation of Method to Assist Identification of Patched Part of the Vulnerable Software Based on Patch Diffing

ASUKA NAKAJIMA[1] REN KIMURA[2] YUHEI KAWAKOYA[1]
MAKOTO IWAMURA[1] TAKEO HARIU[1]

1. 研究の背景と動機

サイバー攻撃の根本的な原因の一つとしてソフトウェアの脆弱性がある．脆弱性とは，ソフトウェアのバグの一種で，第三者から悪用可能なバグのことを指し，マルウェア感染をはじめとした各種サイバー攻撃に用いられる．この脆弱性に起因するサイバー攻撃の被害を防ぐためにも，開発段階で脆弱性を作りこまない努力が各ソフトウェアベンダによってなされているが，それも完璧では無く，脆弱性が残存したまま公開されてしまう場合が多い．そのため，それらの脆弱性をいち早く見つけ修正するためにも，ソフトウェア中から脆弱性を発見するための技術が大事になってくる．

ソフトウェアから脆弱性を発見する技術には，ファジング[1]やシンボリック実行[2]などが挙げられるが，近年はその中でも，コードクローンの脆弱性を対象とした脆弱性発見技術が着目されている[3][4][5][6]．このコードクローンの脆弱性とは，ソフトウェア中に存在した脆弱性部分がソースコードの流用などが原因で，他所にも複製されてしまい，その結果として新たに生まれた脆弱性のことを指す．その数は少なくなく，過去の研究[3]では Debian のパッケージから 145 個の新規のコードクローンの脆弱性を発見している．

このコードクローンの脆弱性発見技術は，既知の脆弱性箇所を基に，検査対象のソフトウェア中に同様の脆弱性が存在するか否かを検査するものである．そのため，事前に既知の脆弱性箇所を把握することが必須ではあるが，修正された脆弱性箇所の情報は脆弱性情報データベース[7]やベンダー公式サイトに掲載されてない場合があり，知り得たい場合には各自で調査する他なかった．

脆弱性箇所の特定方法としては，ソフトウェアのソースコードが入手可能な場合は，修正前後のソースコードの差分(パッチ差分)に着目すれば，ある程度容易に修正箇所を特定できる．しかし，ソースコードが公開されていない実行ファイルのみが入手可能なソフトウェアの場合，脆弱性箇所を特定するのは，逆アセンブルされたプログラムコードを読む能力に加え，コンパイラやリンカの最適化がアセンブリに与える影響などの知見が必要になり，容易ではない．その上，開発者が脆弱性修正以外の修正も加えていた場合，脆弱性修正とそれ以外の部分を区別する必要もあり，それには手間と時間を要した．

上記背景から，過去のコードクローンの脆弱性発見手法では，主にソースコードのパッチ差分に基づき脆弱性箇所を特定し，それを検査に用いていた．しかし，もしプロプライエタリなソフトウェア中からも既知の脆弱性箇所が特定でき，同様に検査に利用することが可能になれば，より

1 NTT セキュアプラットフォーム研究所
2 神戸大学 大学院

多くの脆弱性を発見出来る可能性が高い．

そこで本研究では，実行ファイルを対象とした，パッチ差分に基づく脆弱性箇所の特定支援に関する技術の検討を行う．

2. 従来のパッチ差分解析技術

実行ファイルを対象とした代表的なパッチ差分解析手法としては，BinDiff[8]や TurboDiff[9]のバイナリ比較ツールを用いた手法がある．BinDiff などは実行ファイル間の類似関数の発見・比較を行うツールであり，解析者はそこから得た情報に基づき脆弱性修正箇所の特定を行う．しかし，実行ファイルを対象としたパッチ差分の場合，ソースコード上で修正した箇所以外にも，コンパイラなどの最適化によって変化する場合があり，その判別は解析者の経験に依存していた．その上，脆弱性修正以外の修正も行われていた場合，解析者は変更のあった複数個所の中から，脆弱性修正箇所を見つけ出す必要があった．さらに，例えソースコード上では微小な修正であったとしても，コンパイラなどの最適化の影響から修正前後で大きく変化する場合があり，それが解析をより困難にさせていた．

上記問題を解決するため DarunGrim[10]では，プログラム修正後に追加・変更された部分が脆弱性修正の特徴と一致した場合，それに応じて点数付けを行い，優先的に解析すべき箇所を提示するようにした．具体的には，バッファオーバーフローなどの典型的な脆弱性修正の際に現れる特徴的な機械語命令・関数・即値を特徴とし，その特徴の重要度に応じて加算するべき点数を設定している．

ここで強調すべき点としては，DarunGrim で用いられている脆弱性修正の特徴や加算すべき点数は，全て開発者自身の経験に基づいて定められており，実際の脆弱性修正の現状に基づいているわけではない，という点である．そのため，優先的に解析すべき所として挙げられる箇所が，網羅的では無い可能性がある．

3. 脆弱性修正箇所の特定支援技術の検討

前述のように，実行ファイルを対象としたパッチ差分解析を利用した脆弱性箇所特定は，専門的な知識と多大な労力が必要であった．また，特定支援ツールも存在するものの，開発者独自の知見に基づいた手法のため，網羅性が欠けるものであった．

本問題を解決するため，実際の脆弱性修正の特徴を学習し，その特徴を利用してパッチ差分解析時に優先的に解析すべき箇所を提示する，脆弱性修正箇所の特定支援技術の実現を目指す．本論文では，それに向けた課題の整理と，それらの課題を解決するための検討を行う．

```
@@ -672,10 +675,6 @@ static int do_ssl3_write(SSL *s,

+       if (wb->buf == NULL)
+               if (!ssl3_setup_write_buffer(s))
+                       return -1;
+
        if (len == 0 && !create_empty_fragment)
                return 0;
```

図 1: CVE-2014-0198 の脆弱性修正

```
@@ -92,8 +92,6 @@ X509_REQ *X509_to_X509_REQ(X509 *x,

        pktmp = X509_get_pubkey(x);
+       if (pktmp == NULL)
+               goto err;
        i = X509_REQ_set_pubkey(ret, pktmp);
        EVP_PKEY_free(pktmp);
```

図 2: CVE-2015-0288 の脆弱性修正

3.1 脆弱性修正の特徴と想定する利用形態

一般的に脆弱性を修正する場合，脆弱な状態が発生しないよう，脆弱な状況を生む箇所に対して，その状況を発生させないための最小限の修正が加えられる．脆弱性の種類によりその修正方法も変わってくるが，同種もしくは類似の脆弱性の場合，その修正方法も類似する可能性は高いと考えられる．実際に，OpenSSL に存在していた NULL ポインタ参照の脆弱性である CVE-2014-0198(図 1)と CVE-2015-0288(図 2)に対する脆弱性修正からも，その類似性は見て取れる．具体的に，図 1 と図 2 を比較した時，どちらも同様に，特定のポインタが NULL であるか否かを検査していることが見て取れる．

そこで，既存の各種脆弱性修正から共通する特徴を抽出し，パッチ差分解析時にその特徴に基づき脆弱性修正箇所候補を提示することが可能になれば，従来よりも解析者の負担を軽減することが期待出来る．

上記の実現には，**①既存の脆弱性修正に共通する修正の特徴を抽出する**，**②抽出した特徴を利用して脆弱性修正箇所候補を算出する**，という段階を踏む必要がある．そこで，実現の検討に向け，一方もしくは両方で挙がる課題を次節以降に述べる．

3.2 課題 1: 最適化による変化

実行ファイルを対象としたパッチ差分解析を行う場合，プログラムの変更に応じて生じるコンパイラやリンカの最適化による変化は無視できない．ここでは，実際にどのような変化が起こりうる可能性があるのかについて述べる．

基本ブロックの順序変更 プログラムに変更が加えられた際に，最適化のため関数内の基本ブロックの順序が変更される場合が多々ある．そのため，修正前後での意味的な変化は微小でも，その表記は大きく異なる場合がある．

命令の順序変更/変化 実行の最適化のため，機械語命令の順序が変更される場合がある．さらに，機械語命令そのもの

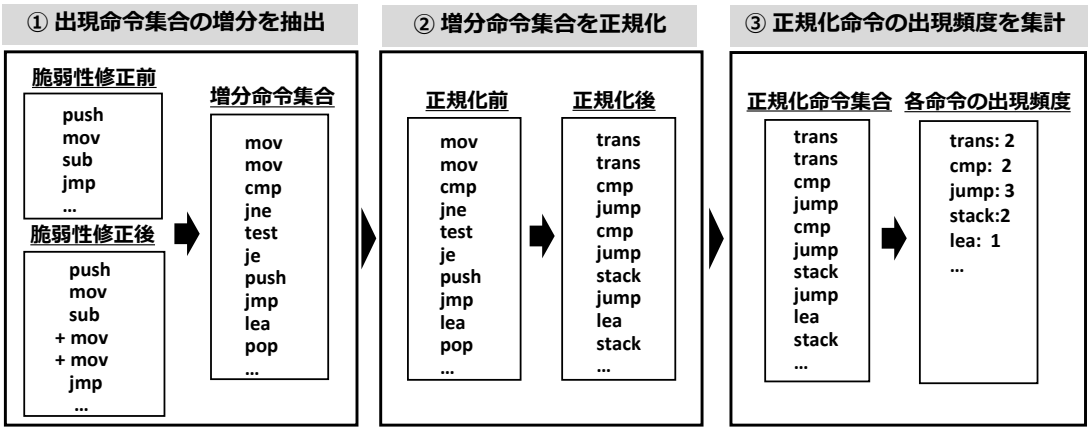

図 3: 各脆弱性修正箇所を特徴ベクトル化する手順

が変化する場合がある．その場合修正前後で内容は同一であるにも関わらず，パッチ差分解析を行った場合，差分として抽出されてしまう．

オペランドの変化 コンパイル環境や修正の方法によって，同一意味を持つ箇所でも，そのオペランドが修正前後で変化する場合が多々ある．具体的には，利用するレジスタの種類や，アクセス先のメモリアドレスなどが変化する．

関数のインライン展開/ループ展開 実行速度の高速化のためプログラム修正後に，関数内で呼び出した関数がインライン展開化される場合や，ループの中身が展開される場合がある.その場合修正前後で意味は同一であるにも関わらず，その表記は変わってくる．

3.3 課題 2：脆弱性修正以外の修正

脆弱性修正を行ったとして公開・配信されたソフトウェア中に，脆弱性とは関係ない変更が加えられている場合が実際には多々ある．例えば，脆弱性修正のついでに機能追加やバグ修正，そしてリファクタリングが行われている場合である．これらの変更は明示的に行われている場合もあるが，暗黙的に行われている場合もある．そして，たとえ明示されていた場合でも，実行ファイル中の機械語命令列の差分情報のみで，何を意図して行われた変更なのかを判断するのは経験者でも容易ではない．

4. 脆弱性修正方法の分類

本章では，前章で挙げた課題を踏まえつつ，3.1 節で述べた①既存の脆弱性修正に共通する修正の特徴を抽出する手法の検討を行うため，まずは脆弱性修正の方法を分類して，実際に修正の共通点などが存在するか否かを検証した．その実験手法と，結果について述べる．

4.1 実験概要

実験では，OpenSSL1.0.1 に適用された脆弱性修正を収集し，それをデータセットとした．具体的には，OpenSSL の公式サイトに掲示されている 1.0.1 版に対する脆弱性修正の情報[11]に基づき，OpenSSL の git レポジトリ中から脆弱性修正に紐づく変更を抽出し，その適用前後で増加した部分を持つ 62 個の脆弱性修正箇所をデータセットとした．またここでは適用前後のソースコードを 32bit 版 Ubuntu14.04 上で，gcc5.4.0 を利用してコンパイルし実行ファイル化したものを扱った．本実験では，数ある分類手法の中でもクラスタ分析手法を用いて，収集した脆弱性修正のデータセットを分類した．具体的には群平均法を適用した階層的クラスタリング手法を用い，その距離尺度としては，ユークリッド距離を利用した．

表 1：命令の正規化規則

正規化命令	命令の種類	正規化対象の命令
jump	分岐	jns, jle, jne, jge, jae, jmp, js, jl, je, jg, ja, jb, jbe
trans	データ転送	movxz,mov,movsx,xchg,cdq
ctrans	条件付きデータ転送	cmovge, cmovae, cmovs, cmovns, cmove, cmovne
stack	スタック操作	push,pop
logical	論理演算	and, xor, or, not
arith	算術演算	sub, add, imul, neg, adc
nop	無演算	nop
bop	ビット/バイト操作	bt, setne, sete
shift	シフト演算	shr, shl, sar
func	関数操作	call, ret
str	文字列操作	repz *
cmp	比較	test, cmp
lea	アドレス計算	lea

4.2 特徴設計

本節では，脆弱性修正の分類を行うために利用した，各脆弱性修正の特徴ベクトルの作成手法について述べる．本実験では，収集したデータセットから下記①～③の手順を踏んで，脆弱性修正時の増分箇所を特徴ベクトル化する．下記手順を図にしたものを図 3 に示す．

①増分命令集合の抽出

手順①では最初に，脆弱性修正前後のソースコードの差分から，脆弱性修正が行われた関数とその名前を特定し，そこを修正箇所と定める．次に，コンパイルして得た脆弱性修正前後の実行ファイルからそれぞれ，脆弱性箇所と定めた関数に含まれる命令を抽出する．

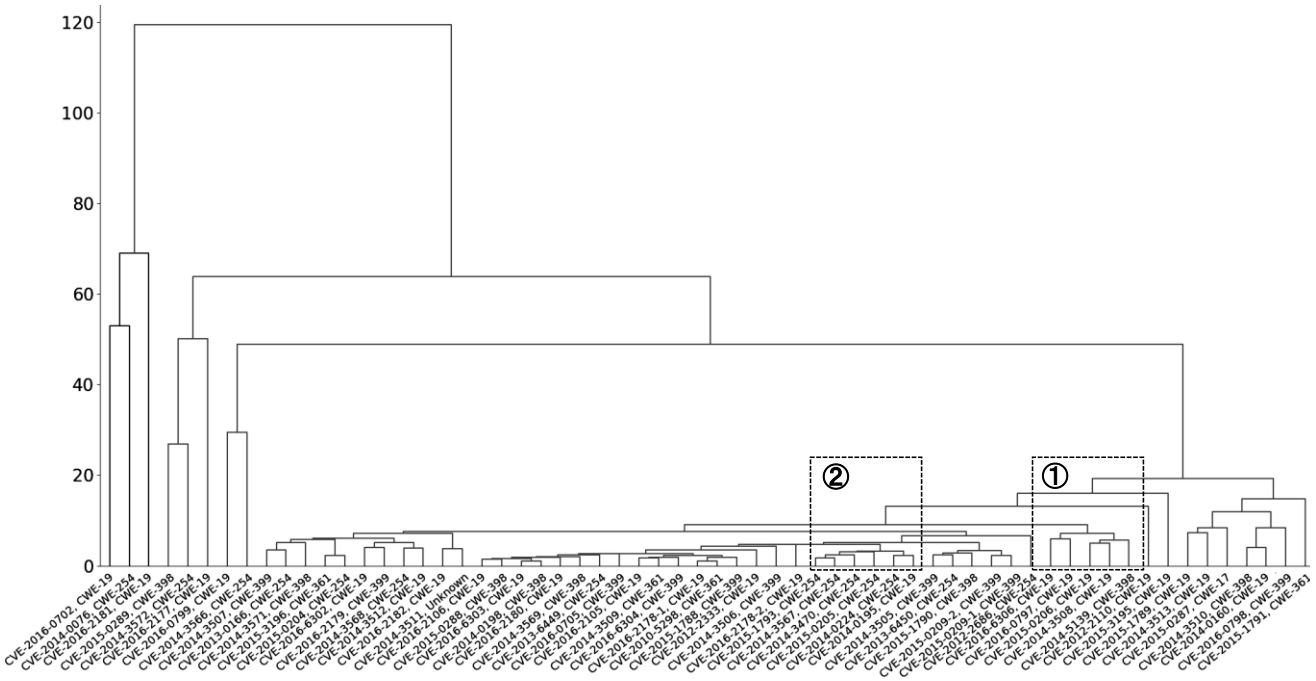

図 4: クラスタリング結果のデンドログラム

最後に，脆弱性の修正前と修正後の脆弱性箇所の命令集合の差分から，修正後に増加した命令を取得する．

②増分命令集合を正規化

次に手順①で抽出した増分命令集合を，筆者らで定めた表 1 の正規化規則に基づき正規化する．正規化を行う理由としては，命令をより抽象的に扱うためである．また，機械語命令の種類は数多あるが，本実験では取得した増分で現れた種類の命令のみを，正規化対象の範囲とする．

③正規化命令の出現頻度を集計

最後に手順③では，手順②で得た正規化命令集合に対して，各命令の出現頻度を集計する．本実験では，この集計した結果を各脆弱性箇所の特徴ベクトルとして取り扱う．

この抽出手法のメリットとしては, 3.2 節で述べたコンパイラなどの最適化によって生じる表記的な変化による影響を最小化するためである．修正前後の正規化された増分命令集合を利用することで, 基本ブロックや命令の順序/変化, オペランドの変化に対応することが言える．ただし本手法では，関数のインライン展開やループ展開による影響は排除出来ないため，改善の余地は残る．

4.3 分類結果とその考察

データセットから抽出した特徴ベクトルに対して，クラスタ分析を行った結果のデンドログラムを図 4 に示す．図では分類結果の傾向を示すため，脆弱性識別子(CVE-ID)に加え，各脆弱性の種類を一意に示す識別子である CWE-ID[12]を各データのラベルとして利用した．ただし今回利用した CWE-ID は，各脆弱性識別子(CVE-ID)に直接紐づけられているものではない．CWE-ID は木構造に割り振られており[13]，各 CWE-ID にはその脆弱性の概念を内包する親となる CWE-ID が存在する．例えば，CWE-119(バッファエラー)や CWE-190(整数オーバーフロー)のルートノードは CWE-19(データ処理)になる．今回は，脆弱性の種類をより抽象化して扱うために CVE-ID に紐づけられた CWE-ID をそのまま使うのでは無く，ルートノードの CWE-ID に変換したものを利用した．また，ルートノードが存在しないものは，変換せずそのまま利用した．利用した CWE-ID 一覧とその説明を表 2 に示す．

分類の結果としては，一部で同種の脆弱性修正が多く含まれているクラスタは見受けられるものの，全体として言える傾向は見られなかった．具体的には，図 4 の①部分で

表 2：CWE-ID と概要

CWE-ID	説明
CWE-17	Code(コード)
CWE-19	Data Handling (データ処理)
CWE-254	Security Features (セキュリティ機能)
CWE-361	Time and State (時間とステータス)
CWE-398	Indicator of Poor Code Quality (貧弱なコード)
CWE-399	Resource Management Errors (リソース管理の問題)

表 3: クラスタ①②の内訳

クラスタ	CWE-ID	CVE-ID
①	CWE-19	CVE-2016-6306
	CWE-19	CVE-2016-0797
	CWE-19	CVE-2015-0206
	CWE-19	CVE-2014-3508
	CWE-398	CVE-2014-5139
②	CWE-254	CVE-2015-1793
	CWE-254	CVE-2014-3567
	CWE-254	CVE-2014-3470
	CWE-254	CVE-2015-0205
	CWE-254	CVE-2014-0224
	CWE-19	CVE-2014-0195

表 4: クラスタ①の各データの内訳

CVE-ID	特徴
CVE-2016-6306	jump: 9, trans: 7, cmp: 6, lea: 4, arith: 3, func: 1
CVE-2016-0797	jump: 7, lea: 4, cmp: 4, trans: 2, arith: 2,
CVE-2015-0206	jump: 7, trans: 5, func: 4, stack: 4, cmp: 4, arith: 2, lea: 1
CVE-2014-3508	jump: 9, trans: 5, cmp: 5, stack: 4, nop: 3, lea: 2, arith: 2, bop: 1, func: 1
CVE-2014-5139	jump: 7, stack: 4, cmp: 3, logical: 2, trans: 2, nop: 2, func': 1

```
@@ -190,11 +189,7 @@ int BN_hex2bn(BIGNUM **bn,
        }
+       for (i = 0; i <= (INT_MAX/4) &&
+            isxdigit((unsigned char)a[i]); i++)
+            continue;
+
+       if (i > INT_MAX/4)
+            goto err;
-       for (i = 0; isxdigit((unsigned char)a[i]); i++) ;
```

図 5: CVE-2016-0797

表 5: クラスタ②の各データの内訳

CVE-ID	特徴
CVE-2015-1793	trans: 4, jump: 3, ctrans: 1, nop: 1, func: 1
CVE-2014-3567	trans: 4, jump: 2, func: 1
CVE-2014-3470	trans: 4, jump: 2, nop: 2, lea: 1, func: 1, cmp: 1
CVE-2015-0205	trans: 5, func: 1
CVE-2014-0224	trans: 5, jump: 3, logical: 2, cmp: 1
CVE-2014-0195	trans: 6, jump: 3, cmp: 1

```
@@ -2512,13 +2512,6 @@ int ssl3_send_client_key_exchange
         int field_size = 0;

+       if (s->session->sess_cert == NULL)
+            {
+            ssl3_send_alert(s,SSL3_AL_FATAL,
                             SSL_AD_UNEXPECTED_MESSAGE);
+            SSLerr(SSL_F_SSL3_SEND_CLIENT_KEY_EXCHANGE,
                             SSL_R_UNEXPECTED_MESSAGE);
+            goto err;
+            }
```

図 6: CVE-2014-3470

は，CWE-19 に紐づく脆弱性修正が連続して並んでいるのが見て取れる．また②の部分では，CWE-254 が多く含まれているのが見て取れる．①，②を抜粋したものを表 3 に示す．次に，それぞれクラスタで見られた特徴について述べた後に，考察を述べる．

ケーススタディ１：クラスタ①(CWE-19 群)の特徴

クラスタ①に含まれる各データの特徴を表 4 に示す．全体的に言える傾向としては,比較命令と分岐命令に加えて，算術演算命令が一定数以上含まれているということである．ただし，CVE-2014-5139 に関しては算術演算命令が含まれて居ないが，それ以外の命令の構成が類似していたため，CWE-398 にも関わらず混ざってしまったものと考えられる．本クラスタの例として，CVE-2016-0797 に対する修正パッチの一部を図 5 に示す．CVE-2016-0797 は整数オーバーフローにあたり，変数の数値を検査する部分などが追加されていた．他の脆弱性修正も確認した所，多くの場合なんらかの値を検査するような修正が加えられていた．

ケーススタディ２：クラスタ②(CWE-254 群)の特徴

クラスタ②に含まれる各データの特徴を表 5 に示す．全体的に言える傾向としては，一定数の転送命令と分岐命令に加えて関数に関係する命令が増えている傾向が見て取れる．本クラスタの例として CVE-2014-3470 に対する修正パッチを図 6 に示す．図 6 の脆弱性修正箇所に加え，他の脆弱性を確認した所，if 文の追加やエラー処理用の関数が追加されている傾向が見られた．

今回前述したように，部分的に同種類の脆弱性修正がまとまる傾向が見られたものの，全体としては無秩序な分類になっている要因の一つとしては，パッチ毎の修正箇所数の違いが挙げられる．パッチによっては，複数箇所に存在している単一の脆弱性を修正している場合などがある．そのため，修正箇所の数が他の同一種の脆弱性と比べ大きく上回る場合，別クラスタとして分類されてしまっていた．

5. 脆弱性修正とその他修正の識別

本章では,3.1 節で述べた②抽出した特徴を利用して脆弱性修正箇所候補を算出する段階で，実際に脆弱性修正とその他修正が識別可能か否かを検証するために行った実験と，その結果の考察について述べる．

5.1 実験概要

実験では，OpenSSL1.0.1 に適用された脆弱性修正と，その他修正を収集し，それを利用して脆弱性修正とその他修正の識別が可能かを検証した．具体的には，線形分類器の一つである SVM(Support Vector Machine)の中でも，誤りを許容するソフトマージン SVM を利用して，収集した脆弱性修正とその他の修正が識別可能か否かを検証した.今回，線形分類器に対する入力データとしては,4.2 節で述べた脆弱性修正の特徴ベクトルに加えて，それと同様の手順で特徴ベクトルを抽出したその他修正箇所を利用する．利用したその他の修正箇所としては，脆弱性修正以外で C のソースコードに対して行われた変更をその他修正と位置づけ，収集したものである．収集したデータセットの内訳を表 6 に示す.

表 6：データセット内訳(括弧内は増分のデータ数)

	脆弱性修正数	その他修正数
OpenSSL1.0.1	77(62)	774(377)

表 7：実験結果

		データ1	データ2	データ3	平均
正解率		0.62	0.54	0.54	0.56
適合率	脆弱性	0.70	0.57	0.57	0.61
	その他	0.59	0.54	0.54	0.55
再現率	脆弱性	0.42	0.39	0.42	0.41
	その他	0.82	0.71	0.68	0.73
F 値	脆弱性	0.53	0.46	0.44	0.47
	その他	0.68	0.61	0.63	0.64

表 6 からも分かるように, 脆弱性修正とその他の修正で, データセットに偏りがあるため, ランダムサンプリングで, 脆弱性修正と同等の数になるように，その他の修正データを抽出し入力データとして扱った．また，今回は SVM のパラメータとして，誤分類の許容度を決定するコストパラメータ C は 10 と設定した. カーネルには RBF 法を利用し，分類の決定境界の複雑さを左右するパラメータγを 0.001 と設定した.

5.2 評価とその考察

本実験では，前節の入力データとパラメータに基づき，10 分割交差検定を行った．脆弱性修正以外のデータセットは，ランダムサンプリングにより抽出されているため，データによって正解率等が左右されることも考慮し，ランダム抽出を 3 回行い，データ毎に交差検定を実施した．その結果得られた全体の正解率(accuracy)と，脆弱性修正とその他の修正に対する適合率(precision)，再現率(recall)，F 値の結果を表 7 に示す.

結果としては，正解率の平均は 56%であり，あまり良いと言える結果にはならなかった．また，脆弱性修正に対する適合率は最高で 70%，平均して 61%とやや高めではあるが, 再現率は平均 41%と低めで, 結果として F 値は平均 0.47 になった. 一方で, その他修正に対する適合率は, 平均 55%と高くは無いが, 再現率は平均 73%と高めの結果となった.

今回，全体として脆弱性修正とその他修正の識別精度が良くは無かった理由としては，正規化された命令のみを特徴として扱ったため，脆弱性修正と，その他修正の違いを捉えきれなかった点にあると言える．例えば，命令だけだと，バグ修正によって追加された比較命令と，脆弱性修正によって追加された比較命令の背景の違いが捕らえきれない．そのため今後は差分命令だけではなく，意味的な違いを捉えるためにも，修正部分のオペランドや構造的な変化も捉えられるような特徴を利用することが方向性として考えられる.

6. 今後の課題とまとめ

本論文では，脆弱性箇所収集を最終目的として，脆弱性修正を含めたプログラム修正が存在する実行ファイル中から，網羅的に脆弱性修正箇所の特定を支援する技術の検討を行った．具体的には，網羅的な脆弱性修正箇所発見を目指して, まずはオープンソースソフトウェアから収集した, 既知の脆弱性修正箇所をクラスタ分析手法で分類し，共通する特徴の有無を調べた．そして次に，線形分類器を利用して，収集した脆弱性修正箇所とその他の修正が識別可能か否かを検証した．その結果，脆弱性修正箇所の分類では部分的に共通する特徴は見られたが，脆弱性修正とその他の修正の識別は正解率 56%と高くはなく，識別に利用する入力データの改善が課題として浮かびあがった.

参考文献

[1] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, Herbert Bos, “VUzzer: Application-aware Evolutionary Fuzzing”, In Proceedings of the Network and Distributed System Security Symposium 2017, 2017

[2] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. “Unleashing Mayhem on Binary Code”, In Proceedings of the 33rd IEEE Symposium on Security and Privacy, 2012.

[3] Jiyong Jang, Abeer Agrawal, and David Brumley, ”ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions”, In Proceedings of the 33rd IEEE Symposium on Security and Privacy, 2012.

[4] Li, Hyuckmin Kwon, Jonghoon Kwon, Heejo Lee, “A Scalable Approach for Vulnerability Discovery Based on Security Patches”, The 5th International Conference on Applications and Techniques for Information Security, Melbourne, Australia, November,2014.

[5] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, Christian Rossow, “Leveraging semantic signatures for bug search in binary programs”, Annual Computer Security Applications Conference, New Orleans, USA, December 2014.

[6] Seulbae Kim, Seunghoon Woo, Heejo Lee, Hakjoo Oh, “VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery”, In Proceedings of the 38th IEEE Symposium on Security and Privacy,2017

[7] CVE-Common Vulnerabilities and Exposures, https://cve.mitre.org/

[8] H. Flake, “Structural comparison of executable objects”, DIMVA, 2004, pp. 161–173.

[9] Turbodiff,CoreSecurity,https://github.com/nihilus/turbodiff

[10] ExploitSpotting: Locating Vulnerabilities Out Of Vendor Patches Automatically, BlackHat USA 2010, Las Vegas, USA. Jeongwook “Matt” Oh

[11] OpenSSL1.0.1 Series Release Notes, https://www.openssl.org/news/openssl-1.0.1-notes.html

[12] CWE-Common Weakness Enumeration, https://cwe.mitre.org/

[13] Development View with Abstractions Highlighted, http://cwe.mitre.org/data/pdf/699_abstraction_colors.pdf