脆弱な正規表現の自動修正技術に関する考察

千田 忠賢[1,a)]　山口 大輔[2,b)]

概要：正規表現エンジンを対象とした DoS 攻撃 (Regular Expression Denial of Service, ReDoS) は正規表現エンジンの振る舞いを悪用し正規表現エンジンを DoS 状態に陥らせる攻撃である．正規表現エンジンは実世界で広く利用されていることから重大な脅威として認識されている．この状況を改善するため既存研究では ReDoS に対して脆弱な正規表現を自動で修正する技術を考案しているが，これらの技術では修正結果を見つけるために幅優先探索を採用しており修正結果の候補を列挙し続けることでしばしば膨大な量のメモリを消費する．本論文では反復深化深さ優先探索を用いた修正アルゴリズムを提案する．反復深化深さ優先探索を用いることで探索中に保持すべき修正結果の候補が探索による再帰の深さ程度に限定される．これにより幅優先探索の場合と比べ探索中に保持すべき修正結果の候補の量が削減され，より少ないメモリ使用量での修正できるようになることが期待できる．評価では実世界の脆弱な正規表現を用い，既存研究との比較によりこれらを実験的に確かめる．

キーワード：脆弱性自動修正，正規表現，正規表現 DoS

On Automatic Repairs of Vulnerable Regular Expressions

Nariyoshi Chida[1,a)]　Daisuke Yamaguchi[2,b)]

Abstract: Regular expression denial of service (ReDoS) attacks are DoS attacks that exploit the behavior of regular expression engines and lead the running time of regular expression engines into a stalling or DoS state. Due to the widespread use of regular expression engines in practice, ReDoS attacks are a significant threat to our society. To rectify this situation, existing work has studied an automatic repair method of regular expressions that are vulnerable to ReDoS attacks. The automatic repair method is based on breadth-first search (BFS) and therefore it may generate and store a lot of candidates for the repair. From this, the BFS-based repair method may consume a lot of memory to store the candidates. In this paper, we introduce an automatic repair based on iterative deepening depth-first search (IDDFS). In the case of IDDFS-based repair, the number of candidates for a repair is at most the depth of the recursion by the search. From this, we expect that the change of the search strategy positively affects the memory consumption. We conducted a comparison of the BFS-based and IDDFS-based repair methods and confirmed how the change of the search strategy affects the memory consumption.

Keywords: Automatic Repairs of Vulnerabilities, Regular Expressions, Regular Expression Denial of Service

1. はじめに

正規表現は文字列のパターンを表す記法として世の中で広く利用されている．例えば，Web アプリケーションでは外部から入力された文字列をサニタイズするため，またテキストデータの中から何らかのパターンに一致する情報を抽出するために正規表現が利用されている．加えて，様々なプログラミング言語が正規表現によるパターンマッチング機能を実装した正規表現エンジンを標準ライブラリとして提供している．そのため，利用者が用途に合わせて様々

[1] NTT セキュリティ・ジャパン
NTT Security Japan
[2] 日本電信電話株式会社
Nippon Telegraph and Telephone Corporation
[a)] nariyoshi.chida@global.ntt
[b)] daisuke.yamaguchi.be@hco.ntt.co.jp

な場面で正規表現を利用できる環境が整っている．

正規表現はその便利さという点で広く世の中の注目を集めている．しかし，2018 年頃から正規表現はその便利さとは異なる点で世の中の注目を集めることとなる．その点とは Regular Expression Denial of Service（ReDoS）攻撃である．ReDoS 攻撃とはバックトラッキングベースの正規表現エンジンを対象とした Denial of Service（DoS）攻撃である．ReDoS 攻撃に対して脆弱な正規表現（以降ではこれを単に脆弱な正規表現と呼ぶ）とその脆弱さを悪用するような文字列を正規表現エンジンに入力することで，正規表現エンジン内でバックトラッキングが多発しパターンマッチングに膨大な時間と資源を費やさなければならない状態に陥らせる．ReDoS 攻撃を受けそのような状況に陥ると，正規表現エンジンおよびその正規表現エンジンを利用しているプログラムは資源を消費し尽くし処理が停止する事態に陥る．ReDoS 攻撃による被害はいくつか報告されている [9], [19]．正規表現エンジンの広い普及状況から，ReDoS 攻撃は我々の社会にとっての重大な脅威として認識されている．

この脅威を取り除くため，ReDoS 攻撃に対して脆弱な正規表現の検知や修正に関する多くの研究が行われている [2], [11], [14], [17], [18], [20], [22]．本論文では脆弱な正規表現の修正に関する研究に着目する．脆弱な正規表現の修正は人が行うには難しいことが知られているため [7]，既存研究では Programming by Examples（PBE）手法を用いて脆弱な正規表現を自動で修正する手法を提案している [6], [13]．特に修正結果が ReDoS 攻撃に対して脆弱でないことを保証するもの [6] については，幅優先探索により修正結果の候補を列挙し，その中から答えを見つけ出す．しかしこの手法では幅優先探索の特徴から探索領域次第では探索中に非常に多くの修正結果の候補を保持しておく必要があるため，そのような場合にはメモリ効率が悪くなることが予想される．

そこで本論文ではメモリ効率の観点から脆弱な正規表現の修正技術についての調査を実施する．また，メモリ効率を改善するための試みとして既存研究の手法をベースに探索方法を幅優先探索から反復深化深さ優先探索に切り替えた手法を提案する．評価により幅優先探索の場合と反復深化深さ優先探索との場合でメモリ効率にどのような違いが出るのか実験的に確かめる．

本論文の以降の構成は次の通りである．2 章では本論文が扱う問題や修正アルゴリズムなどについて大まかに解説する．2 章以降ではそれらを詳細に説明する．まず，3 章では既存研究により定義された修正問題の定義を説明する．次に，4 章ではその修正問題を解くアルゴリズムを紹介する．ここで紹介する修正アルゴリズムは既存研究のものをベースとしているが，それとは異なる探索アルゴリズムを用いる．5 章では前の章で紹介した修正アルゴリズムを実装し既存研究との比較評価を実施した結果を紹介する．6 章ではその結果についての議論する．最後に，7 章では関連研究を紹介する．

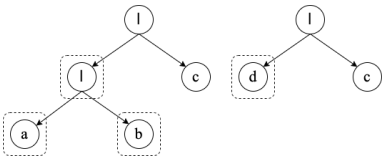

図 1 $r_1 = a|b|c$ の AST(左) と $r_2 = d|c$ の AST(右)

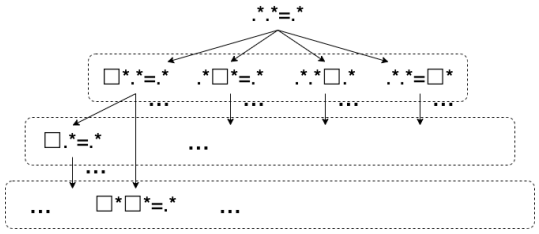

図 2 幅優先探索を用いた場合

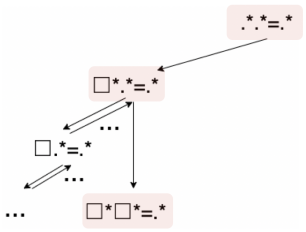

図 3 反復深化深さ優先探索を用いた場合

2. Overview

本論文では Programming by Examples（PBE）手法による脆弱な正規表現の修正問題を扱う[*1]．この修正問題および修正アルゴリズム等について解説するため，Cloudflare の ReDoS インシデント [9] を例として挙げる．このインシデントは，Cloudflare の利用していた Web Application Firewall（WAF）のルールで利用されていた正規表現が ReDoS に対して脆弱であり，かつその脆弱さを突くような入力がきてしまったため ReDoS が起こり Cloudflare のサービスが停止状態に陥ったというものである．このきっかけとなった（脆弱な）正規表現は .*.* = .* という箇所であったと Cloudflare は報告している．ここで .* は任意の文字列に一致する表現であり，= はその文字に一致する表現である．この正規表現は = を含まない十分な長さの文字列などに対して ReDoS を引き起こしうる．

PBE 手法では例を入力として受け取る．例とは利用者の意図，すなわち利用者がどのような修正結果を得たいのか，を自動で修正するプログラムに伝えるためのものである．ここでは正規表現が修正対象であるため，例として修

[*1] より正確には既存研究 [6] にて定義された Real-World Strong 1 Unambiguity（RWS1U）修正問題を扱う．詳細は 3 章にて説明する．

正結果の正規表現に受理して欲しい文字列（以降ではそのような文字列のことを正例と呼ぶ）と拒否して欲しい文字列（以降ではそのような文字列のことを負例と呼ぶ）を用いる．ここでは利用者は .*.* = .* を修正するために以下の正例と負例を用意したものとする．

正例	負例
abcd=	a
efab==	
==a	
a==b==c	

PBE 手法を用いて脆弱な正規表現を自動で修正する既存技術 [6] および本論文で紹介する技術では正規表現，正例，そして負例を入力として受け取り，それぞれの例を満たすような形でかつ ReDoS に対して脆弱でないことを保証した形の正規表現を出力する．これらの修正技術では修正結果となる正規表現を見つけだすため，テンプレートと呼ばれる修正結果の候補を用意し，その形の変形を繰り返し探索を続け答えを探す．既存研究ではこの探索方法として幅優先探索を用いている．すなわち，既存研究では様々な形のテンプレートを Queue と呼ばれるデータ構造[*2]に格納し，順にテンプレートを取り出して答えかどうかの確認する．答えとならないのであればテンプレートの形を変えたものを Queue に格納し探索を続ける．

本論文ではこの探索方法を反復深化深さ優先探索に切り替えた場合について考える．反復深化深さ優先探索では探索を行う際に Queue にテンプレートを格納することはなく，探索により潜る深さの上限を決めつつ注目しているテンプレートのみを深さ優先探索によって変形する．これにより幅優先探索を用いた場合と比べ探索中に保持すべきテンプレートの数が削減されることが期待できるため，修正に必要となるメモリ消費量の改善が期待できる．図 2 と図 3 に上記の例を用いた場合の違いを示す．幅優先探索の場合，少なくとも図 2 に示される点線で囲まれるそれぞれの範囲を全て一時的に Queue に格納し保存する必要がある．反復深化深さ優先探索の場合，図 3 に示されている赤で覆われている箇所のみが一時的に保持される．そのため高々再帰の深さ分しかテンプレートは保持されない．

このようにして探索を続け，答えとなる正規表現を見つけ出す．この例の場合，既存技術 [6] および本論文で紹介する技術は共に
`(?:(?:(?:(?!.).))*)(?:(?:[^=])*)=(?:(?:.)*)`
という正規表現を答えとして出力する．より読みやすくすると，これは `[^=]*=.*` と書ける．この正規表現は上に示されている正例は全て受理し，負例は全て拒否する．加えて，ReDoS に対して脆弱ではない[*3]．

[*2] より正確には，入力の正規表現との距離が小さいものから順に評価するため優先度付き Queue を用いている．

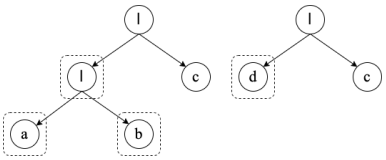

図 4 $r_1 = a|b|c$ の AST(左) と $r_2 = d|c$ の AST(右)

3. 問題定義

本章では本論文で扱う修正問題を紹介する．

3.1 表記

本論文中では以降次の表記を使う．Σ はアルファベットを表す．以降では Σ はアルファベットを表す表記として固定する．$a, b, c \in \Sigma$ は文字を表す．$x, w \in \Sigma^*$ は文字列を表す．ϵ は空文字列を表す．i, j は 0 以上の整数を表す．

3.2 実世界の拡張機能を含む正規表現

正規表現 r の構文を以下に示す．以下の構文において，C は文字の集合を表し，i は 1 以上の整数，x は文字列とする．また，以下の構文の上段に現れている演算子は標準的なものであり，下段に現れている演算子は実世界の拡張機能である．標準的な演算子のみからなる正規表現と実世界の拡張機能までを含んだ正規表現では表現力が異なる．これらを区別するため，以降では標準的な演算子のみからなる正規表現を純粋な正規表現と呼び，実世界の拡張機能までを含むものを実世界の正規表現（もしくは単に正規表現）と呼ぶ．

$$
\begin{aligned}
r ::= {} & [C] \mid \epsilon \mid rr \mid r|r \mid r^* \\
& \mid (r)_i \mid \backslash i \mid (?{=}r) \mid (?!r) \mid (?{<}{=}x) \mid (?{<}!x)
\end{aligned}
$$

$[C]$ は文字の集合 C に属するいずれかの文字に一致するか確認する．以降では $[\{a\}]$ を単に a と表記する．ϵ は空文字列に一致するか確認する．$r_1 r_2$ は連接を表し，r_1 に一致するか確認し，一致するのであれば次に r_2 に一致するか確認する．$r_1|r_2$ は選択を表し，r_1 もしくは r_2 に一致するか確認する．r^* は繰り返しを表し，r を 0 回以上繰り返すことで一致するか確認する．

$(r)_i$ はキャプチャリンググループと呼ばれ，r に一致した文字列を添字 i と関連付けて記録する．$\backslash i$ は後方参照と呼ばれ，同じ添字を持つキャプチャリンググループ $(r)_i$ が添字 i と関連付けた文字列を参照し，それに一致するか確認する．$(?{=}r)$ は肯定先読みと呼ばれ，文字列を消費することなく r に一致するか確認する．$(?!r)$ は否定先読みと

[*3] この正規表現が脆弱でないことはこの正規表現が Real-World Strong 1 Unambiguity（RWS1U）を満たすことにより確認できる．RWS1U を満たす正規表現は ReDoS に対して脆弱でないことが既存研究 [6] により示されている．

呼ばれ，文字列を消費することなく r に一致しないか確認する．$(?<=x)$ は固定長肯定後読みと呼ばれ，文字列を消費することなく今見ている入力文字列上の位置より前の文字列の接尾辞が x に一致するか確認する．$(?<!x)$ は固定長否定後読みと呼ばれ，文字列を消費することなく今見ている入力文字列上の位置より前の文字列の接尾辞が x に一致しないか確認する．これらの演算子の厳密な意味論については既存研究 [6] のものに準拠する．

また実世界で利用される幾つかの糖衣構文は上記の演算子を用いて次のように表現できる: 任意の 1 文字 $. = a_1|a_2|\cdots|a_n$（ここで，$\Sigma = \{a_1, a_2, \cdots, a_n\}$），オプション $r? = r|\epsilon$, 範囲指定の繰り返し $r\{n,m\} = r\cdots(r$ を n 個連続させる $)\cdots rr?\cdots(r?$ を m 個連続させる $)\cdots r?$, そして 1 回以上の繰り返し $r^+ = r^*r$.

正規表現 r の表す文字列集合（すなわち言語）は $L(r)$ と表記され，次のように定義される．厳密な定義は既存研究 [6] のものに準拠する。

$$L(r) = \{w \in \Sigma^* \mid w \text{ は } r \text{ に受理される.}\}$$

3.3 問題

RWS1U 修正問題の定義を以下に示す.

定義 1 (RWS1U 修正問題 [6]) 正規表現 r_1，正例の集合 $P \subseteq \Sigma^*$，負例の集合 $N \subseteq \Sigma^*$ が与えられる．ここで，$P \cap N = \emptyset$ とする．このとき，Real-World Strong 1-Unambiguity 修正問題 (RWS1U 修正問題) とは次の条件を満たす正規表現 r_2 を合成する問題である: (1) r_2 は RWS1U を満たす． (2) $P \subseteq L(r_2)$ となる． (3) $N \cap L(r_2) = \emptyset$ となる． (4) (1) から (3) を満たす r_2 でない全ての正規表現 r_3 について，r_1 と r_3 との距離よりも r_1 と r_2 の距離の方が短い（すなわち，r_1 との距離が最小となる）.
(4) について 2 つの正規表現の距離とは，大まかにはそれぞれの正規表現を抽象構文木（Abstract Syntax Tree, AST）として表した際のそれら 2 つの AST 間で異なる部分木の頂点数の総和である．例えば，正規表現 $r_1 = a|b|c$ と $r_2 = d|c$ を AST として表すと図 4 のようになる．異なる部分木は点線で囲まれる部分であるから r_1 と r_2 の距離は 4 となる．厳密な定義は Pan ら [16] および Chida ら [6] のものに準拠する.

以降では正規表現 r が全ての正例 $w \in P$ について $w \in L(r)$ であるとき，r は正例を満たすと呼ぶ．また，正規表現 r が全ての負例 $w \in N$ について $w \notin L(r)$ であるとき，r は負例を満たすと呼ぶ.

4. 修正アルゴリズム

本章では修正アルゴリズムを紹介する．本修正アルゴリズムは Chida ら [6] の手法をベースにしており，それとの違いは探索方法にある．そのため本章では探索方法について紹介する．その他の処理（すなわち，近似による枝刈，制約解消による答えの有無の確認，テンプレートと呼ばれる構造の変更方法）については Chida らの手法に準拠する．図 5 に探索方法を示す.

```
1: procedure IDDFS(r_i, P, N)
2:     limit = 0
3:     answer = failed
4:     while answer is failed do
5:         answer =DFS(r_i, r_i, P, N, limit)
6:         limit = limit + 1
7:     return answer
8: procedure DFS(r_i, t, P, N, limit)
9:     d =distance(r_i, t)
10:    if d > limit then
11:        return failed
12:    if pruningByApproximation(t, P, N) then
13:        φ =generateConstraints(t, P, N)
14:        if φ は充足可能 then
15:            answer =completion(t, φ)
16:            return answer
17:        n を t の中にある □ の数とする.
18:        for j in 1..n do
19:            for cmp in Components do
20:                t' =replace(t, j, cmp)
21:                result =DFS(r_i, t', P, N, limit)
22:                if result ≠failed then
23:                    return result
24:    m を t を構成する演算子の数とする.
25:    for j in 1..m do
26:        if t の j の演算子を □ に置換可能 then
27:            t' =addHole(t, j)
28:            result =DFS(r_i, t', P, N, limit)
29:            if result ≠failed then
30:                return result
31:    return failed
```

図 5 IDDFS による修正アルゴリズム

本修正アルゴリズムが採用する探索方法は反復深化深さ優先探索であるため，図に示されている探索方法は 2 つの手続き IDDFS と DFS から構成される．IDDFS は修正対象の正規表現 r_i，正例の集合 P，そして負例の集合 N を受け取り，それらの RWS1U 修正問題の答えとなる正規表現を返す．IDDFS は一般的な反復深化深さ優先探索を実施する手続きを行っている．その内部で利用される変数 $limit$ は DFS にて探索を続けることができる深さの上限を

表す．ここでの深さとは入力の正規表現との距離を表す．4-6 行では DFS にて $limit$ の表す深さの上限以下で答えが存在するかどうかを確かめる．$limit$ の値は 0 から順に増加するため答えが RWS1U 修正問題の条件 (4) を満たすことを保証する．

DFS は入力の正規表現 r_i，テンプレート t，正例の集合 P，負例の集合 N，そして深さの上限 $limit$ を受け取り，r_i と t の距離が $limit$ 以下となる領域を探索し答えが存在する場合はその答えを，答えが存在しない場合は答えが存在しないことを表す特殊な文字列 failed を返す．ここでテンプレートとは修正中の正規表現を表すものであり，3.2 節で説明した正規表現の構文にホールと呼ばれる □ で表される演算子を追加したものである．すなわち，テンプレートの構文は次のように定義される．

$$t ::= [C] \mid \epsilon \mid tt \mid t|t \mid t^* \\ \mid (t)_i \mid \backslash i \mid (?{=}t) \mid (?!t) \mid (?{<=}x) \mid (?{<!}x) \mid \square$$

ホール □ は現在その箇所を修正していることを表し，探索の過程で別の形のホールに置換される．加えて，テンプレートが答えを持つかどうか判断する際には，ホールを適切な文字集合 $[C]$ に置換できるかどうかを確認する．3.3 節では正規表現間の距離について説明した．探索の過程では正規表現とテンプレートの距離を利用する．正規表現とテンプレートの距離とは正規表現間の距離と同様に定義される．つまり，それぞれを AST として表記した際の異なる部分木の頂点数の総和として定義される．

DFS が表す手続きは次の通りである．まず最初に入力の正規表現 r_i とテンプレート t の距離を算出する（9 行目）．この距離 d が $limit$ よりも大きい場合はこの状態における探索は打ち切る（10-11 行目）．そうでない場合，探索を続ける．次に，テンプレートに対して Over Approximation と Under Approximation と呼ばれる枝刈りを行う．これは探索ベースの正規表現合成アルゴリズムで広く用いられている枝刈りであり [5], [6], [10], [16]，大まかには Over Approximation とはテンプレート中のホールをなるべく多くの文字列を受理するような表現に置き換え（例えば，.* など），それでも受理できない正例が存在するなら今後ホールをどのような表現に置き換えたとしても正例に一致するような形にはできない（すなわち，答えにはなり得ない）ので探索を打ち切るというものであり，Under Approximation とはテンプレート中のホールをなるべく少ない文字列を受理するような表現に置き換え（例えば (?!.). のような必ず失敗する表現など），それでも拒否できない負例があるなら今後ホールをどのような表現に置き換えたとしても答えにはなり得ないので探索を打ち切るというものである．pruningByApproximation（12 行目）にてこれらの枝刈りを実行し，枝刈りが実行されなかった場合はそのテンプレートは答えを持つ可能性があるため答えを持つかどうかの確認を行う（13-14 行目）．

答えを持つかどうかの確認は Chida [6] らの手法に準拠するが，大まかにはテンプレートのホールを適切な文字集合 $[C]$ に置換することで正例と負例を満たしかつ RWS1U を満たすような正規表現にできるかどうかを制約解消により確認する．もし制約解消の結果答えを持つことが確認できた場合（14 行目），制約解消の結果を利用してテンプレートを答えとなる正規表現に置き換え（15 行目）その正規表現を答えとして返す（16 行目）．答えを持つことが確認できなかった場合，このテンプレートのままでは答えになり得ないためテンプレート中のホールを別の形に置き換え探索を続ける（17-23 行目）．ここで，19 行目の $Components$ はホールの置換対象となるコンポーネントの集合を表し，$Components = \{\square\square, \square|\square, \square^*, (\square)_i, \backslash i, (?{=}\square), (?!\square), (?{<=}\square), (?{<!}\square)\}$ である．加えて，各探索の最後ではテンプレート中の各演算子をホールに置換を試み可能であれば置換し探索を続ける（24-30 行目）．具体的な置換の方法については既存研究のものに準拠するが，大まかにはホールへの置換を試みている演算子が文字集合 $[C]$ なのであればそれをホールに置換する．その他の場合，その演算子の中の表現に文字集合 $[C]$ が無ければホールに置換する．例えば，ホールへの置換を試みている演算子が □|□ の場合これを □ に置換するが，a|□ の場合は文字集合 $a = [\{a\}]$ が含まれるためホールへの置換はしない．

上記の手続きを終えてもなお答えが見つからなかった場合はこの状態では答えがなかったということであるため答えがないことを表す failed を返し探索を打ち切る（31 行目）．

5. 評価

我々は 4 章で紹介した修正アルゴリズムを Java のプログラムとして実装し，Chida ら [6] の幅優先探索による修正アルゴリズムとの比較評価を実施した．本章ではその評価概要と結果を紹介する．以下の評価は以下の環境で実施した．

- **プロセッサ**　2.3GHz クアッドコア Intel Core i7
- **メモリ**　32GB 3733 MHz LPDDR4X
- **OS**　macOS 10.15.7
- **Java**　JDK 9.0.4

本論文では以下の 2 点についての評価を実施した．

- **メモリ効率**　幅優先探索による修正と反復深化深さ優先探索による修正ではメモリ効率にどのような違いが出るか？
- **実行速度**　それぞれの探索方法において，実行速度にはどのような違いがあるか．

これらのアルゴリズムは理論上必ず答えを見つけ出すことができる．実用上は答えを見つけるために非現実的な時間を要求する可能性があるため，タイムアウトを 120 秒に設

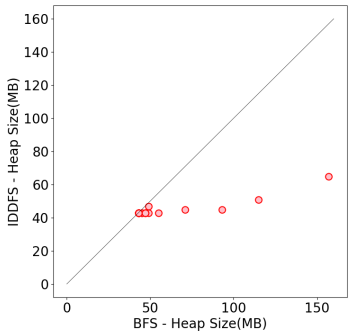

図 6　最小ヒープサイズ: IDDFS(縦軸) と BFS(横軸)

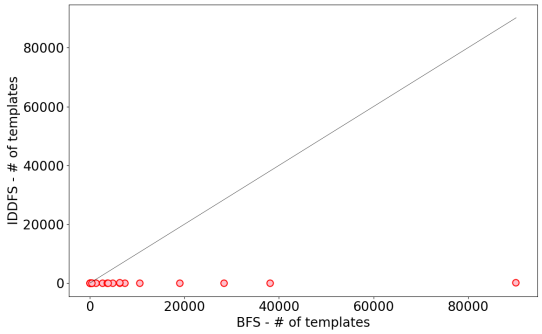

図 7　テンプレート保有数の最大値: IDDFS(縦軸) と BFS(横軸)

定した．

データセット　本評価では Davis らにより収集された実世界で利用されていた脆弱な正規表現からなるデータセットを用いる [7]．このデータセットからランダムに選択した 30 個の正規表現を利用した．このデータセットは正例と負例を持たないため，既存研究 [6] で紹介されている例の自動生成手法により自動で生成したものを例として用いた．

5.1　メモリ効率

探索方法の違いによるメモリ効率の違いを確認するため，それぞれの実装において修正結果を得るために必要な最小のヒープサイズを計測した．修正を得るために必要な最小のヒープサイズを計測する方法は既存研究 [15] で紹介されているものに準拠した．加えて，ヒープサイズに違いがでた場合，探索方法の違いによる探索中に保持すべきテンプレートの数の影響によるものと予想している．この点について実験的に確認するため，修正結果が得られるまでに一時的に保持しておく必要のあったテンプレートの数の最大値も計測した．ここで一時的に保持しておくべき必要のあったテンプレートの数とは，幅優先探索の場合はテンプレートを保持するデータ構造 Queue の修正中のサイズの最大値とし，反復深化深さ優先探索の場合は反復深化深さ優先探索にて呼び出される深さ優先探索の深さの最大値としている．

	修正数 (30)	**平均修正時間** (s)
幅優先探索	23	877.6
反復深化深さ優先探索	22	1419.9

図 8　修正数と平均速度

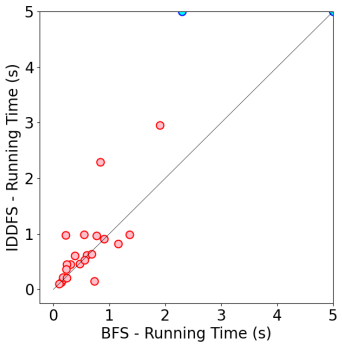

図 9　実行時間: IDDFS(縦軸) と BFS(横軸)

図 6 と図 7 に結果を示す．図 6 では縦軸が反復深化深さ優先探索で修正を行った際の最小のヒープサイズを表し横軸が幅優先探索によるものを表す．いずれか一方でもタイムアウトになったものは最小ヒープサイズを計測できないためプロットしていない．対角線より上に点がある場合は幅優先探索を用いた場合の最小のヒープサイズが反復深化深さ優先探索を用いたものよりも小さかったことを意味する．また，対角線より下に点がある場合はその逆を意味する．本評価においては全てのケースにて反復深化深さ優先探索による修正の方が最小のヒープサイズが小さいという結果になった．

図 7 では縦軸が反復深化深さ優先探索で修正を行った際のテンプレート保有数の最大値であり横軸が幅優先探索で修正を行った際のものを表す．図 6 と同様に対角線の上下でどちらのテンプレートの保有数が多かったのかを確認できる．

5.2　実行速度

次に実行速度の違いを確認する．図 8 にそれぞれの探索方法においてタイムアウトにならずに修正できた正規表現の数とそれらの修正にかかった時間の平均を示す．平均修正時間を求める際には探索方法のどちらか一方でもタイムアウトになったものは除いている．また小数点以下第 2 位を四捨五入している．

また，図 9 にそれぞれの探索方法により実行時間を散布図として表したものを示す．対角線より上にある点は幅優先探索の方が実行速度が速かったことを意味し，対角線より下にある点は反復深化深さ優先探索の方が速かったことを意味する．またグラフのボーダー上にプロットされた青い点は少なくともどちらか一方の探索方法にてタイムアウトしたことを意味する．

6. 議論

実行速度について 5 章では実世界の正規表現を用いて評価を実施し，その範囲においては反復深化深さ優先探索が幅優先探索より遅くなる場合がほとんどであった．これは探索アルゴリズムの動きを考えると反復深化深さ優先探索では深さの上限を更新するたびにその上限以下の範囲を再度探索するため時間がかかるものと考えている．一方でメモリ効率の評価にて上限のヒープサイズを設定した際に幅優先探索よりも反復深化深さ優先探索の方が実行速度が速くなるケースを確認した．これは幅優先探索の場合設定されたヒープサイズでは十分ではなくガベージコレクションが頻発することにより処理が遅くなった結果であると推測している．そうである場合，実行環境によっては幅優先探索よりも反復深さ優先探索を用いた方がメモリ効率・実行速度ともに良くなるケースが存在すると考えられる．

枝刈りについて 本論文では修正に要するメモリ使用量の削減するために反復深化深さ優先探索を導入した．この違いを活かした新たな枝刈りを考えることでメモリ使用量の削減だけではなく探索領域の削減（またそれによる処理速度の改善）も期待できると考えている．例えば，反復深化深さ優先探索では探索領域の深さの上限が決まるため，現在のテンプレートから答えに到達するとしたら最低でも必要となるの距離の最小値などを見積もることができれば現在の距離と合わせることで上限を超えるかどうかが確認できる．超える場合はその上限においては答えとなり得ないため枝刈りができる．このような枝刈りが可能でないか現在検討中である．

7. 関連研究

ReDoS に対して脆弱な正規表現の修正技術

Merwe ら [21] は実世界の拡張機能を含まない正規表現を対象に修正前後で表す言語に影響を与えずに正規表現を ReDoS に対して脆弱でないものに変換する手法を提案している．この手法は正規表現を入力として受け取り，その正規表現を決定性有限オートマトンに変換した後論文中で紹介されている方法で再び正規表現に戻すことで脆弱性を取り除くというものである．しかしこの手法は修正の過程で正規表現を決定性有限オートマトンにしていることから，実世界の拡張機能であって表現力を変えてしまうような演算子を含む正規表現には利用できない．

Li ら [13] は PBE 手法により ReDoS に対して脆弱な正規表現を例を満たす形の 1-unambiguous な正規表現 [3], [4] に変換することで ReDoS に対する脆弱性の緩和を試みる手法を提案した．この技術も実世界の拡張を含まない純粋な正規表現を対象としたものである．加えて，Li ら [12] は正規表現の部分表現を解析し脆弱と思わしき箇所を修正するという方法で ReDoS に対する脆弱性の修正を試みる技術も提案している．

Chida ら [6] は PBE 手法により ReDoS に対して脆弱な正規表現を例を満たす形であって Real-World Strong 1-Unambiguity（RWS1U）を満たす形の正規表現に変換するものである．RWS1U を満たす正規表現は ReDoS に対して脆弱でないことが証明されているためこの提案手法による修正結果は ReDoS に対して脆弱でないことが保証されている．

本論文では Chida らの手法をベースに探索方法を幅優先探索から反復深化深さ優先探索に切り替えることで探索中に保持すべきテンプレートの数を削減する手法を提案した．

その他の ReDoS 対策技術

ReDoS に対して脆弱な正規表現を修正する研究の他に，正規表現を実行する環境に対して工夫をすることで ReDoS による影響の緩和を試みる研究が存在する [1], [8]．これらの研究は正規表現を実行する環境を改良することにより ReDoS への対策を試みるものであるため，実際に世の中で利用するにあたってはその論文で提案されている技術を実装し導入する必要があり，実用上導入するためのハードルが高いものと考えられる．

またこういった研究の他にも実世界で利用されている正規表現では正規表現エンジンの実行時間に対して上限を設けるようなオプションを導入しているものも存在する．設定された上限の時間になっても実行中であった場合，その実行を打ち切りエラーを返すというものである．このオプションは比較的容易に ReDoS による影響の緩和が実現できることが期待できるが，上限を幾つに設定すれば良いのかはその正規表現の ReDoS に対しての脆弱さに依存しており，その脆弱さを求める事は未だ非自明な問題である．上限を適切に設定できない場合，本来受理して欲しい入力を拒否するなどのバグになりえる．加えて，もし上限が大きすぎた場合は ReDoS の影響を受ける [12]．

8. おわりに

本論文では反復深化深さ優先探索による ReDoS に対して脆弱な正規表現の自動修正技術を紹介した．また幅優先探索により実装された既存研究との比較評価により我々の用意したテストケースの範囲では反復深化深さ優先探索の方が修正に必要となる最小のヒープサイズが幅優先探索のものよりも小さくなることを確認した．その理由として修正する際に保持すべきテンプレートの数が削減されることが考えられる．一方で実行時間については幅優先探索の方が反復深化深さ優先探索よりも速くなることを確認した．しかしヒープサイズの上限を指定した場合，一部のケースにおいては実行時間においても反復深化深さ優先探索を用いたものの方が速くなることも確認している．そのため，メモリ環境が制限される状況下では反復深化深さ優先探索による修正アルゴリズムを採用することでよりメモリ効率・

実行速度の両面で効率的に修正できることが期待できる．

参考文献

[1] Bai, Z., Wang, K., Zhu, H., Cao, Y. and Jin, X.: Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks, *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1575–1588 (online), DOI: 10.1109/SP40001.2021.00077 (2021).

[2] Berglund, M., Drewes, F. and van der Merwe, B.: Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching, *Electronic Proceedings in Theoretical Computer Science*, Vol. 151, pp. 109–123 (online), DOI: 10.4204/eptcs.151.7 (2014).

[3] Brüggemann-Klein, A. and Wood, D.: One-Unambiguous Regular Languages, *Information and Computation*, Vol. 140, No. 2, pp. 229–253 (online), DOI: https://doi.org/10.1006/inco.1997.2688 (1998).

[4] Brüggemann-Klein, A.: Unambiguity of extended regular expressions in SGML document grammars, *Algorithms—ESA '93* (Lengauer, T., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 73–84 (1993).

[5] Chen, Q., Wang, X., Ye, X., Durrett, G. and Dillig, I.: Multi-Modal Synthesis of Regular Expressions, *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, New York, NY, USA, Association for Computing Machinery, p. 487–502 (online), DOI: 10.1145/3385412.3385988 (2020).

[6] Chida, N. and Terauchi, T.: Repairing DoS Vulnerability of Real-World Regexes, *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 1049–1066 (online), DOI: 10.1109/SP46214.2022.00061 (2022).

[7] Davis, J. C., Coghlan, C. A., Servant, F. and Lee, D.: The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale, *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, New York, NY, USA, Association for Computing Machinery, p. 246–256 (online), DOI: 10.1145/3236024.3236027 (2018).

[8] Davis, J. C., Servant, F. and Lee, D.: Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS), *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1–17 (online), DOI: 10.1109/SP40001.2021.00032 (2021).

[9] Graham-Cumming, J.: Details of the Cloudflare outage on July 2, 2019 (2019). URL `https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/`.

[10] Lee, M., So, S. and Oh, H.: Synthesizing Regular Expressions from Examples for Introductory Automata Assignments, *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, New York, NY, USA, Association for Computing Machinery, p. 70–80 (online), DOI: 10.1145/2993236.2993244 (2016).

[11] Li, Y., Chen, Z., Cao, J., Xu, Z., Peng, Q., Chen, H., Chen, L. and Cheung, S.-C.: ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection, *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, pp. 3847–3864 (2021).

[12] Li, Y., Sun, Y., Xu, Z., Cao, J., Li, Y., Li, R., Chen, H., Cheung, S.-C., Liu, Y. and Xiao, Y.: RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix, *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, USENIX Association, pp. 4183–4200 (2022).

[13] Li, Y., Xu, Z., Cao, J., Chen, H., Ge, T., Cheung, S.-C. and Zhao, H.: FlashRegex: Deducing Anti-ReDoS Regexes from Examples, *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, New York, NY, USA, Association for Computing Machinery, p. 659–671 (online), DOI: 10.1145/3324884.3416556 (2020).

[14] Liu, Y., Zhang, M. and Meng, W.: Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities, *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1468–1484 (online), DOI: 10.1109/SP40001.2021.00062 (2021).

[15] Mizushima, K., Maeda, A. and Yamaguchi, Y.: Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space, *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, New York, NY, USA, Association for Computing Machinery, p. 29–36 (online), DOI: 10.1145/1806672.1806679 (2010).

[16] Pan, R., Hu, Q., Xu, G. and D'Antoni, L.: Automatic Repair of Regular Expressions, *Proc. ACM Program. Lang.*, Vol. 3, No. OOPSLA (online), DOI: 10.1145/3360565 (2019).

[17] Rathnayake, A. and Thielecke, H.: Static Analysis for Regular Expression Exponential Runtime via Substructural Logics (Extended) (2014).

[18] Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X. and Lu, J.: ReScue: Crafting Regular Expression DoS Attacks, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, New York, NY, USA, Association for Computing Machinery, p. 225–235 (online), DOI: 10.1145/3238147.3238159 (2018).

[19] StackOverflow: Outage Postmortem July 20, 2016 (2016). URL `https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016`.

[20] Sugiyama, S. and Minamide, Y.: Checking Time Linearity of Regular Expression Matching Based on Backtracking, *Information and Media Technologies*, Vol. 9, No. 3, pp. 222–232 (online), DOI: 10.11185/imt.9.222 (2014).

[21] van der Merwe, B., Weideman, N. and Berglund, M.: Turning Evil Regexes Harmless, *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '17, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3129416.3129440 (2017).

[22] Wüstholz, V., Olivo, O., Heule, M. J. and Dillig, I.: Static Detection of DoS Vulnerabilities in Programs That Use Regular Expressions, *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, Berlin, Heidelberg, Springer-Verlag, p. 3–20 (online), DOI: 10.1007/978-3-662-54580-5$_1$(2017).