ReDoS 脆弱な正規表現の修正

情報理工学科 寺内研究室所属 富家功一朗

2023 年 1 月 31 日

概要

ReDoS 攻撃とは文字列のパターンマッチングに膨大な時間がかかるような正規表現を悪用することで，ウェブサイトやサーバーの計算リソースを奪う攻撃のことである．ReDoS 脆弱性を孕んだ正規表現により，実際にサーバーがダウンした事例も少なくない．ReDoS 脆弱な正規表現を修正する手法は既に存在するが，問題点も存在する．そこで，本研究ではその問題点を克服すべく，新たな修正方法を提案する．

目次

第 1 章

はじめに

正規表現は文字列の検索，置換などテキスト処理を行うための強力な手段である．そのような利便性を持つ正規表現であるが, 近年, 正規表現の脆弱性を利用した ReDoS(regular expression denial-of-service) 攻撃が度々発生している．ReDoS 攻撃とは，文字列のパターンマッチングに膨大な時間がかかってしまうような正規表現を悪用し，計算資源を占有することで，ウェブサイトやサーバーの可用性を侵害する攻撃のことである．2016 年には Stack Overflow のサーバーが，空白を 2 万個含む投稿によりダウンした [1]．2019 年には Cloudflare が提供する CDN が，ファイヤーウォールに追加した脆弱性な正規表現によりダウンした [2].

ReDoS 脆弱性に対処するためにはどうしたら良いだろうか．もちろんユーザーが独自に作成した正規表現を使わない，正規表現ライブラリを最新にするなどして予防することはできるだろう．しかし，そのようなことをせずとも正規表現を自動で非脆弱なものに修正できないだろうか．本研究ではこの問いに答えるべく，正規表現の自動修正方法を考案した．

2 章では既存研究，本研究で提案する手法を理解するために必要な知識について説明を行う．具体的には正規表現，NFA，正規表現の等価性判定についての説明を行う．

3 章では ReDoS 脆弱性についての既存研究の紹介を行う．具体的には ReDoS 脆弱性についての定式化，ReDoS 脆弱な正規表現を修正するプログラム REMEDY の紹介を行う．

4 章では ReDoS 脆弱な正規表現の新たな修正方法について提案する．REMEDY と正規表現の等価性判定器 RegEq を使用し，脆弱な正規表現をそれと意味的に等価でかつ非脆弱な正規表現へと修正することを試みる．

5 章では 4 章で提案した手法を用いて，実際に脆弱な正規表現が非脆弱化できるかどう

か実験する.

6 章では本研究のまとめと今後の課題について述べる.

本論文における貢献

- 脆弱な正規表現の修正
  ReDoS 脆弱な正規表現を修正するプログラム REMEDY と正規表現の等価性判定器 RegEq を用いて，脆弱な正規表現を意味的な等しさを保ったまま非脆弱化することに成功した.

第 2 章

準備

2.1 正規表現

正規表現とは言語 (文字列の集合) を一つの文字列で表現する方法である．正規表現は文字列のパターンマッチングなどに使われ，その実用性の高さから多くのプログラミング言語で扱うことができる．実世界のプログラミング言語で使用されるような正規表現の中には，正規言語 (ある正規表現にマッチする文字列全体の集合) 以上の表現力を持つものもある (後方参照の機能を提供している正規表現など) が，本論文では次に示すように純粋な正規表現を対象とする．

2.1.1 正規表現の構文

本論文で扱う正規表現の構文を以下に示す．

$$
\begin{array}{lll}
\text{正規表現} \quad r & := & \varepsilon \\
 & & \mid c \\
 & & \mid r_1 \mid r_2 \\
 & & \mid r_1 r_2 \\
 & & \mid r^*
\end{array}
$$

図 2.1 正規表現の構文

ε は空列を表す定数，c は任意の 1 文字 (本論文では ASCII コードを想定している) を表す定数，r, r_1, r_2 は正規表現を表す変数とする．$r_1 \mid r_2$ は選択，すなわち r_1 に含まれ

る文字列の集合と r_2 に含まれる文字列の集合の和集合を表す．r_1r_2 は連接，すなわち r_1 に含まれる文字列に r_2 に含まれる文字列をつなげてできる文字列の集合を表す．r^* はクリーネ閉包，すなわち r に含まれる文字列を 0 個以上つなげてできる文字列の集合を表す．

また，本論文では先の正規表現で表現可能ないくつかの拡張機能を搭載している．以下にどのような拡張を搭載したか述べる．

- .　任意の 1 文字にマッチすることを表す．
- $[a_ia_{i+1}\dots a_j]$　$a_i, a_{i+1}, \dots, a_j$ のうちのどれか 1 文字にマッチすることを表す．特に，$[a_i\text{-}a_j]$ は文字コード上で i から j 番目にある文字のうちどれか 1 文字にマッチすることを表す．
- $[\hat{}a_ia_{i+1}\dots a_j]$　$a_i, a_{i+1}, \dots, a_j$ 以外のどれか 1 文字にマッチすることを表す．
- $r?$　正規表現 r に属する文字列が 0 回または 1 回出現することを表す．

2.1.2　正規表現を用いた文字列のパターンマッチング例

正規表現を用いた文字列のパターンマッチング例を見る．$a \mid b$ という正規表現には a, b といった文字列がマッチする．ab という正規表現には ab といった文字列がマッチする．a^* という正規表現には $\varepsilon, a, aa, aaa, \dots$ といった文字列がマッチする．最後に，$(a|bc)^*$ という正規表現には $\varepsilon, a, bc, aa, abc, bca, bcbc, \dots$ といった文字列がマッチする．

また，本論文で扱うような正規表現においてパターンマッチングに成功した (正規表現に受理されるとも言う) 文字列集合を正規言語と呼ぶ．そして，ある正規表現 $\mathcal{A}$ に受理される言語のことを $\mathcal{L}(\mathcal{A})$ と書くことにする．

2.2　NFA

NFA(Nondeterministic Finite Automaton) は有限オートマトンの一種で，ある状態から入力を受け取ったとき，次の状態への遷移が一意に決まらないことがあるものである．正規表現はそれと等価な NFA へと変換することが可能であるので，今後は正規表現のままではなくそれと等価な NFA を用いることにより議論を進めることがある．

2.2.1　NFA の定義

先行研究で紹介されていた NFA の定義を以下に示す．

定義 1（(ε 遷移なし)NFA）　NFA $\mathcal{A}$ は 5 つの組 $(Q, \Sigma, \Delta, q_0, F)$ からなる．Q は状態の有限集合，Σ はアルファベット (文字) の有限集合，$\Delta : Q \times \Sigma \to 2^Q$ は遷移関数である．$q_0 \in Q$ を初期状態，$F \subseteq Q$ を受理状態の集合とする．また，もし $q' \in \Delta(q, l)$ なら (q, l, q') をラベルを経由した遷移と呼ぶこととする．

また，関連してパスという用語の定義も行う．

定義 2（(ε 遷移なし NFA における) パス）　NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ のパス π を遷移列，すなわち $(q_1, l_1, q_2), \ldots, (q_{m-1}, l_{m-1}, q_m)$ とする．ただし，$q_i \in Q$,$l_i \in \Sigma$,$q_{i+1} \in \Delta(q_i, l_i)$ である．つまり，π は q_i から始まって q_m で終わるパスを表す．また，$\mathrm{labels}(\pi)$ はラベル列，$(l_1, \ldots, l_{m-1})$ を意味するものとする．

この NFA の定義は一般的ではない．Σ に空列 ε を加えた $\Sigma \cup \{\varepsilon\}$ をアルファベット集合として 5 つ組 $(Q, \Sigma \cup \{\varepsilon\}, \Delta, q_0, F)$ を NFA の定義とすることが多い．すなわち，

定義 3（NFA）　NFA $\mathcal{A}$ は 5 つの組 $(Q, \Sigma \cup \{\varepsilon\}, \Delta, q_0, F)$ からなる．Q は状態の有限集合，Σ はアルファベット (文字) の有限集合，ε は空列，$\Delta : Q \times \Sigma \to 2^Q$ は遷移関数である．$q_0 \in Q$ を初期状態，$F \subseteq Q$ を受理状態の集合とする．また，もし $q' \in \Delta(q, l)$ なら (q, l, q') をラベルを経由した遷移と呼ぶこととする．

また，パスも以下のように書き換える．

定義 4（パス）　NFA $\mathcal{A} = ((Q, \Sigma \cup \{\varepsilon\}, \Delta, q_0, F))$ のパス π を遷移列，すなわち $(q_1, l_1, q_2), \ldots, (q_{m-1}, l_{m-1}, q_m)$ とする．ただし，$q_i \in Q$,$l_i \in \Sigma \cup \{\varepsilon\}$,$q_{i+1} \in \Delta(q_i, l_i)$ である．つまり，π は q_i から始まって q_m で終わるパスを表す．また，$\mathrm{labels}(\pi)$ はラベル列，$(l_1, \ldots, l_{m-1})$ を意味するものとする．

今後はただ NFA とだけ記載した場合，ε 遷移ありの NFA を指す．

NFA に文字列を入力したとき，初期状態 q_0 から始まって一つずつ文字を読み取って遷移していき，ちょうど受理状態に到達するとき，その文字列は NFA に受理されるという．受理状態以外の状態で読み取りが終了した場合，その文字列は NFA に拒絶されるという．NFA に受理される文字列全体の集合を NFA の受理する言語という．

2.2.2 ε 遷移なし NFA の例

(ε 遷移なし)NFA $\mathcal{A}_1$ を $(\{q, q'\}, \{a, b\}, \Delta, q, \{q'\})$ とする．なお，遷移関数 Δ は以下のような状態遷移表で表されるものとする．表の縦軸が状態，横軸がアルファベットを表す．ある状態とあるアルファベットが交差するセルは，ある状態からあるアルファベットを読んで遷移する状態の集合を表している．例えば，状態 q からアルファベット a を読んで遷移する状態の集合は $\{q, q'\}$ である．

表 2.1　遷移関数 Δ

	a	b
q	$\{q, q'\}$	$\{\}$
q'	$\{\}$	$\{q, q'\}$

NFA $\mathcal{A}_1$ をグラフにすると以下のようになる．

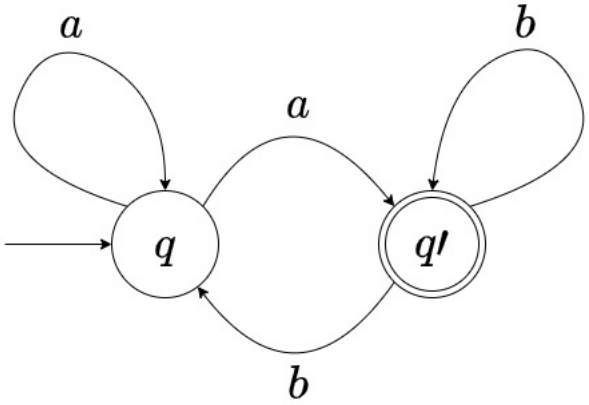

図 2.2　NFA $\mathcal{A}_1$

この NFA が受理する言語は $\{a, aa, ab, aaa, aab, aba, abb, \ldots\}$ である．これは正規表現 $a(a|b)^*$ に対応する．

2.3　正規表現の等価性判定

2 つの正規表現が等価であるとはどういうことか，また等価性判定がどのようなアルゴリズムで行われているのかを述べる．

2.3.1 正規表現の等価性

ある 2 つの正規表現 $\mathcal{A}, \mathcal{B}$ が存在し，それらの正規表現にマッチする文字列の集合 (つまり正規言語) が等しいとき，すなわち $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ であるとき，$\mathcal{A}, \mathcal{B}$ は意味的に等価であるという．以下に例を挙げる．

2 つの正規表現が構文的に等しい場合，例えば $\mathcal{A} = x^*, \mathcal{B} = x^*$ としたとき，$\mathcal{L}(x^*) = \{\varepsilon, x, xx, xxx, \ldots\}$ よりこれらは意味的にも等しい．

2 つの正規表現が構文的に等しくなく，意味的に等しい場合について考える．例えば $\mathcal{A} = x^*, \mathcal{B} = x^*x^*$ としたとき，これらは構文的に等しくないが，$\mathcal{L}(x^*) = \mathcal{L}(x^*x^*) = \{\varepsilon, x, xx, xxx, \ldots\}$ より意味的には等しい．

2.3.2 正規表現の等価性判定アルゴリズム

正規表現の等価性判定を行う方法はいくつか知られている．

例えば，有名なアルゴリズムとして次のようなものがある．まず 2 つの正規表現を NFA 化する．次に 2 つの NFA を DFA(Deterministic Finite Automaton，ある状態からの遷移先が一意に定まった有限オートマトン) 化する．さらに 2 つの DFA を最小化 (不要な状態の削減) する．最小化した際に得られる DFA は一意に定まることが知られている．この事実を利用し，2 つの最小 DFA が一致するかどうかを確認することで等価性判定を行うことができる．上記のようなアルゴリズムは [3] で紹介されている．

また，他にも正規表現の微分 ([4],[5]) を利用した等価性判定アルゴリズム [6] がある．

しかし，本研究では先の 2 つのアルゴリズムではなく，次のようなアルゴリズムを用いることにする (このアルゴリズムも有名で [7] で紹介されている)．

1. 2 つの正規表現 $\mathcal{A}, \mathcal{B}$ を NFA 化し，さらに DFA 化し，それらを $\mathcal{A}_{DFA}, \mathcal{B}_{DFA}$ とする．
2. $\mathcal{A}_{DFA}$ と $\mathcal{B}_{DFA}$ の否定を取った DFA($\overline{\mathcal{B}_{DFA}}$ と表記する) との積を取り，これを $\mathcal{A}_{DFA} \cap \overline{\mathcal{B}_{DFA}}$ と表記する．また，同様に $\overline{\mathcal{A}_{DFA}} \cap \mathcal{B}_{DFA}$ も構成する．なお，正規言語は積集合演算，補集合演算について閉じているので，これらもまた DFA である．
3. $\mathcal{L}(\mathcal{A}_{DFA} \cap \overline{\mathcal{B}_{DFA}}) = \varnothing$ かつ $\mathcal{L}(\overline{\mathcal{A}_{DFA}} \cap \mathcal{B}_{DFA}) = \varnothing$ なら 2 つの正規表現 $\mathcal{A}, \mathcal{B}$ は等価，そうでないなら等価ではない．

ここで DFA の積，否定を取る工程が発生しているが，具体的な構成方法は [7] を参照

されたい．また，正規言語の空性判定についても [7] を参照されたい．

　このアルゴリズムのメリットとして 2 つの正規表現が等価でないことを示す反例を探しやすいということが挙げられる．後に提案する手法ではこの反例を利用する．

第 3 章

既存研究

3.1 ReDoS 脆弱性

文字列のパターンマッチングにかなり時間がかかってしまうような正規表現が存在することが知られている．そのような正規表現を ReDoS(regular expression denial-of-service) 脆弱な正規表現と呼ぶ．本節ではまず文字列のパターンマッチングはどのようなアルゴリズムで行われているかを紹介し，その次に正規表現の脆弱性について定義する．

3.1.1 バックトラック探索

大抵のプログラミング言語が正規表現ライブラリを提供している．そのうち多くの正規表現ライブラリはバックトラック探索アルゴリズムを用いて文字列のパターンマッチングを行っている．このアルゴリズムにおいてバックトラックとはどのようなものなのか，それを [8] の論文で紹介されていた例を用いて説明する．バックトラック探索アルゴリズムの詳細は [9] を参照されたい (プログラミング言語 Java における，バックトラック探索アルゴリズムを用いた文字列パターンマッチングについて述べられている).

a^*a^*b という正規表現について考える．この正規表現と等価な NFA は以下のように作成できる ([8] より引用).

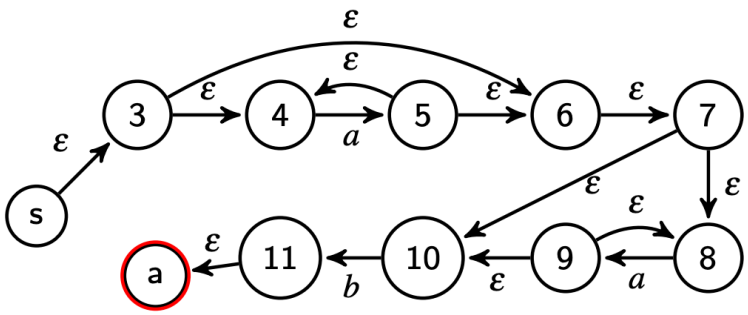

図 3.1　正規表現 a^*a^*b と等価な NFA

この NFA に aaa という文字列を入力として与える (なお，aaa は受理されない).

まず，初期状態 s から遷移できるところまで進む．今回の例では初期状態 s から a という文字を 2 つ消費した後の状態 10 までは遷移できる．これをパスとして表すと以下のようになる．

$$(s, \varepsilon, 3), (3, \varepsilon, 4), (4, a, 5), (5, \varepsilon, 6), (6, \varepsilon, 7), (7, \varepsilon, 8), (8, a, 9), (9, \varepsilon, 10) \tag{3.1}$$

他にも以下のようなパスが存在する．

$$(s, \varepsilon, 3), (3, \varepsilon, 4), (4, a, 5), (5, \varepsilon, 4), (4, a, 5), (5, \varepsilon, 6), (6, \varepsilon, 7), (7, \varepsilon, 10)$$

$$(s, \varepsilon, 3), (3, \varepsilon, 6), (6, \varepsilon, 7), (7, \varepsilon, 8), (8, a, 9), (9, \varepsilon, 8), (8, a, 9), (9, \varepsilon, 10)$$

しかし，状態 10 から先へは文字 b がなければ進めない．このような状況に陥った際，1 手前に戻る，すなわちバックトラックが行われる．具体的に言えば，(3.1) の探索パスで言えば状態 10 から進めないことが分かったので状態 9 に戻ってやり直すということである．状態 9 に戻って，残りの文字 a を読み取った上で状態 a にたどりつけるようなパスを探すが存在しないので，またバックトラックする．このように文字列のパターンマッチングではバックトラックを繰り返し行うことが多々ある．バックトラックの回数が文字列のパターンマッチングの実行時間に大きな影響を与える．

3.1.2　ReDoS 脆弱性の定式化

ReDoS 脆弱性の定式化を行う．Wüstholz らの研究 [10] では，超脆弱な NFA と脆弱な NFA の 2 種類の定義が紹介されていた．本稿では脆弱な NFA についてのみ言及する．なお，本小節での NFA は ε 遷移なし NFA である．

3.1.2.1 脆弱な NFA

脆弱な (ε 遷移なし)NFA について定式化する．

定義 5（脆弱な NFA） NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ が脆弱であることは $\mathcal{A}$ のパスを走査するようなバックトラック探索アルゴリズム MATCH が存在し，MATCH の最悪ケースの複雑さが入力文字列の長さに対して最低でも 2 次関数的になることと同値である．

定理 1 NFA $\mathcal{A}$ が脆弱であることは以下の条件を満たすような 2 つのピボット状態 $q \in Q$ と 3 つのパス π_1, π_2, π_3(ただし，$\pi_1 \neq \pi_2$) が存在することと同値である．

1. π_1 は q で始まり，q で終わる．
2. π_2 は q で始まり，q' で終わる．
3. π_3 は q' で始まり，q' で終わる．
4. labels(π_1)=label(π_2)=label(π_3)．
5. q_0 から q へのパス π_p が存在する．
6. q' から状態 $q_r \notin F$ へのパス π_s が存在する．

証明については [10] を参照されたい．

上記の定理を直感的に表した図を以下に示す ([10] より引用)．

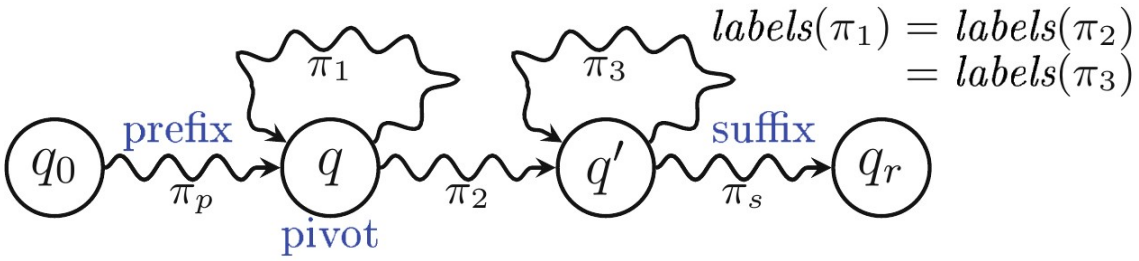

図 3.2　脆弱な NFA の一般的なパターン

定理 1 の性質を満たす NFA が超線形的な動作を引き起こす理由を考える．

$s_0 \cdot s^k \cdot s_1$ という形の攻撃文字列を考える．s_0 が攻撃文字列のプレフィックス labels(π_p)，s_1 が攻撃文字列のサフィックス labels(π_s)，s が攻撃文字列のコア labels(π_1) である．そして，$s_0 \cdot s^k \cdot s_1$ が拒絶される実行パスが存在する ($\pi_p \cdot \pi_1^k \cdot \pi_s$ など)．$T_q(k)$ は q から走査を開始し，文字列 $s^k \cdot s_1$ を拒絶するまでの実行時間 (文字列 s を読むのに 1 単位の時間がかかるとする) を表すものとすると，以下のような漸化式が作れる．

$$T_q(k) = (1 + T_q(k-1)) + (1 + T_{q'}(k-1)) \tag{3.2}$$

式 (3.2) の右辺を詳しくみると，文字列 s は以下の 2 パターンの方法で処理されることが分かる．

1. パス π_1 を通って q に戻る場合，s が 1 単位分処理される．現状 q にいるため，$T_q(k-1)$ 単位分の処理が残る．
2. パス π_2 を通って q' へ行く場合，s が 1 単位分処理される．現状 q' にいるため，$T_{q'}(k-1)$ 単位分の処理が残る．

ここで，$T_{q'}(k)$ の下界は k である．なぜなら，パス $\pi_3^k \pi_s$ は明らかに拒絶状態 q_r に到達できるからである．このことから，

$$T_q(k) \geq T_q(k-1) + k + 1$$

従って，NFA $\mathcal{A}$ において文字列 $s_0 \cdot s^k \cdot s_1$ のパターンマッチングには最低でも k^2 オーダーの時間がかかる．

3.1.2.2 脆弱な NFA の例

脆弱な NFA の例を以下に示す ([10] より引用).

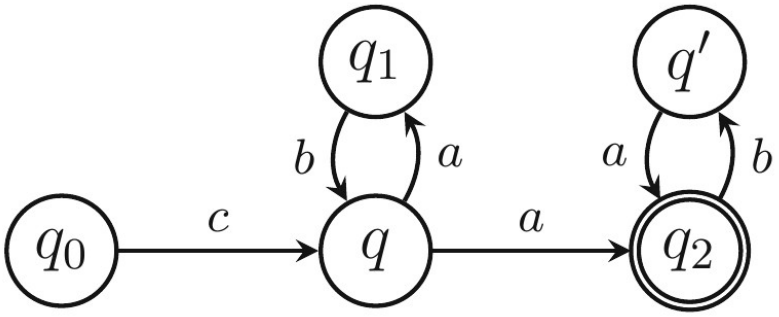

図 3.3　脆弱な NFA の例

π_1 が $(q, a, q_1), (q_1, b, q)$，π_2 が $(q, a, q_2), (q_2, b, q')$，$\pi_3$ が $(q, a, q_2), (q_2, b, q)$ とすると，定理 1 より図 3.3 の NFA は脆弱である．

3.2 REMEDY

プログラマーが ReDoS 脆弱な正規表現を発見したとして，その正規表現をどのように扱いたいだろうか．その正規表現を非脆弱化したいと考える人が多いのではないだろうか．そこで，本章では脆弱な正規表現を非脆弱なものへと変換するツール REMEDY[11] を紹介する．

REMEDY は例を用いて正規表現を修正する programming-by-example(PBE) という手法を用いていて，正規表現の修正手法として多くの研究で注目されている．修正したい正規表現とその正規表現に受理されてほしい文字列例 (positive example，正例) と受理されてほしくない文字列例 (negative example，負例) を用いて正規表現を合成するというのが PBE を用いた正規表現修正のアプローチである．REMEDY は入力として正規表現，positive example，negative example を受けとり，出力として以下の性質を満たすような正規表現を返す．

1. 入力として与えられた positive example，negative example が正しく分類されている (positive example は受理され，negative example は受理されない)
2. real-world strong 1-unambuiguity(RWS1U)
3. 入力として与えられた正規表現に構文的に近い

real-world strong 1-unambuiguity(RWS1U) を満たせば正規表現が非脆弱であることが保証される (今回扱う正規表現は純粋なものなので S1U を満たすだけでも非脆弱であることが保証される)．入力として与えられた正規表現に構文的に近いというのは入力の正規表現 r と出力の正規表現 r' の編集距離が最小になるということである．詳細に言うと，正規表現を構文木としてみなしたとき，編集前の部分木の節点の個数と編集後の部分木の節点の個数の和が最小になるということである．

第 4 章

提案手法

本章では ReDoS 脆弱な正規表現の修正方法を述べる．

4.1 既存の修正方法の問題点

まず，REMEDY は，文字列例を正しく分類し，RWS1U を満たし，入力の正規表現に構文的に近い正規表現を出力として生成するのだった．ここで出力の正規表現は少なくとも入力として与えられた文字列例だけは正しく分類する能力はあるが，その他の文字列については入力の正規表現と同じように分類するかは分からないことに注意されたい．つまり，REMEDY の入力の正規表現と出力の正規表現は意味的に等価でない可能性がある．

正規表現を意味的に等しくかつ非脆弱なものへと修正する方法は既に存在する．[12] では，正規表現と等価な NFA を構成，そして DFA に変換し，その DFA を等価で strongly unambiguous な正規表現に戻すという手法を用いている．しかし，この手法は [12] でも言及されているように，修正後の正規表現のサイズが入力の正規表現のサイズに対して二重指数関数的になる場合もある．正規表現のサイズは文字列のパターンマッチングの処理時間に大きく関係するので，この手法は計算量の観点から望ましくない．

4.2 ReDoS 脆弱な正規表現の新たな修正手法

正規表現のサイズをできるだけ小さく保ったまま，元の正規表現と意味的に等価でかつ非脆弱であるように修正したい．本節ではその修正手法を紹介する．

4.2.1 使用するプログラム

新たな提案手法を実現するために，REMEDY と RegEq[13] というプログラムを使用する．REMEDY については 3 章で説明を行ったとおりである．

RegEq は 2 つの正規表現が与えられたとき，それらが意味的に等価であるか否かを判定するプログラムである．また，等価でない場合，片方の正規表現には受理されるがもう片方の正規表現には受理されない反例を返す．RegEq の等価性判定のアルゴリズムは 2 章で説明を行った [3] のものと同じである．

4.2.2 アルゴリズム

REMEDY と RegEq の 2 つのプログラムを用いて以下のようなアルゴリズムで正規表現を非脆弱化する．

1. (ReDoS 脆弱だと疑われる) 正規表現を REMEDY への入力とする．はじめは positive example，negative example ともに入力として渡さない (つまり positive example の集合，negative example 集合がともに空である).
2. REMEDY から出力された正規表現と元の正規表現を RegEq への入力とし，
   - 等価であるなら修正後の正規表現を返してこのアルゴリズムは終了.
   - 等価でないなら元の正規表現に受理され修正後の正規表現に受理されない文字列を正例に追加し，元の正規表現に受理されず修正後の正規表現に受理される文字列を負例に追加し，それらを修正後の正規表現とともに REMEDY に渡す.
3. 2. を繰り返す.

上記のアルゴリズムを図として表したものを以下に示す.

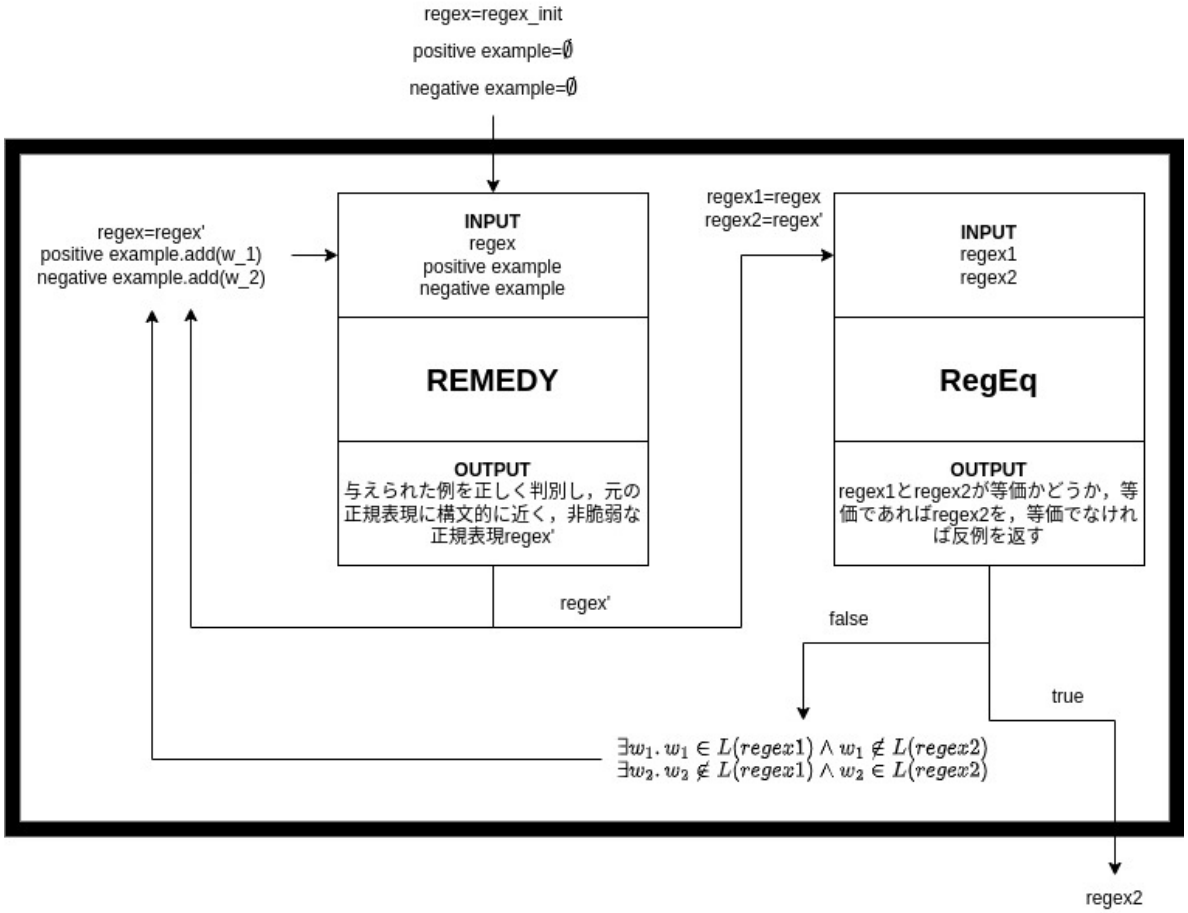

図 4.1　ReDoS 脆弱な正規表現の修正アルゴリズム

このアルゴリズムは停止しない場合がある．それはある正規表現に対してそれと意味的に等価で S1U を満たすようなものが存在しない可能性があるからである．

第 5 章

実験

本章では 4 章で提案した手法を実装し，実際に脆弱な正規表現が非脆弱な正規表現へと修正されていることを確認する．

5.1 実装における諸注意および実行環境

基本的には 4 章で提案した手法をプログラムにしただけである．しかし，RegEq では character set $[a_i a_{i+1} \dots a_j]$ を直接扱うことができないため，これを選択 | でつないだ形で表現している．具体的に言えば，$a_i|a_{i+1}|\dots|a_j$ としている．any . に関しても同様である．

また，実験には計算機として「Xeon Gold6254(3.1GHz)x 2CPU (36 コア), 96GB RAM」を使用した．

5.2 結果

3 章で紹介した，図 3.3 の NFA に対応する正規表現 $c(ab)^*a(ba)^*$ を例に取って実験を行う．当然この正規表現は脆弱である．本研究で実装したプログラムに正規表現 $c(ab)^*a(ba)^*$，positive example = $\varnothing$，negative example = $\varnothing$ を入力すると，以下のような正規表現が出力として返ってきた (実行結果の詳細は付録を参照されたい)．

Listing 5.1 $c(ab)^*a(ba)^*$ を非脆弱化した正規表現

```
1 ca(?:(ba)*)
```

なお，non-capturing group ?: が含まれているが，本研究で扱う純粋な正規表現では意味

を持たないので削除してしまって構わない．従って，整形すると $ca(ba)^*$ という正規表現が得られる．

そして，$ca(ba)^*$ は $c(ab)^*a(ba)^*$ と意味的に等価で，かつ非脆弱 (量化子 $*$ が 1 つしかない，すなわち NFA に変換したときループを 1 つしか持たないため，3 章で紹介した定理 1 を満たさないことが直感的に分かる) であることが分かる．また，修正にかかった時間は 0.533 秒だった．以上のことを表にまとめると以下のようになる．

表 5.1　脆弱な正規表現 $c(ab)^*a(ba)^*$ の修正結果

修正前の正規表現	修正後の正規表現	修正にかかった時間 [s]
$c(ab)^*a(ba)^*$	$ca(ba)^*$	0.533

しかし，この例は REMEDY による初回の修正後の正規表現が偶然修正前の正規表現と意味的に等価であったため，1 ループ目で終了してしまった (REMEDY を実行，その後 RegEq を実行という工程を 1 ループとする)．数ループした果てに目的の正規表現が得られる例を見る．そこで $.^*a^*=$ という正規表現を考える．$.^*a^*=$ を入力として本研究で実装したプログラムを走らせると 10 ループ目にして以下のような結果が得られた (実行結果の詳細は付録を参照されたい)．

Listing 5.2　$.^*a^*=$ を非脆弱化した正規表現

```
1 (?:([^=])*)[=](?:((?:([^=])*)[=])*)
```

整形すると，$[\hat{}=]^*=([\hat{}=]^*=)^*$ のようになる．修正にかかった時間は 55.000 秒だった．以上のことを表にまとめると以下のようになる．

表 5.2　脆弱な正規表現 $.^*a^*=$ の修正結果

修正前の正規表現	修正後の正規表現	修正にかかった時間 [s]
$.^*a^*=$	$[\hat{}=]^*=([\hat{}=]^*=)^*$	55.000

第 6 章

おわりに

6.1 まとめ

正規表現を非脆弱化するプログラム REMEDY と言語等価性判定器 RegEq を用いて，ReDoS 脆弱な正規表現を意味的に等価でかつ非脆弱なものに修正する手法を提案した．また，そのアルゴリズムを実装し，プログラムを走らせることで ReDoS 脆弱な正規表現を意味的な等しさを保ったまま非脆弱化することができた．

6.2 今後の課題

今回のアルゴリズムでは REMEDY への初回の入力を修正したい正規表現としていた．そうではなく次のような手法を試したい．まずはじめに RegEq にて正規表現 ε と修正したい正規表現の比較を行い，そこで得られた反例と正規表現 ε を REMEDY への入力とする．得られた正規表現を修正したい正規表現と RegEq にて比較し，反例を得る．この工程を繰り返し行う．ε を基準に修正したい正規表現と等価な正規表現を生成していく流れとなる．ε から修正したい正規表現へと近づけていくのでよりサイズが小さく非脆弱な正規表現が得られる可能性がある．

また，RegEq の実装上，比較する 2 つの正規表現をどちらも DFA に変換している．しかし，本アルゴリズムでは修正元の正規表現は 1 回 DFA 化するだけで十分である．ループが繰り返されるたびに DFA 化するのは非効率的なので今後改良したい．

さらに，そもそも等価性判定のアルゴリズムとして今回採用したものが適しているのかも疑問に残る．理由は 2 つある．1 つ目は今回使用したアルゴリズムより効率よく等価性判定を行うことのできるものが存在するかもしれないからである．2 つ目は等価性判定器

が等価でないと決定する際に得られる反例が良いものでない可能性があるからである．しかし，現段階ではどのような例を REMEDY に渡すとより小さくより速く非脆弱な正規表現が得られるのかよく分かっていないので今後の課題としたい．

最後に，5.1 節でも述べたとおり，character set を選択 | でつなげた形に変換していた．この方法は恐らく計算量的に好ましくない．RegEq を character set を直接扱えるように変更する，ないしは character set を直接扱えるような言語等価性判定器を 1 から自作する必要があるだろう．

謝辞

本研究を進めるにあたり，担当教員である早稲田大学情報理工学科の寺内多智弘教授には熱心なご指導をいただきました．また，本研究に使用したプログラム REMEDY の作成者千田さんをはじめ，寺内研究室のメンバーからの助言の数々には頭が上がりません．この場を借りて深く感謝申し上げます．

付録 A

実行結果の詳細

実行結果の詳細について示す.

A.1 実行ログ

「remedy」の右隣に書かれている数値は 1 回あたりの REMEDY の実行時間 (秒),「regeq」の右隣に書かれている数値は 1 回あたりの RegEq の実行時間 (秒) を表す．5.1 節でも述べたが，REMEDY と RegEq で扱える正規表現には差異があり，それを埋めるためのプログラムが「pure」である．「pure」の右隣に書かれている数値は 1 回あたりの「pure」の実行時間 (秒) である.

最終行から 2 行目には非脆弱化した正規表現，最終行には修正するのにかかったトータルの時間を示している.

Listing A.1 $c(ab)^*a(ba)^*$ を入力したときの実行ログ

```
remedy 0.4174873169977218
pure 0.0027732611633837223
regeq 0.11208686418831348
ca(?:(ba)*)
0.5333753989543766
```

Listing A.2 $.^*a^*=$ を入力したときの実行ログ

```
remedy 0.3885219458024949
pure 0.004464931786060333
regeq 1.419098210055381
remedy 0.38785919081419706
```

```
pure 0.0045299839694052935
regeq 1.3890138799324632
remedy 0.3998173091094941
pure 0.004595744889229536
regeq 4.988238276913762
remedy 0.4057066470850259
pure 0.004617673112079501
regeq 2.94695325801149
remedy 0.3773885320406407
pure 0.0044965429697185755
regeq 5.4774684340227395
remedy 0.4433436091057956
pure 0.004422834841534495
regeq 3.137833727989346
remedy 0.623568732989952
pure 0.004489025101065636
regeq 2.9904427719302475
remedy 0.6126141811255366
pure 0.004606247181072831
regeq 4.881942786974832
remedy 0.6278217800427228
pure 0.004618596052750945
regeq 10.262604668969288
remedy 2.1400269609875977
pure 0.004332051845267415
regeq 11.046329517150298
(?:([^=])*)[=](?:((?:([^=])*)[=])*)
54.99987591896206
```

A.2 REMEDY に追加された positive example および negative example

$.^*a^*=$ を修正する際に REMEDY に追加された positive example および negative example について以下に示す．一番上の行が修正したい正規表現，その下から「＋＋＋」までが positive example，さらにその下から「－－－」までが negative example である．

Listing A.3　$.^*a^*=$ を修正する際に REMEDY に追加された positive example および negative example

```
1  .*a*=
2  -=
3  0=
4  ==
5  =
6  --=
7  0-=
8  =-=
9  =0=
10 +++
11
12 -
13 0
14 =-
15 ---
```

参考文献

[1] Stack Exchange Network Status — Outage Postmortem - July 20, 2016. https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016. Accessed: 2023-01-21.

[2] Details of the cloudflare outage on july 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/. Accessed: 2023-01-21.

[3] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[4] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, Vol. 11, No. 4, pp. 481–494, oct 1964.

[5] SCOTT OWENS, JOHN REPPY, and AARON TURON. Regular-expression derivatives re-examined. *Journal of Functional Programming*, Vol. 19, No. 2, pp. 173–190, 2009.

[6] Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, Vol. 49, pp. 95–106, 2012. published online March 2011.

[7] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.

[8] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018.

[9] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. *Electronic Proceedings in Theoretical Computer Science*, Vol. 151, , 05 2014.

[10] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions (extended version). *CoRR*, Vol. abs/1701.04045, , 2017.

[11] Nariyoshi Chida and Tachio Terauchi. Repairing dos vulnerability of real-world regexes. 2020.

[12] Brink van der Merwe, Nicolaas Weideman, and Martin Berglund. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '17, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Regeq. `https://bakkot.github.io/dfa-lib/regeq.html`. Accessed: 2023-01-21.