第 1 章

既存研究

1.1 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．

1.1.1 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 [1] 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 [2] 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 ([1] より引用).

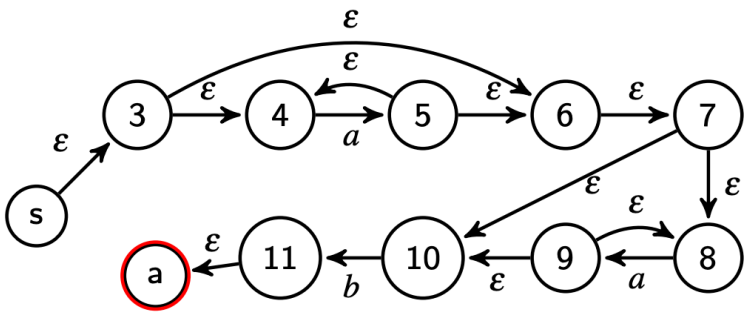

図 1.1　正規表現 a^*a^*b と等価な NFA

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．

$$(s, \varepsilon, 3), (3, \varepsilon, 4), (4, a, 5), (5, \varepsilon, 6), (6, \varepsilon, 7), (7, \varepsilon, 8), (8, a, 9), (9, \varepsilon, 10) \tag{1.1}$$

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，(1.1) 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．

1.1.2　ReDoS 脆弱性の定式化

ReDoS 脆弱性の定式化を行う．Wüstholz らの研究 [3] 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，本節での NFA は ε 遷移なし NFA である．

1.1.2.1 脆弱な NFA

次に，脆弱な (ε 遷移なし)NFA について定式化する．

定義 1（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 [3] 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 ([3] より引用)．

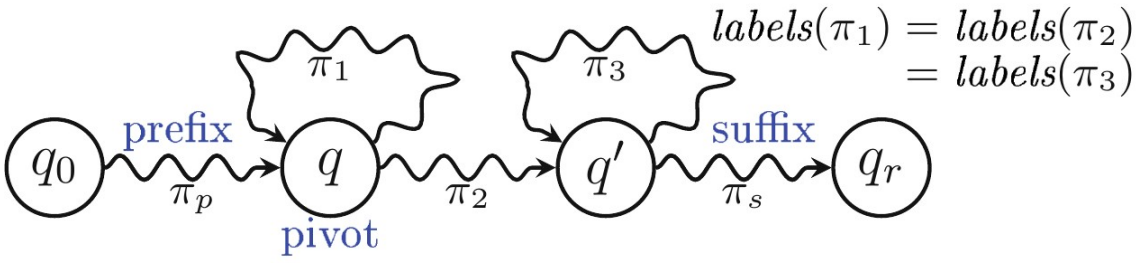

図 1.2　脆弱な NFA の一般的なパターン

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．

$$T_q(k) = (1 + T_q(k-1)) + (1 + T_{q'}(k-1)) \tag{1.2}$$

式 (1.2) 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．

1.1.2.2 脆弱な NFA の例

脆弱な NFA の例を以下に示す ([3] より引用).

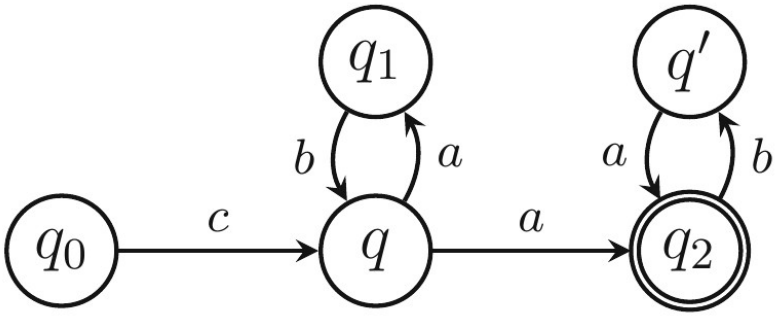

図 1.3　脆弱な NFA の例

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 1.3 は脆弱である．

1.2 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[4] 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．

参考文献

[1] 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.

[2] 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.

[3] 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.

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