インジェクション脆弱性検知のための Prioritized Streaming String Transducer の逆像計算の高速化

千田 忠賢[1,a)]　山口 大輔[2]　上川 先之[1]

概要：
インジェクション脆弱性とは攻撃者が用意したコードを標的の環境に挿入して実行することを許す脆弱性であり，情報漏洩や情報の改竄などの深刻な被害をもたらす重大な脅威である．この脅威を克服するため，インジェクション脆弱性の検知技術に関する研究が盛んに行われている．インジェクション脆弱性を検知するには，外部入力に対する文字列操作の処理内容を正確に捉えることが重要となる．ところが，多くの既存研究では文字列操作において広く利用されている正規表現のマッチング処理の優先度を考慮できておらず，誤検知の原因となる．この不正確さを改善するため，文字列ソルバーの研究領域では正規表現のマッチング処理における優先度を正確に捉えた制約解消に関する技術が研究されており，この技術を既存のインジェクション脆弱性検知技術と組み合わせることで優先度を正確に捉えた検知が可能となる．しかし，実世界で発見されたインジェクション脆弱性を対象とした場合，最先端の文字列ソルバーを利用しても処理速度が非常に遅く，現状では実用に耐えない．そこで，本研究では実世界で発見されたインジェクション脆弱性の検知において，最先端の技術を用いても処理速度が低下する原因を報告し，処理速度の低下を防ぐための高速化手法を提案する．高速化手法を取り入れた検知技術を実装し，実世界で発見されたインジェクション脆弱性を対象に評価を実施したところ，最先端の文字列ソルバーと比較した場合でも少なくとも約 209 倍以上処理速度が改善されることを確認した．

キーワード：インジェクション脆弱性，文字列制約解消，Prioritized Streaming String Transducer

Speeding Up Pre-Image Computation of Regular Languages under Prioritized Streaming String Transducer for Injection Vulnerability Detection

Nariyoshi Chida[1,a)]　Daisuke Yamaguchi[2]　Hiroyuki Uekawa[1]

Abstract:
Injection vulnerabilities are a significant threat that allows attackers to inject and execute malicious code within a target environment, resulting in information leakage, data tampering, etc. In response to this threat, many approaches have been proposed for detecting injection vulnerabilities. To detect the vulnerabilities accurately, it is important to model string operations applied to the user's input. However, many existing approaches fail to capture the precedence of regular expression (regex) matching, which is widely used in string operations. To address this issue, recent work has proposed a technique to model the precedence of regex matching in string solving. However, prior work reported that the incorporation of the technique into injection vulnerability detections does not work in real-world scenarios due to the slow running times. To overcome this situation, in this work, we report the causes of the degradations and introduce optimization techniques to mitigate the degradations. We evaluated our optimization techniques in a real-world scenario, and the result shows a speedup of at least 209× compared to the state-of-the-art string solver.

Keywords: Injection Vulnerability, String Solving, Prioritized Streaming String Transducer

1. はじめに

インジェクション脆弱性とは攻撃者が用意したコードを外部入力を経由して標的の環境に挿入し実行することを許す脆弱性であり，情報漏洩や情報の改竄など深刻な被害をもたらす重大な脅威である．代表的なインジェクション脆弱性としては Cross-Site Scripting（XSS）や SQL インジェクションなどが知られている．この脅威を克服するため，インジェクション脆弱性を検知する技術に関する研究が盛んに行われている [2, 5, 7–11, 15, 16]．

これらの研究の多くはインジェクション脆弱性の有無を判定するため，外部入力を受け取る箇所（ソース）から得られた入力が内部処理を経て外部操作（例：画面への出力）に用いられる箇所（シンク）に到達するまでの過程を解析し，入力を工夫することで脆弱性を悪用する文字列がシンクに到達するか判定する [3]．そのため，これらの手法の検知精度は入力文字列が辿る過程の解析の精度に影響を受ける．ところが，既存技術ではその過程で広く利用されている正規表現のマッチング処理[*1]における優先度（以降，マッチング優先度）を正確に捉えることができておらず誤検知の原因となっている．

この課題は既に認識されており，解決に向けた研究が行われている．Chen らは，正規表現を用いた文字列操作を，そのマッチング優先度を考慮して正確に表現できる変換器 *Prioritized Streaming String Transducer*（PSST）を考案した [4]．PSST を用いることで，入力文字列に適用される正規表現を用いた文字列操作を正確に扱える．そのため，Chen らの手法を既存のインジェクション脆弱性検知技術と組み合わせることで正規表現のマッチング優先度を考慮したインジェクション脆弱性検知が可能となるように思われる．しかし，先行研究ではこれらの素朴な組み合わせは実世界で発見されたインジェクション脆弱性を対象とした場合に処理速度が非常に遅くなることを報告しており [17]，このままでは実用に耐えない．

本論文では，実用に耐える正規表現のマッチング優先度を考慮した正確なインジェクション脆弱性検知技術の実現を目指す．この実現に向けて，Chen らの手法と既存のインジェクション検知技術の素朴な組み合わせによる処理速度が遅くなる原因を解明するための予備実験を実施したところ，全体の処理を遅くする原因はその内部で利用されている PSST による正規言語の逆像計算にあることを確認した．この原因を踏まえ，本論文では PSST による正規言語の逆像計算を高速化する手法を考案する．考案した高速化手法を適用した逆像計算と適用していない逆像計算をそれぞれ実装し，処理速度の比較を行い高速化手法の有効性を評価したためその結果を報告する．また，正規表現のマッチング優先度を扱える最先端の文字列ソルバーである OSTRICH [4] とも処理速度を比較し，少なくとも約 209 倍以上の処理速度の改善を確認したためその結果を報告する．

本論文の構成は次の通りである．まず，第 2 節では Chen らの手法を用いた正規表現のマッチング優先度を正確に捉えたインジェクション脆弱性検知技術の概要を説明し，PSST による正規言語の逆像計算がどのように利用されているのかを説明する．次に，第 3 節では PSST の定義および逆像計算を説明し，第 4 節では逆像計算の高速化手法を提案する．最後に，第 5 節にて評価結果を報告し，第 6 節で関連研究について述べる．

2. 背景

本節では下記の具体例を用いて既存のインジェクション脆弱性検知技術の説明，及びそれに対して Chen らの手法を組み合わせた場合の説明を行う．

```
1 let n = document.getElementById("name").value;
2 n = n.replace(/<img.*>/g, ""); // sanitize
3 document.documentElement.innerHTML = n;
```

このコードは，1 行目のソースから外部入力を受け取り，受け取った文字列は最終的に 3 行目のシンクに流れ込む．その過程では，外部入力から受け取った文字列に対して 2 行目で文字列操作を適用し加工を加えている．

既存のインジェクション検知手法では，このソースから外部入力を受け取ると，その入力の流れを追跡し，2 行目で適用された文字列操作（すなわち，`replace` 関数）をログとして記録しておく．そして，外部入力がシンクに到達した場合，それまでにログとして記録した一連の文字列操作を解析する．この解析により，外部入力を工夫することで一連の文字列操作の結果としてシンクに既知の攻撃用文字列（例えば，`<img src="/" =_=" title="onerror='prompt(1)'">` [12] など）を到達させることができるか判定する．これを判定するため，既存手法ではログに記録された各文字列操作を Finite-State Transducer (FST) として表現し，シンクに到達するか判定したい攻撃用文字列 w を部分文字列に含む文字列集合を正規表現 $.^*w.^*$ として表現する．そして，それらの FST における正規言語の逆像計算を最後に適用された文字列操作から逆順に繰り返し行い，最終的に得られた逆像が空集合かどうか判定する．逆像が空集合であるなら，どのような文字列をソースから入力したとしても既知の攻撃用文字列を含む文字列をシンクに到達させることができないため，用いた攻撃用文字列の範囲でインジェクション脆弱性はな

[1] NTT 社会情報研究所
NTT Social Informatics Laboratories
[2] NTT コンピュータ&データサイエンス研究所
NTT Computer & Data Science Laboratories
[a)] na.chida@ntt.com
[*1] 正規表現のマッチング処理とは，正規表現エンジンのマッチングアルゴリズムにおいて複数の可能性を順次試行する処理を指す．

いと判定する．逆像が空集合でないなら，そこに属する要素をソースに入力することで文字列操作の結果として既知の攻撃用文字列を含む文字列をシンクに到達させることができるため，インジェクション脆弱性があると判定する．

ところが，この既存手法ではこの例に含まれるような正規表現を用いた文字列操作のマッチング優先度を考慮していないため誤検知が発生しうる．例えば，上記の例において攻撃用文字列`<img src=1 onerror=alert(`hacked`)>`を含む文字列をシンクに到達させることができるか考える．この場合，そのような文字列はシンクに到達しない．なぜなら，`replace`関数のパラメタとして設定されている正規表現`<img.*>`における`.*`は貪欲な繰り返しであるため，`<img`という文字列が出現した場合，そこから最後の`>`までを全て削除してしまうからである．ところが，既存手法では正規表現のマッチング優先度を考慮せずに`replace`関数を FST に変換するため，優先度を考慮しないのであれば`<im<img>g src=1 onerror=alert`1`>`などの文字列をソースに入力することで`replace`関数により`<img>`が削除され，結果としてシンクに`<img src=1 onerror=alert`1`>`を到達させることができる．

ここでの問題は FST を用いた方法では正規表現のマッチング優先度を考慮できていないことであるため，Chen らの考案した PSST を用いることでこれを解決できる．Chen らは正規表現依存の文字列関数の PSST による表現や PSST における正規言語の逆像の求め方を示しており，それらを実装した文字列制約ソルバー OSTRICH も公開している．これにより，ログとして記録した文字列操作を OSTRICH で扱うことのできる制約に書き換えることで正規表現のマッチング優先度を考慮したインジェクション脆弱性の検知技術を実現できる．ところが，我々の先行研究にて報告したとおり，Chen らの方法を素朴に組み合わせるだけでは実用的な処理速度の達成が難しい [17]．現に，第 5 節で示す通り，実世界で発見されたインジェクション脆弱性を含むサニタイザを対象に，実世界で利用される攻撃用文字列を用いて脆弱性の有無を検証した場合，Chen らの公開している OSTRICH は PSST による正規言語の逆像計算に膨大な時間がかかることがあることを確認している．そこで，第 4 節では逆像計算を遅くする原因となる箇所を改善する 4 つの高速化手法を提案する．

3. PSST と逆像計算

本節では PSST の定義と PSST による正規言語の逆像計算を説明する．

3.1 Prioritized Streaming String Transducer

Prioritized Streaming String Transducer (PSST) は Chen らによって考案されたものであり，正規表現に依存した文字列関数による文字列から文字列への変換を表す変換器である [4]．大まかに言えば，PSST とは有限状態オートマトンに出力と文字列を格納する変数を付け加え，各遷移は正規表現のマッチング優先度を表すための優先度を持つように変更を加えたものと見ることができる．以下に PSST の定義を示す．

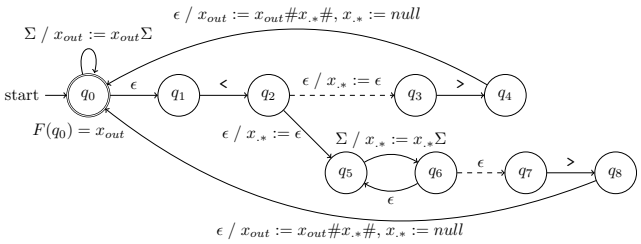

図 1 `replace(/<(.*)>/g, "#$1#")` を表す PSST

定義 1 (PSST)**.** PSST は 8 つ組 $(Q, \Sigma, X, \delta, \tau, E, q_0, F)$ として表され，Q は状態の有限集合，Σ はアルファベット，X は変数の集合，$\delta : Q \times \Sigma \to Q$ は ϵ 遷移を除く遷移関数，$\tau : Q \to Q \times Q$ は ϵ 関数，$E : (Q \times \Sigma \times Q) \to (X \cup \Sigma)^*$ は変数を更新するための関数，$q_0 \in Q$ は初期状態，$F : Q \to (X \cup \Sigma)^*$ は出力関数を表す．ここで，$\tau(q) = (q_1, q_2)$ は状態 q から状態 q_1 (resp. q_2) への高い (resp. 低い) 優先度を持つ ϵ 遷移が存在することを意味する．

上記の定義は Chen らの定義に対して軽微な変更を加えている．具体的には，Chen らの定義において PSST の遷移関数は文字もしくは空文字列から状態のリストへの関数として定義されているのに対して，上記の定義における遷移関数は文字もしくは空文字列から単一の状態への関数として定義されている．この差異は本論文で紹介する高速化手法に影響を与えないため，今後の説明を簡潔にするためにこのような変更を加えている．

図 1 に PSST の例を示す．この PSST は`replace(/<(.*)>/g, "#$1#")`を表すものであり，文字列中に出現する HTML タグの `<` および `>` を `#` に書き換えることで無害化することを意図している．残念ながら，この`replace`関数は正規表現のマッチング優先度を正しく考慮できていないため意図したサニタイズに失敗している．例えば，この`replace`関数に文字列`<<script>>`を与えた場合，`#<script>#`に変換され`script`タグが無害化されていない．

図 1 において，ノードは状態を表し，矢印は遷移を表す．各遷移は次の形のラベルを持っている．c / $x_1 := w_1, \cdots, x_n := w_n$．ここで，$c$ は $\Sigma \cup \{\epsilon\}$ の要素であり，各 $i \in \{1, \cdots, n\}$ について $x_i \in X$ かつ $w_i \in (\Sigma \cup X)^* \cup \{\texttt{null}\}$ を満たす．$x_i := w_i$ は変数への値の割り当てであり，変数 x_i に記録されている文字列を w_i で表される文字列に置き換える．w_i で表される文字列とは，w_i に出現する各変数をその値に置き換えることで得られる文字列である．全ての変数の更新は同時に実施される．また，変数は初期値とし

て空文字列を持つ．図を簡略化するため，各変数 $x \in X$ について $x := x$ となるラベルは省略している．また，図では $\Sigma \ / \ x := x\Sigma$ を用いている．これは，$\Sigma = \{a_1, \cdots, a_n\}$ としたとき，$a_1/x := xa_1$，$\cdots$，$a_n/x := xa_n$ の略記である．

PSST には 3 種類の遷移が存在する．具体的には，優先度が高い ϵ 遷移，ϵ ではない遷移，優先度が低い ϵ 遷移の 3 種類である．PSST は優先度が高い ϵ 遷移を最も優先して選択し，次に ϵ ではない遷移を選択する．最後に，優先度が低い ϵ 遷移を選択する．これらの優先度を用いて正規表現のマッチング優先度を表現している．図 1 において，優先度の高い ϵ 遷移および ϵ ではない遷移は実線で，優先度の低い ϵ 遷移は点線で表した．

二重線のノードは受理状態を表す．図 1 においては q_0 が受理状態であり，その出力は x_{out} である．すなわち，x_{out} に記録された文字列がこの PSST の出力となる．本論文では文字列 $w \in \Sigma^*$ に対する PSST T の出力を $T(w)$ と表記する．

同じ入力文字列に対して受理状態に到達するための経路（実行）が複数存在する場合，最も高い優先度を持つ実行のみが受理される．例えば，図 1 に示す PSST において入力文字列<a>>を与えた場合を考える．このとき，PSST には少なくとも次の 2 つの実行が存在する．$\pi_1 = q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{<} q_2 \xrightarrow{\epsilon} q_5 \cdots \xrightarrow{\epsilon} q_0$ 及び $\pi_2 = q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{<} q_2 \xrightarrow{\epsilon} q_3 \cdots \xrightarrow{\epsilon} q_0$ である．この場合，π_1 は π_2 よりも高い優先度を持つ．なぜなら，q_2 までの経路はどちらも同じであり，それに続く π_1 における q_2 から q_5 の遷移は π_2 における q_2 から q_3 の遷移よりも高い優先度を持つ遷移であるからである．

3.2 逆像計算

PSST の逆像計算の概要を説明する前に，有限状態オートマトンの定義を述べる．（決定性）有限状態オートマトンは 5 つ組 $(Q, \Sigma, \delta, q_0, F)$ として表される．ここで，Q は状態の有限集合，Σ はアルファベット，$\delta : Q \times \Sigma \to Q$ は遷移関数，$q_0 \in Q$ は初期状態，そして F は受理状態の集合である．有限状態オートマトンの文字列 $a_1 \cdots a_n$ に対する実行とは，列 $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{n-1} \xrightarrow{a_n} p_n$ であって，全ての $i \in \{1, \cdots, n\}$ に対して $(p_{i-1}, a_i, p_i) \in \delta$ が成り立つもののことを指す．ここで，次の条件を満たす実行が存在するとき，その有限状態オートマトンはその文字列を受理する: $p_0 = q_0 \wedge p_n \in F$.

PSST の逆像計算の概要を述べる．PSST $T = (Q_T, \Sigma, X, \delta_T, \tau, E, q_{0,T}, F_T)$ と有限状態オートマトン $A = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$ が与えられたとき，T における A の逆像を表す有限状態オートマトン $B = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$，すなわち，$B = \{u \mid (\exists v \in A).T(u) = v\}$，を構築できることが Chen らにより示されている（ [4] の補題 5.5）．Chen らの補題の証明では，次の形からなる状態を持つ B を構築している． $(q_T, \rho, \Lambda, S) \in Q_T \times \mathcal{P}(Q_A \times Q_A)^X \times \mathcal{P}(\mathcal{E}(\tau)) \times \mathcal{P}(Q_T)$．ここで，$\mathcal{E}(\tau) = \{(q, q'), (q, q'') \mid (q', q'') \in \tau(q)\}$ であり，$\mathcal{P}(C)$ は C の冪集合である．直感的には，状態の各要素は以下の役割を担っている．q_T はこれまでの計算により到達した T の状態を表す．変数 $x \in X$ について，$(q_{i,A}, q_{j,A}) \in \rho(x)$ は A において x に記録された文字列を用いて状態 $q_{i,A}$ から状態 $q_{j,A}$ に遷移可能でることを意味する．Λ は文字を消費せずに通過した ϵ 遷移を記録しており，ϵ 遷移による無限ループを回避するために用いられる．S は同じ文字列で到達可能な今の状態よりも高い優先度を持つ T 上の実行を記録している．PSST は全ての実行の中でも最も優先度が高いものだけを受理するため，q_T が受理状態に到達した場合でも S に属する状態が受理状態に到達しているならその状態は受理されない．

また，B における受理状態 F_B は次のように定められる．状態 $q_B = (q_T, \rho, \Lambda, S) \in Q_B$ が以下の条件を全て満たすとき，$q_B \in F_B$ となる． (1) $q_T \in F_T$，(2) $q_T \notin S$，及び (3) $(q_{0,A}, p) \in \rho(x_{out}) \wedge p \in F_A$.

4. 高速化手法

本節では PSST における正規言語の逆像計算に対する 4 つの高速化手法を提案する．

4.1 高速化手法 1 と 2

T，A，及び B をそれぞれ第 3 節で紹介した PSST，有限状態オートマトン，T による A の逆像を表す有限状態オートマトンとする．このとき，以下が成り立つ．

補題 1. B の文字列 $w = a_1 \cdots a_n$ に対する実行 $\pi = q_1 \xrightarrow{a_1} \cdots q_{n-1} \xrightarrow{a_{n-1}} q_n$ について，以下の条件のいずれかを満たす $q_i = (q_T, \rho, \Lambda, S)$ が存在するなら B は w を受理しない．条件 (1) $q_T \in S$ 及び (2) $(q_{0,A}, \_) \notin \rho(x_{out})$.

これらが成り立つことを確認する．実行に $q_T \in S$ を満たす状態 (q_T, ρ, Λ, S) が含まれる場合，残りの文字列に対する遷移について，第 1 要素の q_T が受理状態に到達しないならその文字列は受理されない．受理状態に到達するのであれば S に属する q_T も同様に受理状態に到達する．逆像を表す有限状態オートマトン B の受理状態の定義 (2) $q_T \notin S$ より，そのような状態は受理状態にはならない．そのため，$q_T \in S$ を満たす状態が実行に含まれる場合，その文字列は受理されない．

また，実行に $(q_{0,A}, \_) \notin \rho(x_{out})$ を満たす状態が含まれる場合，Chen らの論文で示された PSST の構築方法上，それ以降の遷移において x_{out} に $(q_{0,A}, \_)$ の要素は追加されることはない．そのため，逆像を表す有限状態オートマトン B の受理状態の定義 (3) $(q_{0,A}, p) \in \rho(x_{out}) \wedge p \in F_A$ が成り立たない．よって，その文字列は受理されない．

ここで，1 つ目の条件 $q_T \in S$ に関しては一般の PSST に対して成り立つものの，2 つ目の条件 $(q_{0,A}, \_) \notin \rho(x_{out})$ に関しては Chen らの PSST の構築方法に依存したもので

あることに注意されたい．

高速化手法 1 は補題 1 を利用し，逆像を表す有限状態オートマトン B を構築する際に，補題 1 に示される条件 (1) を満たす状態 (q_T, ρ, Λ, S) を以降の構築から除外する．同様に，高速化手法 2 は条件 (2) を満たす状態を除外する．それらの条件を満たす状態は受理される文字列の集合（言語）に影響しないため，逆像に影響を与えることなく除外することができる．

4.2 高速化手法 3

高速化手法 3 は PSST に含まれる不要な変数を除外することにより ρ のサイズを削減する．具体的には次の通りである．Chen らにより示された正規表現依存の文字列関数から PSST への変換方法は，正規表現の各部分表現に対応する変数を用意する．これは，正規表現がキャプチャグループを含む場合，そのキャプチャグループが抽出する文字列を PSST 上でも変数への格納という形で同様に抽出するためである．しかし，それらの変数は正規表現もしくは置換パターンにその変数に対応する参照が存在しないのであれば利用されない．そのため，x_{out} か置換パターンに出現する参照に影響を与える変数以外は PSST から除外する（高速化手法 3）．

例えば，図 1 に示される PSST は，図中では図が複雑になることを避けるため省略されているが，正規表現の部分表現 `<` に対応する変数 $x_{<}$ や `>` に対応する変数 $x_{>}$ などが存在している．しかし，それらの変数 $x_{<}$ と $x_{>}$ は x_{out} や置換パターン `#$1#` に出現する参照 `$1` のいずれにも影響を与えない．そのため，これらを PSST から除外する．これにより，逆像計算においても ρ の定義域からそれらの変数に関するものを除外される．

4.3 高速化手法 4

高速化手法 4 も ρ のサイズの削減を狙うものである．逆像計算において，ρ は T の出力が A に受理されるかを判定するために用いられる．これを判定するため，各変数 $x \in X$ に対して，ρ は A の全ての状態 q に対して q から x に格納された文字列を用いて到達する状態を記録する．しかし，実用上この判定には必ずしも A の全ての状態に対してこれを記録する必要はない．例えば，図 1 に示した PSST について考える．出力変数 x_{out} について，T の出力が A に受理されるかを判定するには A の初期状態 $q_{0,A}$ から x_{out} に格納された文字列で到達可能な状態を記録しておけば十分である．なぜなら，その PSST の変数の更新を確認すると，x_{out} はいずれも先頭でしか呼び出されていない．また，出力の際にも先頭で呼び出されている．そのため，x_{out} は常に $q_{0,A}$ からどこに遷移するのかという情報しか利用されない．そのため，A における $q_{0,A}$ 以外の状態から x_{out} に記録された文字列によりどこに遷移するのかを計算しても利用されることはない．また，変数 $x_{.*}$ についても，A において文字 `#` で遷移して到達可能な状態に限定して行き先の状態を記録すれば十分である．変数の更新において変数 $x_{.*}$ は常にその前に文字 `#` が出現しているからである．よって，ρ における $x_{.*}$ は常にそのような状態集合のみを対象にするような制限を加える．

具体的には，次のように ρ を制限する．各変数 $x \in X$ について，文字列の集合 W を次の方法で構築する．W に $\emptyset$ をセットし，W を以下の方法で更新する．各 $w \in \mathtt{ran}(F_T)$ 及び $w = E(q, \_, q')$ について，$w = w_0 \cdot x_0 \cdot w_1 \cdot x_1 \cdots w_n \cdot x_n \cdot w_{n+1}$ が x を持つなら（すなわち，$x_i = x$ となる $i \in \{1, \cdots, n\}$ が存在するなら） 以下を実施する．$x_i = x$ であるような各 $i \in \{1, \cdots, n\}$ について，$w_i \neq \epsilon$ なら W を $W \cup \{w_i\}$ に更新する．そうでない，すなわち，$w_i = \epsilon$ なら，W を Σ^* に更新する．全ての更新が完了したとき，$W \neq \Sigma^*$ なら，次の条件を満たす A の状態 q から構成される集合 C を用意する．条件：A 上の状態 q' であって，q' から q に対して文字列 $w \in W$ で遷移可能である．逆像計算において ρ の代わりに ρ を C に制限したもの，すなわち，$\rho \restriction C = \{(u, v) \mid (u, v) \in \rho, u \in C\}$，を用いる（高速化手法 4）．$W = \Sigma^*$ の場合は高速化手法 4 は適用できない．

5. 評価

本節では第 4 節で考案した 4 つの高速化手法を評価した結果を示す．具体的には，本論文では以下の 2 つの観点での評価を実施した．

RQ1 高速化手法は処理速度を改善するか．

RQ2 どの高速化手法が処理速度の改善に役立つか．

RQ1 に答えるため，全ての高速化手法を適用した逆像計算を実装したプログラム $\textsc{Pre}^+$ 及び高速化手法を全く適用しない逆像計算を実装したプログラム Pre を用意し，それぞれの処理速度を比較した．また，正規表現のマッチング優先度を考慮している最先端の文字列ソルバーである OSTRICH（バージョン 1.3.5）とも比較を行った．OSTRICH には逆像計算の前後に処理時間を計測するための処理を追加している．この結果は第 5.2 節で報告する．

RQ2 に答えるため，各高速化手法に対するアブレーション分析を実施した．この結果第 5.3 節で報告する．本実験は Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz のマシン上で行った．また，初期ヒープサイズは 2 GB，最大ヒープサイズは 32 GB であり，タイムアウトは 30 分に設定している．

5.1 データセット

本実験では次の形からなる制約を用いた．$\forall w \in \Sigma^*.\ w.\mathtt{replace}(/reg/g, rep) \in L$．ここで，$reg$ は正規表現，rep は置換パターン， L は攻撃用の文字列集合を表す正規言語である．正規表現と置換パターンは [7] に掲載さ

表 1　評価結果

No.	replace 関数	攻撃用文字列集合	Pre$^+$（秒）	Pre（秒）	OSTRICH（秒）
1	replace(/r_1/g, "")	.*w_1.*	1.44 (sat)	OOM	621.54 (sat)
2	replace(/r_2/g, "")	.*w_2.*	5.81 (sat)	OOM	timeout
3	replace(/r_3/g, "")	.*w_3.*	3.72 (unsat)	OOM	timeout
4	replace(/r_3/g, "")	.*w_4.*	0.57 (unsat)	OOM	172.07 (unsat)
5	replace(/r_3/g, "")	.*w_5.*	0.46 (unsat)	OOM	121.80 (unsat)
6	replace(/r_1/g, "")	.*w_6.*	3.69 (sat)	OOM	timeout
7	replace(/r_2/g, "")	.*w_7.*	20.56 (sat)	OOM	timeout
8	replace(/r_1/g, "")	.*w_8.*	0.38 (unsat)	OOM	79.70 (unsat)
9	replace(/r_1/g, "")	.*w_9.*	1.11 (sat)	OOM	1082.00 (sat)
10	replace(/r_2/g, "")	.*w_{10}.*	7.93 (unsat)	OOM	timeout

れている実世界で発見されたインジェクション脆弱性のあるサニタイザから収集したものに一部単純化や変形を施したものを用いた．攻撃用の文字列集合を表す正規言語 L は正規表現 $.^*w.^*$ で表される言語を用いた．ここで，$w \in \Sigma^*$ は既知の攻撃用文字列であり，この正規表現は部分文字列に w を含む文字列全体の集合を表す．既知の攻撃用文字列はそのような文字列を掲載しているウェブサイト [12] から取得した．評価で用いた具体的な値については第 5.2 節にて示す．

5.2　RQ1: 処理速度

高速化手法による処理速度の改善を確認するため，Pre$^+$，Pre，及び OSTRICH の処理速度を比較した結果を表 1 に示す．表 1 において OOM（Out of Memory）はメモリ不足によるエラーで処理を完了できなかったことを意味する．timeout は 30 分以内に処理が完了しなかったことを意味する．また，時間内に解を得ることができたものについては時間の隣に充足可能性の判定結果（充足可能であることを表す sat もしくは充足可能でないことを表す unsat）を記載している．表 1 における正規表現（r_1，r_2，及び r_3）は以下の通りである．

- r_1 = `<script.*?>|<[/](a|img)[^>]*`
- r_2 = `<script[^>]*>(.*?)<[/]script>`
- r_3 = `<(?:[/][a-zA-Z0-9]|[a-zA-Z0-9])(?:[^']|'[^']*')*>`

また，攻撃用の文字列（w_1 から w_{10}）は以下の通りである．

- w_1 = `<IMG onmouseover=alert('xss')>`
- w_2 = `<<script>alert('XSS');//<<script>`
- w_3 = `<META HTTP-EQUIV='refresh' CONTENT='0;url=data:text/html base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K'>`
- w_4 = `<body background=javascript:alert('XSS')>`
- w_5 = `<img lowsrc=javascript:alert('XSS')>`
- w_6 = `<script src=[···]/index.js></script>` [*2]
- w_7 = `<a href='javascript:alert(String.fromCharCode(88,83,83))'>Click Me!</a>`
- w_8 = `<img onmouseover=alert('xxs')>`
- w_9 = `</script><script>alert('XSS');</script>`
- w_{10} = `</title><script>alert('XSS');</script>`

本評価において，高速化手法を適用した Pre$^+$ は 10 件全てを時間内に解くことができた．一方で，高速化手法を全く適用しない Pre は全てのテストケースにおいてメモリを枯渇させ，いずれも解くことができていない．これは，逆像計算において，逆像を表す有限状態オートマトンのサイズが膨大なものになることに起因する．そして，最先端の文字列ソルバーである OSTRICH は 10 件中 5 件を解くことができた．

RQ1 のまとめ

本評価においては高速化手法により処理速度が大幅に改善されたことを確認した．特に，Pre$^+$ と OSTRICH の両方が解けたテストケースにおいて，最先端の文字列ソルバーである OSTRICH と比較しても本提案手法である Pre$^+$ は**少なくとも約 209.7 倍，最大では約 974.7 倍の処理速度の改善を確認した**．

実装の正しさに関する検証． Pre$^+$ と OSTRICH の両方で解くことができたテストケースについてはどちらも充足可能性の判定結果が一致していることを確認した．また，充足可能と判定されたものについてはその証拠となる攻撃用文字列を生成させ，その文字列を対象の replace 関数に与えることで実際に攻撃用文字列集合に属する文字列が得られることも確認した．例えば，**No. 6** のテストケースにおいて，Pre$^+$ は sat である証拠として次の文字列を提示した．文字列：`<scrip<script>t src=[···]/index.js></script>`．このテストケースにおける replace 関数は `replace(/<script.*?>|<[/](a|img)[^>]*/g, "")` であるため，一見 script タグは削除されるように思える．しかし，Pre$^+$ が示した証拠のように script タグを分断する形で script タグを埋め込むことで，replace 関数が

[*2] w_6 は具体的な URL を含む文字列であるため，URL の一部を [···] に置き換えている．省略部分の長さは 50 文字である．

埋め込まれた script タグを削除し，結果として script タグを含む文字列を生成することができる．

5.3 RQ2: 各高速化手法の影響

各高速化手法が処理速度に与える影響を確認するため，アブレーション分析を実施した．結果を図 2 に示す．4 つの図のうち，左上は高速化手法 1 のみ，右上は高速化手法 2 のみ，左下は高速化手法 3 のみ，右下は高速化手法 4 のみを適用しなかった場合の処理速度を示している．各図において，横軸は $\textsc{Pre}^{+}$ の処理速度，縦軸は対象の高速化手法を外した場合の処理速度を表す．点はテストケースを表しており，対角線よりも上に点がある場合はその高速化手法を外すことにより処理速度が遅くなったことを表すため，その高速化手法は有効であると考えられる．逆に，対角線よりも下に点がある場合はその高速化手法を外したことにより処理速度が速くなったことを表すため，その高速化手法は有効でないと考えられる．タイムアウトやメモリ不足によるエラーとなったものに関しては青い点として枠線に重なるように表示している．これらの結果から，高速化手法 3 と 4 についてはその影響が顕著であることが確認できる．また，高速化手法 1 については一部のテストケースで処理速度の改善が見られた．高速化手法 2 に関しては本評価においては十分な影響を確認できなかった．

RQ2 のまとめ

本評価においては，高速化手法 3 と 4 が特に処理速度の改善に役立っていることを確認した．

6. 関連研究

インジェクション脆弱性は古くから脅威として認識されている脆弱性であるため，その検知技術に関しては多くの研究が存在している [2, 5, 10, 11, 13, 14, 16]．本節では本技術と関係が深いものに限定して紹介する．

DOM 型 XSS の検知. DOM 型 XSS は 2005 年に Amit Klein により報告された脆弱性である [1]．この脆弱性を検知するため，ブラウザに動的テイント解析機能を追加する拡張を施し，ソースからシンクに外部入力が到達すか判定することにより DOM 型 XSS の有無を判断するツール DOMinator が登場した [6]．しかし，このツールが採用する手法ではソースからシンクまでの過程で適用される文字列操作を考慮しないため，サニタイザが正しく配置されており入力が無害化された状態でシンクに到達するような場面であって脆弱性があると判定されうる．

この課題を解決するため，Lekies らは外部入力に対して適用される標準的なエンコーディング等（encodeURI，encodeURIComponent，及び escape 等）の情報をテイントに付加して脆弱性有無の判断に利用する方法を考案した [8]．この方法は Melcher らによる DOM 型 XSS の実態調査でも採用されている [9]．ところが，この手法ではプログラミング言語が標準的にサポートしている一部の関数等を対象としたものであるため，それら以外の無害化処理や第三者が作成したライブラリ，そしてユーザ自身が作成した簡易的なサニタイザなどは扱うことができないという課題が残されている．

Klein らの研究では，外部入力がシンクに到達するまでの過程で適用された文字列操作をログとして記録し，記録された文字列操作における攻撃用文字列の集合を表す DFA の逆像計算を繰り返すことで DOM 型 XSS の有無を判定する手法を考案した [7]．ところが，この手法では正規表現のマッチング優先度が考慮されておらず，それに起因する誤検知が考えられる．

文字列ソルバー. Chen らは正規表現のマッチング優先度を考慮した文字列ソルバー OSTRICH を考案した [4]．また，Chen らは PSST による正規言語の逆像が計算できることを示し，逆像計算を用いて文字列制約の充足可能性を判定する方法を考案している．考案された手法はインジェクション脆弱性を含む文字列操作に関する広いバグや脆弱性の検知に応用が期待される．しかし，インジェクション脆弱性検知技術に組み合わせた場合，実世界で発見されたインジェクション脆弱性を対象として評価において処理速度が遅いことが報告されている [17]．本研究ではこの原因を調査し，状況を緩和するための高速化手法を考案した．

7. おわりに

本論文では正規表現のマッチング優先度を考慮したインジェクション検知技術の高速化を図る 4 つの手法を提案した．それらを実装したプログラムを用いて評価を行い，本高速化手法を用いると，最先端の文字列ソルバーと比較して少なくとも約 209.7 倍処理速度が高速化されることを確認した．今後は本手法を用いた実装を用いて，正規表現のマッチング優先度が実世界のインジェクション脆弱性検知に与える影響を調査することを検討している．

参考文献

[1] Amit, K.: DOM Based Cross Site Scripting or XSS of the Third Kind, URL http://www.webappsec.org/projects/articles/071105.shtml (2005).

[2] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C. and Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications, *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 387–401 (online), DOI: 10.1109/SP.2008.22 (2008).

[3] Bultan, T., Yu, F., Alkhalaf, M. and Aydin, A.: *String Analysis for Software Verification and Security*, Springer Publishing Company, Incorporated, 1st edition (2018).

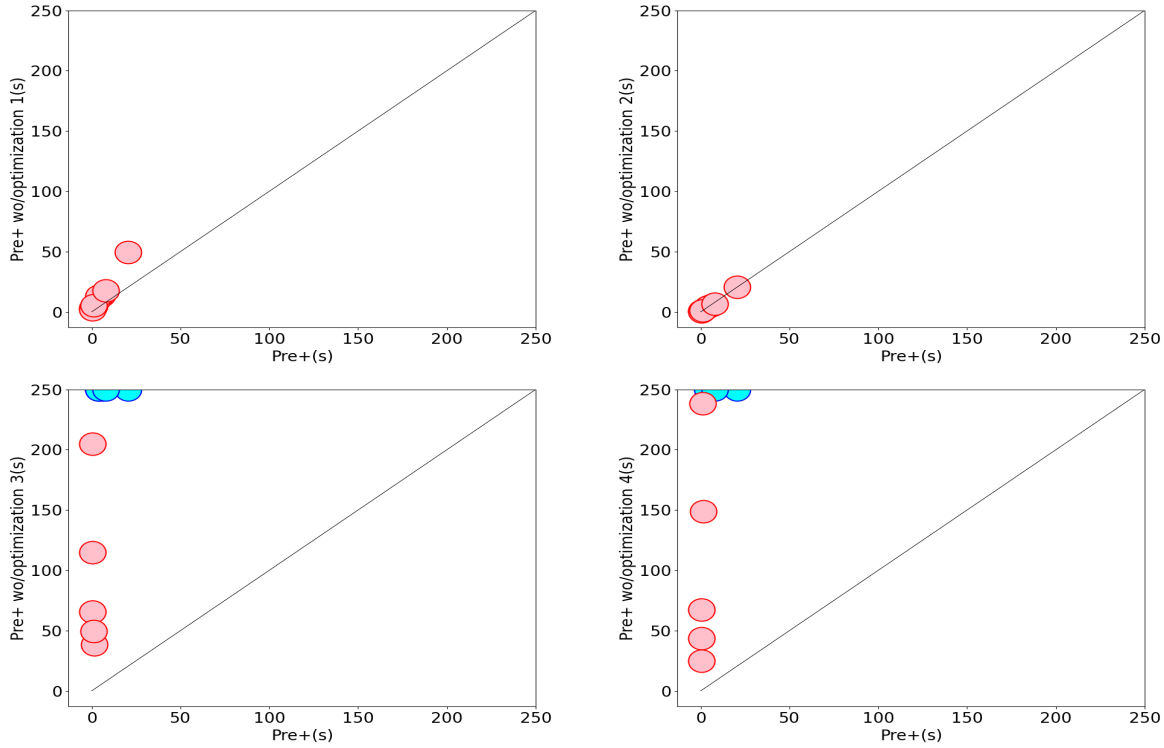

図 2 RQ2: 各高速化手法の有無による処理速度への影響

[4] Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A. W., Rümmer, P. and Wu, Z.: Solving string constraints with Regex-dependent functions through transducers with priorities and variables, *Proc. ACM Program. Lang.*, Vol. 6, No. POPL (online), DOI: 10.1145/3498707 (2022).

[5] Christensen, A. S., Møller, A. and Schwartzbach, M. I.: Precise Analysis of String Expressions, *Static Analysis* (Cousot, R., ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 1–18 (2003).

[6] Di, P.: DominatorPro: Securing Next Generation of Web Applications, URL `https://dominator.mindedsecurity.com/` (2012).

[7] Klein, D., Barber, T., Bensalim, S., Stock, B. and Johns, M.: Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions, *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pp. 236–250 (online), DOI: 10.1109/EuroSP53844.2022.00023 (2022).

[8] Lekies, S., Stock, B. and Johns, M.: 25 million flows later: large-scale detection of DOM-based XSS, *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, New York, NY, USA, Association for Computing Machinery, p. 1193–1204 (online), DOI: 10.1145/2508859.2516703 (2013).

[9] Melicher, W., Das, A., Sharif, M., Bauer, L. and Jia, L.: Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting, *Network and Distributed System Security Symposium*, (online), available from ⟨https://api.semanticscholar.org/CorpusID:3389782⟩ (2018).

[10] Melicher, W., Fung, C., Bauer, L. and Jia, L.: Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning, *Proceedings of the Web Conference 2021*, WWW '21, New York, NY, USA, Association for Computing Machinery, p. 2684–2695 (online), DOI: 10.1145/3442381.3450062 (2021).

[11] Minamide, Y.: Static approximation of dynamically generated Web pages, *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, New York, NY, USA, Association for Computing Machinery, p. 432–441 (online), DOI: 10.1145/1060745.1060809 (2005).

[12] OWASP: XSS Filter Evasion Cheat Sheet, URL `https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html` (2025).

[13] Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A. and Saxena, P.: DexterJS: robust testing platform for DOM-based XSS vulnerabilities, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA, Association for Computing Machinery, p. 946–949 (online), DOI: 10.1145/2786805.2803191 (2015).

[14] Stock, B., Lekies, S., Mueller, T., Spiegel, P. and Johns, M.: Precise Client-side Protection against DOM-based Cross-Site Scripting, *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, USENIX Association, pp. 655–670 (online), available from ⟨`https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/stock`⟩ (2014).

[15] Su, Z. and Wassermann, G.: The essence of command injection attacks in web applications, *SIGPLAN Not.*, Vol. 41, No. 1, p. 372–382 (online), DOI: 10.1145/1111320.1111070 (2006).

[16] Wassermann, G. and Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities, *SIGPLAN Not.*, Vol. 42, No. 6, p. 32–41 (online), DOI: 10.1145/1273442.1250739 (2007).

[17] 千田忠賢，山口大輔，上川先之：実世界の文字列操作を正確に捉えた DOM-based XSS の検知技術の実現に向けて，情報処理学会論文誌 66 巻 10 号.