# Repairing DoS Vulnerability of Real-World Regexes

Nariyoshi Chida[1,2] and Tachio Terauchi[2]
[1]NTT Security Japan※; [2]Waseda University

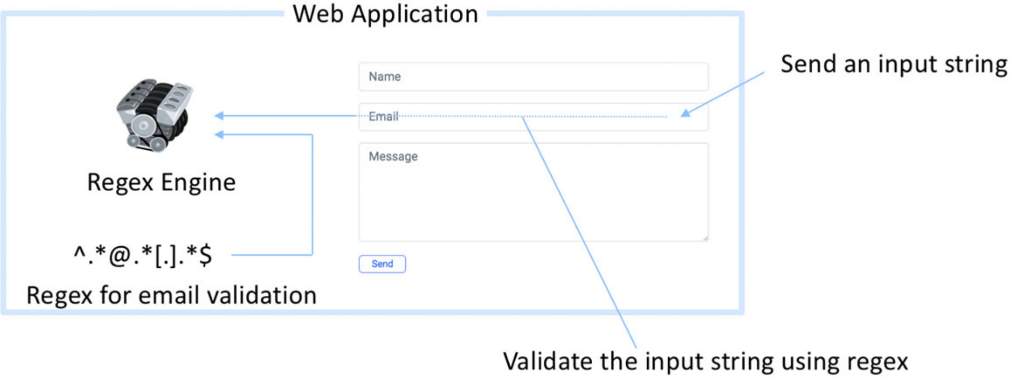
@S&P 2022
25 May 2022

※ Temporary transfer from NTT Corporation

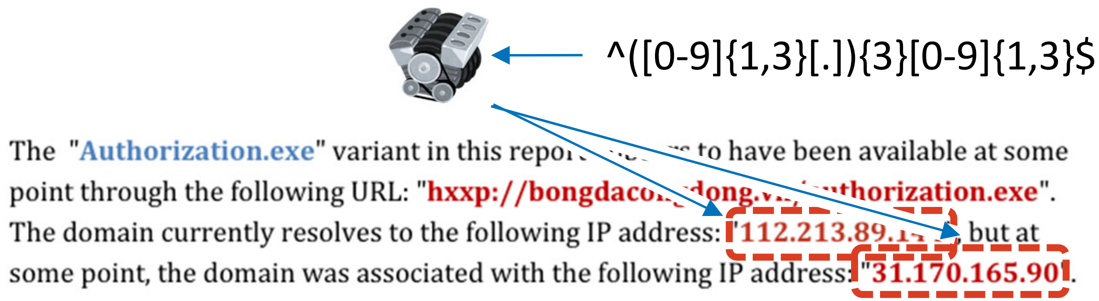
# Regular Expressions (Regexes)

Regexes are ubiquitous in modern software development.

**Sanitizing user inputs**:




**Extracting data from unstructured text**:




IoC extraction from threat reports

**General purpose libraries**:

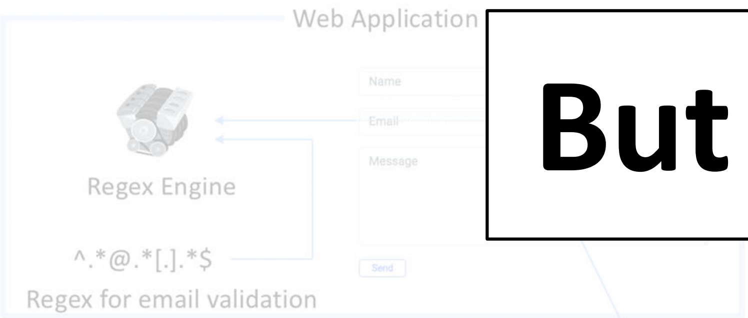
: regex engine

node JS · Java · python™ …

# Regular Expressions (Regexes)

Regexes are ubiquitous in modern software development.

**Sanitizing user inputs:**




Validate the input string using regex

**Extracting data from unstructured text:**

-9]{1,3}[.]){3}[0-9]{1,3}$

s to have been available at some
dong.vn/authorization.exe".
The domain currently resolves to the following IP address: "112.213.89.144", but at some point, the domain was associated with the following IP address: "31.170.165.90".

IoC extraction from threat reports

**General purpose libraries:**

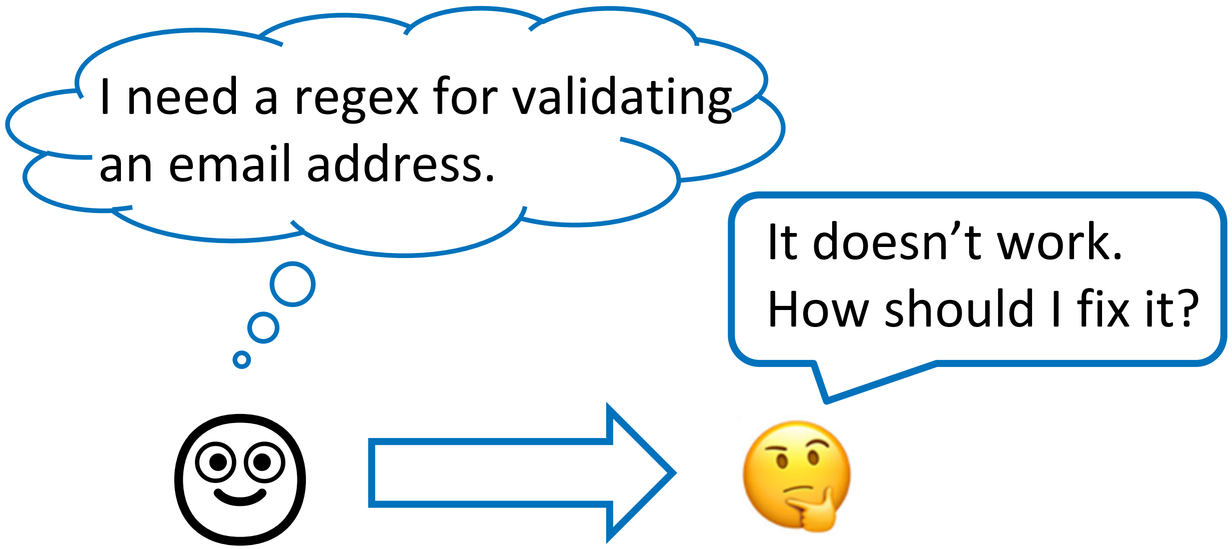
: regex engine

node JS · Java · python ...

# But unfortunately...

# Regexes are Hard!

Writing (or repairing) regexes are difficult... [Michael+ 2019]




Prepare the regex
(write manually or reuse it from StackOverflow)

Too complex to repair...

**Prepared regex**:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08
\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(
:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5
0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][
-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-
x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Validation failed

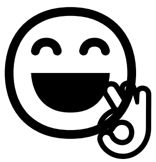
**Test cases**:

| Test cases for the regex | |
|---|---|
| **Correct email address** | **Incorrect email address** |
| foo@example.com<br>B_a.R@example.com<br>... | fooexample.com<br>... |

# Programming-By-Example (PBE)

One prominent approach to improve this situation is
**writing regexes with PBE method.**

[Lee+ 2016] [Pan+ 2019] [Chen+ 2020]...



**Prepared regex**:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08
\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(
:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5|
0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][(
-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\
x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

**Test cases**:

| Test cases for the regex | |
|---|---|
| **Correct email address** | **Incorrect email address** |
| foo@example.com<br>B_a.R@example.com<br>... | fooexample.com<br>... |

PBE tools → Repaired regex

Consistent with all examples (i.e., **correct** regex).

Reflect users' intentions by examples.

positive examples (strings to be accepted)

negative examples (strings to be rejected)

# Now, are we free from the difficulties of regexes?

Now, are we free from the difficulties of regexes?

**No!**

**We are still facing the difficulties of their vulnerabilities called regex denial of services (ReDoS).**

Indeed, the existing PBE methods may generate vulnerable regexes.

[Li+ 2020]

# Regex Denial of Service (ReDoS)

ReDoS is the vulnerability that causes the regex matching algorithm to take super linear time.

**ReDoS is a significant threat to our society**

**Details of the Cloudflare outage on July 2, 2019**

2019/07/13

On July 2, we deployed a new rule in our WAF Managed Rules that caused CPUs to become exhausted on every CPU core that handles HTTP/HTTPS traffic on the Cloudfla

**Stack Exchange Network Status**

On July 20, 2016 we experienced a 34 minute outage starting at 14:44 UTC. It took 10 mi identify the cause, 14 minutes to write the code to fix it, and 10 minutes to roll out the fix t where Stack Overflow became available again.

**CVE-2019-14232: Denial-of-service possibility in `django.utils.text.Truncator`**

django — The web framework for perfectionists with deadlines.

If `django.utils.text.Truncator`'s `chars()` and `words()` methods were passed the `html=True` argument, they were extremely slow to evaluate certain inputs due to a catastrophic backtracking vulnerability in a regular expression. The `chars()` and `words()` methods are used to implement the `truncatechars_html` and `truncatewords_html` te

| CVE-ID | |
|---|---|
| **CVE-2021-41817** | Learn more at National Vulner • CVSS Severity Rating • Fix Informatior |
| **Description** | |
| Date.parse in the date gem through 3.2.0 for Ruby allows ReDoS 2.0.1. | |

# In this talk

We introduce a tool called REMEDY that rectifies this situation.

**Input**:

(Possibly incorrect and vulnerable) real-world regex

Positive and negative examples

→ **REMEDY** →

**Output**:

**Correct** and **invulnerable** real-world regex

# In this talk

We introduce a tool cal

Importantly, our method can handle the regexes that have **real-world extensions** such as

- lookarounds,
- capturing groups, and
- Backreferences.

**Input**:

(Possibly incorrect and vulnerable) **real-world** regex

Positive and negative examples

**REMEDY**

**Correct** and **invulnerable** real-world regex

# Challenges & Contributions

1. **The Definition of ReDoS vulnerability of real-world regexes**

   A novel formal semantics and the time complexity of backtracking matching algorithm for real-world regexes

2. **The Repair Problem**

   A novel condition called real-world strong 1-unambiguity (RWS1U) and formalize the corresponding PBE repair problem (RWS1U repair problem)

3. **Algorithm**

   An algorithm for solving the RWS1U repair problem

# Outline

- Real-World Regexes
- ReDoS Vulnerabilities of Real-World Regexes
- RWS1U and Repair Problem
- Repair Algorithm and Evaluation
- Conclusion

# Outline

- Real-World Regexes
- ReDoS Vulnerabilities of Real-World Regexes
- RWS1U and Repair Problem
- Repair Algorithm and Evaluation
- Conclusion

# Real-World Regexes

**Syntax:**

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

# Real-World Regexes

Standard

**Syntax:** $r ::= [C] \mid \varepsilon \mid rr \mid r|r \mid r^*$

$\mid (r)_i \mid \backslash i \mid (?{=}r) \mid (?!r) \mid (?{<}{=}x) \mid (?{<}!x)$

Real-world extensions

# Real-World Regexes

**Syntax:**

$$r \quad ::= \quad [C] \mid \varepsilon \mid rr \mid r|r \mid r^* \\ \mid (r)_i \mid \backslash i \mid (?{=}r) \mid (?!r) \mid (?{<}{=}x) \mid (?{<}!x)$$

[C] is a character set. C is a set of characters.
We sometimes write a for [{a}].

**Example**:

[a-cz] matches one of the characters a, b, c, and z.

# Real-World Regexes

**Syntax:**

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

$(r)_i$ is a capturing group and it stores the matched string with the index i.
¥i is a backreference and it refers to the string captured by $(r)_i$.

**Example:**

<(.*)$_1$>.*</¥1>

<html>…</html> ⇨ accept
<html>…</body> ⇨ reject (¥1 fails)

(.*)$_1$ captures the string html

¥1 refers to the string html

# Real-World Regexes

**Syntax:**

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

Lookaheads attempt to match r without any character consumption.
Positive lookahead (?=r) succeeds if r succeeds.
Negative lookahead (?!r) succeeds if r fails.

**Example**:

(?=.*@).*

foo@example.com ⇨ accept
fooexamle.co.jp ⇨ reject

((?=.*@) fails since there is no @)

# Real-World Regexes

**Syntax:** $r \;::=\; [C] \mid \varepsilon \mid rr \mid r|r \mid r^*$
$\mid\; (r)_i \mid \backslash i \mid (?{=}r) \mid (?!r) \mid (?{<}{=}x) \mid (?{<}!x)$

Fixed-length Lookbehinds look back and attempt to match x without any character consumption.

**Example:**

.*(?<!jp)

foo@example.com ⇨ accept
fooexamle.co.jp ⇨ reject

((?<!jp) fails since the suffix is “jp”.

# Outline

- Real-World Regexes
- ReDoS Vulnerabilities of Real-World Regexes
- RWS1U and Repair Problem
- Repair Algorithm and Evaluation
- Conclusion

# ReDoS Vulnerabilities of Real-World Regexes

ReDoS vulnerability concerns the complexity of backtracking matching algorithm.

Therefore, **we define the semantics that models the behavior of backtracking matching algorithm by the matching relation** $\rightsquigarrow$ .

Real-world regex

Position on the input string

$$(r, w, p, \Gamma) \rightsquigarrow \{(p_1, \Gamma_1), (p_2, \Gamma_2), \cdots, (p_n, \Gamma_n)\}$$

Input string

Environment (that stores strings for each index)

Set of matching results

# Rules of the Matching Relation

$$\frac{p < |w| \qquad w[p] \in C}{([C], w, p, \Gamma) \rightsquigarrow \{(p+1, \Gamma)\}} \ \text{(Set of characters)}$$

$$\frac{p \geq |w| \vee w[p] \notin C}{([C], w, p, \Gamma) \rightsquigarrow \emptyset} \ \text{(Set of characters Failure)}$$

$$\frac{}{(\varepsilon, w, p, \Gamma) \rightsquigarrow \{(p, \Gamma)\}} \ \text{(Empty String)}$$

$$\frac{(r_1, w, p, \Gamma) \rightsquigarrow \mathcal{N} \qquad \forall (p_i, \Gamma_i) \in \mathcal{N},\ (r_2, w, p_i, \Gamma_i) \rightsquigarrow \mathcal{N}_i}{(r_1 r_2, w, p, \Gamma) \rightsquigarrow \bigcup_{0 \leq i < |\mathcal{N}|} \mathcal{N}_i} \ \text{(Concatenation)}$$

$$\frac{(r_1, w, p, \Gamma) \rightsquigarrow \mathcal{N} \qquad (r_2, w, p, \Gamma) \rightsquigarrow \mathcal{N}'}{(r_1 | r_2, w, p, \Gamma) \rightsquigarrow \mathcal{N} \cup \mathcal{N}'} \ \text{(Union)}$$

$$\frac{\begin{array}{c}(r, w, p, \Gamma) \rightsquigarrow \mathcal{N} \\ \forall (p_i, \Gamma_i) \in (\mathcal{N} \setminus \{(p, \Gamma)\}),\ (r^*, w, p_i, \Gamma_i) \rightsquigarrow \mathcal{N}_i\end{array}}{(r^*, w, p, \Gamma) \rightsquigarrow \{(p, \Gamma)\} \cup \bigcup_{0 \leq i < |(\mathcal{N} \setminus \{(p, \Gamma)\})|} \mathcal{N}_i} \ \text{(Repetition)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathcal{N}}{((r)_j, w, p, \Gamma) \rightsquigarrow \{(p_i, \Gamma_i[j \mapsto w[p..p_i)]) \mid (p_i, \Gamma_i) \in \mathcal{N}\}} \ \text{(Capturing group)}$$

$$\frac{\Gamma(i) \neq \bot \qquad (\Gamma(i), w, p, \Gamma) \rightsquigarrow \mathcal{N}}{(\backslash i, w, p, \Gamma) \rightsquigarrow \mathcal{N}} \ \text{(Backreference)}$$

$$\frac{\Gamma(i) = \bot}{(\backslash i, w, p, \Gamma) \rightsquigarrow \emptyset} \ \text{(Backreference Failure)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathcal{N}}{((?{=}r), w, p, \Gamma) \rightsquigarrow \{(p, \Gamma') \mid (\_, \Gamma') \in \mathcal{N}\}} \ \text{(Positive lookahead)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset, \emptyset, \{(p, \Gamma)\})}{((?!r), w, p, \Gamma) \rightsquigarrow \mathcal{N}'} \ \text{(Negative lookahead)}$$

$$\frac{(x, w[p - |x|..p), 0, \Gamma) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset, \{(p, \Gamma)\}, \emptyset)}{((?{<=}x), w, p, \Gamma) \rightsquigarrow \mathcal{N}'} \ \text{(Positive lookbehind)}$$

$$\frac{(x, w[p - |x|..p), 0, \Gamma) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset, \emptyset, \{(p, \Gamma)\})}{((?{<}!x), w, p, \Gamma) \rightsquigarrow \mathcal{N}'} \ \text{(Negative lookbehind)}$$

Fig. 11: Rules of the matching relation $\rightsquigarrow$

# Example:

# Matching of (a*)* with the input string ab

$$
\dfrac{
\dfrac{
\dfrac{0 < |ab| \quad a \in \{a\}}{(a, ab, 0) \leadsto \{1\}}
\qquad
\dfrac{\dfrac{1 < |ab| \quad b \notin \{a\}}{(a, ab, 1) \leadsto \emptyset}}{(a^*, ab, 1) \leadsto \{1\}}
}{(a^*, ab, 0) \leadsto \{0, 1\}}
\qquad
\dfrac{\dfrac{1 < |ab| \quad b \notin \{a\}}{(a, ab, 1) \leadsto \emptyset}}{((a^*)^*, ab, 1) \leadsto \{1\}}
}{((a^*)^*, ab, 0) \leadsto \{0, 1\}}
$$

# ReDoS Vulnerabilities of Real-World Regexes

Definition (Running time):

For a regex $r$ and a string $w$, we define the *running time* of the backtracking matching algorithm on $r$ and $w$, $Time(r,w)$, to be the size of the derivation tree of $(r,w,0,\emptyset) \rightsquigarrow \mathcal{N}$.

Definition (Vulnerable regexes):

We say that a regex $r$ is *vulnerable* if $Time(r,w) \notin O(|w|)$.

# Outline

- Real-World Regexes
- ReDoS Vulnerabilities of Real-World Regexes
- RWS1U and Repair Problem
- Repair Algorithm and Evaluation
- Conclusion

# How can we guarantee ReDoS invulnerability?

The root cause of ReDoS is backtrackings due to the ambiguity.

⇨ We modify a regex to eliminate the ambiguity.

**Regex for an email address**:

.*@.*[.].* ⇨ [^@]*@[^@]*[.].*

Disambiguate

It's too relaxed!

(.* and .* do not need to accept the character @.)

# How can we guarantee ReDoS invulnerability?

The root cause of ReDoS is backtrackings due to the ambiguity.

⇨ We modify a regex to eliminate the ambiguity.

We defined a grammatical condition sufficient to ensure ReDoS invulnerability called **real-world strong 1-unambiguity** (**RWS1U**).

Extension of strong 1-unambiguity [Koch and Scherzinger 2007].
We'll explain this next (see paper for formal def.)

# Real-World Strong 1-Unambiguity (RWS1U)

**Lookahead removal**:

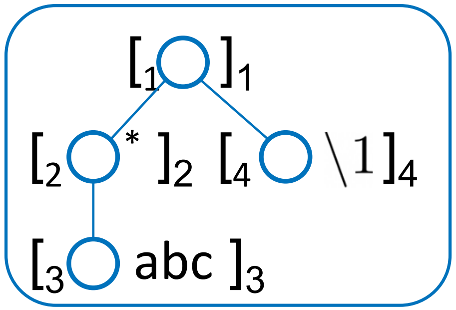
$[abc]^*(?=a)\backslash 1$

Replace lookaheads with ε

$[abc]^*\backslash 1$

**Bracketing**:

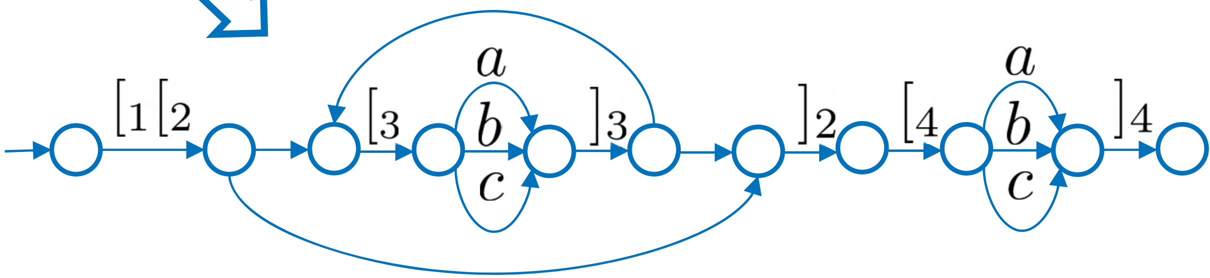
$[_1[_2([_3 abc]_3)^*]_2[_4\backslash 1]_4]_1$




AST

**Extended NFA translation**:

RWS1U enforces that the matching algorithm can determine which subexpression to match next by looking at the next character in the input string.



# Real-World Strong 1-Unambiguity (RWS1U)

**Lookahead removal**:

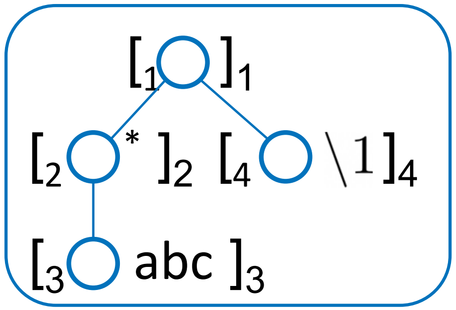
$[abc]^*(?=a)\backslash 1$

⇩ Replace lookaheads with ε

$[abc]^*\backslash 1$

**Bracketing**:

⇨ $[_1[_2([_3abc]_3)^*]_2[_4\backslash 1]_4]_1$

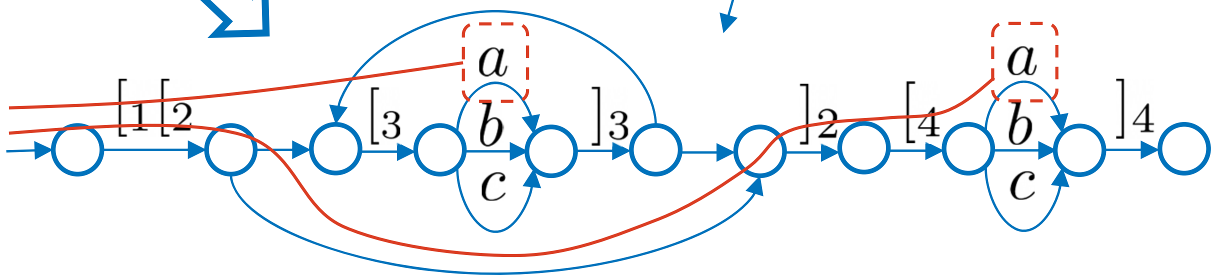



AST

**Extended NFA translation**:

RWS1U enforces that the matching algorithm can determine which subexpression to match next by looking at the next character in the input string.

RWS1U violation




The character $a$ has two bracketing-only paths.
(Therefore, violates RWS1U.)

# RWS1U Repair Problem

**Input**:

- (Possibly incorrect and vulnerable) regex : $r$
- Set of positive examples : P
- Set of negative examples : N

**Output**:

- Regex $r'$ that is consistent with examples, ← For correctness
  satisfies RWS1U, and ← For ReDoS invulnerability
  the edit distance from the input regex is minimal.

For quality →

# RWS1U Repair Problem

**Input**:

- (Possibly incorrect and vulnerable) regex : $r$
- 
- 

**Outp**

- 

**The RWS1U repair problem is NP-hard!**

We can show this by a reduction from Exact Cover
which is NP-complete.
The proof can be found in our paper.

orrectness

satisfies RWS1U, and ← For ReDoS invulnerability

the edit distance from the input regex is minimal.

For quality →

# Outline

- Real-World Regexes
- ReDoS Vulnerabilities of Real-World Regexes
- RWS1U and Repair Problem
- Repair Algorithm and Evaluation
- Conclusion

# Repair Algorithm

Our repair algorithm is

**Enumerative Search**

**+**

**Pruning by Approximations**

**+**

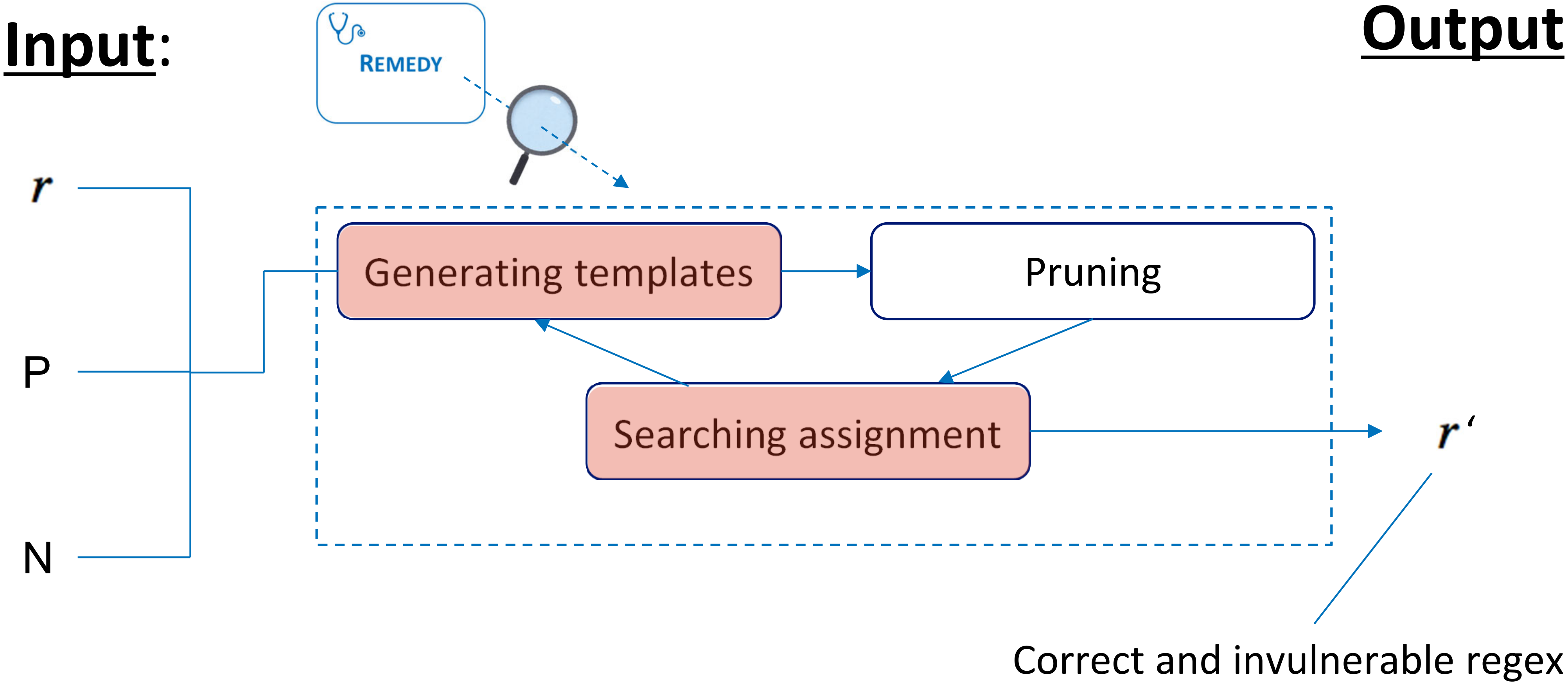
**SMT-based Constraint Solving**

Builds on [Pan+ 2019].

# High Level Repair Algorithm

**Input:**

**Output:**



# Repair Algorithm: Example

Generating templates

Searching assignment

**Input**:

$r$ = $< (.^*)_1 > .^* < /\backslash 1 >$

P = {<ab></ab>, <a>ab</a>}

N = {<a></b>, <a><b></b></a>, <a><ab></a>}

# Repair Algorithm: Example

**Generating templates**

Searching assignment

**Input**: $r = < (.^{*})_1 > .^{*} < /\backslash 1 >$, P = {<ab></ab>, <a>ab</a>}, N = {<a></b>, <a><b></b></a>, <a><ab></a>}

- Replace the subexpressions with holes □

$$< (.^{*})_1 > .^{*} < /\backslash 1 > \quad \Rightarrow \quad < (\square_1^{*})_1 > \square_2^{*} < /\backslash 1 >$$

After some iterations

# Repair Algorithm: Example

Generating templates

**Searching assignment**

**Input**: $\boldsymbol{r} = < (.^*)_1 > .^* < /\backslash 1 >$, P = {<ab></ab>, <a>ab</a>}, N = {<a></b>, <a><b></b></a>, <a><ab></a>}

- Checks if the template can be instantiated to a regex that satisfies the required conditions by replacing its holes with some sets of characters

$$< (\square_1^*)_1 > \square_2^* < /\backslash 1 >$$

Try to replace □ with [C].

# Repair Algorithm: Example

Generating templates

**Searching assignment**

**Input**: $\boldsymbol{r} = <(.^*)_1>.^*</\backslash 1>$, P = {<ab></ab>, <a>ab</a>}, N = {<a></b>, <a><b></b></a>, <a><ab></a>}

- Checks if the template can be instantiated to a regex that satisfies the required conditions by replacing its holes with some sets of characters

$$<(\square_1^*)_1>\square_2^*</\backslash 1> \xrightarrow{\textbf{Generate constraints}} (\phi_p^1 \wedge \phi_p^2) \wedge (\neg\phi_n^1 \wedge \neg\phi_n^2 \wedge \neg\phi_n^3)$$

# Repair Algorithm: Example

Generating templates

**Searching assignment**

**Input**: $r =< (.^*)_1 > .^* < /\backslash 1 >$, P = {<ab></ab>, <a>ab</a>}, N = {<a></b>, <a><b></b></a>, <a><ab></a>}

- Checks if the template can be instantiated to a regex that satisfies the required conditions by replacing its holes with some sets of characters

$$< (\square_1^*)_1 > \square_2^* < /\backslash 1 >$$

Generate constraints

$$(\phi_p^1 \wedge \phi_p^2) \wedge (\neg\phi_n^1 \wedge \neg\phi_n^2 \wedge \neg\phi_n^3)$$

Constraints for P

P = {<ab></ab>, <a>ab</a>}

$$(v_1^a \wedge v_1^b) \wedge (v_1^a \wedge (v_2^a \wedge v_2^b))$$

$\square_1$ can be replaced with the character set that contains a and b.

# Repair Algorithm: Example

Generating templates

**Searching assignment**

**Input**: $\boldsymbol{r} = < (.^*)_1 > .^* < /\backslash 1 >$, P = {<ab></ab>, <a>ab</a>}, N = {<a></b>, <a><b></b></a>, <a><ab></a>}

- Checks if there are multiple bracketing-only paths for for each $[_i$ that reach a same character

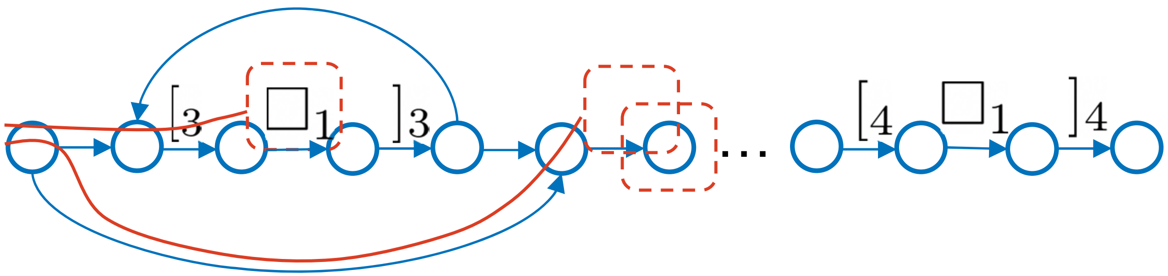
$< (\square_1^*)_1 > \square_2^* < /\backslash 1 >$

Generate constraints

$(\phi_p^1 \wedge \phi_p^2) \wedge (\neg\phi_n^1 \wedge \neg\phi_n^2 \wedge \neg\phi_n^3) \wedge \phi_l$

Constraints for RWS1U

$\neg v_1^> \wedge \neg v_2^<$



# Repair Algorithm: Example

Generating templates

**Searching assignment**

**<u>Input</u>**: $r = <(.^*)_1>.^*</\backslash 1>$, P = {<ab></ab>, <a>ab</a>}, N = {<a></b>, <a><b></b></a>, <a><ab></a>}

- Checks if the template can be instantiated to a regex that satisfies the required conditions by replacing its holes with some sets of characters

$$<(\square_1^*)_1>\square_2^*</\backslash 1>$$

Generate constraints

$$(\phi_p^1 \wedge \phi_p^2) \wedge (\neg\phi_n^1 \wedge \neg\phi_n^2 \wedge \neg\phi_n^3) \wedge \phi_l$$

Solve the constraints by an SMT solver.

**If it is satisfiable, replace □ with the character set.**

$$<([\text{^}>]^*)_1>[\text{^}<]^*</\backslash 1>$$

# Evaluation

**Research Questions**:

1. Can REMEDY repair vulnerable regexes efficiently?
2. Can REMEDY find high-quality regexes?

…

**Benchmark**:

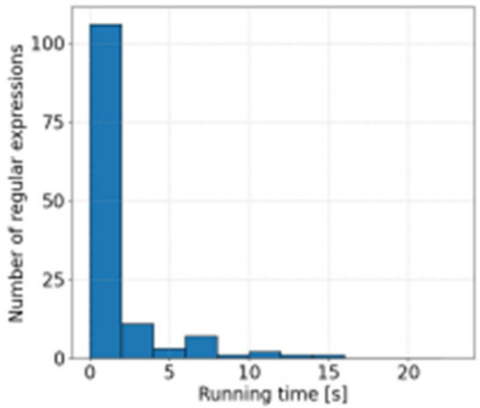
ReDoS data set [Davis+ 2018]

- It contains real-world regexes in Node.js (JavaScript) and Python core libraries.

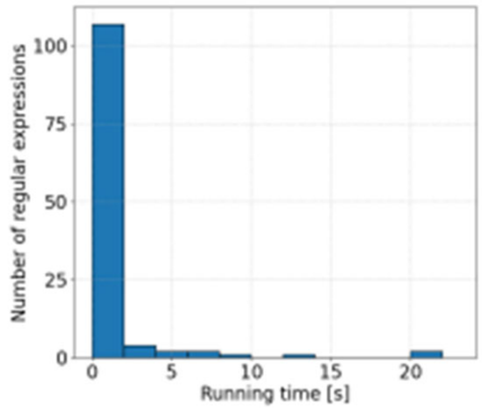
# RQ1. Can REMEDY repair vulnerable regexes efficiently?

REMEDY could solved 82.1% of regexes within 0.97 seconds on average.

| | Solved(179) | Average(s) |
|---|---|---|
| REMEDY | 132 | 1.54 |
| REMEDY-o | 119 | 1.08 |
| REMEDY-h | 147 | 0.97 |




(a) Running times of REMEDY




(b) Running times of REMEDY-o

Fig. 6: Results of the repairs.

## Regarding RQ2:

# What is "high-quality"?

In PBE scenario, the user wants to obtain what the user intended as output.

Therefore, the repairs that are similar to the original ones are often considered good in PBE scenario [Pan+ 2019].

**Regex written by the user**:

.*@.*[.].*

**What the user intended**:

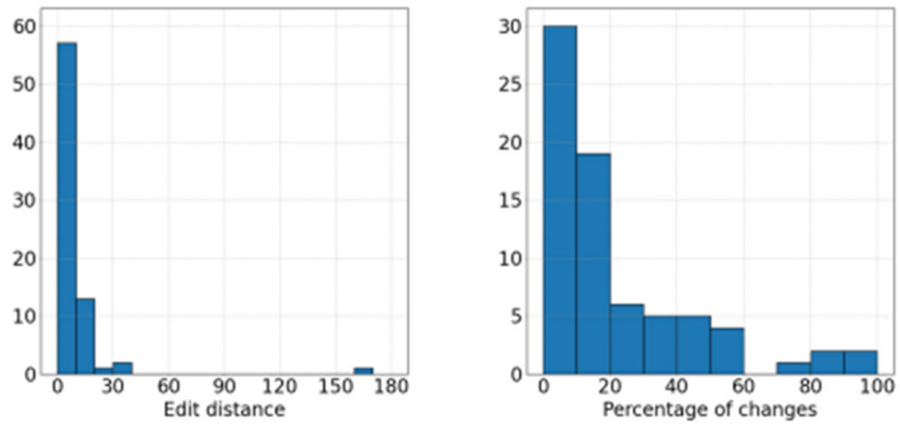
[^@]*@[^@]*[.].*

It is vulnerable but similar to what the user intended.

## RQ2. Can REMEDY find high-quality regexes?

About 81% of regexes were repaired within the small edit distances (12).

Additionally, the average ratio of change was 24.3%.




(a) Edit distances. (b) Percentages of changes.

Fig. 9: Histograms for repair quality.

# Outline

- Real-World Regexes
- ReDoS Vulnerabilities of Real-World Regexes
- RWS1U and Repair Problem
- Repair Algorithm and Evaluation
- Conclusion

# Conclusion

We introduced

1. **the definition of ReDoS vulnerability** of real-world regexes,
2. the condition for ReDoS invulnerability and **the repair problem**, and
3. **the algorithm** for solving the repair problem.

**Artifact**:

Artifact (REMEDY) is available!

or

github.com/NariyoshiChida/SP2022