Improving Developers' Understanding of Regular Expression Denial of Service (ReDoS) through Anti-Patterns and Fix Strategies

Sk Adnan Hassan, **Zainab Aamir**, Dongyoon Lee, James C. Davis, Francisco Servant

Stony Brook University
Computer Science

INTRODUCTION

- Regular expressions are a common tool for text manipulation tasks such as input validation. They can also lead to a security vulnerability called ReDoS (Regular Expression Denial of Service). This attack is caused by a worst case super-linear regex matching time.
- Proposed defenses include automated tools to detect these regexes and fix them. Their usability has not been studied.
- In our work, we introduce a set of anti-patterns to identify vulnerable regexes and fix strategies to repair the regexes. The anti-patterns are based on concept of infinite ambiguity in regexes.
- We investigate how our anti-patterns and strategies help developers' understanding of ReDoS and the outcome of existing detection and repair tools.

BACKGROUND

- Regexes have various degrees of ambiguity: none, finite or infinite. Infinite ambiguity (IA) leads to a super-linear time complexity during regex matching. IA regexes can have a polynomial degree of ambiguity (PDA) or an exponential degree of ambiguity (EDA) .
- An example of a regex with EDA is **/^(a+)+$/** , an input that would trigger super-linear behavior is "**aaaaaaaa!"**

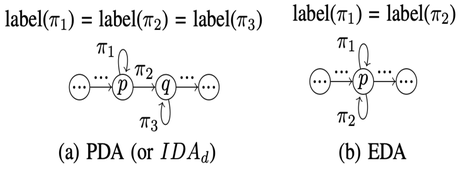

Figure 1: PDA and EDA in the NFA

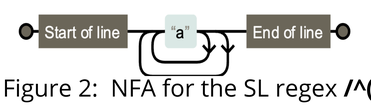

Figure 2: NFA for the SL regex **/^(a+)+$/**

METHODOLOGY AND RESULTS

- The antipatterns were evaluated on Kleene-regular regexes to detect IA regexes using the largest available dataset of real-world regexes[1]. They were compared with state-of-the-art anti-patterns[2]. Our anti-patterns led to an increase in precision and recall and no false positives. Results in Figure 4.
- To study the effectiveness of our anti-patterns in detecting IA regexes, we interviewed 20 software developers and asked them to perform simple regex composition tasks and detect IA in their regexes using SOA anti-patterns and our anti-patterns. Our anti patterns outperformed the SOA by 50% effectiveness as seen in Figure 5.
- To study how well our anti-patterns complement existing automatic tools, we interviewed 9 software developers who had written IA regexes in open-source projects. For detection and repair, the developers first used the tool alone and then the tool combined with our anti-patterns(detection) and fix strategies(repair). Results showed that subjects reported much higher understanding, going from "very weakly" understanding to "strongly" for detection and "very strongly" for repair.

Anti-pattern	Thm.	Description	Example
Concat 1	2	`R = ...P*Q*...` (R has a sub-regex `P*Q*`) — The two quantified parts `P*` and `Q*` can match some shared string s.	`\w*\d*` — both classes can match digits [0-9].
Concat 2	2	`R = ...P*SQ*...` — The two quantified parts `P*` and `Q*` can match a string s from the middle part S.	`\w*0\d*` — the repeated classes `\w` and `\d` can match the middle part 0.
Concat 3	2	`R = ...P*S*Q*...` — Advanced form of Concat 1. Since `S*` includes an empty string, the ambiguity between `P*` and `Q*` can be realized.	`\w*:*\d*` — The classes `\w` and `\d` overlap, and the intervening `:*` can be skipped.
Star 1	1, 4	`R*, R= (P|Q|...)` — There is an intersection between any two alternates, *i.e.,* both match some shared strings.	`(\w|\d)*` — both classes match digits [0-9].
Star 2	3	`R*, R= (P|Q|...)` — You can make one option of the alternation by repeating another option multiple times or by concatenating two or more options multiple times.	`(a|b|ab)*` — The 3rd option, *ab*, matches combinations of the first and second options.
Star 3	3	`R*, R= (...P*...)` — Nested quantifiers, provided `RR` follows any of the Concat anti-patterns.	Expanding `R=(0?\w*)*` to `RR` yields `0?\w*0?\w*`, which is IA by Concat 3. Similarly, `R=(xy*)*` yields `xy*xy*`; this is not IA by any Concat anti-pattern.

Figure 3: Our proposed anti-patterns

Anti-patterns	Precision	Recall
Our IA anti-patterns	100%	99%
SOA anti-patterns [2]	50%	87%

Figure 4: Comparison with SOA anti-patterns

	SOA first, IA after ($N = 10$)		IA first, SOA after ($N = 10$)		All orders ($N = 20$)	
Task	SOA	IA	SOA	IA	SOA	IA
1	100%	100%	100%	100%	100%	100%
2	10%	100%	0%	100%	5%	100%
3	20%	100%	20%	100%	20%	100%
4	30%	100%	20%	100%	25%	100%
5	100%	100%	100%	100%	100%	100%
All	52%	100%	48%	100%	50%	100%

Figure 5: Results of the first experiment

Detection Task

	Output of automatic detection tool	How strongly do you understand what makes this regex vulnerable?	Explain your reasoning
PDA regex	*Output of Weideman's detection tool* [19]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[...]
EDA regex	*Output of Weideman's detection tool* [19]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[...]
Non-IA regex	*Output of Weideman's detection tool* [19]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[...]

Fixing Task

	Output of automatic fixing tool	How strongly do you understand what makes the resulting fixed regex not vulnerable?	Explain your reasoning
Their vulnerable regex in context	*Output of van der Merwe's fixing tool* [27]	[Very strongly, Strongly, Neutral, Weakly, Very weakly, Not Vulnerable]	[...]

Figure 6: Tasks for the second experiment

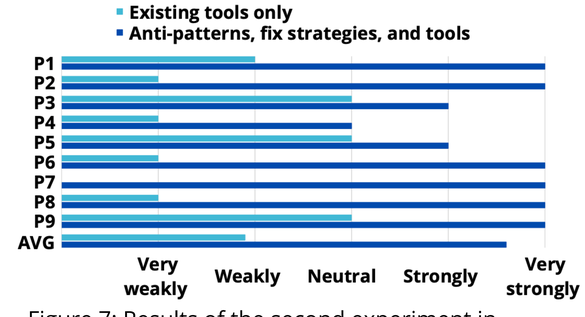

Figure 7: Results of the second experiment in repairing IA regexes

REFERENCES

[1] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *(ESEC/FSE)*, 2019.

[2] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale," in *(ESEC/FSE)*, 2018

ACKNOWLEDGEMENTS

Partial support provided by NSF award #2135156 and URJC award C01INVESDIST.