Static Analysis for Regular Expression Denial-of-Service Attacks

Asiri Rathnayake, Hayo Thielecke and James Kirrage

University of Birmingham, UK

Overview

Background

- Regular expressions are everywhere (e.g. validation, sanitization, etc.)
- Most matchers are based on backtracking (.NET, Java, PCRE, Perl, etc.)

Overview

Background

- Regular expressions are everywhere (e.g. validation, sanitization, etc.)
- Most matchers are based on backtracking (.NET, Java, PCRE, Perl, etc.)

The problem

- Backtracking can lead to exponential runtimes
- This can be exploited: denial-of-service attack (REDoS)

Overview

Background

- Regular expressions are everywhere (e.g. validation, sanitization, etc.)
- Most matchers are based on backtracking (.NET, Java, PCRE, Perl, etc.)

The problem

- Backtracking can lead to exponential runtimes
- This can be exploited: denial-of-service attack (REDoS)

The solution / Our contribution

- What property causes exponential runtimes?
- A static analysis (tool)

Example

Validate a 24-hour formatted string:

```
^(([01][0-9]|[012][0-3]):([0-5][0-9]))*$
```

Example

Validate a 24-hour formatted string:

```
^(([01][0-9]|[012][0-3]):([0-5][0-9]))*$
```

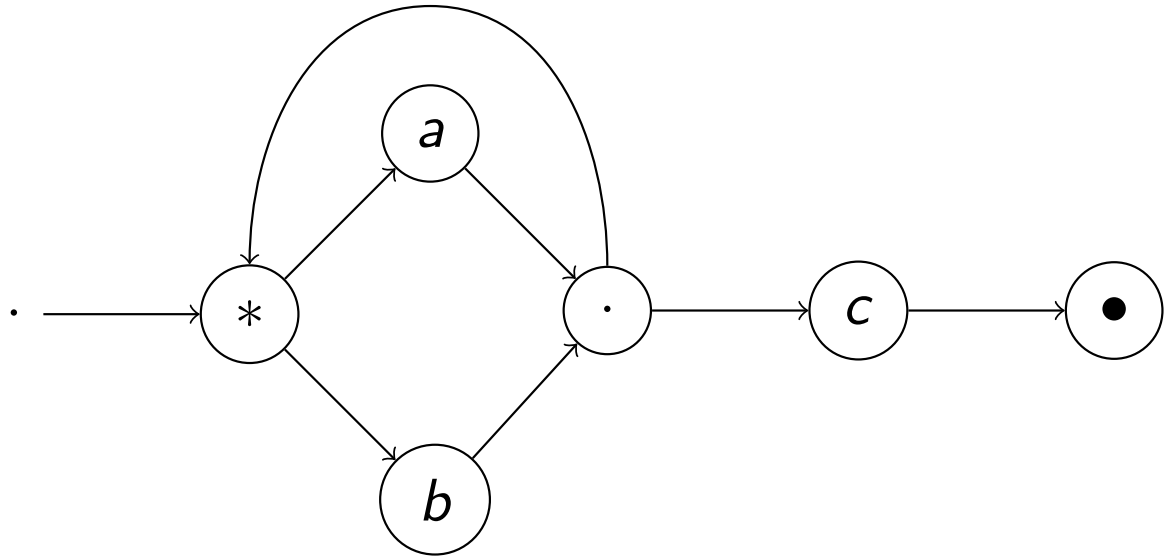
Tool output:

- Prefix: ε
- Pumpable: `13:59`
- Suffix: `/`

Attack string: $(\texttt{13:59})^n\texttt{/}$

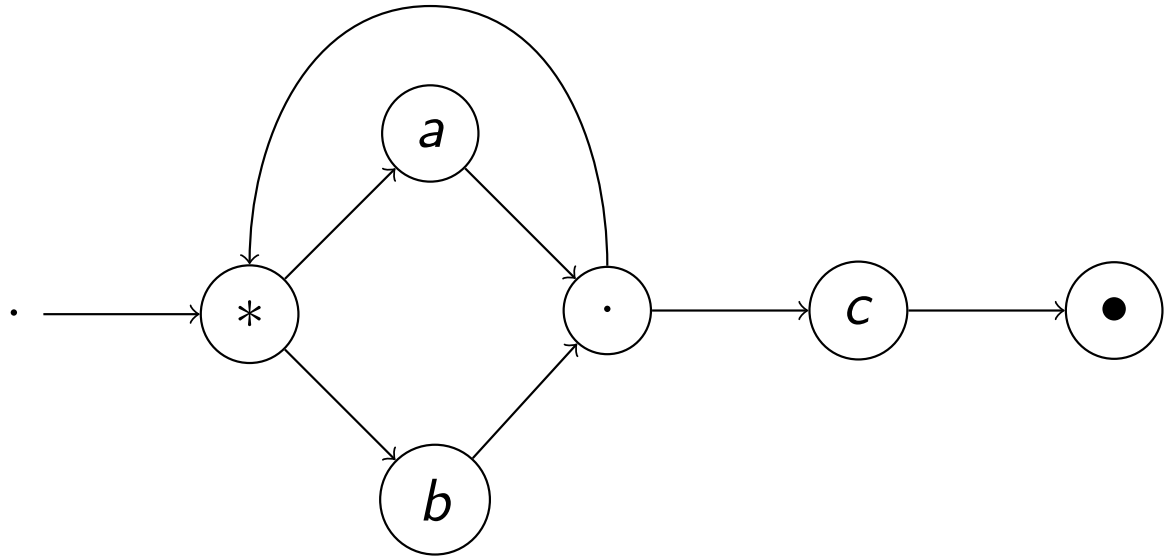
Backtracking

Example: $(a|b)^*c$



Backtracking

Example: $(a|b)^*c$



Backtracking:

- Try one branch first
- If it fails, try the next one

The cause of exponential runtime

- Exponential runtime - because of backtracking ?

The cause of exponential runtime

- Exponential runtime - because of backtracking ?
- Culprit: Non-deterministic Kleene sub-expressions

The cause of exponential runtime

- Exponential runtime - because of backtracking ?
- Culprit: Non-deterministic Kleene sub-expressions
- Example: $(a \mid ab \mid b)^*$
  - Input the string: ab

Exponential behaviour

- Vulnerable expression (simplified): $e_1 e_2{}^* e_3$

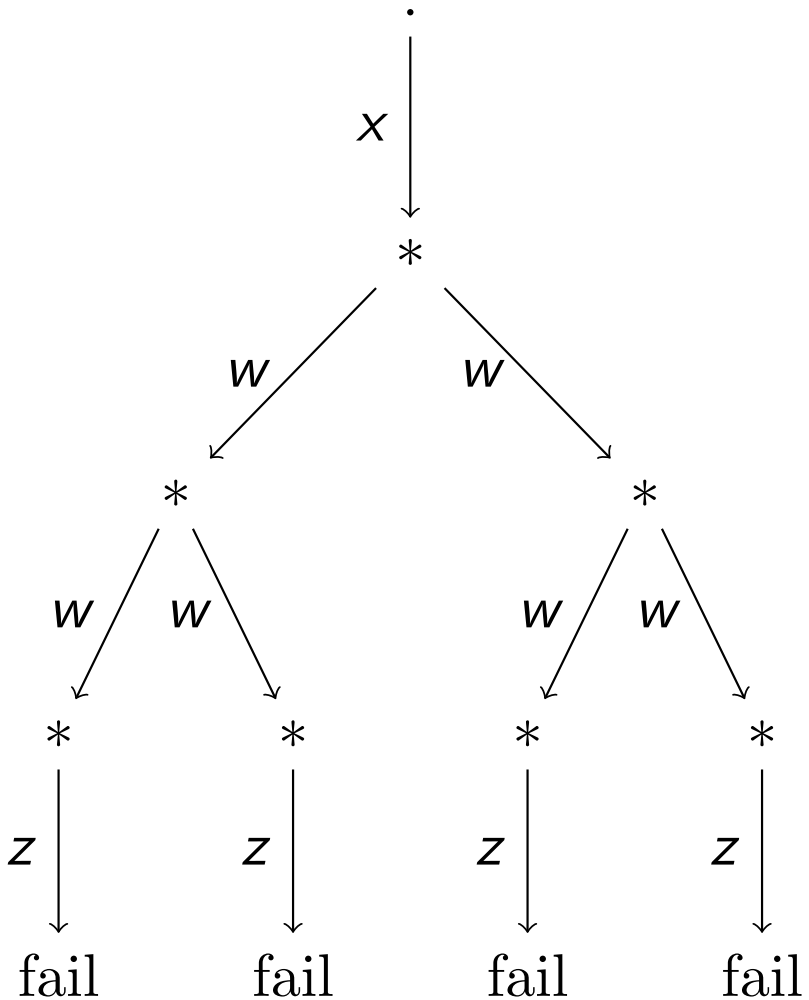
Exponential behaviour

- Vulnerable expression (simplified): $e_1 e_2{}^* e_3$
- Attack string (tool output): $xw^n z$

Exponential behaviour

- Vulnerable expression (simplified): $e_1 {e_2}^* e_3$
- Attack string (tool output): $x w^n z$
  - x - Prefix ($x \in e_1$)
  - w - Pumpable string ($w \in {e_2}^*$: in more than one way)
  - z - Failure suffix ($x w^n z \notin e_1 {e_2}^* e_3$)

Exponential behaviour



Experimental results

	RegExLib	Snort
Total patterns	2994	12499
Analyzable (only regular constructs)	2213 (˜74%)	9408 (˜75%)
Uses Kleene star	1103 (˜50%)	2741 (˜30%)
Pumpable Kleene and suffix found	127 (˜12%)	15 (˜0.5%)
Total classification time (Intel Core 2 Duo 1.8 MHz, 4 GB RAM)	40 s	10 s

Experimental results

	RegExLib	Snort
Total patterns	2994	12499
Analyzable (only regular constructs)	2213 (˜74%)	9408 (˜75%)
Uses Kleene star	1103 (˜50%)	2741 (˜30%)
Pumpable Kleene and suffix found	127 (˜12%)	15 (˜0.5%)
Total classification time (Intel Core 2 Duo 1.8 MHz, 4 GB RAM)	40 s	10 s

- Orders of magnitude faster than Microsoft's SDL Fuzzer (micro-seconds vs minutes per regular expression!)

Conclusions

- An effective tool for detecting REDoS vulnerabilities
- REDoS is real, many real-world examples exist
- Our analysis is magnitudes faster than fuzzing, fast enough to be a plug-in for an IDE

Future work

- Improve the analysis (prefix / suffix generation)
- Support non-regular constructs. Example:
  - Pattern (non-regular): `/(c|d)(ab|a|b)*\g1/`
  - Attack string: $c(ab)^n d$
- More information / updates / downloads:
  http://www.cs.bham.ac.uk/~hxt/research/rxxr.shtml

Thank you!