Finding catastrophic backtracking

Statically (and dynamically)

quasilyte @ DevFest Vladivostok 2019

About me

Before: Intel-Go
Now: VK infrastructure

#GolangKazan
Go GDE Russia

Why we are here?

Regular expressions...
Good, bad and (sometimes) slow.

Terminology note

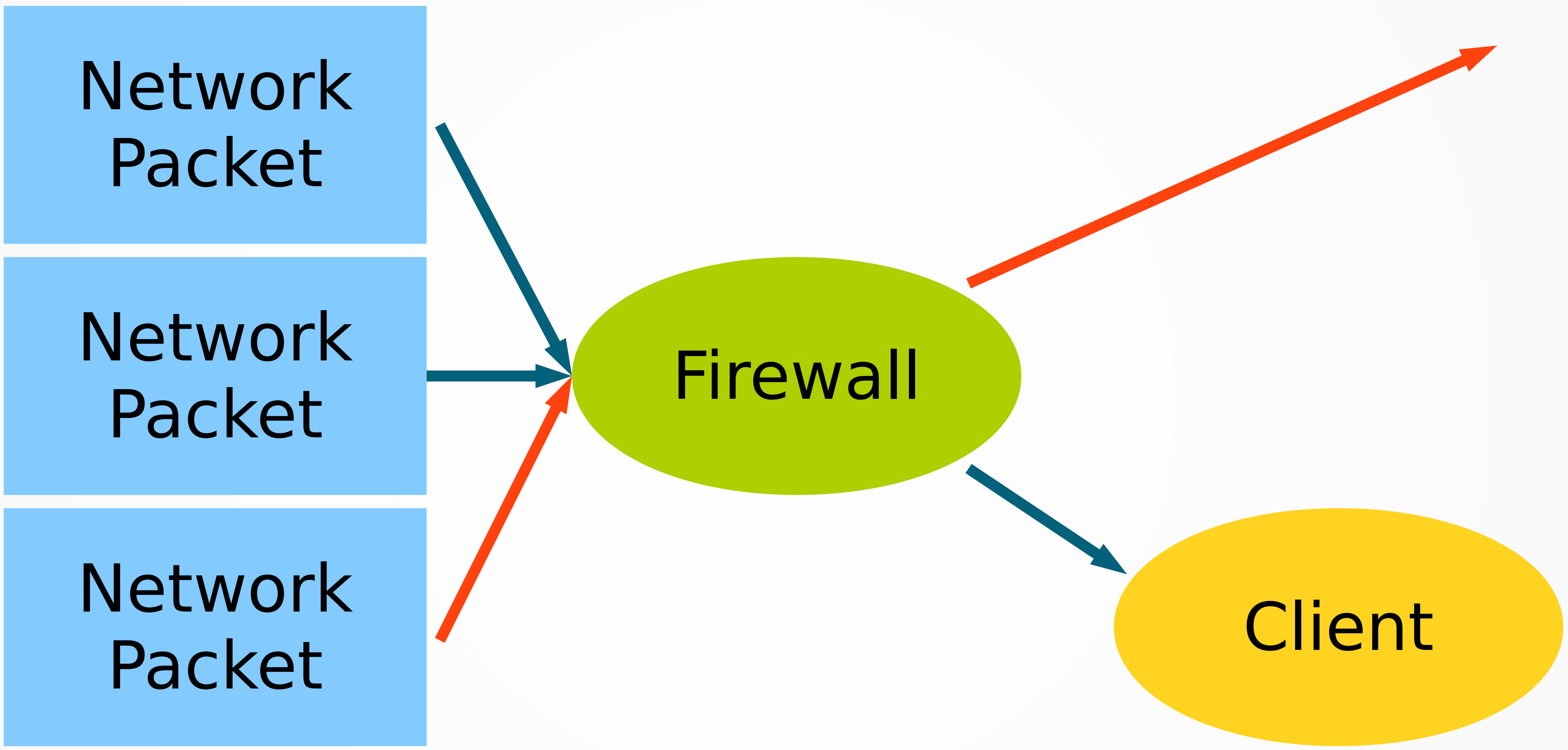
I usually use “regexp” term when speaking about “regular expressions”.

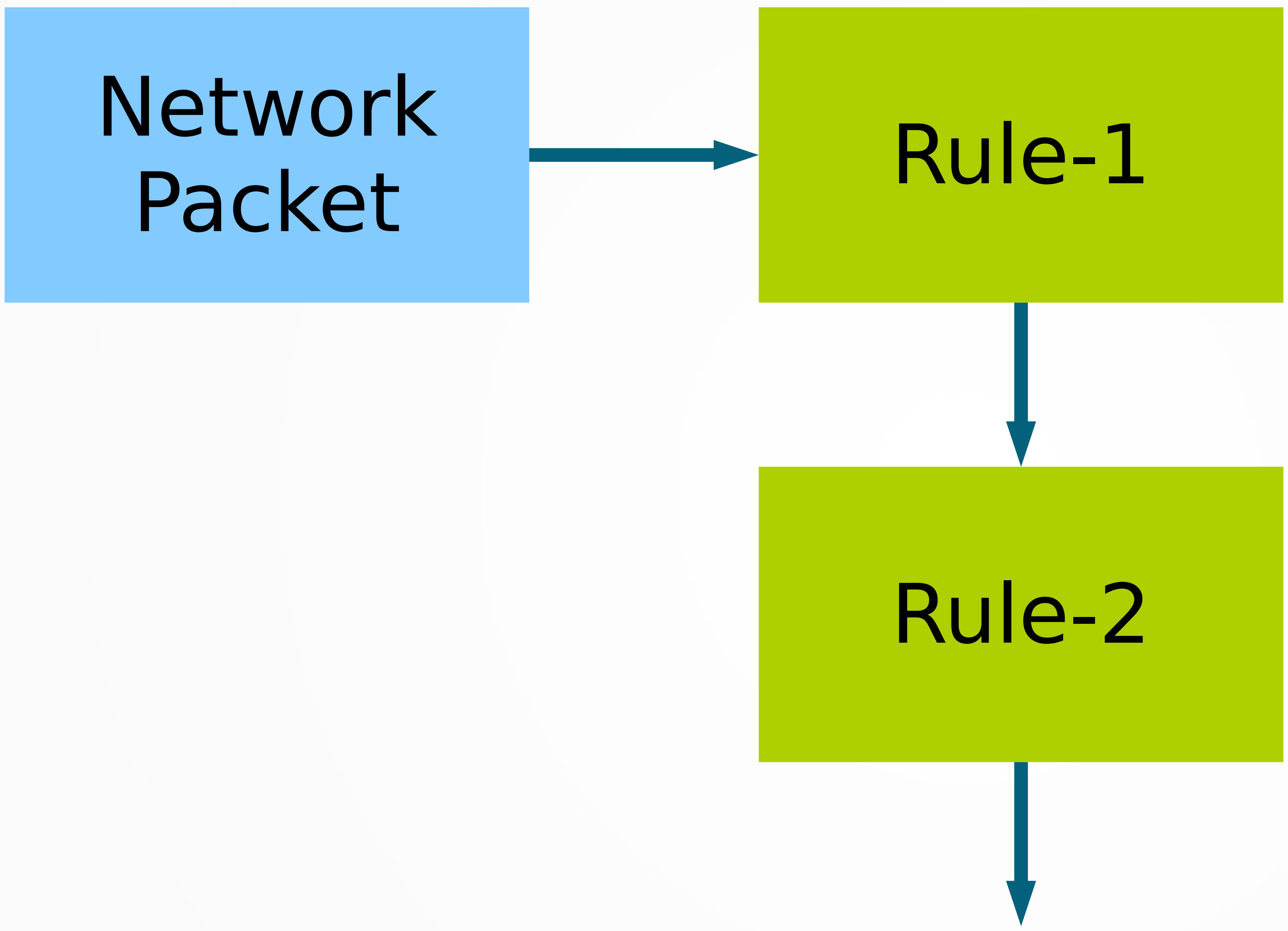
Solving imaginary task

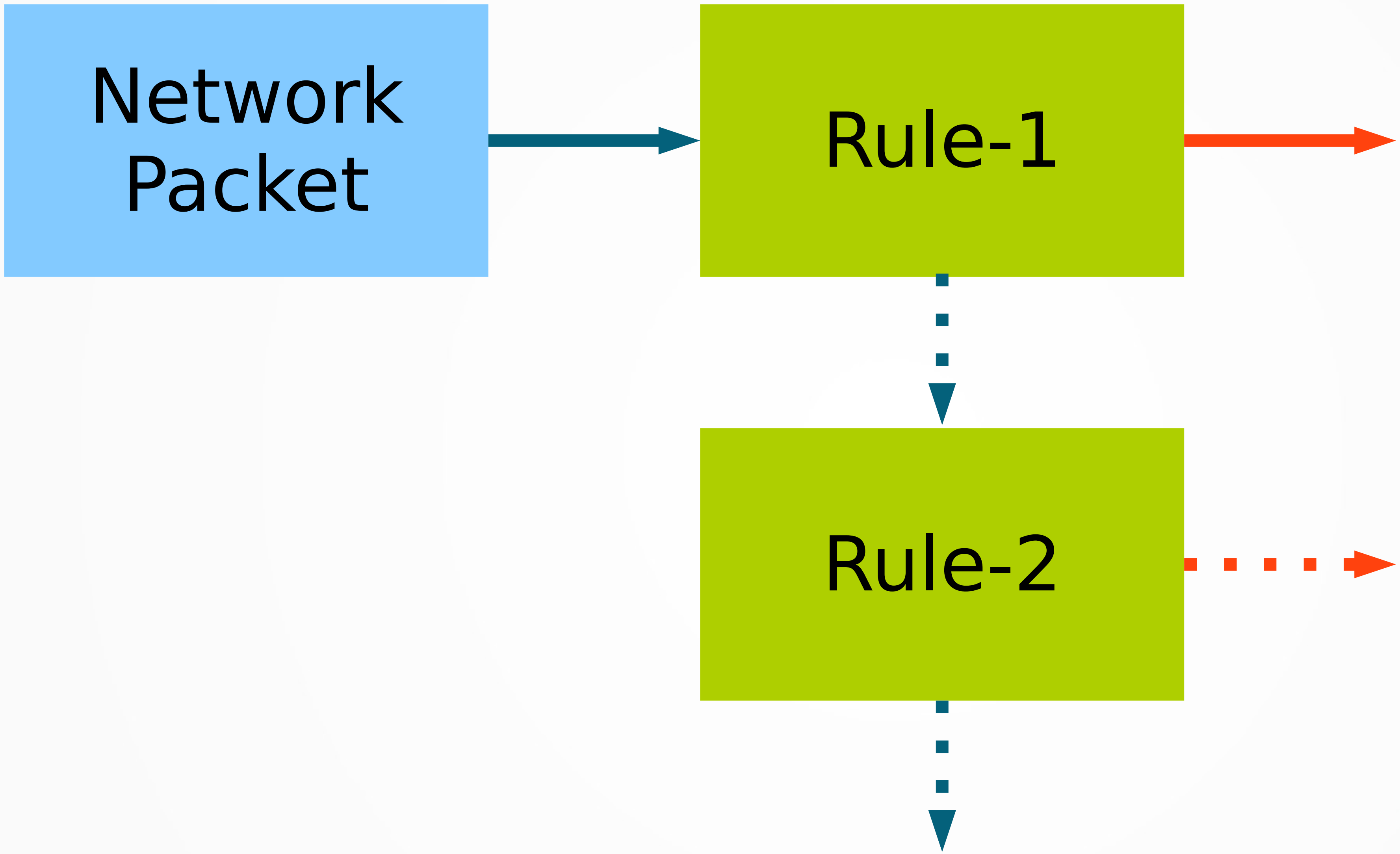
Let’s design a firewall-like system for network traffic.

Network firewall

Filtering network traffic to find malicious requests and payloads before routing it to the clients.


Network Packet
Network Packet
Network Packet
Firewall
Client


Network
Packet
Rule-1
Rule-2


Network
Packet
Rule-1
Rule-2

How to implement “rules”?

We need to match arbitrary text against some predefined patterns…

Enter: regular expressions

- Available almost everywhere
- Recognized by most programmers
- Solving a task almost without coding
- Have good backing theory

What can go wrong?

Cloudflare outage: July 2, 2019

https://bit.ly/2xJ2YQV

Lessons learned

Regular expressions can be slow,
they can introduce security issues.

Questions to answer

- When/why regular expressions are slow?
- How to detect a slow regular expression?
- How to fix those regular expressions?
- Are all regexp engines so dangerous?

Questions to answer

- When/why regular expressions are slow?
- How to detect a slow regular expression?
- How to fix those regular expressions?
- Are all regexp engines so dangerous?

But before that...

What ***is*** catastrophic backtracking?

Catastrophic backtracking

A situation where regexp matcher scans the input string a lot of times.

Usually, the performance degrades exponentially.

Pathological regexp

A pattern that behaves much-much worse than “normal” patterns.

All catastrophic backtracking cases belong to them.

We want to know more!

Let’s see whether this is a global problem and where it comes from.

Regexp libraries

- PHP - “pcre” module
- JS - RegExp class
- Python - “re” module
- C++ - “regex” library
- Go - “regexp” package

Regexp libraries

- PHP - “pcre” module **BACKTRACKING**
- JS - RegExp class **BACKTRACKING**
- Python - “re” module **BACKTRACKING**
- C++ - “regex” library **BACKTRACKING**
- Go - “regexp” package

Danger Zone

Regexp libraries

- PHP - “pcre” module
- JS - RegExp class
- Python - “re” module
- C++ - “regex” library
- Go - “regexp” package

Safe ʕ⊙ϖ⊙ʔ

Regexp implementation

- Perl-like engine with backreferences
- RE2-like engine

Regexp implementation

- **Perl-like engine with backreferences**
- RE2-like engine

If you’re unlucky:

- Exponential dependency on input length
- Can eat a lot of memory
- “Optimistic”

Regexp implementation

- Perl-like engine with backreferences
- **RE2-like engine**

Always:

- Linear run time, O(N) over input
- Constant memory consumption
- “Pessimistic”

Benchmarking Python

```
import re
rows = [
    '0,0,0,0,0,0,0,0,0',
    '0,0,0,0,0,1,0,0,0',
    '0,1,0,0,0,0,1,0,0',
]
haystack = '\n'.join(rows)
```

Benchmarking Python

```
pat = ''.join(['(.*,)*1(,[^\n]*)?',
               '\n',
               '([^\n]*,)*1,?'])

print(re.search(pat, s, flags=re.S))
```

Benchmarking Python

```
(.*,)*1(,[^\n]*)?
\n
([^\n]*,)*1,?
```

Benchmarking Python

```
$ time python regexp.py
<_sre.SRE_Match object>
real 0m0,046s
user 0m0,038s
sys  0m0,008s
```

Benchmarking Python

```
import re
rows = [
    '0,0,0,0,0,0,0,0,0',
    '0,0,0,0,0,0,0,0,0',
    '0,1,0,0,0,0,1,0,0',
]
haystack = '\n'.join(rows)
```

Benchmarking Python

```
$ time python regexp.py
None
real 0m7,075s
user 0m7,063s
sys  0m0,012s
```

0.046s vs 7s

Non-matching case was ~150x times slower.

Benchmarking Python

```
import re
rows = [
    '0,0,0,0,0,0,0,0,0,0,0',
    '0,0,0,0,0,0,0,0,0,0,0',
    '0,1,0,0,0,0,1,0,0,0,0',
]
haystack = '\n'.join(rows)
```

Benchmarking Python

```
$ time python regexp.py
```

Benchmarking Python

```
$ time python regexp.py
```

Benchmarking Python

```
$ time python regexp.py
```

Benchmarking Python

```
$ time python regexp.py
None
real 6m26,291s
user 6m26,286s
sys  0m0,004s
```

6 MINUTES!

Benchmarking PHP

```php
<?php
$rows = [
    '0,0,0,0,0,0,0,0,0,0,0',
    '0,0,0,0,0,0,0,0,0,0,0',
    '0,1,0,0,0,0,1,0,0,0,0',
];
$s = implode($rows, "\n");
```

Benchmarking PHP

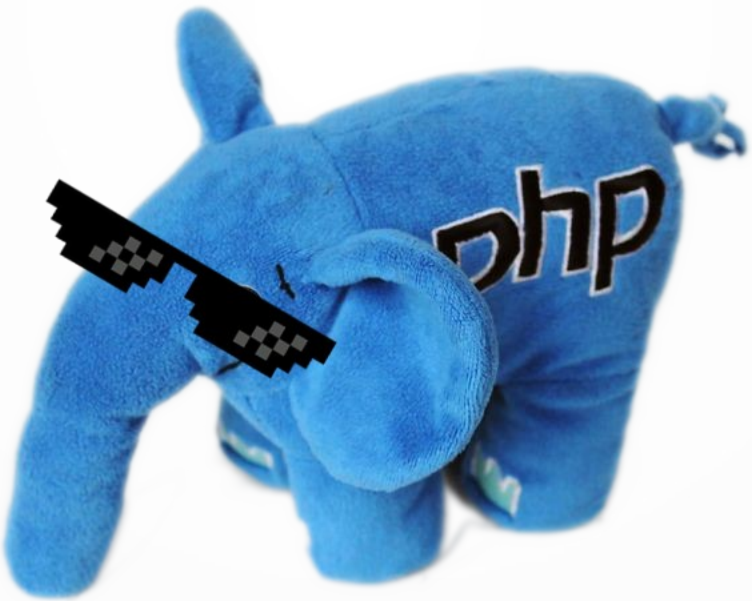
```
$pat = implode([
    '(.*,)*1(,[^\n]*)?',
    '\n',
    '([^\n]*,)*1,?',
], '');

var_dump(preg_match("/$pat/s", $s));
```

Benchmarking PHP

```
$ time php -f regexp.php
bool(false)
real 0m0,024s
user 0m0,016s
sys  0m0,008s
```

Wow! PHP is fast!

Our regexp executed in 0.024s!



But what does “false” mean?

Let’s see what docs say.

preg_match return values

1 - string is matched

0 - string is not matched

false – error occurred

Analyzing preg error

```
var_dump(preg_last_error());
// => int(2)
// => PREG_BACKTRACK_LIMIT_ERROR
```

What exactly takes time here?

Let’s see how excessive “backtracking” hurts the efficiency.

Let's backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 0: # initial state
  A0 → (a) → (b) → (a)
(s)                     (m)
  A1 → (a) → (b) → (b)
```

Let's backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 1: # try alt0...
  A0 → (a) → (b) → (a)
(s)                     (m)
  A1 → (a) → (b) → (b)
```

Let's backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 2: # alt0 matched "a"
  A0 → (a) → (b) → (a)
(s)                      (m)
  A1 → (a) → (b) → (b)
```

Let’s backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 3: # alt0 matched "ab"
  A0 → (a) → (b) → (a)
(s)                        (m)
  A1 → (a) → (b) → (b)
```

Let's backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 4: # alt0 failed to match "aba"
  A0 → (a) → (b) → (a)
(s)                         (m)
  A1 → (a) → (b) → (b)
```

Let's backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 5: # backtrack to (s)
  A0 → (a) → (b) → (a)
(s)                        (m)
  A1 → (a) → (b) → (b)
```

Let’s backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 6: # alt1 matched "a"
  A0 → (a) → (b) → (a)
(s)                      (m)
  A1 → (a) → (b) → (b)
```

Let’s backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 7: # alt1 matched "ab"
  A0 → (a) → (b) → (a)
(s)                       (m)
  A1 → (a) → (b) → (b)
```

Let’s backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 8: # alt1 matched "abb"
  A0 → (a) → (b) → (a)
(s)                     (m)
  A1 → (a) → (b) → (b)
```

Let's backtrack together!

```
regexp = "aba|abb"
input  = "abb"


Step 9: # input matched!
  A0 → (a) → (b) → (a)
(s)                     (m)
  A1 → (a) → (b) → (b)
```

Determenistic approach

https://bit.ly/1Fo3RaY

NFA with parallel matching

```
regexp = "aba|abb"
input  = "abb"


Step 0: # initial state
  A0 → (a) → (b) → (a)
(s)                     (m)
  A1 → (a) → (b) → (b)
```

NFA with parallel matching

```
regexp = "aba|abb"
input  = "abb"


Step 1: # try both alts
   A0 → (a) → (b) → (a)
(s)                       (m)
   A1 → (a) → (b) → (b)
```

NFA with parallel matching

```
regexp = "aba|abb"
input  = "abb"


Step 2: # both alts matched "a"
  A0 → (a) → (b) → (a)
(s)                    (m)
  A1 → (a) → (b) → (b)
```

NFA with parallel matching

```
regexp = "aba|abb"
input  = "abb"


Step 3: # both alts matched "ab"
  A0 → (a) → (b) → (a)
(s)                        (m)
  A1 → (a) → (b) → (b)
```

NFA with parallel matching

```
regexp = "aba|abb"
input  = "abb"


Step 4: # alt1 matched, alt0 not
  A0 → (a) → (b) → (a)
(s)                   (m)
  A1 → (a) → (b) → (b)
```

NFA with parallel matching

```
regexp = "aba|abb"
input  = "abb"


Step 5: # input matched!
  A0 → (a) → (b) → (a)
(s)                    (m)
  A1 → (a) → (b) → (b)
```

How to fix slow regexp?

- Use better regexp engine (like re2)
- Try regexp optimizers

If you don’t fix slow regexps,

your system can be vulnerable.

When system is vulnerable?

- Patterns from the user input
- Target string comes from the user input

But if you have to use regexps,

there are ways to minimize the risk.

Reducing risks

- Patterns from the user input
  - Use regexp run time limits
- Target string comes from the user input
  - Limit input strings size

When everything else is not working,

be as suspicious as possible.

Regex101 site uses timeouts

So you can’t shut it down using your craziest regexp.

How to find scary regexps?

There are both *static* and *dynamic* ways to do it.

Finding them statically

By *statically* I mean “before the real program is executed”.

But regexp evaluation can be considered as a kind of dynamic analysis (we need to run them).

Finding them statically

Algorithm:

- Collect all analyzable (known) regexps

Finding them statically

Algorithm:

- Collect all analyzable (known) regexps
- For every regexp, do cost evaluation:
  - Parse regexp

Finding them statically

Algorithm:

- Collect all analyzable (known) regexps
- For every regexp, do cost evaluation:
  - Parse regexp
  - Build the hardest case input string

Finding them statically

Algorithm:

- Collect all analyzable (known) regexps
- For every regexp, do cost evaluation:
  - Parse regexp
  - Build the hardest case input string
  - Collect stats (regexp lib dependent)

Finding them statically

Algorithm:

- Collect all analyzable (known) regexps
- For every regexp, do cost evaluation:
  - Parse regexp
  - Build the hardest case input string
  - Collect stats (regexp lib dependent)
- Analyze how run time cost increases with N

Generating input string

We need to cover all possible “worst” scenarios.

- All alternation permutations should be tested
- Regexp should fail in the end

Input generator should be able to generate strings of different length, depending on the N argument.

Generating input string

- For *x+* we generate at least 1 match
- For *x** we usually generate at least 1 match
- For *x?* we generate 1 match

To control result length, *x+* and *x** generate a number of matches that satisfy a given N argument.

Generating input string

```
gen_matching("a+a+b", n=3)
# => aab


gen_matching("a+a+b", n=6)
# => aaaaab


gen_matching("a*a*b", n=2)
# => ab
```

Making it non-matching

Last regexp op (part) should generate a non-matching char.

Non-matching strings

```
gen("a+a+b", n=3)
# => aax

gen("a*a*b", n=2)
# => ax


# Both fail on the last character.
```

Evaluation

If regexp run time increases exponentially with gen func N argument, issue a warning about problematic pattern.

Finding them dynamically

Algorithm:

- Wrap all your library regexp-match calls
- Log problematic regexps after a timeout

After some time you'll know which

regexps cause more troubles during the run time.

Regexp performance linter

Coming soon
Not open sourced yet...

Conclusions

Regexps are not slow

Regexp engine can be flawed though.

Don’t be afraid of regexps

But don’t rely on
non-determenistic regexp engines
too much.

If unsure, check statically

If you already have CI with linters, add one that checks your regexps complexity.

Quote for unconvinced

Given a choice between being predictable and fast on all inputs and being quick on some of them while taking years on others, the decision should be easy.

(Shortened from Russ Cox.)

```
pat.match("questions?")
```

Thank you for attention

quasilyte @ DevFest Vladivostok 2019