Ruby master - Feature #17837
Add support for Regexp timeouts
04/27/2021 11:39 PM - sam.saffron (Sam Saffron)

Status: Open
Priority: Normal
Assignee: 
Target version: 

Description

Background

ReDoS are a very common security issue. At Discourse we have seen a few through the years. 

In a nutshell there are 100s of ways this can happen in production apps, the key is for an attacker (or possibly innocent person) to 
supply either a problematic Regexp or a bad string to test it with.

/A(B|C+)+D/ =~ "A" + "C" * 100 + "X"

Having a problem Regexp somewhere in a large app is a universal constant, it will happen as long as you are using Regexp

Currently the only feasible way of supplying a consistent safeguard is by using Thread.raise and managing all execution. This kind of 
pattern requires usage of a third party implementation. There are possibly issues with jRuby and Truffle when taking approaches like this.

Prior art

.NET provides a MatchTimeout property per: 
https://docs.microsoft.com/en-us/dotnet/api/system.text.regexes.regex.matchtimeout?view=net-5.0

Java has nothing built in as far as I can tell: https://stackoverflow.com/questions/910740/cancelling-a-long-running-regex-match

Node has nothing built in as far as I can tell: https://stackoverflow.com/questions/38859506/cancel-regex-match-if-timeout

Golang and Rust uses RE2 which is not vulnerable to DoS by limiting features (available in Ruby RE2 gem)

irb(main):003:0> r = RE2::Regexp.new('A(B|C+)+D')
=> #<RE2::Regexp /A(B|C+)+D/>
irb(main):004:0> r.match("A" + "C" * 100 + "X")
=> nil

Proposal

Implement Regexp.timeout which allow us to specify a global timeout for all Regexp operations in Ruby.

Per Regexp would require massive application changes, almost all web apps would do just fine with a 1 second Regexp timeout.

If timeout is set to nil everything would work as it does today, when set to second a "monitor" thread would track running regexps and 
time them out according to the global value.

Alternatives

I recommend against a "per Regexp" API as this decision is at the application level. You want to apply it to all regular expressions in 
all the gems you are consuming.

I recommend against a move to RE2 at the moment as way too much would break

See also:

https://levelup.gitconnected.com/the-regular-expression-denial-of-service-redos-cheat-sheet-a78d0ed7d865
I like this idea. It might be better to use SIGVTALRM instead of a monitor thread. However, it may affect the performance of a program that repeatedly uses a small and trivial regexp. Anyone can try to implement it and evaluate the performance?

I wonder if even a lightweight SIGVTALRM may be too much of a performance hit? On the upside though not needing to think about a background thread is nice!

That way every 100ms (or something else reasonable) we would walk all threads checking for a particular regexp execution. If you see the same globally increasing "regexp run number" twice you know it has been running for at least 100ms.

That way ultra short regexps pay almost no cost (log regexp_run_number++ in thread local storage is about all)

Invoking a thread implicitly in the interpreter is troublesome. Previously, Ruby had a timer thread, but as far as I know, it was (almost) abandonned by normalperson (Eric Wong). If you try to revive a timer thread, you should learn the complex history first.

Another idea suggested by naruse (Yui NARUSE): simply recording the start time of onig_search, calculating the elapsed time at CHECK_INTERRUPT_IN_MATCH_AT, and raising an exception if the time limit exceeded. This approach is very tractable because it does not use any asynchronous things like SIGVTALRM nor a thread.

Even without the DoS aspect, it's all too easy to create regexps with pathological performance that only manifests in certain edge cases, usually in production. It would be very useful if some kind of timeout exception was raised. Ideally that exception should have references (attr_reader) to both the regexp and string that caused the timeout.

Shouldn't an app have a global timeout per request anyway, and that would catch regexps and other things too?

Could such regexps emit a warning since they can result in pathological backtracking?

Or is it too difficult to detect them / the problematic patterns evolve over time?

So that would be one way: warn for those regexps that cannot be implemented without backtracking, but that's probably more restrictive than needed.

Sort of, it gets complicated. Unicorn is easy cause it is single threaded. Killing off threads in Puma is much more fraught, in Sidekiq the old pattern of killing off was nuked by Mike cause he saw it as way too risky

https://github.com/mperham/sidekiq/commit/7e094567a585578fad0b6d0c8669efb46643f653.

Or is it too difficult to detect them / the problematic patterns evolve over time?

Sadly I think they are very hard to predict upfront.

I do hear you though, a zero cost when no timeout is defined and very cheap cost when a timeout is defined seems non trivial to implement.

Eregon (Benoit Daloze) wrote in #note-5:

Shouldn't an app have a global timeout per request anyway, and that would catch regexps and other things too?
I agree that it is much better to have. Still, I think this proposal is good-to-have because, IMO, it mitigates ReDoS generally. But I admit that this is less important than per-request timeout.

Capturing the time in regexp interrupt checks is easy but sounds fairly expensive.

Agreed, this is my main concern. Thus I think this proposal requires careful performance evaluation.

#8 - 05/03/2021 02:57 PM - byroot (Jean Boussier)

I admit that this is less important than per-request timeout.

I’d like to second what Sam said on this. With what is now the most common way of deploying Ruby in production, namely a threaded web-server like Puma, there is no good way to have a global request timeout. The only mechanism that is semi-working is Timeout.timeout so ultimately Thread.raise which is very likely to leave the process in a corrupted state. Only forking servers can actually provide a reliable request timeout through SIGTERM / SIGKILL.

#9 - 05/04/2021 12:12 AM - duerst (Martin Dürst)

sam.saffron (Sam Saffron) wrote in #note-6:

Or is it too difficult to detect them / the problematic patterns evolve over time?

Sadly I think they are very hard to predict upfront.

In general, yes. But for an extremely large set of regular expressions, it’s easy to predict that they are harmless. And some specific patterns in regular expressions are clear signs that something might go wrong.

I do hear you though, a zero cost when no timeout is defined and very cheap cost when a timeout is defined seems non trivial to implement.

I very strongly suggest that this feature be voluntary, e.g. as an additional flag on the regular expression.

#10 - 05/04/2021 10:31 AM - Eregon (Benoit Daloze)

sam.saffron (Sam Saffron) wrote in #note-6:

Sort of, it gets complicated. Unicorn is easy cause it is single threaded. Killing off threads in Puma is much more fraught, in Sidekiq the old pattern of killing off was nuked by Mike cause he saw it as way too risky

https://github.com/mperham/sidekiq/commit/7e094567a585578fad08bd0c8669efb46643853.

I think fixing Timeout.timeout might be possible. The main/major issue is it can trigger within ensure, right? Is there anything else? We could automatically mask Thread.raise within ensure so it only happens after the ensure body completes. And we could still have a larger "hard timeout" if an ensure takes way too long (shouldn't happen, but one cannot be sure). I recall discussing this with schneems (Richard Schneemann) some time ago on Twitter.

#11 - 05/04/2021 12:55 PM - Dan0042 (Daniel DeLorme)

duerst (Martin Dürst) wrote in #note-9:

I very strongly suggest that this feature be voluntary, e.g. as an additional flag on the regular expression.

If you have to turn it on for each regexp, that would make the feature kinda useless. I agree with the OP that this decision is at the application level. You want it either on or off for all/most regexps. Although it would make sense to be able to override the default timeout for a few specific regexps that are known to be time-consuming or performance-critical.

Rather than CHECK_INTERRUPT_IN_MATCH_AT would it be feasible to check for timeouts only when backtracking occurs?

#12 - 05/04/2021 02:22 PM - byroot (Jean Boussier)

The main/major issue is it can trigger within ensure, right? Is there anything else?

That would fix most issues, but not all. It can also trigger in places where exception are entirely unexpected, so there's just no ensure.
Also I'm not clear on the details, but some C extensions (often various clients) can't be interrupted by Thread.raise.

#13 - 05/05/2021 01:41 AM - duerst (Martin Dürst)
- Related to Feature #17849: Fix Timeout.timeout so that it can be used in threaded Web servers added

#14 - 05/05/2021 02:02 AM - duerst (Martin Dürst)

Eregon (Benoit Daloze) wrote in #note-10:

I think fixing Timeout.timeout might be possible. The main/major issue is it can trigger within ensure, right? Is there anything else? We could automatically mask Thread.raise within ensure so it only happens after the ensure body completes. And we could still have a larger "hard timeout" if an ensure takes way too long (shouldn't happen, but one cannot be sure). I recall discussing this with schneems (Richard Schneeman) some time ago on Twitter.

I created a separate issue for the improvement of Timeout.timeout: #17849. Please feel free to discuss there. My guess is that there are all kinds of other issues that can happen in a Web application, so it would be better to solve this for the general case.

Dan0042 (Daniel DeLorme) wrote in #note-11:

duerst (Martin Dürst) wrote in #note-9:

I very strongly suggest that this feature be voluntary, e.g. as an additional flag on the regular expression.

If you have to turn it on for each regexp, that would make the feature kinda useless. I agree with the OP that this decision is at the application level.

I have no problems with making it possible to switch this on at the application level.

You want it either on or off for all/most regexps. Although it would make sense to be able to override the default timeout for a few specific regexps that are known to be time-consuming or performance-critical.

Yes. My assumption is that when writing a regular expression, the writer should make sure it's well behaved. So in general, timeouts would only be needed for regular expressions that come from the outside.

Rather than CHECK_INTERRUPT_IN_MATCH_AT would it be feasible to check for timeouts only when backtracking occurs?

In a backtracking regular expression engine, backtracking occurs very often. There are many cases of backtracking that are still totally harmless. Ideally, a regular expression engine would deal with most regular expressions in a way similar to what RE2 (or any DFA-based implementation) does, and only use a timeout for those that a DFA-based strategy cannot handle (backreferences,...). But that would require quite a bit of implementation work.

(Of course all the above discussion is predicated on the assumption that timeouts cannot be added to regular expressions with negligible speed loss.)

#15 - 05/06/2021 01:30 PM - Dan0042 (Daniel DeLorme)

duerst (Martin Dürst) wrote in #note-14:

In a backtracking regular expression engine, backtracking occurs very often. There are many cases of backtracking that are still totally harmless.

Even if backtracking occurs very often, my thinking was that it occurs less often than CHECK_INTERRUPT_IN_MATCH_AT. But now that I'm looking at regexec.c I'm not so sure anymore. I can't make heads or tails of that code. But still, the slowness of a regexp is directly correlated to how much backtracking occurs, so it would make sense to tie the timeout into that. Like, check the timeout at every 256th backtrack or something like that. If anyone can figure out what constitutes a "backtrack" in the regexec code.

Ideally, a regular expression engine would deal with most regular expressions in a way similar to what RE2 (or any DFA-based implementation) does

Yes, but that's rather out of scope for this ticket isn't it?

#16 - 05/06/2021 03:20 PM - Eregon (Benoit Daloze)

I noticed there is a reply in [ruby-core:103730] by normalperson (Eric Wong) that unfortunately doesn't seem to be mirrored on Redmine:
http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/103730
I think that backtracking limit would be better than timeout.

I've created a simple prototype of Regexp.timeout= by a polling approach.

Conclusion first. It brings about 5% overhead in micro benchmark, unfortunately. I guess it is unlikely to be significant in a real application, but not good anyway.

The following is about my patch, just for the record.

https://github.com/ruby/ruby/compare/master...mame:regexp-timeout-prototype

Implementation approach:

1. When starting regexp matching, the current time is recorded by using clock_gettime(CLOCK_MONOTONIC)
2. At CHECK_INTERRUPT_IN_MATCH_AT, the elapsed time is calculated and an exception is raised if expired

Example:

Regexp.timeout = 1 # one second
/^((a-z))+$/ =~ "abcdefghijklmnopqrstuvwxyz@" => regexp match timeout (RuntimeError)

Benchmark:

The following simple regexp matching becomes 4.8% slower.

10000000.times { /(abc)+/ =~ "abcabcabc" }

# The minimum time in 10 executions
# before: 1.962 s
# after: 2.056 s

The following complex regexp matching becomes 5.2% slower.

/^((a-z))+$/ =~ "abcdefghijklmnopqrstuvwxyz@"

# The minimum time in 10 executions
# before: 2.237 s
# after: 2.353 s

mame (Yusuke Endoh) not sure if the compiler takes care of this but maybe we can avoid calls to GET_THREAD if the static reg_match_time_limit is not set, just bypass all of this if the static is not set?

I tested with:

diff --git a/thread.c b/thread.c
index 47e43ecb63..811b6e88a8 100644
--- a/thread.c
+++ b/thread.c
@@ -1573,25 +1573,29 @@ rb_thread_reg_match_time_limit_get()
 void
 rb_thread_reg_match_start(void)
 {
- rb_thread_t *th = GET_THREAD();
- if (reg_match_time_limit) {
- th->reg_match_end_time = rb_hrtime_add(reg_match_time_limit, rb_hrtime_now());
- }
- else {
- th->reg_match_end_time = 0;
+ rb_thread_t *th = GET_THREAD();
+ if (reg_match_time_limit) {
+ th->reg_match_end_time = rb_hrtime_add(reg_match_time_limit, rb_hrtime_now());
+ }
+ else {
+ th->reg_match_end_time = 0;
+ }
+ }
void rb_thread_reg_check_ints(void)
{
    rb_thread_t *th = GET_THREAD();
    if (reg_match_time_limit) {
        rb_thread_t *th = GET_THREAD();
        if (th->reg_match_end_time && th->reg_match_end_time < rb_hrtime_now()) {
            VALUE argv[2];
            argv[0] = rb_eRuntimeError;
            argv[1] = rb_str_new2("regexp match timeout");
            rb_threadptr_raise(th, 2, argv);
        }
    }
    rb_thread_check_ints();
}

'10000000.times { /(abc)+/ =~ "abcabcabc" }

Before (min over 10 runs): 1.590 after 1.610 ~ 1.2% slower

I can't figure out though how to squeeze back the perf on the big regex.

#21 - 05/11/2021 11:25 AM - mame (Yusuke Endoh)
Interesting. I've tested your patch, but it is not so small on my machine.

'10000000.times { /(abc)+/ =~ "abcabcabc" }
Before (min over 10 runs): 2.037 after 1.962 ~ 3.8% slower

It may depend on the performance of branch prediction of CPU.

#22 - 05/11/2021 12:23 PM - nobu (Nobuyoshi Nakada)
I made a patch for Regexp#backtrack_limit=.
This seems no significant performance difference.

https://github.com/ruby/ruby/compare/master...nobu:Regexp.backtrack_limit?expand=1

I don't think timeout per regexp is a good idea.
To avoid DoS on server side, timeout should be per-session, I think.

#23 - 05/11/2021 10:22 PM - sam.saffron (Sam Saffron)
nobu (Nobuyoshi Nakada) I follow but unfortunately there are many ways in which thread.raise can corrupt internal state.
See: https://github.com/mperham/sidekiq/issues/852

Discussion goes back to 2008 on this one: http://blog.headius.com/2008/02/ruby-threadraise-threadkill-timeoutrb.html
Not providing any mechanism for safe timeouts on certain operations means you would need to drain+recycle entire processes on timeout for multi-threaded services (like the very popular Sidekiq and Puma).

Databases provide timeouts on operations as do many other tools. They allow for partial mitigation. Regex is a very common oversight.
Just in the last few weeks we had a whole Rails security issue on this exact problem + 3 items on Discourse. It is incredibly common.

Thanks so much for the backtrack_limit patch, I will test it out. Honestly most apps would do just fine with a 30 second timeout which may end up simply being setting backtrack limit to an outrageously high number.

#24 - 05/11/2021 10:23 PM - sam.saffron (Sam Saffron)
An alternative may be something like:
Thread.safe_raise which allows for raising in places we consider "safe" like mid-regex. Not sure...

#25 - 05/12/2021 03:13 PM - schneems (Richard Schneeman)
I commented on https://bugs.ruby-lang.org/issues/17849 which I would LOVE to see some movement on. I support being able to have high-level
"safe" timeouts. I also support a separate effort to improve this pathological regex DoS problem though I don't have specific opinions on low-level
details on implementation yet.

#26 - 05/12/2021 10:18 PM - Dan0042 (Daniel DeLorme)
nobu (Nobuyoshi Nakada) wrote in #note-22:

I made a patch for Regexp#backtrack_limit=.
This seems no significant performance difference.

This is really perfect isn't it? It's much better than a wall clock timeout. If you have 2 active threads then a 1s timeout really means 0.5s of execution
time; this approach is closer to a cpu time based timeout, and there's no syscall overhead.

#27 - 05/21/2021 06:00 PM - mame (Yusuke Endoh)
We discussed this issue in today's dev-meeting. We agreed that, if we can find a good enough threshold, Regexp#backtrack_limit= is better than
Regexp#timeout=. For example,

- the threshold should stop almost all practical Regexps that run in about one minute, and
- the threshold should NOT stop almost all practical Regexps that ends at most in a few seconds.

Of course, it depends on the input and regexps, so we need to evaluate this in practical settings. We will wait for sam.saffron (Sam Saffron)'s
experiment.

#28 - 07/14/2021 11:59 AM - duerst (Martin Dürst)
nobu (Nobuyoshi Nakada) wrote in #note-22:

I made a patch for Regexp#backtrack_limit=.
This seems no significant performance difference.

https://github.com/ruby/ruby/compare/master...nobu:Regexp.backtrack_limit?expand=1

I have looked at this patch. I think this is the general direction to go. I also think that the interface/API looks good, maybe having a keyword argument
on Regexp.new, too, would be a good addition.

I installed the patch and ran some very experiments. I started with a very slow Regexp that I use to show my students. It can be made of any length n,
but gets really slow when n grows, to the order of O(2**n). I realized that it actually may be some kind of worst case, because it's not really doing
much except for backtracking. That means that the overhead of counting the backtracks will show very clearly. Any more 'average' example should
slow down quite a bit less.

So here is the program I used:

```ruby
HAS_BACKTRACK_LIMIT = Regexp.respond_to? :backtrack_limit

def slow_find (n)
  s = 'a' * n
  r = Regexp.compile('a?* n + s)
  m = nil
  t1 = Time.now
  10.times { m = s.match(r) }
  t2 = Time.now
  print "n: #\{n\}, time: \{t2-t1\}/10"
  print ", backtrack_count: \{m.backtrack_count\}" if HAS_BACKTRACK_LIMIT
  puts (25..29).each do |i|
    slow_find i
  end
end
```

You can easily adjust this by changing the part (25..29) (the range of s n's used) and the two instances of '10' (the number of times a match is run in a
row).

Here are the results for the patch:

duerst@Kloentalersee:/nobuRuby$ ./try_backtrack_limit.rb
n: 25, time: 2.8453695/10, backtrack_count: 33554431
n: 26, time: 5.6392941/10, backtrack_count: 67108863
n: 27, time: 11.3532755/10, backtrack_count: 134217727
n: 28, time: 24.1388335/10, backtrack_count: 268435455
n: 29, time: 49.084651/10, backtrack_count: 536870911
duerst@Kloentalersee:/nobuRuby$ ./try_backtrack_limit.rb
And here are the results without the patch:

duerst@Kloentalersee:~/ruby3$ ./ruby ../nobuRuby/try_backtrack_limit.rb
n: 25, time: 2.6384167/10
n: 26, time: 5.2395088/10
n: 27, time: 10.5159888/10
n: 28, time: 22.5549276/10
n: 29, time: 45.600226/10

duerst@Kloentalersee:~/ruby3$ ./ruby ../nobuRuby/try_backtrack_limit.rb
n: 25, time: 2.5845849/10
n: 26, time: 5.2094378/10
n: 27, time: 10.2203586/10
n: 28, time: 22.115786/10
n: 29, time: 45.0094087/10

duerst@Kloentalersee:~/ruby3$ ./ruby ../nobuRuby/try_backtrack_limit.rb
n: 25, time: 2.5993792/10
n: 26, time: 5.2897985/10
n: 27, time: 11.2203586/10
n: 28, time: 23.115786/10
n: 29, time: 45.0094087/10

These results were obtained on a WSL2/Ubuntu install on Windows 10. All other user programs were switched off, which on Windows doesn't mean there's nothing else running, of course. It should be clear from the above results that the difference is around 5%, maybe a bit higher, but not 10%.

As I already said, that's for what I think is pretty much the worst case. All this Regexp does in backtrack in a binary tree of depth n, testing out all the combinations of choosing 'a' or not 'a' in the first half of the Regexp (which is just "a?a?a?a?...."). Every time it looks for an 'a' in that part, it finds one. But then (except for the very very last try) it cannot match the second part of the Regexp (just n "a"s) to the rest of the string (which is also just n "a"s). For that, it actually doesn't need time, because this part is optimized with a Boyer-Moore algorithm, which means it just checks that the last "a" in the Regexp is beyond the actual string and so there's no match. This can be seen from the debug output when compiling Ruby with

#define ONIG_DEBUG_PARSE_TREE
#define ONIG_DEBUG_COMPILE

in regint.h, which results in the following:

`. /ruby -e 'Regexp.new("a?a?a?aaa")'
  `RubyGems' were not loaded.
  `error_highlight' was not loaded.
  `did_you_mean' was not loaded.

  <list:55e37d443dc0>  
    <quantifier:55e37d443d80>{0,1}
    <string:55e37d443d40>a
    <quantifier:55e37d443e40>{0,1}
    <string:55e37d443e00>a
    <quantifier:55e37d445030>{0,1}
    <string:55e37d444ff0>a
    <string:55e37d4450b0>aaa
  optimize: EXACT_BM
  anchor: []
  sub anchor: []

each: [aaa]: length: 3
code length: 26

So this should give everybody some indication of the worst slowdown with this new backtrack_limit feature. Results for some more "average" scenarios would also help.

#29 - 09/02/2021 07:06 AM - duerst (Martin Dürst)
So if we have 536870911 backtracks per 48.6898231/10 seconds, that comes out to roughly 110M backtracks per second. How about fixing a safe limit of 60s -> 6600M backtracks? Since it only stops the most pathological regexp after 60s, that means it will definitely NOT stop all practical Regexps that ends at most in a few seconds. If this was incorporated in ruby 3.1 it would allow testing in real-world applications in order to find a lower threshold that should stop almost all practical Regexps that run in about one minute.

In summary, there are two proposals, Regexp.timeout= and Regexp.backtrack_limit=, which have a trade-off.

- Regexp.timeout= is easy to use in practical applications, but makes the regexp matching slow (especially in simple regexp cases)
- Regexp.backtrack_limit= introduces little runtime overhead, but is difficult to decide its good defacto limit.

ko1 (Koichi Sasada) suggested mixing the two ideas; enabling the time limit after 10,000 backtracks occurred. This will introduce no overhead for many simple regexps, and provide easy-to-use time-based API.

I will give it a try to implement and experiment it later. (Or contribution is welcome.)

In short, #{untrusted_input} =~ something will be vulnerabile against DoS even after this API is used.

What if the time between two backtracks is much larger for some Regexp, isn't that possible with many characters being matched and then at the end a possible backtrack? (e.g., something like /(a{100000}|b{100000})*/) If so, it sounds like 10000 backtracks could be either microseconds or seconds, i.e., not necessarily related to time, and the approach would not work for some Regexps which backtrack.

IMHO a better solution to this is https://youtu.be/DYPckR7Nax?t=1231 / https://eregon.me/blog/assets/research/just-in-time-compiling-ruby-regexps-on-truffleruby.pdf slide 18 (which is what TruffleRuby does). i.e., use a automaton-based regexp engine (which always matches in linear time) and warn for regexps which can't be run by it (called "slow regexps"). Those slow regexps should then be reviewed and ideally rewritten so they can be matched by the automaton-based regexp engine. They could also have a timeout if needed, with much less impact than on all regexps.

In my understanding, this feature is just a workaround to prevent Regexp DoS pracitcally. We can craft a regexp skirting this feature, but it would have worked well against regexps that I have ever seen as DoS issues.

I agree that using automaton-based regexp engine is a smarter solution, but it requires changes in Ruby code. On the other hand, this feature is never a best solution, but will mitigate many Regexp DoS issues with minimum incompatibility.

I hope we can introduce this feature as a short-term DoS mitigation in near future (if possible, Ruby 3.1).
What if the time between two backtracks is much larger for some Regexp, isn't that possible with many characters being matched and then at the end a possible backtrack? (e.g., something like /\(a{100000}\)|\(b{100000}\))/)
If so, it sounds like 10000 backtracks could be either microseconds or seconds, i.e., not necessarily related to time, and the approach would not work for some Regexps which backtrack.

I don't think we need to worry that much about a regexp custom-made to be slow. ReDoS is about custom-made strings that trigger backtracking in very plain, regular-looking regexps. In CVE-2021-22880, a regexp as simple as /^(\^-?\D+\[\d,\]+\.\d{2}\)/ was the source of the trouble. I think it's ok to think of ReDoS protection in terms of such real-life regexps like that one, and not the realm of all possible weird regexps. And I think these real-life regexps will have a predictable relationship between number of backtracks and time. (edit: mame (Yusuke Endoh) beat me to it...)

IMHO a better solution to this is use a automaton-based regexp engine (which always matches in linear time)

It may indeed be "better", but when will it be available? Regexp.backtrack_limit= is available right now, which makes it "better" by default, IMHO.

The Regexp.backtrack_limit= approach is

- simple
- deterministic
- almost no overhead
- available now

Regexp.timeout= sounds "easy to use in practical applications" but it's also a bit arbitrary. What timeout to use? 5 seconds? Why 5? In reality we should measure how long regexps take to execute and then fix a limit based on the largest valid measured value. And at that point there's no reason why time is easier to measure than backtracks.