Fuzzing Regex Engines

LUCAS DU, University of California, Davis, USA

Abstract

Regular expressions are a commonly used programming tool, particularly in security-critical domains (i.e. in firewalls, in user input sanitizers, and for heuristic malware detection). As such, the correctness of regular expression construction and compilation has widespread implications, both on software security and on general programmer productivity.

This project focuses on the correctness of regular expression compilation by investigating the use of *grammar-aware random generation* for randomized differential testing of real-world regex engines, specifically engines that have RE2-based syntax and semantics. We provide an OCaml implementation of a grammar-aware random testing (i.e. fuzzing) framework for regex engines, which includes: a random regex generator, which produces syntactically valid RE2-style regexes, a random input string generator, and a testing harness for automatically running random tests on different engines and comparing outputs. We also evaluate our implementation on two real-world RE2-based engines—Rust's regex crate and Go's regexp package—and discuss limitations and future work.

1 Introduction

Regular expressions are everywhere. Nearly every mainstream language has a built-in regular expression library and many programmers reach for regular expressions as a general string-processing tool. They are often used in security-sensitive domains such as validation of arbitrary user input, so errors in the construction of regular expressions have major security implications. However, in pursuit of more expressiveness and more performance, modern regex engines include many features that go beyond the studied theory of regular expressions (and so can recognize more expressive languages than regular languages) and build in various optimizations to improve the speed of compiled regexes. This creates a situation that is ripe for implementation error.

Many approaches to improving regular expression correctness have focused on analyzing the regular expressions themselves, either by helping a programmer avoid carelessly crafted regexes that accept unexpected inputs[1], by attempting to synthesize regexes by example [2], or by developing techniques to catch or mitigate ReDoS attacks, which exploit the possibly the unbounded time complexity of modern (and more expressive) regexes on certain inputs (as a result of backtracking features like lookarounds and backreferences that go beyond traditional regular expressions) to execute a denial-of-service attack [7, 9]. There is comparatively little work on the correctness of the underlying regular expression engine, which compiles a syntactic regular expression into an actually executable program.

The purpose of this project is to take a closer look at the correctness of widely-used regular expression engines in the real world by using randomized differential testing on black-box targets. Specifically, we target two engines based on Google's RE2 regex engine: Rust's regex crate and Go's regexp library. We focus particularly on inconsistent output between these engines (i.e. one engine finds a match, while another does not), which indicates a very serious error. That said, we are also interested in other kinds of errors, such as crashes or results that are inconsistent with documented guarantees.

[1] https://github.com/0xacb/recollapse

March 14, 2025.
Author's Contact Information: Lucas Du, lzdu@ucdavis.edu, University of California, Davis, USA.

2 Motivating Example

Say we have a regex defined as follows: `^ *v [0-9]`. We should expect this to match the string `v 10`. In this case, it would be very bad if a regex engine produced a program that didn't return a match!

While it may seem inconceivable that this could happen, errors like this *do* occur in the real world. Regex engines are performing translation while incorporating various optimizations—human error is usually inevitable in such situations. In fact, an error very similar to the example above was found in Rust's regex crate[2].

3 Technical Approach

3.1 Grammar-Aware Random Generation

The primary strategy is to randomly generate **only** syntactically valid regular expressions through a grammar-based approach.

Regular expressions seem like a good target for grammar-aware generation: like standard compilers or interpreters (the classic domain of grammar-aware input generation), regular expression engines have to parse structured syntax that has defined semantics and then translate it down to a lower executable level of representation. The core language of regular expressions is also well-defined and supported by all production regex engines (although in practice, there are some significant differences in the semantics and syntax due to various language extensions), which aids differential testing.

In our case, we define a grammar for a common subset of regex syntax between RE2-based engines, assign a weight to each grammatical production, and then probabilistically pick productions to use next based on their weights. This follows the approach in Csmith and other grammar-based random generators [6, 13].

We also focus on a core subset that includes only ASCII characters—Unicode support in regex engines is inconsistent and testing that support is left for future work. We also avoid features that have no known non-backtracking implementation, i.e. lookarounds and backreferences, in accordance with our engines under test: RE2 and related variants have the explicit goal of avoiding such features in order to get predictable performance and guarantee a certain class of safety (specifically, no unbounded backtracking means guaranteed linear-time performance and immunity to ReDoS attacks).

The OCaml definition of our grammar looks like this:

```
type charset =
  | None
  | Chars of string
  | Any
  | Digit
  | AnyLetter
  | CapLetter
  | LowLetter

type regex =
  | Empty
  | CharSet of charset
  | Not of charset
```

[2]https://github.com/rust-lang/regex/issues/1070

```
| And of charset * charset
| StartsWith of regex
| EndsWith of regex
| Concat of regex * regex
| Or of regex * regex
| Optional of regex
| KleeneStar of regex
| Repeat of regex * int
| RepeatAtLeast of regex * int
| RepeatRange of regex * int * int
```

This includes character sets (the charset type), operations on those character sets (Not and And), compositions on regexes (Concat and Or), and repetitions of regexes (Optional, KleeneStar, Repeat, RepeatAtLeast, RepeatRange). This grammar follows the one defined in prior work [2], with some modifications to match the supported syntax of our engines under test and the specific implementation of our random generator.

One difficulty with random generation of a common regex for different engines lies in the fact that different engines often have subtle differences in syntax and semantics, i.e. greedy vs. lazy matching. We avoid much of this difficulty by focusing on RE2 variants (Rust's regex and Go's regexp are both RE2 variants), which gives us a degree of consistency in both syntax and semantics.

We recursively generate regexes from this grammar, in accordance with a set of chosen weights for the grammar's productions. Specifically, we have a function that takes all of the productions, along with their weights, and randomly picks a production using these weights. At each step of our random regex generator, we call this function to pick our next production. We then recursively call the random regex generator to generate the subexpressions of the chosen production.

As an example, say that we randomly pick Concat. Concat takes two regexes as subexpressions, so our generator will then recursively call itself twice to produce these subexpressions. For productions with an integer argument, such as Repeat, our generator will randomly pick an integer within some range.

As a minor implementation consideration, we have a separate random generator for character sets that we use for productions with a subexpression of `charset` type. Note that this random generator is *not recursive*, since character sets are terminal in our grammar.

3.2 Picking Parameters for Regex Generation

There are several heuristic parameters for our random generator that need to be set: the maximum (recursive) depth of the regex, the probabilistic weights used to choose productions, and the integer ranges used to choose numbers of repetitions (for the Repeat, RepeatAtLeast, and RepeatRange productions).

Our naive approach is to simply choose weights for each production in the grammar by hand and hard-code those weights. This approach is easy to implement, but is entirely based on vague intuitions about whether the generated regexes "look like" the ones that people usually write.

A slightly more sophisticated approach, which is the one taken in "Inputs From Hell" [8], is to learn weights and other parameters based on some real-world corpus of inputs. It would be interesting to investigate the efficacy of this approach by learning parameters from a large public collection of realistic regular expressions[3], but this is left for future work.

[3]https://github.com/lorisdanto/automatark/tree/master/regex

3.3 Random Input Generation

To generate random input strings, our approach is to simply convert a random stream of bytes generated by OCaml's Random module to ASCII characters. The character length of the generated strings can be modified, but currently it is set to 128.

Another interesting idea is to produce random inputs by first generating a set of matching inputs—i.e. using an SMT solver with a theory of regular expressions like Z3—and then mutating those inputs. The hypothesis is that, by generating matching input strings and then mutating those strings for further tests, we would be exercising more states in the underlying finite automaton representation of the regular expression. This approach is left to future work.

3.4 Test Harness

The test harness generates a wrapper program for each regex engine under test: the wrapper pre-compiles a regular expression and takes input strings to match on that regular expression as a command line argument. For each random regular expression we test, this wrapper program is dynamically generated and compiled. The harness will then run a set number of randomly generated input strings through these wrappers, comparing the outputs between engines.

More specifically, our generator will produce some random regular expression E that we want to test. Our harness will then produce and compile a wrapper for each of the regex engines that: A) pre-compiles E and B) accepts an input string as a command line argument that the compiled E will match on.

The harness will then hook up our random input string generator to these wrappers: for each random input string I_s we generate, we will run I_s through each of the wrappers (as a command line argument) and compare the output. Mismatches, along with their corresponding regex wrappers, will be saved for analysis.

We run tests for different regular expression E concurrently on a single core to get more test throughput. We can also run tests in parallel on separate cores to further improve performance, but this part of the implementation is left for future work.

There is a good deal of technical annoyance in translating the symbolic regexes produced by our random generator to concrete syntax accepted by each specific engine. For example, different engines and their host languages handle character escapes and string encodings slightly differently. As a result, some care must be taken to perform a faithful translation from a symbolic regex to a semantically equivalent concrete regexe that is actually executable for each engine.

Go's regexp library is also much less forgiving in its parsing, specifically when it comes to unnecessary square brackets `[]`. This required some cleaning up of our concrete syntax translation to ensure that square brackets were only introduced when absolutely necessary.

4 Related Work

4.1 Existing Fuzzing Frameworks for Regex Engines

There are existing fuzzers for regular expression engines: in particular Google's OSSFuzz actively fuzzes popular engines like RE2, PCRE2, and Rust's regex crate. While OSSFuzz is an interesting project and a public good for open source software, it employs an inefficient and somewhat naive brute force approach to system fuzzing: throwing completely random, unstructured bytes at a program as input and hoping that something will break. This results in waste of computational resources.

The closest work in spirit to our project is described in a series of blog posts on secret.club [4]. Those posts describe work that modifies fuzzer harnesses in OSSFuzz for rust-regex and PCRE2 to try and improve code coverage and bug-finding power. The rust-regex harness improvements

are particularly similar to our approach: they use Rust's `arbitrary` crate to interpret the random bytes that rust-regex's default fuzzing input generator, libFuzzer, produces as grammatically valid regular expressions, ensuring that all random inputs are meaningful. This is a direct parallel to our work on generating syntactically valid inputs from a core grammar, although it approaches the problem in a "top-down" way—re-interpreting randomness by using =arbitrary= as a parser over random bytes—while we take a more "bottom-up" track, generating inputs directly from a grammar using arbitrarily defined weights for each production.

Another key difference is in the assumptions made about the programs under test. The secret.club approach is white-box, incorporating dynamic code coverage information to guide mutations of random input. This coverage-guided mutation follows the spirit of popular fuzzing frameworks like AFL++ and has been very successful at guiding generation of interesting random inputs.

Our project instead views the engines under test as black-boxes, avoiding the need to instrument the engine binaries. This makes it easier to extend our harness to new regex engines, in line with our explicit focus on differential testing of engines to find correctness issues.

Generally, our work is focused on A) differential testing between different engines (not just different versions or options in the same engine) and B) developing a grammar-aware random generator specifically for regular expressions to aid experiments in regex-specific input generation (instead of adopting the "top-down" approach that the secret.club work uses in order to integrate well with the existing OSSFUzz runtime).

4.2 Grammar-Aware Random Generation

4.2.1 Applications of Grammar-Based Fuzzing. There is a large body of work on random generation of inputs for fuzzing and property-based testing. Generally, grammar-aware testing is not new—the problem of meaningless inputs from purely random generation has significant impact on fuzzing efficiency and grammar-aware generation is a simple, but effective way of getting around this problem by generating only valid, meaningful input by-construction [1, 6, 13]. By generating inputs that satisfy the definition for a grammar, random testing frameworks can produce more meaningful random inputs (i.e. inputs that won't be immediately rejected by the program under test) that, for white-box code-coverage guided fuzzers, are also able to explore "deeper" parts of the programs under test. This technique has found particular success in compiler and interpreter fuzzing, since these programs take highly-structured input.

4.2.2 Technical Refinements. There is also a good deal of work refining techniques for grammar-aware or otherwise structure-aware random generation, some of which it would be interesting to explore and possibly incorporate into our project. For example, there are ways to learn weights or distributions for grammar-based random generation from a sample set of real-life, or otherwise common, inputs [8]. This makes it possible to tailor random generation to produce new inputs that are similar to the inputs in the sample set or that are /very dissimilar/ to the inputs in the sample set (by inverting probabilities).

There is also some work on more efficient representation and less biased sampling of random inputs in a grammar-aware setting [10] that involves converting the input grammar to finite-state automaton. The automaton representation, which is referred to as a *grammar automaton*, restructures the production rules in a way A) that eliminates the inherent sampling bias of a context-free grammar approach and B) that allows for faster and more aggressive mutation compared to the standard parse-tree representation of inputs.

Finally, there is work in the property-based testing space on more effective and more ergonomic random generation libraries. Property-based testing libraries like Haskell's QuickCheck [3] and OCaml's qcheck already offer a variety of combinators from which to build random generators,

but there are also libraries that aim to do things in an even more advanced and automated way. Of particular interest is Feat (Functional Enumeration of Algebraic Types), which is a Haskell library that can automatically do random sampling and exhaustive enumeration of algebraic types in an efficient way [5]. In this project, we implement a random generator by hand, specifically for the grammar of regular expressions. It could be interesting to either A) use these libraries to implement our regular expression generator or B) try and take ideas from these random generation libraries to make our current implementation more flexible and more efficient.

5 Experimental Evaluation

5.1 Methodology

We ran the implemented fuzzing framework on two different real-world regex engines—Rust's regex crate and Go's regexp module—for around 20 hours. We generated random, valid regexes and ran 2,048 random input strings through each of them, comparing the outputs between the engines to check for inconsistencies. Around 33,000 regexes were generated and tested in the 20 hour testing run.

Our regex generator had max regex depth set to 5. Weights were hand-set as follows (in OCaml code):

```
let regex_weights =
  [
    (CharSet None             , 4);
    (Not None                 , 2);
    (And (None, None)         , 1);
    (StartsWith Empty         , 2);
    (EndsWith Empty           , 2);
    (Concat (Empty, Empty)    , 3);
    (Or (Empty, Empty)        , 2);
    (Optional Empty           , 2);
    (KleeneStar Empty         , 2);
    (Repeat (Empty, 0)        , 2);
    (RepeatAtLeast (Empty, 0) , 1);
    (RepeatRange (Empty, 0, 0), 1);
  ]

let charset_weights =
  [
    (Chars "" , 15);
    (Any      , 1);
    (Digit    , 3);
    (AnyLetter, 1);
    (CapLetter, 2);
    (LowLetter, 2);
  ]
```

Repetition parameters are set as follows, with example OCaml code below:

- Choose a random number of repetitions for Repeat between 0 and 15
- Choose a random number of minimum repetitions for RepeatAtLeast between 0 and 5
- For RepeatRange, choose a random start for the range between 0 and 5 and a random end for the range between the choice of start and 15

```
...
    | Repeat (_, _)         -> Repeat(gen_regex depth', Random.int 15)
    | RepeatAtLeast (_, _)  -> RepeatAtLeast(gen_regex depth', Random.int 5)
    | RepeatRange (_, _, _) ->
       let start_range = Random.int 5 in
       let end_range = Random.int_in_range ~min:start_range ~max:15 in
       RepeatRange(gen_regex depth', start_range, end_range)
```

If an inconsistency is found, we save the inputs on which the inconsistency occurred, along with all the engine wrappers involved (which includes the regex being tested). We then manually check to see if each inconsistency is really the result of a bug or if there is some documented behavior that is causing it.

We ran these tests on a single core of a 10-core 13th Gen Intel® Core™ i5-13500H CPU, with 4 concurrent threads.

5.2 Results and Discussion

After about 20 wall-clock hours of test runtime, we found 7 regexes that have inconsistent output between Rust's regex crate and Go's regexp library. All 7 were due to errors in the wrapper for Go's regexp library.

6 were due to a documented limitation on maximum repetition counts in regexes accepted by Go's regexp engine. Specifically, Go's regexp engine does not allow repetition counts above 1000. Here is one of the randomly generated regexes that triggered an error due to this limitation: `((((([^a-z]){4,}){0,6}){6}){10}`. Notice that this regex explicitly allows for $4 \times 6 \times 6 \times 10 \geq 1000$ repetitions.

The last inconsistency was likely due to an error in the concurrent test harness. Specifically, the Go regexp wrapper for the regex `((C@){0,})*` produced errors on all inputs during the test run, but subsequent manual testing of the compiled regex on a randomly selected subset of 10 of the 2,048 inputs revealed no error and no inconsistency with the version compiled by the Rust regex wrapper.

5.3 Limitations

There are a number of limitations to our work and our evaluation.

- We only ran our test harness for 20 wall-clock hours, on 4 threads on a single core. There is room here to cover much more of the state space of possible regexes by A) running more tests concurrently and in parallel, B) using a more powerful CPU, and C) simply running the harness for much longer.
- We take the naive approach to both generating random regexes and producing random input. The tight restrictions on the grammar (which does not cover all of the valid syntax for RE2-based engines), the hand-set production weights, the limit of regex depth to 5, and the restriction to printable ASCII characters in the input string all limit the parts of the regex engines we actually test to parts that are likely quite well-understood and already very well-tested.

6 Conclusion and Future Work

We present a functional implementation of a framework for randomized differential testing of RE2-style regex engines. We the evaluate our implementation by running a 20 hour testing campaign against Rust's regex create and Go's regexp library. We ultimately find 7 regexes resulting inconsistent output in that time span, but all 7 are not due to errors in the underlying engines.

6.1 Future Work

There are many interesting directions for future work. In the short-term, fully parallelizing the harness and running it for longer on better hardware is an immediate improvement that will result in faster and better coverage of the regex state space.

We could experiment with improved random regex generation in the following ways:

- Learn weights for grammar productions, as well as regex depth and repetition parameters, from a large, real-world set of regexes in the style of Inputs From Hell [8].
- Take inspiration from the enumeration-based approach of the FEAT libraries for Haskell and OCaml [5] to try and automatically and more flexibly generate random regexes from a grammar, which is an approach that has been investigated recently for SMT solver fuzzing [12]. This offers a more principled and more expressive way to do random grammar-based test generation.
- Cover a larger (hopefully maximal) subset of RE2 syntax that is common between all RE2 variants.
- Look more deeply into regex engine internals to get a better sense of where optimizations often go wrong and try to heuristically target those optimizations.

It would also be interesting to see if we can improve the input string generation strategy:

- In particular, it would be interesting to use an SMT solver with a theory of regular expressions (i.e. Z3 or CVC5) to generate a matching string input for a certain regex [11] and then mutate that guaranteed-valid input for other tests. This would make the input generation process a bit more principled and directed, instead of simply randomly throwing bytes at a compiled regex.
- It would also be possible to add more Unicode characters to the options for string input. Unicode support in regex engines is usually quite limited and often experimental, which likely means that there is low-hanging fruit for finding bugs.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *NDSS*.

[2] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 487–502.

[3] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[4] Addison Crump. 2024. Ring Around The Regex: Lessons learned from fuzzing regex libraries (Part 1). https://secret.club/2024/06/30/ring-around-the-regex-1.html. Accessed: 2025-02-16.

[5] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices* 47, 12 (2012), 61–72.

[6] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.

[7] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. 2022. Regulator: Dynamic Analysis to Detect {ReDoS}. In *31st USENIX Security Symposium (USENIX Security 22)*. 4219–4235.

[8] Esteban Pavese, Ezekiel Soremekun, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2018. Inputs from hell: generating uncommon inputs from common samples. *arXiv preprint arXiv:1812.07525* (2018).

[9] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 225–235.

[10] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256.

[11] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 498–507.

[12] Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2378–2401.

[13] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.