Approach to Benchmarking Regex Engines

PARTH PATIL, JACK BLOWERS, KANIKA SHRIMALI, GRAYSON NOCERA

Purdue University

1. Abstract

INTRODUCTION: This project aims to assess the current state of benchmarking regex engines, focusing on expanding available tools and creating a tool for robust testing and comparison amongst engines. We recognize the importance of regexes in validating user input, parsing data, and ensuring software security.
STATE-OF-THE-ART: We provide an analysis of existing regex benchmarking systems, methods, and applications. This paper also examines existing work in detecting regex-based vulnerabilities, examining real world implications of these vulnerabilities and existing literature in engine and regex optimization.
PROPOSED CONTRIBUTION: The proposed contributions include the following:

- Analysis of current methods for benchmarking regex engines
- Analysis of use cases and importance of proper implementation of regexes
- A tool for comparing regex engines
- Preliminary results for crafted test spaces, exploring behaviors and generalizability of pattern-haystack pairs on given engines.

PROPOSED METHOD: This project will introduce small testing spaces, composed of regexes from various accessible sources and input strings varying in features and length. A Docker-based tool utilizes these test spaces by measuring the time to match for each regex-string pair and producing contour graphs of the result for each engine.

2. Introduction

Regular expression engines are important because of their applications in URL Matching, Input Sanitization, Log Analysis, Web Scraping, Email Validation, Password Strength, Text Cleaning, etc. Improper implementation in a regex engine or regex evaluation that is too slow can lead to several security risks and functional failures, such as Regular Expression Denial of Service (ReDoS), Injection Attacks, and Catastrophic Backtracking. Thus, there is a sufficient driving force to develop efficient regex expressions as well as develop engines that can evaluate these regexes optimally in a context-specific way. However, no existing frameworks exist to evaluate or compare performance across regex engines. We seek to address this research gap by defining a methodology, problem space, and tools for benchmarking regex engines.

However, an element of subjectivity arises when trying to compare different regex engines because of the variety of applications, dependency on regex optimization, and interest in specific optimization criteria. Also, regexes themselves are not a "Lingua Franca," meaning they don't entirely translate across programming languages or developers [1]. Previous work on regexes have mainly focused on optimization, generalizability, and vulnerability detection. Benchmarking engines has been attempted with minor success, but novel proposals to the problem like examining potential computational bounds of engines are still considerably valuable.

We propose a method of regex evaluation that explores the relationship between regexes, a given engine, an input string length, and the resulting time to match. The underlying assumption is that a metric roughly corresponding to the complexity of a regular expression is a function of the expression and the engine. Thus, varying the engine across subsets of consistent regexes and

input strings will allow us to benchmark the performance of the engines. We can extend this further by saying that every possible task a regex could use forms a Cartesian plane, and every task occupies a subspace on this plane. By examining specific tasks or subspaces on this plane, we can also make context-specific statements about engines.

In order to achieve this, we created sets of regex expressions to test against different engines using our metrics. We also created a corpus of text against which the regexes are run. The selection and composition of these test spaces will be described more thoroughly in the Methodology section; in short, they are common and accessible for the given use case, compiled through various public regex sources. The combination of dimensional metrics, appropriate regex problem space, and a sufficient body of text allowed us to formalize an approach to benchmarking regex engines. It also allows potential users to generate graphs to propose potential computational bounds on real systems. Finally, our team created an interactive regex benchmarking tool to create an accessible and scalable benchmarking system that provides the test spaces used, custom testing, and data generation.

Our work shows initial results in the effort to benchmark a large number of engines, and while the created test spaces have limitations in the depth of the conclusions we are able to draw, we believe they provide above-surface-level insights, and the created tool allows for a scalable and tailorable system for individual users. Limitations in the system used for testing, the tradeoffs between the generalizability of regular expressions used, and feature usage by the engine all limit the results of the work; however, the sheer amount of data generated and possible data to be generated indicates a success in the available resources and tools to benchmark regex engines.

3. Motivation

The gap between user knowledge of regex systems and current implementations is present in both real-world examples and theoretical/literature-based vulnerabilities. Our project bridges this gap by giving a scalable tool for testing regexes, and general information about current in-use engines.

3.1. The Dangers of Regexes in Real-World Cases

Improper implementation or limited understanding of regexes and regex engines can lead to tangible damage. For example, a postmortem provided by Cloudflare details an incident wherein a change in the WAF (web application firewall) Rule included a poorly written regex and caused the company's service to drop, losing %80 of their network traffic in a matter of minutes [2].

In another example, the Snyk Reacher team investigated the ReDoS vulnerability in UAParser, a popular Javascript package. The package at the time had roughly 6 million downloads, and a vulnerability was found in a section of code meant for identifying specific browsers. Appending long strings that terminated in exclamation points allowed for an attack to force catastrophic backtracking and could disable the service client side and potentially the server side. This vulnerability was made known to UAParser but shows the presence of regex misimplantation. [3]

Another danger with regexes is their lack of generalizability across systems, use cases, engines, and users. Additionally, developers have vocalized the sentiment that regex complexity causes regexes to be nearly impossible to debug, leaving testing as the only practical means to do so [4].

3.2. The Dangers of Regexes from Literature

The Impact of Regular Expression Denial of Service (ReDoS) in Practice [5] examines ReDoS vulnerabilities in JavaScript and Python, identifying and analyzing super-linear regexes through static and dynamic methods. It exposes widespread ReDoS risks in over 10,000 modules, challenging the adequacy of current anti-pattern detection methods. The research underscores that while using anti-patterns to identify super-linear (SL) regexes is crucial, it alone is insufficient. Regular Expression Matching Can Be Simple And Fast(but is slow in Java, Perl, PHP, Python, Ruby, ...) [6] dives into the difference between the two major types of regex engine algorithms:

backtracking and non-backtracking. It shows the immense performance difference between the two algorithms for certain regexes and haystacks. This article increased our knowledge of the two most-common regex algorithms and alerted us to the pitfalls of the backtracking approach. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers [7] studies how non-backtracking automata-based regex matches are vulnerable to ReDoS attacks even though they are suggested as a method for mitigation of ReDoS. They do this by developing a method of developing inputs that caused the matches to perform poorly. The research of this paper is of particular interest to us because the selection of databases used would be considered important in the context of benchmarking or regex engines, especially one focused in the area of mitigating ReDoS attacks and other security vulnerabilities.

FlashRegex: Deducing Anti-ReDoS Regexes from Examples [8] introduces a method to transform unsafe regexes into safe regexes or synthesize safe regexes. ReDOS is a major issue when it comes to regexes, so a paper that can transform regexes to be anti-ReDOS is very impressive. We used this paper to pass in both unsafe and safe (in terms of ReDOS) regexes into our benchmarking system to see how each engine handles them. Exploring Regular Expression Comprehension [9] investigates the understandability of regexes in software engineering. Through an empirical study with 180 participants, the authors identify code smells and evaluate the comprehensibility of different regex representations. They find that certain representations, such as using ranges instead of default character classes, are more understandable. Testing Regex Generalizability And Its Implications: A Large-Scale Many-Language Measurement Study [10] aims to examine the generalizability of regex practices across different programming languages. It uses eight metrics to analyze regexes, like length, character class count, quantifier count, etc. These metrics were selected to capture various aspects of regex representation, string language diversity, and match complexity across different programming languages. The authors report significant differences in regex behavior/syntax between some languages.

4. Background and Related Work

Below, we introduce the topic of regular expression engines, their pitfalls, and why and how they can be benchmarked.

4.1. Background

Developers and researchers have conducted extensive work to develop regular expression engines that improve performance. Additionally, regular expression engineers often add features or “syntactic sugar" to their engines to aid developers. However, attempts at benchmarking regex engines are fewer and farther between, due to the difficulty of the problem statement. The attempts that have been made are often flawed in some way, such as only considering CPU time as a metric or not benchmarking important features of engines.

4.2. Previous Benchmarking Attempts

Rebar is a biased attempt at benchmarking popular regex engines, written by the author of the Rust regex engine. It was a great starting point for us as we built our system, as it has extensive documentation regarding its methodology, metrics, CLI, etc. However, this project has important drawbacks that we improved upon, such as a limited list of regex features that it tests, bias toward the Rust regex engine, and difficulty of use and installation. [11]

The repository mariomka/regex-benchmark [12] tests a long list of regex engines with three regexes (Email, URI, IPv4) against a large body of text. While not as exhaustive as rebar [11], this project was still helpful to gain a better understanding of how we would implement our benchmarking system. We relied heavily on this repository for its Docker system and environment setup for each engine. The standardization of the environment via a Docker container makes our system easy to setup and use.

We used both of these previous attempts to build a system that comprised the best features of both attempts. For example, we drew upon rebar's [11] benchmarking methodology but relied on regex-benchmark's [12] environment setup to create an easy-to-use tool.

4.3. Browser-Based Regex Testers

Multiple browser-based regex testing systems exist with the goal of allowing users to experiment with custom regexes and inputs in order to check the functionality of a given regex or potentially change the correctness of the implementation. The two most prominent when comparing our use case are Regex101 [13] and RegExr [14]. RegExr only allows for the use of JavaScript and PCRE but contains information about a number of matches in an expression as well as time to match. It also provides tools about specific aspects of the regex such as quantifiers and capture groups, making it a very cohesive tool for single expression testing. Regex101 provides a greater sweep of languages to use and a broader explanation about building regexes, returning time, and number of matches for one regex and string.

These browser-based regex testers show what the current accessible benchmarking tools are and allow us to justify the novelty of our benchmarking tool. They typically operate 1-1 with input strings and tested regexes, with the intention of validating the operation of a single regex, our tool creates larger test spaces with multiple input regexes and strings to measure the engine above all else. Ideally, developers should be able to test a potential regex any many strings to capture all potential performance profiles of the regex.

4.4. Regular Expression Engine Performance

A Search for Improved Performance in Regular Expressions [15] focuses on optimizing regular expressions (regexes) using Genetic Programming (GP) to improve performance while maintaining functionality. It employs metrics like execution time and functional equivalence to measure the effectiveness of the optimization. The study demonstrates that GP can significantly enhance regex performance across various cases. ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures [16] details more of the hardware side of regex matching and automata. This paper improves our understanding of the hardware constraints along with the nature of automata. The paper also describes automata by certain benchmarks. While this would be difficult to do ourselves, one could attempt to draw a correlation between the performance of a regex and the characteristics of its automata. Rex: Symbolic Regular Expression Explorer [17] implements an analyzer of regex constraints. While the complex math did not aid us in creating our benchmark system, it was useful in learning more about finite automata and their role in regex matching.

Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics [18] details the implementation of .NET's new regex engine, which uses a nonbacktracking regex algorithm while still maintaining a rich feature set. This paper gave us more insight into an engine that we benchmarked. Symbolic Regex Matcher [16] details a new engine called SRM. It is built on top of the .NET regex engine but with minor changes and additional features. A Prefiltering Approach to Regular Expression Matching for Network Security Systems [19] discusses a new approach for regex matching in network security applications. Thus, through this paper, we gain insight into the goals of a regex engine built for network security.

5. Methodology

5.1. Problem Statement

Our project addresses a lack of a unified, effective benchmarking system for regex performance, crucial for software security and efficiency. This gap underscores the potential for vulnerabilities

and performance bottlenecks due to the absence of a comprehensive benchmarking framework capable of spanning diverse regex engines and application contexts.

A successful solution, as per our study, would provide a comprehensive, standardized framework that accurately assesses regex performance across various engines, focusing on metrics relevant to real-world applications.

Our project's findings could significantly enhance software engineering by providing a standardized approach to regex benchmarking, leading to more secure and efficient software systems. This would not only mitigate common vulnerabilities but also streamline software development, impacting both the creation and maintenance of applications across various industries.

5.2. Overview of approach

We built a regex benchmarking system that can measure metrics of certain regex engines and display these to the user.

5.2.1. Engine Selection

Our first step was to decide which regex engines to benchmark. In making this decision, we considered the following characteristics of engines.

USER BASE: Engines differ widely in the size of their user base. For example, the regex module built into Python is certainly used by more people globally than other engines we benchmarked, like Crystal's engine. In general, we made an assumption that the user base of the regex engine itself was directly correlated to the user base of its language, provided that the engine is part of the standard library of that language. For example, data about the use of C++'s standard regex module would be difficult to collect, but we can infer from C++'s wide usage that its built-in engine is likely widely used as well.

ALGORITHM: Regex engines are broken up broadly into two categories: backtracking and non-backtracking. While intricacies exist within each of these subcategories (along with engines that do not neatly fit into these categories), we focused on ensuring that these two types of engines were included in the benchmark. Future work could be done to better classify the algorithms that should be benchmarked.

DEVELOPMENT STATUS: For our benchmark, we focused on those engines that are actively maintained. Generally, due to the user bases of the engines that we benchmarked, they are all actively maintained.

GOALS: Engines differ widely in their goals. Regexes used in Network Intrusion Detection Systems (NIDs) focus on maximizing throughput, while others focus on programmability or maintainability. Some emphasize performance, while others aim to have a rich feature set (these two emphases are often at odds in regex engines). We included at least one engine for each of these goals.

Our goal in engine selection was to include engines that varied widely along all four of these axes. The final list of engines used can be seen below.

Engine	Reason for Choosing
Crystal	This engine was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it. Additionally, Crystal recently upgraded its regex engine, motivating us to try out its new engine, which uses PCRE2 under the hood. [20]
C++ STL	The standard regex engine of C++ is used widely in a broad range of applications and systems.
C++ Boost	This engine was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
C++ SRELL	This engine was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it. Because the APIs of the C++ regex engines are interchangeable, adding benchmarks for them was simple and quick.
Grep	This engine is very widely used as a utility for searching through text. This utility specializes in return lines in the input text that match a given regex. Thus, we would like to test it in this context and other contexts to observe its performance compared to the other engines.
Re2	This engine is very influential in the regex engine space. It is a non-backtracking engine written in C++ that also includes a rich feature set.
Rust	Rust is a language that is quickly gaining popularity in many different spaces. Its regex engine uses a non-backtracking approach and lacks a rich feature set like re2. While extensive work has been done to benchmark it, these efforts have come from individuals within the organization itself, so we wanted to provide an unbiased benchmark of the engine. [11] [21]
Hyperscan	This regex engine is often used in Deep Packet Inspection (DPI) for network security use cases. Thus, the goals of this engine are generally to maximize throughput with often stringent time and space constraints. Due to the unique goals of this engine, we chose to include it in the benchmark.
Ruby	Due to Ruby's popularity as a scripting langauge, we decided to include it in this list.
Perl	Perl, though dated, still exists widely in production codebases. Furthermore, it's built-in regex engine is at the core of what the language does well. Its engine employs a nonderministic finite state-machine (backtracking) approach.
PHP	PHP is used widely in production codebases, especially for web applications. Thus, its engine has a large enough userbase to be included in this list.
Python 2	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
Python 3	Python's built-in re module is incredibly popular given the language's use in a wide variety of contexts. It uses a backtracking algorithm.

Table 1. Benchmarked Engines

Python PyPy2	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
Python PyPy3	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it. Additionally, the APIs of the Python engines are interchangeable, making it easy to setup all of them at the same time.
Java	Java's regex engine has a very large user base, as the language still exists widely in production codebases in many different areas. Furthermore, Java's regex engine has been used in bioinformatics applications as well. [22]
JavaScript	JavaScript is used widely for web development (both frontend and backend). Its regex engine is used often both client-side and server-side, so we felt that it should be included in this list. It uses a backtracking algorithm.
Nim	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
Nim Regex	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
C#	C# has a broad set of features and superficially relates to the .NET framework. Thus, understanding its performance is important in .NET application usage.
D dmd	D allows for Perl-compatible regex syntax with feature enhancements. It was also included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
D ldc	See reason for including D dmd.
Dart Native	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
Go	Go uses a non-backtracking engine and thus will likely have different performance characteristics of interest to us in this project.
Julia	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
Kotlin	This was included because the regex-benchmark repository had functionality for it already. Thus, it was easy to get tests setup for it.
PCRE2	Perl Compatible Regex Engines (PCRE) provides a set of functions in C for evaluating regular expressions. Other engines, such as Crystal, use it under the hood. Thus, its influence in the regex engine space motivated our choice to include it in our benchmark.

Table 2. Benchmarked Engines (cont.)

Below lists the test spaces with what application they are attempting to represent as well as the number of input regexes and input strings. The backtracking and backreference tasks are comprised of regexes and strings meant to trigger both behaviors and compare preformative in optimal and sub-optimal conditions to get a sense of worse-case time constraints. The e-mail, date-time, and URI test spaces are comprised of common and accessible regexes found from literature, Regexlib, stack overflow [23], and strings representing multiple possible formats and

input lengths for each of the respective categories. These are meant to be very general and possibly overly simple in order to allow results to be generated across all engines and keep output results within a reasonable time bound by the constraints of our system. The backtracking and backreference test cases are meant to examine specific types of behavior, and the e-mail, date-time, and URI cases are meant to examine general behavior with multiple types of real-world use cases.

Test Name	# of Regexs	# of Strings
Backtracking	7	30
Backreference	25	26
Email	31	100
Date and time	23	38
URI	26	36

Table 3. Test spaces

5.3. Procedure

With our given selection of engines, we will run test spaces against each, which will consist of a selection of regexes meant to encapsulate possible and likely uses for a given task such as input validation against strings of varying length and feature, mainly with emphasis on creating linearly scaling inputs to test time to match for a given set of regexes. The analysis will focus on the evaluation time of all regexes on all strings as well as understanding the number of total regexes in the test space that are runnable in the given language. These will be tested through a Docker container that handles the necessary requirements and prepares the engines in order to programmatically handle the testing and result generation. Each engine first compiles the regex, starts a timer, searches the text for all matches of the regex, and finally checks the time elapsed since starting the timer. It does this x times and averages the result, where x is an input to our system. The resulting times will be outputted into a CSV file which will allow for the conversion to a three-dimensional plane in order to examine general trends of regex performance and allow for a more visually comparative way to examine regex engine performance across multiple engines.

6. Results

We have divided our results into two sections. The first one is description of the final state of the tool and a demonstration of use and results while the second examines the data extracted from the tests but in an aggregated form.

6.1. Benchmarking application

The tool created for regex benchmarking allows users to not only test and generate output data on the test spaces we designed but also implement their own regex. This has much of the functionality of websites such as Regex101 and RegExr but boasts a larger list of engines to select from and generates the output result data in the multidimensional analysis format that was used in our own benchmarking analysis. This makes our project scalable and more accessible, combining the best aspects of existing tools expressly used for benchmarking and tools for browser-based regex experimentation.

Below is the process for using the engine and the results. First, a user selects the input regexes

and strings, either by creating their own or using the test spaces we provided. Then, the user selects which engines they want to run these tests on, as seen in Figure 1.

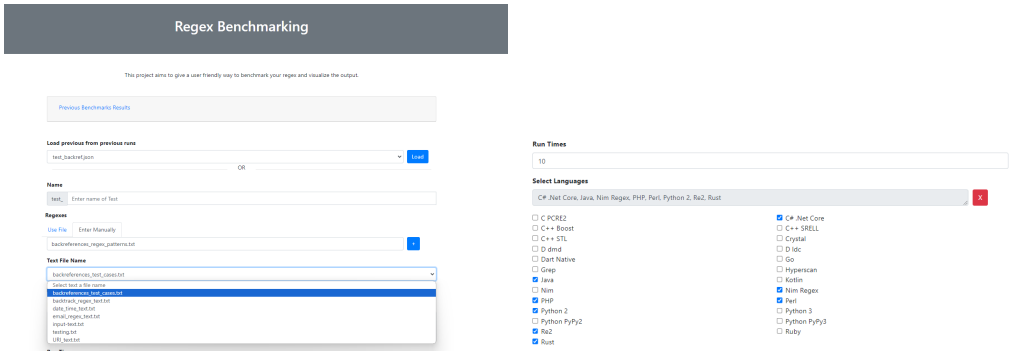


Fig. 1. Regex, regex sets, input strings, and engine selection in our UI

Next the tool compiles and runs the user-selected inputs as seen in Figure 2.

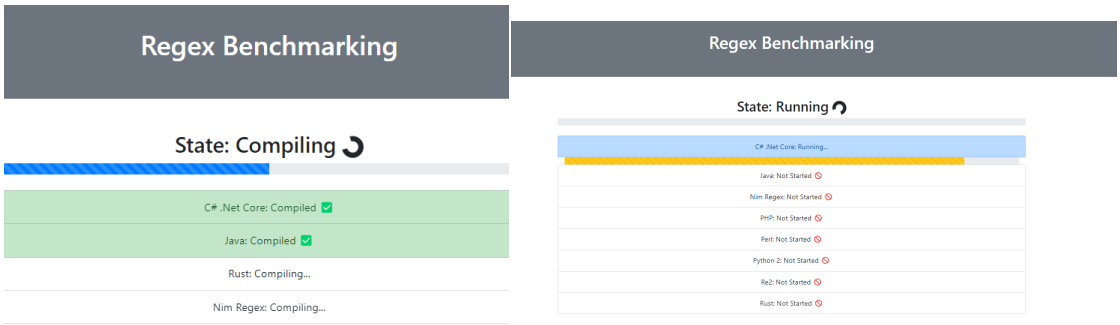


Fig. 2. UI compiling and running

Finally, the user outputs are displayed in the format seen in Figure 3 with results being displayed in a graphic format as well as a CSV format for data analysis of all user-selected engines.

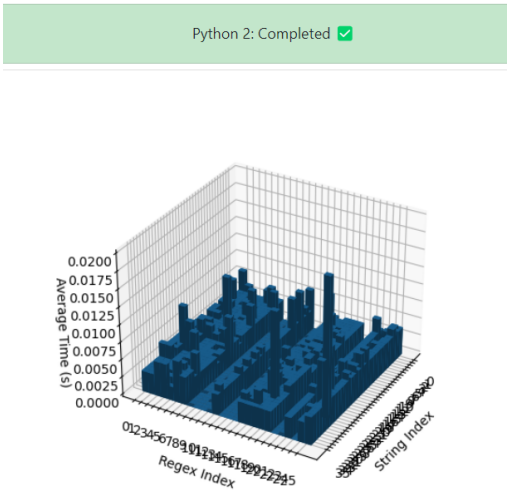


Fig. 3. UI results

The tool also saves runs and configuration for easy of use and to keep track of results across multiple test runs and inputs.

6.2. Engine comparison

URI, Date-Time, and Email The following figures contain the results for aggregated testing of URI, date-time, and e-mail on most of the engines. The aggregated figures make it slightly challenging to state any generalized result. However, they do allow for some general results for individual engines and help in identifying outliers in certain tasks. A portion of the engines measured were not able to run all of the regexes in this test set. In this case, the CSV file records -1. For the purposes of attempting coherent graphing of the data, these values have been zeroed out.

Starting with URI, almost all of the engines are able to perform the task with relatively low time overhead, notably with Rust beginning to experience a heavy linear growth with string length (Figure 4). Interestingly the majority of engines appear to vary more in the dimension of regexes rather than inputs. This is likely due to the fact that the initial regexes in the URI case validate relatively simple strings such as IP addresses, file endings, and file paths.

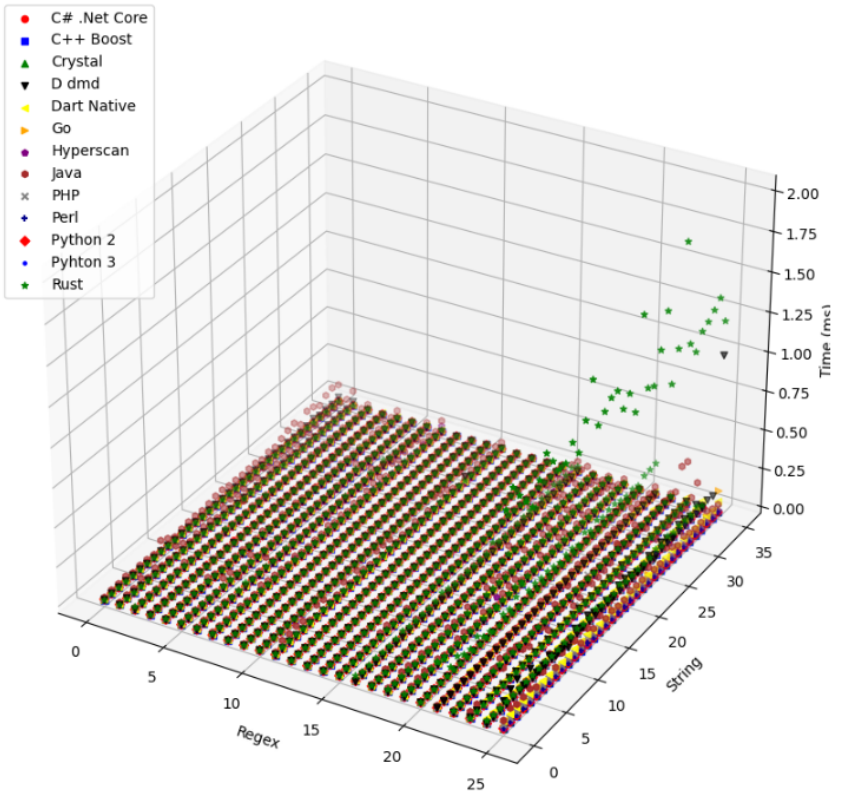


Fig. 4. Aggregated URI results on select engines

The date-time test set appears to have similar results. Notably, grep on average is much slower than any of the other engines shown (Figure 5). It should be noted that grep is unique among the engines benchmarked due to its CLI interface. As such, there was not an easy way to decouple compile time from search time, like there was for the other engines. Thus, the results of grep reflect the time to both compile and search, while the other engines merely reflect the time to search a haystack for all matches. Rust exhibits a time growth as the input string grows in the URI test case. Nim also noticeably exhibits string dependent growth far more than regex dependent growth, which is interesting as it is based on PCRE (Perl Compatible Regular Expressions), and many of the other PCRE engines do not behave exactly this way. Further investigation into this behavior is needed.

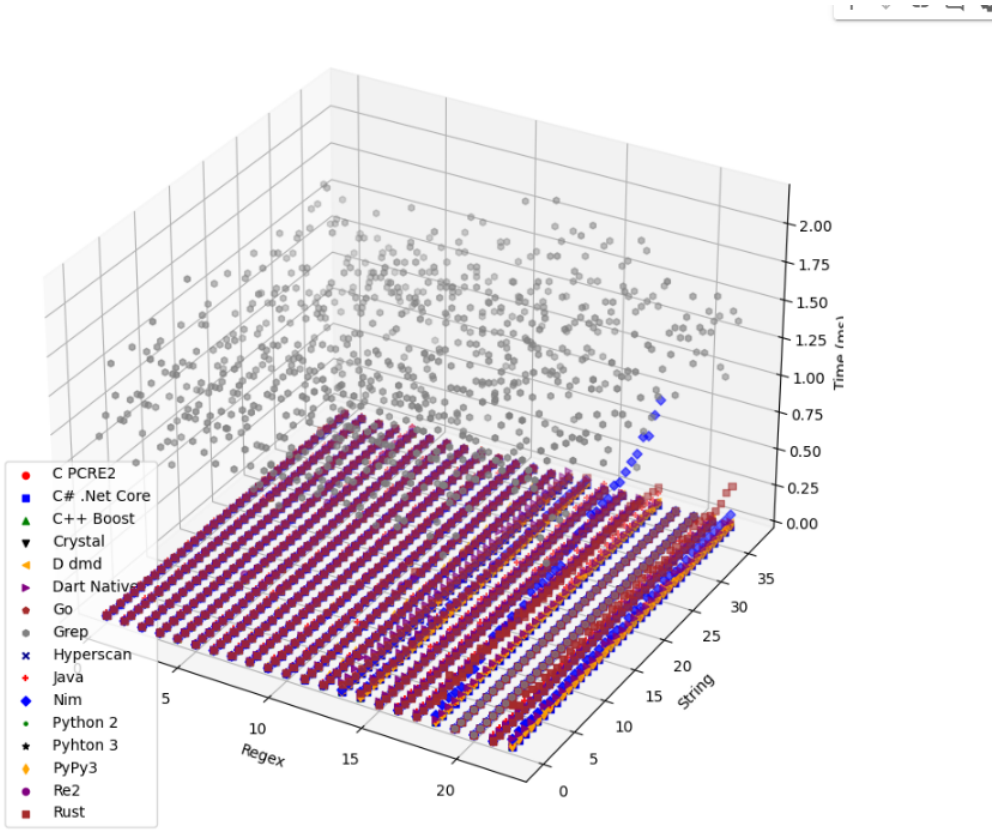


Fig. 5. Aggregated Date-Time results on select engines

The email test results are interesting in that they are slightly harder to parse as the email test space has the largest number of input strings and the longest length of input strings. Some particularly noticeable regex-string pairings causing heavy time spikes for certain engines. Due to catastrophic backtracking and overall stress testing, we had to build a timeout in order to run even single engines. With this in mind, limited results can be gathered due to the computationally intensive process and aggregation of data leading to a heavy number of outliers. Nonetheless, we believe as a test space and tool this is more helpful than some of the other general test spaces provided, as it does take more computation time and does begin to approach edge case testing for our system in terms of input-regex pairings.

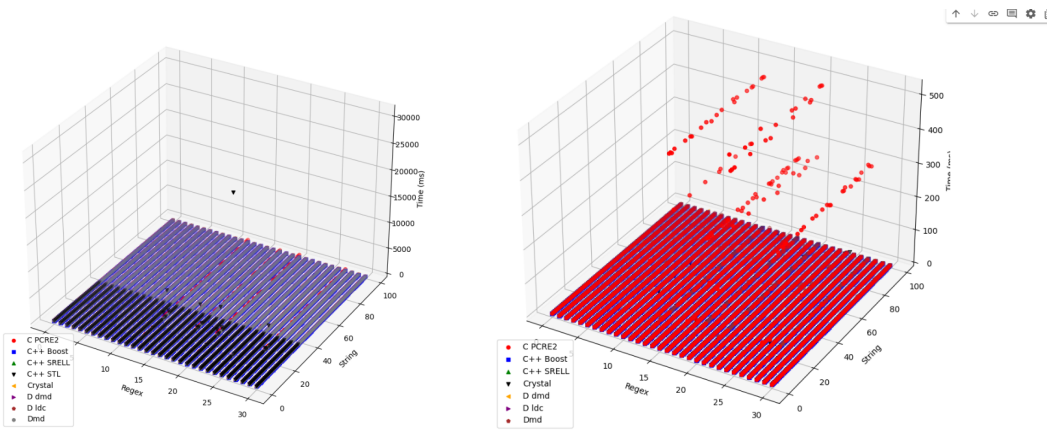


Fig. 6. Aggregated email results on select engines with outliers

The aggregated results with outliers essentially provide a descending hierarchy of time to match per engine but make aggregated results complicated to process. In the limited selection of engines shown, C++ STL > C PCRE2 > Crytal > others by factors of x10 to x1000 (from C++ STL to C PCRE2) (Figure 6, Figure 7).

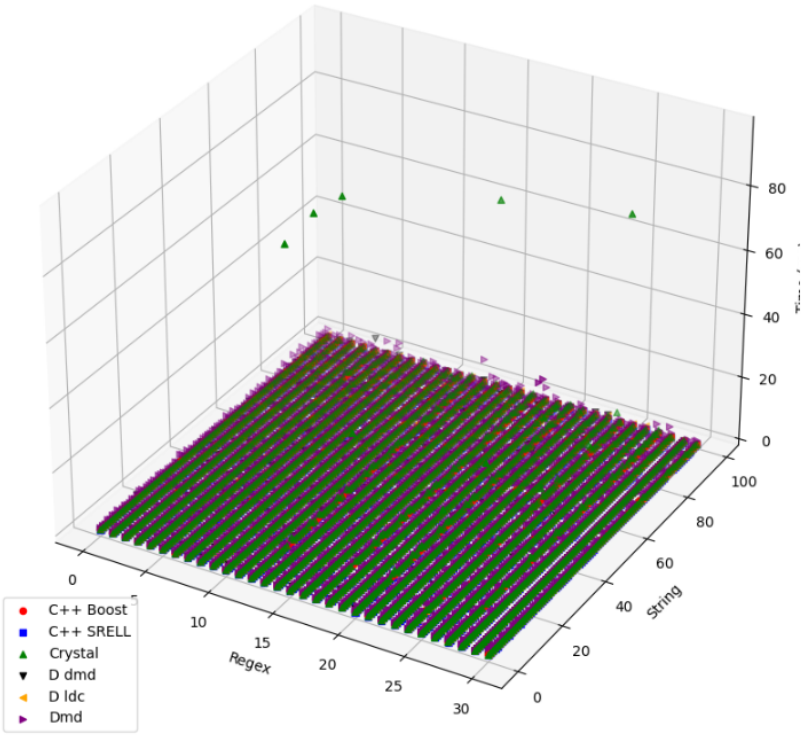


Fig. 7. Aggregated email results on select engines with outliers

Backreference Backreference provided interesting results as it created characteristic graphs among engines that roughly fell into 3 categories. First, there were engines that errored out on

the entire test case, those being Re2, Nim, Hyperscan, and Go. These engines do not support the behavior. Second, engines like Grep, Java, and PHP had higher overall run times and on average experienced a longer time to match. The remaining engines appeared to only have higher run times on certain regex-string pairs in the group which is a common phenomenon seen in these tests. This difference can be seen when examining Figure 8.

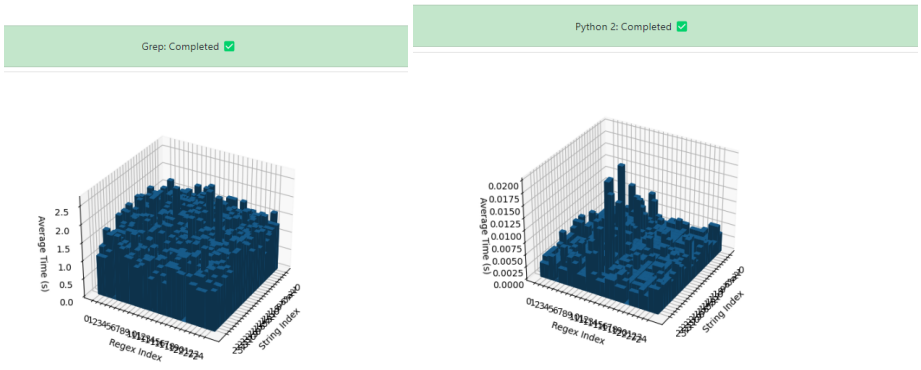


Fig. 8. Backreference on Grep vs Python 2

Backtracking The backtracking tests did not provide particularly interesting results as string length was not appropriately scaled to a size large enough to trigger the exponential behavior in vulnerable engines. Part of the reason this happened is while developing the test cases a balance between length of strings in the input space and time constraints for running the system made extensive backtracking testing slightly less viable. When looking back at the real world ReDos vulnerability testing in the UAParser package we can see that strings on the order of magnitude of 5000 characters are appropriate to trigger catastrophic backtracking. This is a challenging task for us to realize in our current system. The results can be seen in appendix D.

Further results, such as individual engine/task pairings can be found in Appendix D.

7. Validity

The main limitation in our implementation of the regex benchmark system is generating test spaces that can be said to completely encapsulate the space they hope to measure. Our approach was to carefully select regexes that have several expanding levels of complexity in the sense that they contain more capture groups or use more sophisticated features, but this is subjective and may not accurately reflect the actual use case of the test space. This could be solved by the inclusion of a greater set of regexes and a better understanding of engine-specific features. Secondly, the Docker implementation can make times differ across users as opposed to running the system on a machine. However, we saw this as a worthy trade-off in order to streamline the process across engines and inputs and to make the tool easier to use for regex developers. Finally, since several of these measurements are operating in relatively small time scales, the number of times a regex should be run against a string in order to correctly average out a meaningful result is not exact and adds significant time to the execution of the program. Examining outliers also poses a challenge, as test cases may prove of significant importance or may cloud analysis of overall performance. Some engines also save results of evaluation, so implementing a correct time measure proves an extra challenge because averaging across times does not mean the same thing compared to an engine that does not save the time result.

8. Conclusion

The initial goal of the project was to provide an analysis of the current method for benchmarking regex engines, an analysis of use cases and the importance of proper implementation of regexes, and finally, a tool for comparing regex engines and preliminary results from crafted test spaces exploring behaviors and generalizability of expression regex-input on given engines. In this regard, we believe we have succeeded in delivering a system that compares regex engines. Due to the many different implementations, use cases, and levels of documentation for regex engines, creating a complete benchmark for all potential systems proves challenging. However, some general trends can be extrapolated from our results that seem to indicate that almost every engine makes a trade-off in regard to some aspect of its design. For example, backtracking engines can exhibit catastrophic backtracking with a poor regex and a reasonably long string, while nonbacktracking approaches can take significant time to create the automata, given a large regex. The results we were able to draw from the experience are limited in our ability to read them or make strong generalized claims; however, they open an avenue for large amounts of data and testing to be done. While our results mainly show engine-specific behavior in regards to sensitivity to regexes versus input strings, the scalability of the work allows for a distinctly broader result.

The potential implications for individuals in research or industry to test not only individual regexes but sets of regexes and sets of input strings on a given engine are far-reaching. It allows for users to create their own models of systems or use existing data to gain insight into any system using a regular expression and test potential implementation changes in a much more streamlined and scalable context. For our future work, creating more test spaces, developing the current test spaces to be better reflections of the use case, and refining the method for handling invalid regexes in an engine seem to be the most logical next steps. Adding better handling for timeout conditions and polishing the UI also have been discussed, but ultimately, the team is happy with the final state of the tool. The process of developing and expanding the test cases as well as enriching the insights able to be drawn from them will likely be most successfully undertaken by individuals with a deeper understanding of given engines as well as a broader base of in-use regex and regex applications.

References

1. J. C. Davis, L. G. M. IV, C. A. Coghlan, *et al.*, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," CoRR **abs/2105.04397** (2021).
2. J. Graham-Cumming, "details-of-the-cloudflare-outage-on-july-2-2019," Accessed: 4/18/2024.
3. snyk, "Redos protecting your application from malicious regular expressions," Accessed: 4/18/2024.
4. J. Maurer, "Why regular expressions are super powerful, but a terrible coding decision," Accessed: 4/18/2024.
5. 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 *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* (Association for Computing Machinery, New York, NY, USA, 2018), ESEC/FSE 2018, p. 246–256.
6. R. Cox, "Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...)," (2007).
7. L. Turoňová, L. Holík, I. Homoliak, *et al.*, "Counting in regexes considered harmful: Exposing ReDoS vulnerability of nonbacktracking matchers," in *31st USENIX Security Symposium (USENIX Security 22),* (USENIX Association, Boston, MA, 2022), pp. 4165–4182.
8. Y. Li, Z. Xu, J. Cao, *et al.*, "Flashregex: deducing anti-redos regexes from examples," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering,* (Association for Computing Machinery, New York, NY, USA, 2021), ASE '20, p. 659–671.
9. C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE),* (2017), pp. 405–416.
10. J. C. Davis, D. Moyer, A. M. Kazerouni, and D. Lee, "Testing regex generalizability and its implications: A large-scale many-language measurement study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE),* (2019), pp. 427–439.
11. BurntSushi, "rebar," .
12. Mario Juárez, "regex-benchmark," .
13. F. Dib, "Regular expressions 101," Accessed: 4/18/2024.

14. gskinner, "Regexr is an online tool to learn, build, test regular expressions," Accessed: 4/18/2024.
15. B. Cody-Kenny, M. Fenton, A. Ronayne, *et al.*, "A search for improved performance in regular expressions," in *Proceedings of the Genetic and Evolutionary Computation Conference,* (Association for Computing Machinery, New York, NY, USA, 2017), GECCO '17, p. 1280–1287.
16. J. Wadden, V. Dang, N. Brunelle, *et al.*, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *2016 IEEE International Symposium on Workload Characterization (IISWC),* (2016), pp. 1–12.
17. D. Moseley, M. Nishio, J. Perez Rodriguez, *et al.*, "Derivative based nonbacktracking real-world regex matching with backtracking semantics," Proc. ACM Program. Lang. **7** (2023).
18. O. Saarikivi, M. Veanes, T. Wan, and E. Xu, "Symbolic regex matcher," in *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I,* (Springer-Verlag, Berlin, Heidelberg, 2019), p. 372–378.
19. T. Liu, Y. Sun, A. X. Liu, *et al.*, "A prefiltering approach to regular expression matching for network security systems," in *International Conference on Applied Cryptography and Network Security,* (2012).
20. B. Ziliani, "Heads up: Crystal is upgrading its regex engine," (2023).
21. Rust-Leipzig, "Rust-leipzig/regex-performance: Performance comparison of regular expression engines." (2017).
22. F. Marass and C. Upton, "Sequence searcher: A java tool to perform regular expression and fuzzy searches of multiple dna and protein sequences," BMC research notes **2**, 1–3 (2009).
23. M. Partridge, "How can i recognize an evil regex?" Accessed: 4/18/2024.
24. D. Gotterbarn, "Acm code of ethics and professional conduct," (2018).

A. Appendix

Appendix A enumerates and describes the various artifacts associated with the project, which involved the analysis and comparison of regular expression engines. The artifacts include survey protocols, URLs to GitHub repositories, websites detailing engine comparisons, and open-source documentation that was crucial to the project's completion.

Project Documentation and Comparison

- **General Information on Regular Expression Engines**
  - Comprehensive overview of regular expression engines: https://www.regular-expressions.info/engine.html
- **Comparison of Different Regular Expression Engines**
  - Detailed comparison including features and limitations:https://gist.github.com/CMCDragonkai/6c933f4a7d713ef712145c5eb94a1816
- **Issues Related to Regular Expression Engines**
  - Discussion on catastrophic backtracking, a critical issue in regex engines:https://www.regular-expressions.info/catastrophic.html

Code Repositories

- **Benchmarking and Analysis Repositories**
  - Repository for benchmarking different regex engines: https://github.com/mariomka/regex-benchmark
  - Analysis of regular expression handling in the Rust language: https://github.com/BurntSushi/rebar
  - Benchmarking of a small subset of engines by the official Rust GitHub: https://github.com/rust-leipzig/regex-performance
- **Testing and Safety Analysis**
  - Testing suite for safe regular expressions: https://github.com/davisjam/safe-regex/blob/master/src/test/regex.spec.js

Additional Resources

- **Regular Expression Database**
  - Searchable database of regular expressions for various tasks and patterns: https://regexlib.com/Search.aspx?AspxAutoDetectCookieSupport=1

B. Appendix

B.1. Jack Blowers

The main ethical concern I can see with a project like this, comes from my experience working on the project, and that is users of this may not understand the full suite of testing as well as what those tests actually mean . Hypothetically an engine they tested through our tool covers all potential use cases that they may encounter and thus is their implementation will work completely fine for all inputs forever. In order to ethically remedy this or at least provide a warning, I think that our limitations section should be a pop-up, almost like terms and conditions for the use of the tool. This is obviously not a feasible idea, but to some extent, a warning label in the flavor of these results does not guarantee much of anything other than this is what we saw when we ran the experiments, and these are what the experiments look like under the hood.

B.2. Grayson Nocera

Below, I address three sections of the ACM Code of Ethics.
AVOID HARM: Section 1.2 of the ACM Code of Ethics succinctly states "Avoid harm." [24] The writers of the Code even point out that harm can arise from "well-intended actions," such as our choice to build a regex benchmarking system. Ethically, better testing around regexes could benefit a large number of people. Indeed, ReDoS attacks and other regex-related bugs continue to plague developers and, by extension, users. However, my concern is that engineers could begin to use tools like this as a crutch that it is not meant to be. Thus, by creating a tool that theoretically makes testing regex performance easier, we could unintentionally lead to more harm, if someone uses the tool improperly.
PROFESSIONAL COMPETENCE: An additional concern comes from Section 2.2 of the ACM Code of Ethics: "Maintain high standards of professional competence, conduct, and ethical practice." [24] This concern relates with the description of engines used and the conclusions drawn from the data. No one on our team is an expert in regexes or finite automata. In fact, this project involved very little reading of regex engine source code. Thus, my concern is that we did not maintain a high standard of competence in the area of engines when building the benchmarking system. We could be interpreting some of our results incorrectly simply due to lack of knowledge about the guts of the engines we benchmarked. We could also have picked an incomplete set of engines to benchmark that do not cover certain algorithms or use cases. One could argue that only regex developers that are very knowledgeable about the space should be considered competent enough to make a regex benchmarking system.
HONESTY: Finally, both concerns above must be documented extensively in our system to adhere to Section 1.3 of the ACM Code of Ethics: "Be honest and trustworthy." [24] We should detail in the documentation of our system the caveats about the benchmark. For example, we must emphasize that the benchmark should not be used to confirm that a regex is bulletproof. We also should acknowledge our lack of knowledge about finite automata or the inner workings of engines.

Regular expressions are tricky, and developers cannot look to this system or any benchmarking system as a cure for all regex-related ailments. Nonetheless, I believe benchmarking systems such as our own will be very beneficial to developers, especially from an ethical standpoint.

B.3. Kanika Shrimali

The regex benchmarking tool we have created facilitates running comprehensive performance tests on a diverse set of regex engines and test strings, encompassing a range of use cases like date and time, URI, and email processing. By analyzing the time complexities based on string length and regex engine, we aim to provide users with a clear understanding of which regex engine

performs best in different scenarios. For me personally, this objective resonates strongly with the IEEE principles of Public Welfare, Productivity and Quality, and Professional Competence. The broader goal of this project is to empower individuals and companies to make informed decisions when selecting regex engines, thereby helping them avoid production downtimes and prevent DoS attacks, such as the well-known 2018 Cloudflare outage caused by a regex flaw.

B.4. Parth Patil

Regexes are widely used in all kinds of software. Thus, vulnerabilities in regex affect a large domain of things. This project will enable software engineers with little background to design and run benchmarks for various regex engines. The hope is to uncover possible flaws from such testing. Ethically, it is crucial to ensure that these recommendations serve the broader user community by enhancing the reliability and security of software applications. A real-world example is the adoption of safer, more efficient regex libraries in web frameworks following benchmark studies, which helps prevent vulnerabilities like those exposed in the 2013 Rails regex DoS vulnerability, ensuring applications are more secure and performant for end-users.

C. Appendix

Comparing our project's final result to the proposed work, we as a team have delivered more than proposed in some areas and less than hoped in others. The creation of a benchmarking UI was not initially part of the proposed scope of the project, but it has turned out to be a great tool and likely the most inheritable part of the work. The large selection of engines, as well as the accessibility of the tool makes it extremely helpful for undergoing a task like this and allows users to focus more on a test space, that is, a set of regexes and a set of potential strings, than the actual implementation allowing for a quicker experimentation process and better understanding of the system attempting to be implemented. One of the challenges faced in this process was developing the test spaces and making them reasonable encapsulate real-world scenarios without creating so much data that any form of meaningful analysis would be challenging. In this pursuit, the simplicity of the test suite was prioritized in order to validate meaningful progress on the project, and some of the more qualitative metrics, such as the feature richness of an engine, became harder to incorporate in meaningful and significant ways.

C.1. Failure 1 - What tests with what engines?

One of the biggest challenges of this project is balancing the size and complexity of an input space and being able to measure its results and extrapolate meaningful data. One of the failures of this project thusly might be considered to be the scope of work and number of engines selected. Because the benchmarking tool has a large selection of languages to work with and runs tests against, it seemed beneficial for completeness of analysis; however, it creates problems getting the individual instances of engines running as well as creates an issue determining whether a test space should be set up so that only regexes that run in all engines should be used in order to promote fairness and the result, or should we shift our analysis to incorporate these under some form of measuring generalizability of the engine. Looking back, this stems from how our team structured its approach to this problem as having a tool that both examines a large number of engines and examines a large number of engines with any number of test spaces, which themselves have any number of regexes varying in complexity (from the engine's perspective) and trying to have a level of granularity and fairness towards each engine and its respective features. This may have been more achievable by simply breaking the engines into the backtracking and non-backtracking categories and using engines as a specific example instead of treating each engine as a category itself, but we felt that this limited the extent of what a regex benchmark should be.

C.2. Failure 2 - What do we do with all this data?

Secondly, another thing that could be considered a failure of the project, or at least an area where this project could be expanded further, is a more coherent and descriptive approach to the metrics of measuring the output of given engines. This is essentially a dimensionality problem that can be traced back to the project scope, but it also distinctly remains present throughout the project development. Given a test space, a 3-dimensional view allows for more detail in the results being shown, both in the growth of computation time as a function of string length but also allows for potentially hazardous regexes to be identified almost immediately or subtle regex-string pairs that may show an unconsidered use case. To some extent, this could probably be done with an approach as shown in a class by creating a regex language and string language and making sure we were complete under both languages, but the dimensionality of the data makes it tricky to represent and analyze in a meaningful way. Specific cases are easy to identify, and for larger trends, our visual but broad statements about an engine performance are more challenging, and reducing the data to a single metric of total time to pass a test case defeats a little bit of the initial goal set out by the team. The result of this is almost too much data to meaningfully convey,

and analysis on an engine-by-engine, case-by-case basis seems to be more appropriate but also unreasonable in regards to the reader's attention span and conveying the meaningful results of the work. Finding a balance between descriptive metrics of the data was a challenge we as a team identified very early on in the project and have not concluded what the most meaningful solution would be, as we feel it might be better to allow for users to simply have the data and see if it works for their given use case.

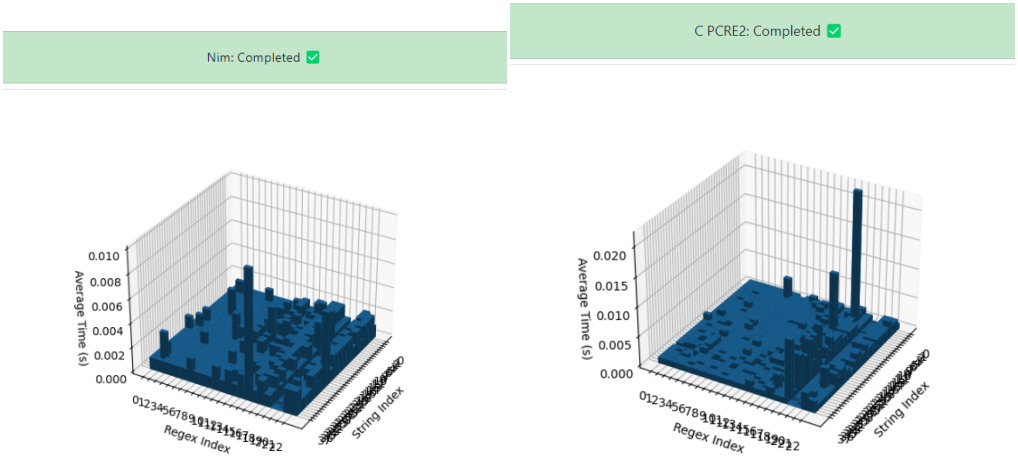
Since both of these problems seem related to how the team scoped the project, there can also be a case made that with more time to experiment with test case development, creating meaningful test cases, and data representation, the end result would be more coherent and allow for more definitive takeaways. Trying to expand on rebar's approach worked as a great starting point but also a massive challenge as probably our biggest source of inspiration. The work done in it is very thorough. Since the team took the approach that having more data is better than less data, we found ourselves in a unique position in regards to meaningfully conveying the data, justifying that this data was sound, and drawing meaningful insights pertinent to a given engine pragmatically. What has now become apparent, and probably should have been from the start, is that no definitive computational bound exists for all given regex for any given engine as it appears to be very challenging to posit in the general case, and empirically working backward also creates several limitations in the extent of what conclusions can be drawn.

There were also several choke points and development in terms of the actual implementation and getting the code base working involving engine-specific problems, regex, and engine interactions, the presence or lack of certain features in regexes, correctly configuring the docker, standardizing of regex inputs or lack thereof, and scenarios in which the test was too time-intensive to gather results. All of these posed problems at some point in working on this project and were handled to the best of our ability with the goal of presenting accurate results of the experiments being run. Those results were descriptive and insightful for someone doing analysis on the engine or comparing engines.

D. Appendix

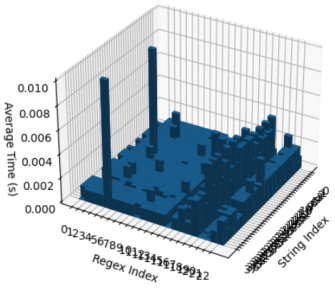
Extra graphs

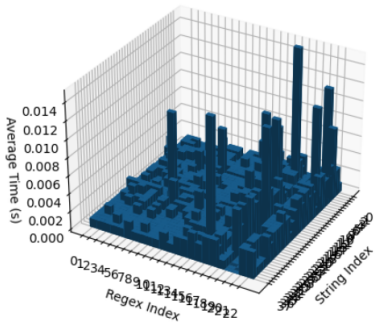
D.1. Date Time on Select Engines



Crystal: Completed ✅

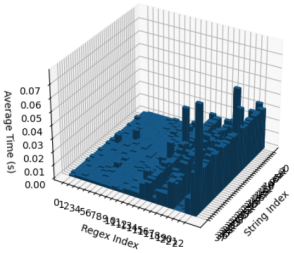
C++ SRELL: Completed ✅

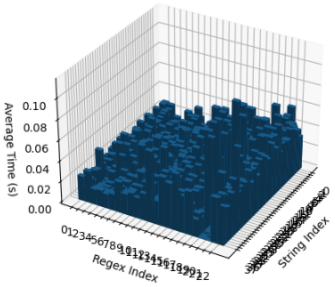




D ldc: Completed ✅

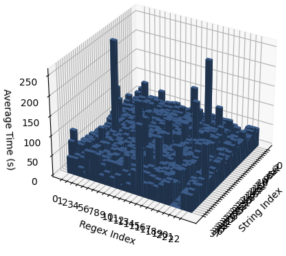
Java: Completed ✅

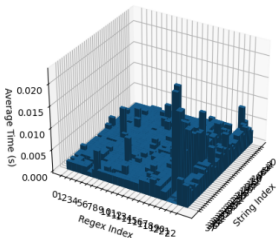




Kotlin: Completed ✅

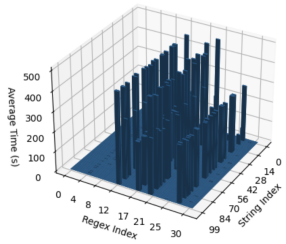
Python 3: Completed ✅



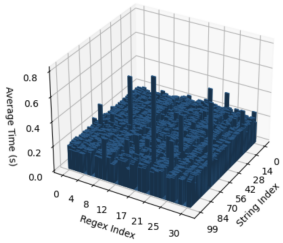


D.2. Email on Select Engines

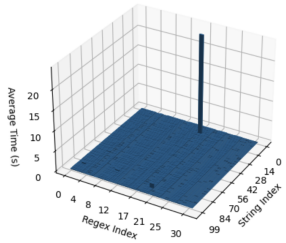
C PCRE2: Completed ✅



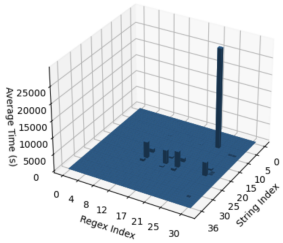
C++ Boost: Completed ✅



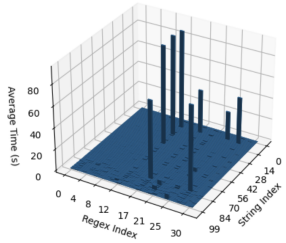
C++ SRELL: Completed ✅



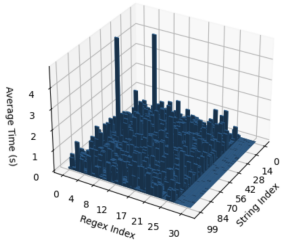
C++ STL: Completed ✅



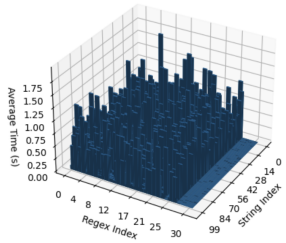
Crystal: Completed ✅



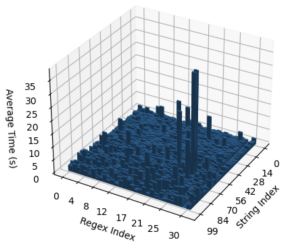
D dmd: Completed ✅



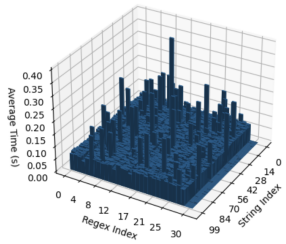
D ldc: Completed ✅



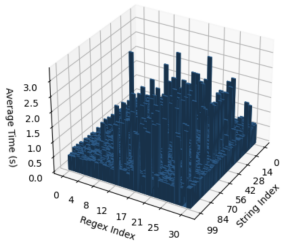
Dart Native: Completed ✅



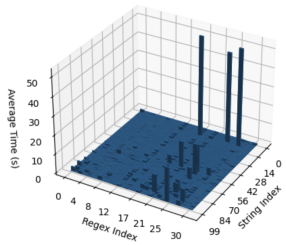
Go: Completed ✅



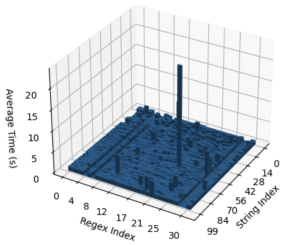
PHP: Completed ✅



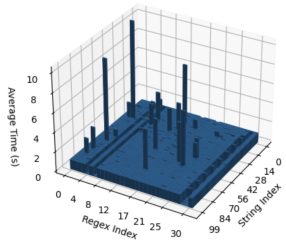
Python 2: Completed ✅



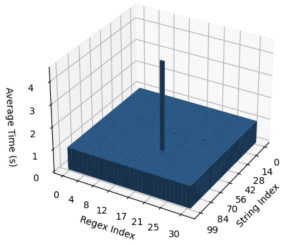
Python 3: Completed ✅



Python PyPy2: Completed ✅

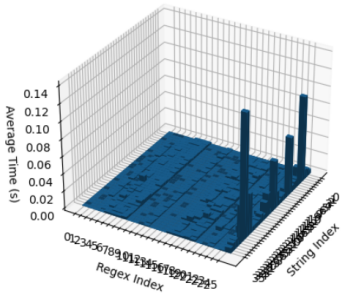


Python PyPy3: Completed ✅

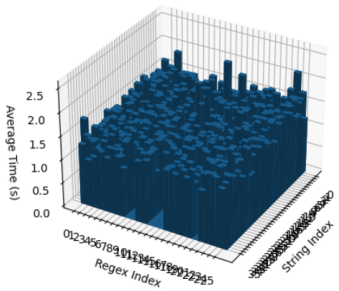


D.3. URI on Select Engines

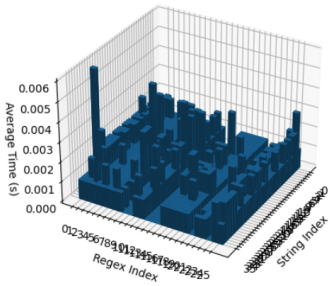
Go: Completed ✅



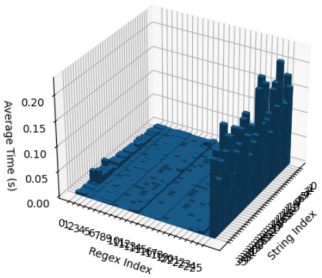
Grep: Completed ✅



C++ Boost: Completed ✅

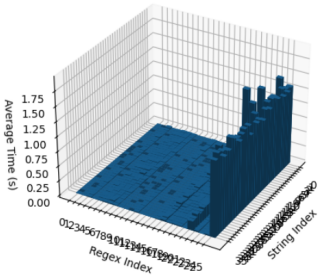


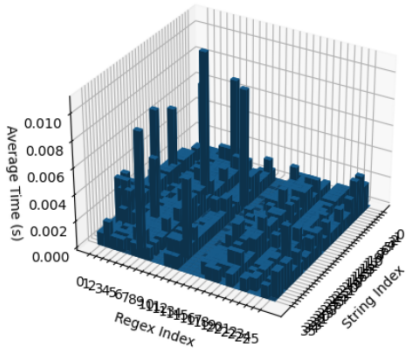
D ldc: Completed ✅



C PCRE2: Completed ✅

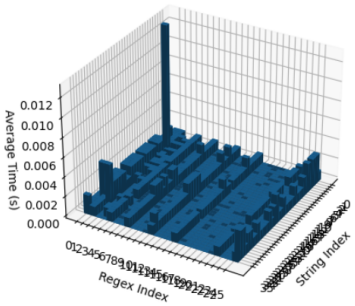
Rust: Completed ✅

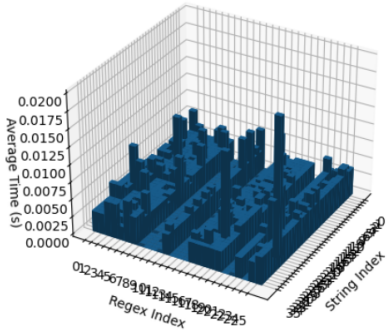




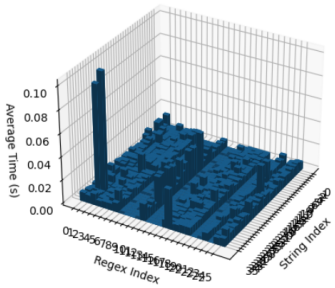
Python 2: Completed ✅

C++ SRELL: Completed ✅



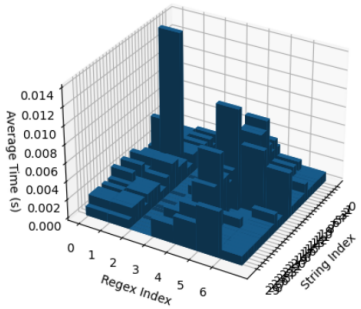


Python PyPy2: Completed ✅

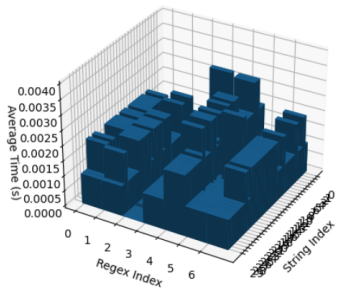


583 ## *D.4. Backtracking on Select Engines*

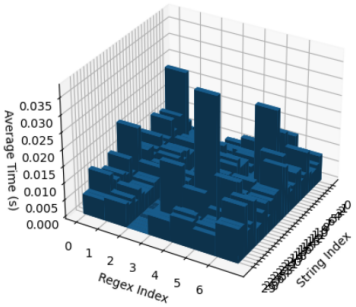
C++ Boost: Completed ✅



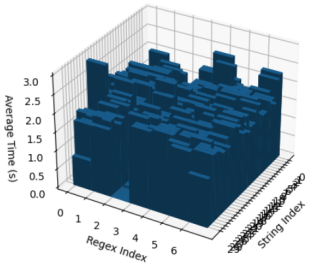
C++ SRELL: Completed ✅



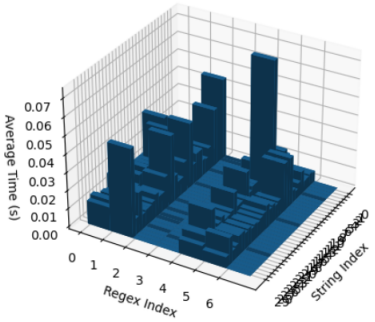
C# .Net Core: Completed ✅



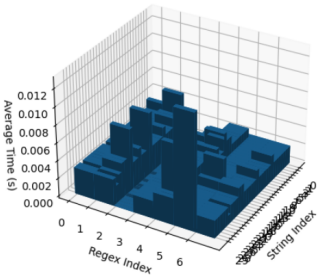
Grep: Completed ✅



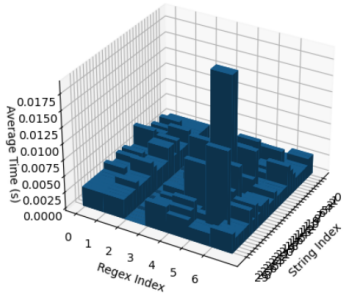
Hyperscan: Completed ✅



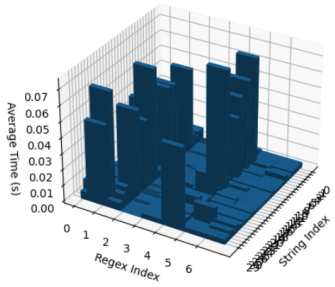
Perl: Completed ✅



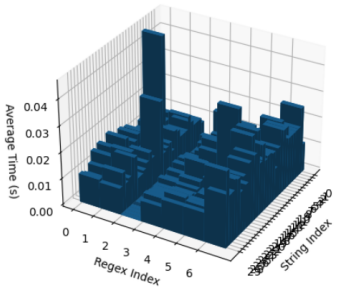
Python 3: Completed



Ruby: Completed

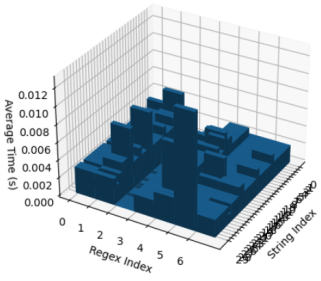


Rust: Completed

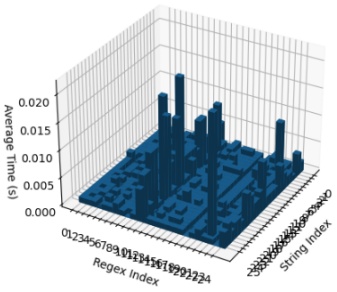


D.5. Back-reference on Select Engines

Perl: Completed

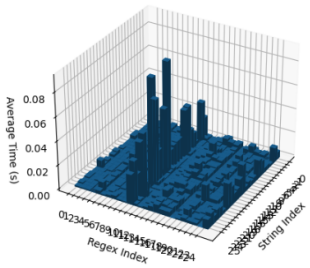


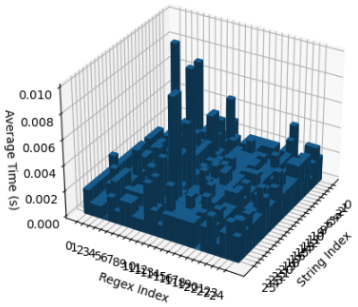
C++ SRELL: Completed



Crystal: Completed

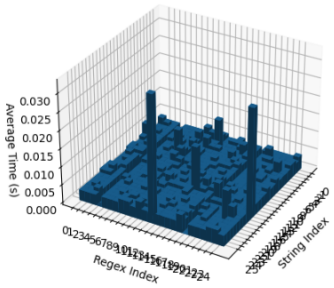
C++ STL: Completed

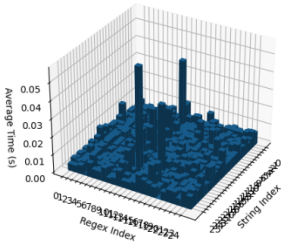




C# .Net Core: Completed

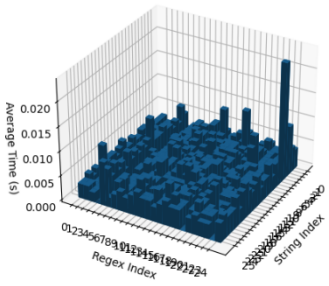
Dart Native: Completed

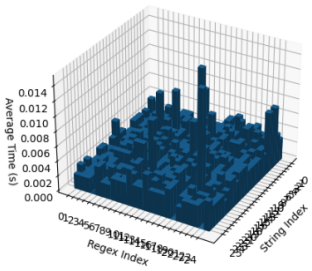




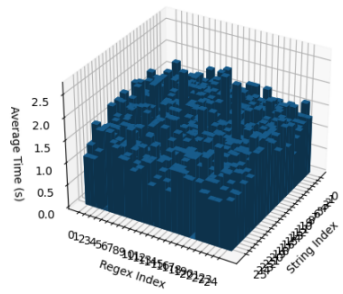
D dmd: Completed

D ldc: Completed

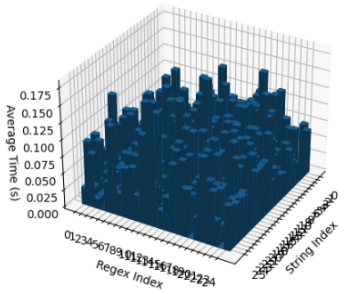




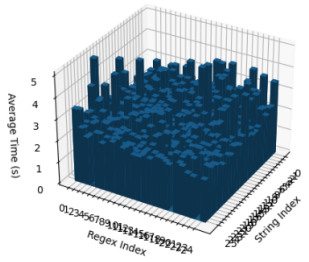
Grep: Completed ✅



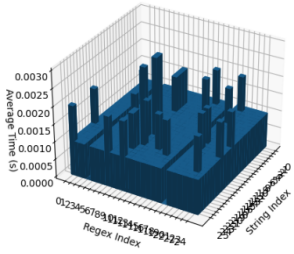
Java: Completed ✅



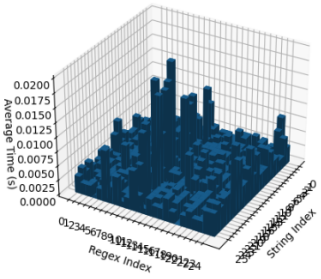
Kotlin: Completed ✅



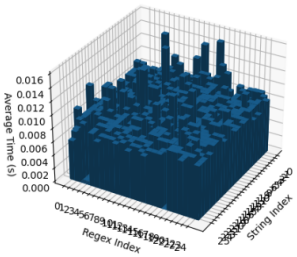
Nim: Completed ✅



Perl: Completed ✅

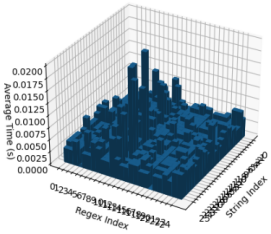


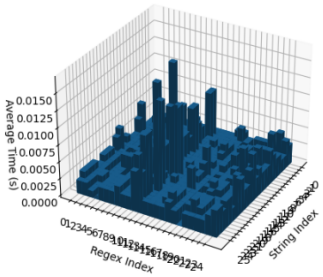
PHP: Completed ✅



Python 3: Completed

Python 2: Completed





Ruby: Completed