International Journal of Applied Engineering and Technology ISSN: 2277-212X
An Open Access, Online International Journal Available at http://www.cibtech.org/jet.htm
2022, Vol.12, pp.128-138/Vijay et al.
Research Article

SOFTWARE QUALITY IMPROVEMENT FOR REGULAR EXPRESSIONMATCHING TOOLS USING AUTOMATED TESTING

***M. Vijay ananth kumar[1], Manikandan[2] and C. Senthil kumar[3]**
Department of MCA Roever Engineering College
**Author for Correspondence: vijayananthmca@gmail.com*

ABSTRACT
Regular expression matching tools (grep) match regular expressions to lines of text. However, because of the complexity that regular expressions can reach, it is challenging to apply state of the art automated testing frameworks to grep tools. Combinatorial testing has shown to be an effective testing methodology, especially for systems with large input spaces. In this dissertation, we investigate the approach of a fully automated combinatorial testing system for regular expression matching tools CoRE (Combinatorial testing for Regular Expressions). CoRE automatically generates test cases using combinatorial testing and measures correctness using differential testing. CoRE outperformed AFL and AFLFast in terms of code coverage testing icGrep, GNU grep and PCRE grep.

***Keywords**: Regular Expression, Grep, Automated testing, Combinatorial testing, Regular Expression Generator; String Generator, Differential testing*

INTRODUCTION

Ever since regular expressions were first used to match text in 1968 by Ken Thompson [1], regular expressions have experienced a remarkable rise in popularity [2, 3]. A regular expression is a specific kind of text pattern that you can use with many modern applications and programming languages such as verifying input patterns, finding text that matches the pattern within a larger body of text, replacing text matching the pattern with other text and many other applications.

Today, almost all popular programming languages like Java, C and Python include a powerful regular expression library, or even have regular expression support built right into the language [4]. Many developers have taken advantage of these regular expression features to provide users of their applications the ability to search or filter through their data using a regular expression. The adaptation of regular expressions in different tools and the differences in supported features between these tools resulted in different regular expression syntaxes (sometimes called flavors). Thus, creating additional challenges to the attempt of testing regular expression matching tools [5].

In this dissertation, we reveal an approach to an automated testing framework for regular expression matching tools (grep) using automated combinatorial testing and differential testing. Over the past few years, combinatorial testing has shown to be an effective testing strategy [6] [7] [8]. Combinatorial testing is considered a black box testing technique. It requires no knowledge of the system's implementation relying on the knowledge of input space model. Some system problems only occur when a combination of input parameters interact. For 2-way or pairwise testing, every pair of input parameters must be tested at least once in the test suite. The same concept applies to k-way testing. There are algorithms and tools like Automated Combinatorial Testing for Software ACTS [6] to help generate all different combinations of parameters to satisfy k-way testing. It takes the input parameters for the system under test and, using covering arrays, produces abstract combinatorial tests. These abstract test cases then need to be transformed into concrete test cases ready to run on the system under test. But for systems with a large input space, it would take a lot of time and effort to write concrete combinatorial test suites and thus may only be feasible when applied to small systems or critical parts of bigger systems [8].

While ACTS generates raw combinatorial test cases, our contribution relies on transforming these combinations into ready-to-run test cases.
In order to evaluate our approach, we implemented a tool CoRE (Combinatorial testing for Regular Expressions) that tests regular expression matching tools like GNU grep and icGrep. GNU grep is a grep tool implemented by GNU organization that supports GNU basic regular expression syntax BRE as well as GNU extended regular expression syntax ERE. On the other hand, icGrep is a powerful regular expression matching tool with support of GNU BRE and ERE syntaxes along with Unicode RE syntax.
Grep tools normally take three inputs. A regular expression, an input file and command line options such as Case insensitive mode or count mode. A regular expression is a sequence of characters that define a search pattern.
To reach full automation of combinatorial testing for regular expression matching tools, we applied two main techniques:

- Automated transformation of ACTS abstract combinatorial test cases into concrete grep test cases.
- Automation of result evaluation and error detection using differential testing.

There have been some efforts to use combinatorial testing to test grep tools [9]. Borazjany showed that applying such technique on a system like grep can improve fault detection and software quality. Borazjany manually transformed ACTS output to test cases hand writing regular expressions as well as input files.
In terms of Automated Testing, there are fuzzing tools like American Fuzzy Lop (AFL) [10] and others [11] [12] which rely on generating extensive tests and looking for crashes. AFL takes an initial test suite and mutates input using sequential bit manipulation to explore new execution paths. AFLFast is an extension of AFL with a different techniqueto mutate initial test suite using Markov chains [11].
The goal of this dissertation is to evaluate our proposed methodology against existing approaches and observe the impact of our approach on the quality of the regular expression matching tool under test. To do so, we compare the code coverage CoRE reaches testing grep tools to the requirement based manually written test suites. We evaluate CoRE testing icGrep, GNU grep and PCRE grep. icGrep is a powerful regular expression matching tool based on Parabix, a parallel computing framework, supporting different regular expression syntaxes [12]. We are interested in testing icGrep because it relies on LLVM JIT compilation [13] which makes static analysis techniques used by fuzz testing tools challenging. We also evaluate CoRE against two fuzzing tools, AFL and AFLFast comparing statement and function coverage.
Additionally, we evaluated CoRE testing GNU grep performing differential testing with FreeBSD grep. Both GNU grep and FreeBSD grep follow the same syntax and should be returning identical results. We also evaluated CoRE testing PCRE Grep.
In next Chapter , we discuss the history regular expressions, the growth of interest in regular expressions and how they evolved to different flavors over different applications. After that, we discuss previous efforts in the combinatorial testing and the automated testing fields as well as some previous work on applying combinatorial testing on systems like grep. Chapter 3 discusses the design and methodology of CoRE starting from the input space modeling to regular expression generation and input file generation ending with composing the test suite and comparing results with different grep tools. In Chapter 4, we evaluate the effectiveness of automating combinatorial testing for regular expression matching tools and observe its effect on system quality. Chapter 5 concludes this dissertation with a summary of the contribution of our work. It also discusses possible future work to further expand the benefits of such an approach.

Background and Overview

Regular Expression Matching

A regular expression is a pattern that consists of one or more character literals and operators. Regular

expression matching tools like icGrep search plain-text data sets for lines that match a regular expression.
Ken Thompson used regular expressions to match patterns in a text editor in 1968 [1]. In the 1980's, the Perl programming language incorporated regular expressions as first-class elements of the programming language. Perl provided several innovative extensions of regular expressions that became common. Since then, many regular expression matching tools have emerged and started adding new features to regular expressions like POSIX Character classes and Unicode support. Perl Compatible Regular Expressions (PCRE) is a regular expression C library, originated in 1997, inspired by the regular expression capabilities in the Perl programming language [13]. PCRE expanded regular expressions' capabilities and features.
In 2014, Robert Cameron et al. introduced the regular expression matcher icGrep [3]. icGrep uses bitwise data parallelism to achieve high performance regular expression matching. The way icGrep is implemented makes it more challenging to test. icGrep relies on LLVM, "a collection of modular and reusable compiler and toolchain technologies" [14], to generate the match function at runtime (JIT). This means that automated testing tools that rely on static code instrumentation such as AFL may not perform as well on icGrep as it would on other grep tools that do not rely on JIT code generation.

Automated Software Testing

Software testing automation can reduce costs dramatically by saving testers time and energy and directing their efforts on other areas. There are several white box automated testing techniques that have shown success in fault detection.
One successful automated testing technique is fuzzing [11] [19]. Fuzz testing involves providing randomly generated inputs in an attempt to make the software under test SUT fail [12]. This kind of testing is achieved by using a varietyof strategies and algorithms to mutate the test suite of the SUT [11]. American Fuzzy Lop (AFL) is a tool which uses code analysis to mutate inputs to explore new paths in the control flow of the software under test.
AFLFast is an extension of AFL and is also considered gray box testing, mid- level between white box and black box testing. AFLFast requires no program analysis. Instead of analyzing code as in AFL, AFLFast produces new tests by mutating a seed input and tracking if the test visits interesting paths in the program. If so, the test is added to the set of seeds and otherwise discarded. AFLFast claims to be more efficient than AFL.
The problem with using fuzzers to test grep tools is that regular expressions are syntactically constrained. Open and closed parenthesis and brackets have to match while other Meta characters require a value from a pre-defined set. These constrains make fuzzers hit a syntax error more often than not. Although testing these incidents are important to know if the grep tool under test manages syntax errors correctly, most bugs are found in tests with proper formed regular expressions. Another downside to fuzzers is their inability to perform useful differential testing to find correctness bugs for grep tools because of the need for input files containing matches to the generated regular expression.

Design and Methodology

Throughout this section, we use icGrep as an example to illustrate the design and methodology to test a grep toolusing CoRE. All of what is explained here applies to most if not all grep tools similarly.

Input Space Modeling

Prior to performing combinatorial testing to the software under test, we must model the system's input space in a way that captures all input parameters for grep tools. A regular expression is a sequence of characters that form a pattern. These characters can either match themselves, i.e. 'abc' and '123', or can have a special property like '+' or '\s' and are called metacharacters. We decided to categorize combinatorial parameters based on metacharacters. Doing so gives us the ability to test metacharacters with different combinatorial settings. Metacharacters only appear in the test if their value was not set to 'off' in the combinatorial test. Table 3.1 Show the combinatorial parameters of a combinatorial testing tool (ACTS) for regular expressions.
For Boolean parameters, the feature exists in the regular expression used for the test only if the value is

"True". There are some parameters that have enumerated values for the parameters. For example, the values for the Property parameter are categorized based on the property type [31]. *Enumeration* properties have enumerated values which constitute a logical partition space. *Binary* properties are a special case of Enumeration properties, which have exactly two values: Yes and No (or True and False) while *Numeric* properties specify the actual numeric values for digits and other characters associated with numbers in some way. Finally, *String* typed Properties take a character class or a regular expression as a value of the property itself. Expanding our combinatorial parameter to capture all these types of parameters increases the coverage of Unicode properties. Our proposed methodology makes adding new features and metacharacters easy. Doing so requires adding the appropriate parameters in the combinatorial testing tool as well as some code to transform the parameter into the appropriate metacharacter.

Table 1: Regular Expression parameters for icGrep

Parameter	**Value Type**	**Description**
Any	Boolean	'.' matches any single character.
Zero or One	Boolean	'?' makes the preceding pattern optional.
Zero or More	Boolean	'*' makes the preceding pattern matched zero or more times.
One or More	Boolean	'+' makes the preceding pattern matched one or more times.
Repetition {n}	Unum = {small, medium,large}	'{n}' makes the preceding pattern matched n times.
Repetition {n,m}	Enum = {small-small, small-medium, small-large, medium-large, large}	'{n,m}' makes the preceding pattern matched between nand m times.
Repetition {n,}	Enum = {small, medium, large}	'{n,}' matches the preceding pattern n or more times
Repetition {,m}	Enum = {small, medium, large}	'{n,}' matches the preceding patter at most m times
Alternation	Boolean	'|' matches either the preceding pattern or the followingpattern
List	Boolean	'[xyz]' matches either x or y or z.
Not List	Boolean	'[^xyz]' matches any character except x, y and z.
Range	Boolean	'[1-9]' matches a character from 1 until 9.
Posix Bracket Expression	Enum = {off, alnum, alpha, blank, digit, graph, lower, upper, print, punct,xdigit}	Special kind of character classes. For example, '[:alpha:]'matches any alphabet character.
Word Character	Boolean	'\w' matches word constituent

Not Word Character	Boolean	'\W' matches non-word constituent
Whitespace	Boolean	'\s' matches the whitespace character.
Not whitespace	Boolean	'\S' matches any non-whitespace characters.
Tab	Boolean	'\t' matches the horizontal tab character.
Digit	Boolean	'\d' matches a digit character.
Not Digit	Boolean	'\D' matches a non-digit character.
Property	Enum = {off, binary, enum, string, numeric}	'\p{property}' matches any character with the specifiedUnicode property.
Not Property	Enum = {off, binary, enum, catalog, numeric}	'\P{property}' matches any character not having the specified Unicode property.
Name Property	Boolean	'\N{Name}' matches the named character.
Unicode Codepoint	Boolean	'\uFFFF' where FFFF are four hexadecimal digits, matchesa specific Unicode codepoint.
Lookahead	Boolean	'(?=pattern)' Matches at a position where the pattern insidethe lookahead can be matched. Matches only the position. It does not consume any characters or expand the match.
Negative Lookahead	Boolean	'(?!pattern)' Similar to positive lookahead, except that negative lookahead only succeeds if the regex inside thelookahead fails to match.
Lookbehind	Boolean	'(?<=pattern) Matches at a position if the pattern inside the lookbehind can be matched ending at that position.
Negative Lookbehind	Boolean	'(?<!pattern)' Matches at a position if the pattern inside the lookbehind cannot be matched ending at that position.
Start	Boolean	'^' matches the empty string at the beginning of a line.
End	Boolean	'$' matches the empty string at the end of a line.
Back Referencing	Boolean	'\n' where 1 ☐ n ☐ 9, match the same text as previouslymatched by the nth capturing group.

Table 2: Regular Expression parameters for icGrep

Parameter Type	**Parameter**	**Value Type**	**Description**
Regular Expression Interpretation	Case Insensitive	Boolean	'-i' Ignores case distinctions in the pattern.
	Regular Expression Syntax	Enum = {off, -G, -E, -P}	'-G', '-E' and '-P' specify the regular expression syntax used.
	Word Regular Expression	Boolean	'-w' requires that whole words be matches.
	Line Regular Expression	Boolean	'-x' requires that entire lines be matched.

Input Options	Multiple Regular Expressions	Boolean	'-e pattern' is used to match multiple regularexpression or with '-f'.
	Regular Expression File	Boolean	'-f File' is used to read regular expression from File line by line.
Output Options	Count	Boolean	'-c' displays only the number of matches.
	Inverted Match	Boolean	'-v' selects non-matching lines.
icGrep SpecificFlags	Threads	Enum ={off,1,2,3,4}	'-t=n' where n is a digit, specifies the numberof threads used.
	Block Size	Enum = {off, 64,128,265,512}	'-BlockSize=n' where n is a digit, specifiesthe processing block size.

Table 2 shows the parameters for the combinatorial testing for the command line flags. The flags can affect thematched regular expression, like case insensitive match and regular expression syntax. Other types of flags provide input and output options like counting matches. There are flags which are icGrep specific like segment size and thread count. icGrep supports GNU basic regular expression syntax "-G" as well as their extended regular expression syntax "-E". icGrep also supports Unicode ICU regular expression syntax. Some combinatorial parameters may not be supported in the basic and extended regular expression syntaxes. These parameters can be modified depending on the grep tool under test.

Test Case Generation

After setting all the parameters for the grep tool under test, we generate test cases from the model using combinatorial testing tools such as ACTS. These test cases are abstract test cases because the parameters and values in the model are abstract. Thus, it is necessary to derive concrete test cases from these abstract test cases before the actual testing can be performed. Note that an abstract test case typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing.
We show in Figure 1 The architecture of the proposed methodology. In order to perform this testing technique, we process the abstract test cases one by one. At each cycle, we first transform the abstract test case into a concrete test case using the regular expression generator and the string generator. The regular expression generator takes the values of the test case generated from ACTS and generates a corresponding regular expression and command line flags to be tested. The string generator takes the generated regular expression as input and generates a file containing a match to the regular expression

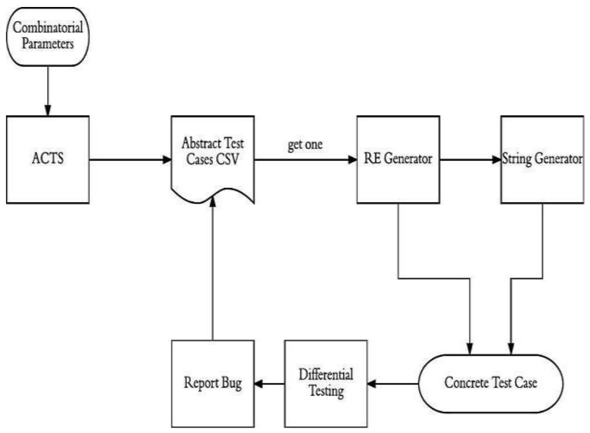
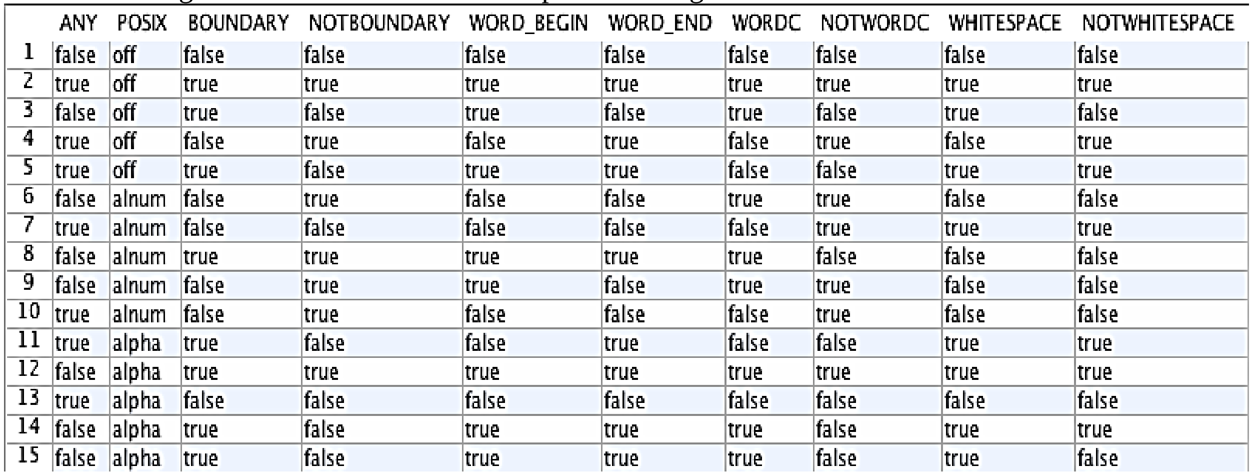


Figure 1: Architecture of the proposed methodology

Regular Expression Generator: The first step to transform the ACTS raw test cases into ready-to-run test cases is the regular expression generator. The regular expression generator transforms a set of parameter values in ACTS into a regular expression along with the command line flags.
Looking at Figure 2, a snapshot of icGrep project on ACTS, the first row represents the parameters specified in the input space modeling phase where the figure only shows a subset of the parameters. Each of the following rows is a set of values that represent a single test case.



	ANY	POSIX	BOUNDARY	NOTBOUNDARY	WORD_BEGIN	WORD_END	WORDC	NOTWORDC	WHITESPACE	NOTWHITESPACE
1	false	off	false	false	false	false	false	false	false	false
2	true	off	true	true	true	true	true	true	true	true
3	false	off	true	false	true	false	true	false	true	false
4	true	off	false	true	false	true	false	true	false	true
5	true	off	true	false	true	true	false	false	true	true
6	false	alnum	false	true	false	false	true	true	false	false
7	true	alnum	false	false	false	false	false	true	true	true
8	false	alnum	true	true	true	true	true	false	false	false
9	false	alnum	false	true	true	false	true	true	false	false
10	true	alnum	false	true	false	false	false	true	false	false
11	true	alpha	true	false	false	true	false	false	true	true
12	false	alpha	true	true	true	true	true	true	true	true
13	true	alpha	false	false	false	false	false	false	false	false
14	false	alpha	true	false	true	true	true	false	true	true
15	false	alpha	true	false	true	true	true	false	true	false

Figure 2: A snapshot of icGrep project on ACTS

First, we set the syntax of the regular expression based on the associated parameter value where –G is the GNU basic regular expression syntax, -E is the GNU extended regular expression syntax [32] and –P is the default icGrep regular expression syntax that follows the Unicode ICU regular expression syntax [33].

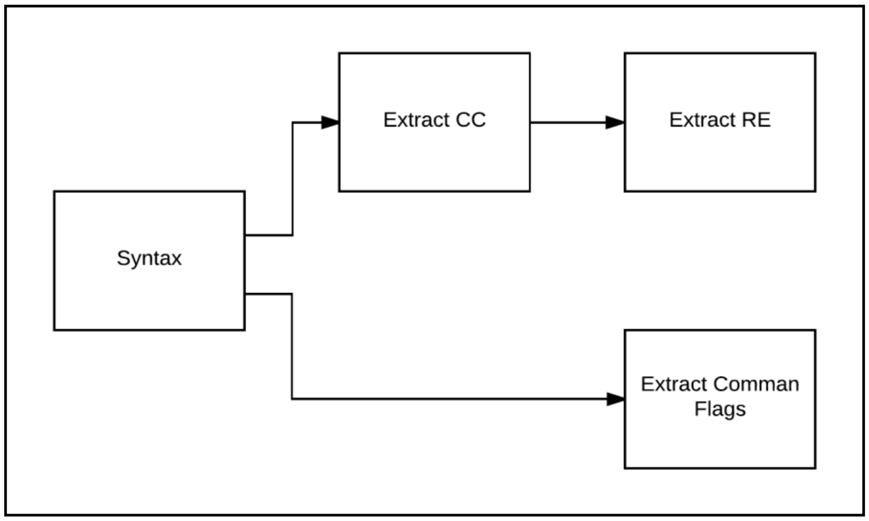


Figure 3: Regular Expression Generator Flow Chart

Second, we collect a set that contains the parameters representing character classes whose value is not "false". For each of these parameters, the syntax of the corresponding regular expression metacharacter may differ depending on the regular expression syntax. The GNU basic regular expression syntax only supports a few metacharacters that are considered character classes. The "Any" metacharacter is transformed to the metacharacter "." in all syntaxes. The same applies to the whitespace character which transforms to "\s" and its complement "\S". The rest of the metacharacters supported by the GNU basic

regular expression syntax an operational behavior and will be discussed further down this section.
Finally, the Start and End parameters are added to the beginning and the end of the RE set respectively if their value is "true". These metacharacters would have no meaning if present in the middle of a regular expression, therefore are added after shuffling the RE set. The regular expression is the combination of each of the elements in RE in an ordered matter.
After transforming the raw ACTS combinatorial values into a regular expression, the regular expression generator iterates through the flags parameters storing each command line flag in a set (F).

Results and Evaluation

In order to thoroughly evaluate CoRE, we first run CoRE over different t-way combinatorial configurations andcalculate the statement and coverage rate for each run. We also measure the fail rate by the formula:

$$\text{Fail Rate} = \frac{\text{Failed test cases}}{\text{All test cases}}$$

Once we find a configuration where adding more combinatorial constraints does not increase code Coverage nor the fail rate, we will use this configuration to evaluate CoRE against two different testing techniques. The first is the manually written test suite. It is written by icGrep developers based on the icGrep requirement specification. The other testing techniques we evaluate CoRE against are two automated testing tools AFL and AFL and AFLFast. AFLFast is an extension of the state of the art fuzzer American Fuzzy Lop. AFLFast performed better than AFL with a better fault detection rate on GNU binutils, a collection of binary tools widely used for the analysis of program binaries [11]. We will evaluate CoRE against AFL and AFLFast to determine how well CoRE performs against state of the art automated testing methods.
Experimental Infrastructure: We ran our experiments on a MacBook Air with a 2.2 GHz Intel Core i7 processor with 4 cores and 8GB RAM. We ran the testing tools on icGrep revision 5720. We ran each test 10 times and we used Gcov tool to measure code coverage. We also ran AFL and AFLFast on all 4 cores to maximize its performance. Weset 10 seconds to be the time limit for each test run. GNU grep version 3.1 and the library ICU4C version 59 were used in CoRE for differential testing.

Combinatorial Level	**Number of Tests**	**Statement Coverage %**	**Function Coverage %**	**Fail Rate %**	**ElapsedTime**
1-way	10	66.8	73.3	20	12s
2-way	64	67.1	73.4	22.2	1m39s
3-way	394	67.8	73.6	26.5	12m5s
4-way	2228	68.1	74.3	29.1	1h8m46s
5-way	10926	68.4	74.4	32.3	5h42m21

Table 2 Code Coverage and Bug Rate for Different Levels of Combinatorial Testing On Core

Table 2 shows the code coverage for CoRE running different levels of t-way combinatorial testing. It also shows the number of tests generated in each level as well as the execution times for different levels of combinatorial interaction on CoRE testing icGrep with differential testing with GNU grep and ICU RegexMatcher.
Another note is that 6-way was dropped out of the evaluation process despite the possibility of having even more complex tests than 5-way. The reason we did not evaluate 6-way is because we could not generate a 6-way combinatorial test suite in ACTS before running out of memory.
From Table 2, we notice that the combinatorial interaction level has almost no effect on statement or

function coverage. But when we look at the fail rate, we find that as we increase the level of combinatorial testing, the percentage of failed tests increases. For this reason, we evaluate CoRE against manually written test suits, AFL and AFLFast using 5-way combinatorial testing.

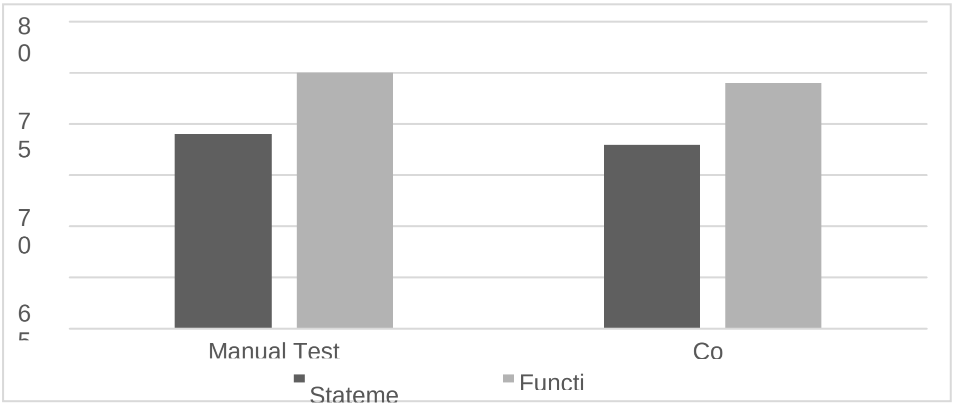


Figure 4: Code coverage for manual test suites and CoRE on 5-way combinatorial level

From Figure 4, we note that CoRE has almost the same score for both statement coverage and function coverage as the manual test suites. Manual test suites scored 1% higher in both statement and function coverage. Even when comparing manual test suites with 2-way combinatorial level on CoRE to have similar execution times (manual tests take 1m41 seconds to execute), the difference in about 1% lower coverage in statements and functions.

Limitations: The string generator in CoRE relies on icGrep's parser to construct the regular expression abstract syntax tree. This limits CoRE to test grep tools with syntaxes similar to the ones supported by icGrep. Also, relying on icGrep's parser adds some bias to the generated strings since the string generator will only generate characters that are defined in the character class by icGrep's parser.

Another limitation is that the string generator only generates strings that match a regular expression. This means ifa regular expression matching tool returns all line as matches and returns nothing when inverted match is invoked would theoretically pass all tests generated by CoRE. Adding a feature to the string generator to generate non- matching lines would eliminate this concern.

Future work

CoRE is an implementation of a fully automated combinatorial testing methodology for regular expression matching tools. It showcases our idea of a fully automated combinatorial testing approach to testing regular expression matching tools. There are different areas to continue our research in.

One way to enhance CoRE is to expand the capabilities of core enabling it to perform differential testing between grep tools with different syntaxes testing the common features between them. We could generate a regular expression AST instead of a full regular expression. Then, we could transform the AST to different syntaxes based on the grep tools under test. For instance, any regular expression written in the GNU BRE syntax could be transformed into a GNU ERE syntax. This give us the ability to test grep's GNU BRE syntax against its ERE syntax.

The experiments we have done show an advantage of CoRE over AFL and AFLFast testing icGrep, GNU grep and PCRE grep in statement and function coverage. Having more time and resources would give us the chance to evaluate CoRE against other automated testing techniques like Nezha. Nezha uses fuzzing as well as differential testing to test for correctness but it requires writing code to make it work on icGrep and other grep tools.

Another interesting take on CoRE would be to use differential testing to measure performance differences between regular expression matching tool. Performance is an important aspect of regular expression

matching tools and would be interesting to know which test cases cause performance problems to icGrep compared to other grep tools and which test cases give icGrep the performance advantage.
We would like to further enhance CoRE by identifying unique bugs. This requires little instrumentation of the system under test to track the control flow of each test case and only report bugs that explore new paths in the control glow graph.
Finally, Expanding our methodology for a fully automated combinatorial testing solution and testing systemswith complex input spaces other than regular expression matching tools.

Conclusion

In this dissertation, we presented a methodology to reach fully automated testing for regular expression matching tools. We implemented CoRE, a testing tool based on our proposed approach testing icGrep, GNU grep and PCRE grep.
To reach full automation, we implemented a regular expression generator and a string generator to generate test cases. We also performed differential testing on the generated test cases.
We evaluated CoRE against hand written test suites and two fuzzing tools, AFL and AFLFast testing icGrep and measuring code coverage and bug detection rate.
CoRE outperformed AFL and AFLFast in both statement coverage and function coverage in icGrep, GNU grep and PCRE grep. CoRE also found bugs that were not caught by manual test suites nor AFL or AFLFast.CoRE also detected a bug in FreeBSD grep running Mac OSX 10.1.
Our proposed approach to automated combinatorial testing for regular expression matching tools showed to be effective and can improve the quality of regular expression matching tools.

REFERENCES

K. Thompson (1968). Programming Techniques: Regular Expression Search Algorithm, vol. 11, no. 6, pp. 419-422.
B. Kernighan (2007). A Regular Expressions Matcher, in *Beautiful Code*, O'Reilly Media.
R. D. Cameron, T. C. Shermer, A. Shrirman, K. S. Herdy, D. Lin, B. R. Hull and D. Lin (2014). Bitwise data parallelism in regular expression matching, in *The 23rd international conference on Parallel architectures and compilation*, Edmonton.
T. Stubblebine (2003). Regular Expression Pocket Reference, Sebastopol, CA: O'Reilly & Associates, Inc,.
J. Goyvaerts and S. Levinthan (2012). Regular Expressions Cookbook, O'Reilly Media, pp. 1-2.
D. R. Kuhn, R. N. Kacker and Y. Lei (2013). Introduction to Compinatorial Testing, Chapman and Hall/CRC.
M. N. Borazjany, G. Laleh, Y. Lei, R. Kacker and R. Kuhn (2013). An Input Space Modeling MEthodology for Combinatorial Testing, in *2nd International Workshop on Combinatorial Testing*, Luxembourg. **D. R. Kuhn and M. J. Reilly(2012).**An investigation of the Applicability of Design of Experimentsto Software Testing, in *27th Annual Nasa Goddard/IEEE*.
M. N. Borazjany, (2013). "Applying Combinatorial Testing to Systems with A Complex Input Space," **M.Zalewski (2017)**AmericanFuzzyLop[Online].Available: http://lcamtuf.coredump.cx/afl/README.txt. [Accessed 2 11].*Thesis* (Ph. D). **M. Bohme, V. Pham and A. Roychoudhury (2016).** Coverage-based Greybox Fuzzing as Markov Chain in SIGSAC Conference on Computer and Communications Security.
R. D. Cameron, N. Medforth, D. Lin, D. Denis and W. N. Sumner (2015). Bitwise Data Parallelism with LLVM: The ICgrep Case Study.
P. Hazel(1999).Exim and PCRE: How free software hijacked my life, [Online].
Available: http://www.ukuug.org/events/winter99/proc/PH.ps. [Accessed 3 11 2017].
D. M. Cohen, S. R. Dalal, J. Parelius and C. Patton (1996). The Combinatorial Design to Automatic Test Generation, vol. 13, no. 5, pp. 83-88, 9.
M. Grindal, J. Offutt and S. F. Andler (2005). Combination Testing Strategies: A Survey, vol. 15, no. 3, pp. 167-199.

Y. Lei, R. Kacker, D. R. Kuhn, V. Okun and J. Lawrence (2007). "IPOG/IPO -D: Effecient Test Generation for Multi-Way Combinatorial Testing," vol. 18, no. 3, pp. 125-148

R. W. D. Kuhn and R. Gallo (2004). Software fault interactions and implications for software testing,vol. 30, no. 6, pp. 418-421,

C. Lemieux and K. Sen, FairFuzz:Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testinig Coverage, [Online]. Available: https://arxiv.org/pdf/1709.07101.pdf. [Accessed 3 11 2017].

T. Petsios, A. Tang, S. Stolfo, A. Keromytis and S. Jana (2017). "NEZHA: Efficient Domain-Independent Differential Testing.

W. M. McKeeman (1998). Differential Testing for Software,vol. 10, no. 1, pp. 100-107

C. Brubaker, S. Jana, B. Ray, S. Khurshid and V. Shmatikov (2014). Using frankencerts forautomated adversarial testing of certificate validation in SSL/TLS implementations, in IEEE Symposium on Security and Privacy

Y. Chen and Z. Su (2015). Guided differential testing of certificate validation in SSL/TLS implementations, in The 10th Joint Meeting on Foundations of Software Engineering,

S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis and S. Jana (2017). HVLearn: Automates Black-Box Analysis of Hostname Verification in SSL/TLS Implementations," in IEEE Symposium on Security and Privacy, San Jose.

X. Yang, Y. Chen, E. Eide and J. Regehr (2011). Finding and Understanding Bugs in C Compilers," in The 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.

Y. Chen, T. Su, C. Sun, Z. Su and J. Zhao (2016). Coverage-directed differential testing of JVM implementations, in The 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.

G. Argyros, I. Stais, S. Jana, A. D. Keromytis and A. Kiaylias (2016).SFADiff: AutomatedEvasion Attacks and Fingerprinting using Black-box Differential Automata Learning, in ACM SIGSAC Conference on Computer and Communications Security

V. Srivastava, M. D. M. K. S. Bond and V. Shmatikov (2011). A Security Policy Oracle: Detecting Security Holes using Multiple API Implementations," ACM SIGPLAN Notices, vol. 46, no. 6.

S. Jana and V. Shmatikov (2012). Abusing File Processing in Malware Detectors for Fun and Profit, in IEEE Symposium on Security and Privacy,

D. Brumley, J. Caballero, Z. Liang, J. Newsome and D. Song (2007). Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection andFingerprint generation, in The 16th USENIX Security Symposium.

Unicode.org, "Unicode Character Database," The Unicode Consortium, 14 6 2017. [Online]. Available: http://www.unicode.org/reports/tr44/. [Accessed 3 11 2017].

GNU Grep Manual, GNU, 9 2 2017. [Online]. Available: https://www.gnu.org/software/grep/manual/grep.html. [Accessed 3 11 2017].

ICU User Guide (Regular Expressions), [Online]. Available: http://userguide.icu-project.org/strings/regexp. [Accessed 3 11 2017].

R.Expressions.info.[Online]. Available: https://www.regular-expressions.info/posixbrackets.html#class. [Accessed 3 12 2017].