Improving Software Quality for Regular Expression Matching Tools Using Automated Combinatorial Testing

by

Fahad Aldebeyan

B.Sc., King Saud University, 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Fahad Aldebeyan
SIMON FRASER UNIVERSITY
Spring 2018

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Approval

Name:	**Fahad Aldebeyan**
Degree:	**Master of Science**
Title:	**Improving Software Quality for Regular Expression Matching Tools Using Automated Combinatorial Testing**
Examining Committee:	**Chair:** Dr. Kay Wiese Associate Professor

Dr. Robert Cameron
Senior Supervisor
Professor

Dr. William Sumner
Supervisor
Assistant Professor

Dr. Thomas Shermer
Supervisor
Professor

Dr. Fred Popowich
Internal Examiner
Professor

Date Defended/Approved: December 19, 2017

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; String Generator; Differential testing

Acknowledgements

First, I would like to express my greatest gratitude and appreciation to my senior supervisor Dr. Rob Cameron for providing me with the opportunity to complete this thesis providing me knowledge and guidance throughout this thesis. I have considered it a great honour.

I would also like to thank Dr. Nick Sumner and Dr. Thomas Shermer for their assistance, guidance, and suggestions that have been invaluable in the completion of this work.

I would like to thank The Saudi Cultural Bureau and The Kingdom of Saudi Arabia for granting me a scholarship to complete my thesis.

Finally, I must express my very profound gratitude to my parents Ahmad and Susan and wife Eman for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This thesis is dedicated to them.

Table of Contents

Approval ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Tables vi
List of Figures vii

Chapter 1. Introduction 1

Chapter 2. Background and Overview 4
2.1. Regular Expression Matching 4
2.2. Combinatorial Testing 4
2.3. Automated Software Testing 6
2.4. Differential Testing 7

Chapter 3. Design and Methodology 9
3.1. Input Space Modeling 9
3.2. Test Case Generation 13
3.2.1. Regular Expression Generator 14
Unicode Metacharacters 16
Example 20
3.2.2. String Generator 22
3.3. Differential Testing 23

Chapter 4. Results and Evaluation 26

Chapter 5. Future work 34

Chapter 6. Conclusion 36

Bibliography 37

List of Tables

Table 3.1. Regular Expression parameters for icGrep 11
Table 3.2. Command-line Parameters for icGrep 12
Table 3.3. Parameter to metacharacter Transformation Based on Regular Expression Syntax 16
Table 3.4. Transformation of ACTS operator and assertion parameters to regular expressions for icGrep supported syntaxes 19
Table 3.5. Example of one test case from ACTS for icGrep 21
Table 4.1. Code coverage and bug rate for different levels of combinatorial testing on CoRE 27
Table 4.2. Bug in FreeBSD Grep found by CoRE 31

List of Figures

Figure 2.1. AETG pair-wise test cases [9]. P: parameter ABC: values 5
Figure 2.2. IPO pair-wise test cases [9]. P: parameter ABC: values 6
Figure 3.1. Architecture for CoRE 13
Figure 3.2. Snapshot of ACTS Combinatorial Testing 14
Figure 3.3. Regular Expression Generator Flow Chart 15
Figure 3.4. Pseudocode to extract the Character Class set 18
Figure 3.5. Pseudocode to extract the Regular Expression set 18
Figure 3.6. Pseudocode for the string generator 22
Figure 4.1. Code coverage for manual test suites and CoRE on 5-way combinatorial level 28
Figure 4.2. Code coverage for AFL, AFLFast and CoRE testing icGrep 29
Figure 4.3. Code coverage for AFL, AFLFast, CoRE and Manual Tests testing GNU Grep 30
Figure 4.4. Code coverage for AFL, AFLFast, CoRE and Manual Tests testing PCRE Grep 30
Figure 4.5. Fail rate of CoRE with different combinatorial levels before and after eliminating the carry manager bug in icGrep 32

Chapter 1. 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, 5]. 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 investigate an approach to an automated testing framework for regular expression matching tools (grep) using combinatorial testing and differential testing. Our hypothesis is that our approach would expose defects in grep tools that are hard to detect using existing automated testing tools.

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 combinatorial testing tools like ACTS generate 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 technique to mutate initial test suite using Markov chains [11].

To evaluate our approach, 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 supports three different syntaxes with Unicode support for its default syntax. 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 Chapter 2, 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.

Chapter 2. Background and Overview

2.1. 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 some automated testing tools may not perform as well on icGrep as it would on grep tools that only rely on compile time code.

2.2. Combinatorial Testing

Software failures are often the result of a faulty interaction between input parameters. Testing all the combinations of the input parameters is often impossible for large and/or complex software systems due to resource constraints. Combinatorial (or t-way) testing covers every combination of any t parameter values at least once [15]. Empirical studies suggest that combinatorial testing can be very effective for fault detection in practice. A NIST study suggests that all the faults in several applications are caused by interactions among six or fewer parameters [6].

Combinatorial testing has shown to be an effective strategy to test grep tools [9]. Borazjany studied applying combinatorial testing to system with complex input spaces like grep and found that t-way testing is more reliable and has better performance than modeled-random testing.

Many combinatorial test generation strategies have been proposed to generate test sets that are as small as possible but still satisfy t-way coverage. Grindal et al. [16] surveyed fifteen important strategies that have been reported in the literature. Two representative strategies, i.e., the AETG strategy [15] and the IPO strategy [17], are described as follows. The AETG (Automatic Efficient Test Generation) strategy adopts a greedy framework for combinatorial test generation. In this algorithm, a test is created to cover as many uncovered t-way combinations as possible. First, it selects a value of a parameter that appears in the most uncovered combinations. Second, it randomly selects another parameter from the rest of the parameters to cover the most number of uncovered pairs. Third, it includes values of the remaining parameters one by one, with the policy used at the second step. Fourth, it repeats the above steps to generate a certain number of candidate tests, and picks the candidate test that covers the most uncovered t-way combinations as the final test. It repeats these steps until all t - way combinations have been satisfied. Figure 2.1 shows AETG strategy for pair-wise test cases.

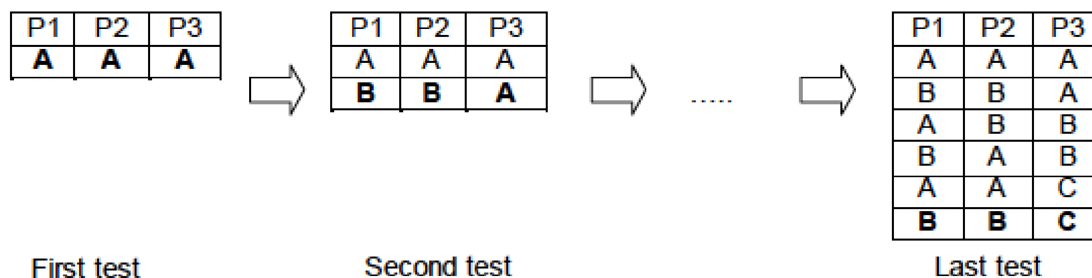

Figure 2.1. AETG pair-wise test cases [9]. P: parameter ABC: values

Lei et al. [17] proposed another t-way testing strategy called In-Parameter-Order (IPO). The IPO strategy generates a t-way test set to cover all the t-way combinations between the first t parameters and then extends the test set to cover all the t-way combinations of the first t+1 parameters. This process is repeated until the test set covers all the t -way combinations of the parameters. In this dissertation, we used a tool

called ACTS to generate our test cases. ACTS implements the IPO strategy. Figure 2.2 demonstrates the IPO strategy for pair-wise test cases.

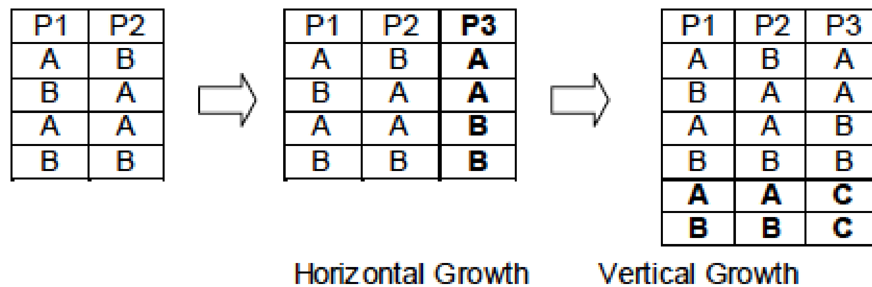

Figure 2.2. IPO pair-wise test cases [9]. P: parameter ABC: values

Kuhn et al. [18] reported a study of several fault databases and found that all the faults in these databases are caused by no more than six factors. They analyzed 329 error reports of a large system with a number of subsystems in NASA. Different systems such as database, server, and browser with various sizes (LOC) from 3000 to $2{\times}10^6$ are used in this study. Faults are characterized in a database by date submitted, severity, priority for fix, the location where found, status, the activity being performed when found, and several other features. The summary of all failures reviewed in this paper were triggered by no more than six factors. This study suggested that if all errors in a particular class of software are triggered by finite combinations of t parameters or less, then testing all combinations of t parameters would provide a form of pseudo–exhaustive testing.

2.3. 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 variety of 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 a modification of AFL designed to improve AFL's performance. 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 metacharacters 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.

2.4. Differential Testing

Differential testing is a type of testing to measure correctness. It requires that two or more comparable systems be available to the tester. These systems are presented with an exhaustive series of mechanically generated test cases. If the results differ or one of the systems loops indefinitely or crashes, the tester has a candidate for a bug-exposing test.

Differential testing is the use of two or more programs with similar functionality to test one program against the rest [20, 21]. Differential testing finds semantic bugs by using different implementations of the same functionality as cross-referencing oracles, pinpointing differences in their outputs across many inputs: any discrepancy between the program behaviors on the same input is marked as a potential bug.

Many different domains like SSL/TLS implementations [22, 23, 24], C compilers [25], JVM implementations [26], Web application firewalls [27], security policies for APIs [28], and antivirus software [20, 29] have taken advantage of differential testing. It has also been used for automated fingerprint generation from different network protocol implementations [30].

Differential testing addresses a specific problem—the cost of evaluating test results. If a single test is fed to several comparable programs (In our case, several grep tools), and one program gives a different result, a bug may have been exposed. However, Differential testing usually depends on a source of test generator or a test suite. NEZHA is a differential testing tool that relies on fuzzing to generate tests [20]. But it would be ineffective to apply NEZHA on grep tools because of the missing step of string generation explained in our methodology. In other words, fuzzing tools do not generate a text file for each test to find differences in matches between the two grep tools under test.

Chapter 3. Design and Methodology

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

3.1. 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.

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 n and 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 following pattern
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 specified Unicode 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	'\u{FFFF}' where FFFF are four hexadecimal digits, matches a specific Unicode codepoint.

Parameter	Value Type	Description
Lookahead	Boolean	‘(?=pattern)’ Matches at a position where the pattern inside the 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 the lookahead 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 $1 \le n \le 9$, match the same text as previously matched by the nth capturing group.

Table 3.1. 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 regular expression 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 Specific Flags	Threads	Enum ={off,1,2,3,4}	'-t=n' where n is a digit, specifies the number of threads used.
	Block Size	Enum = {off, 64,128,265,512}	'-BlockSize=n' where n is a digit, specifies the processing block size.

Table 3.2. Command-line Parameters for icGrep

Table 3.2 shows the parameters for the combinatorial testing for the command line flags. The flags can affect the matched 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.

3.2. Test Case Generation

After setting all the parameters for the grep tool under test, we generate test frames from the model using combinatorial testing tools such as ACTS. These test frames 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 frames before the actual testing can be performed. Note that a test frame typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing. Borazjany manually transformed the generated test frames from ACTS into concrete test cases.

We show in Figure 3.1 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.

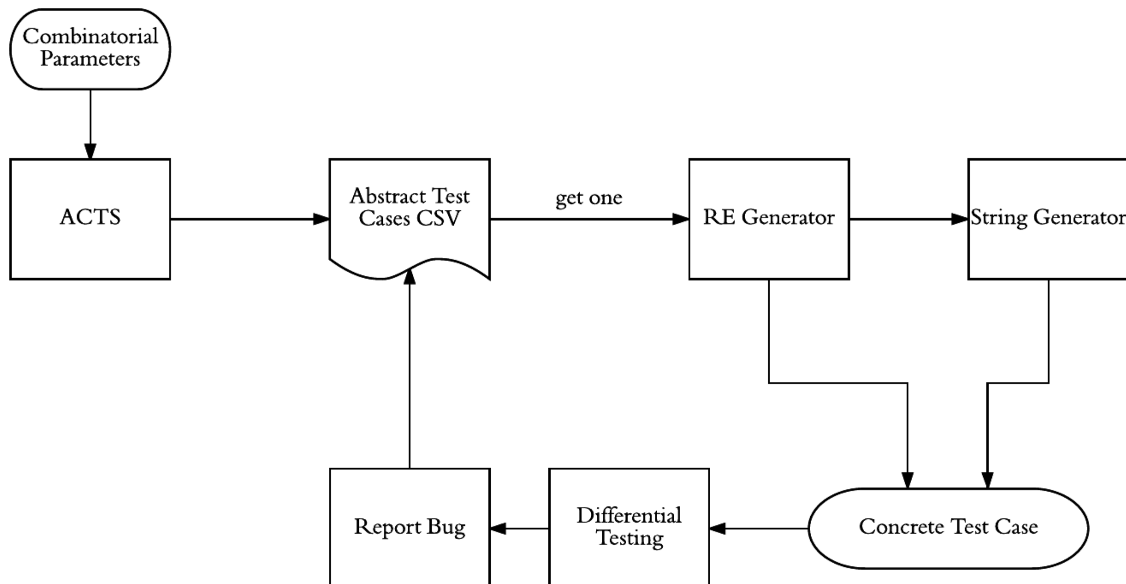

Figure 3.1. Architecture for CoRE

3.2.1. Regular Expression Generator

The first step to transform the ACTS raw test frames 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 3.2, 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 3.2. Snapshot of ACTS Combinatorial Testing

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 3.3 shows the flow chart for the regular expression generator.

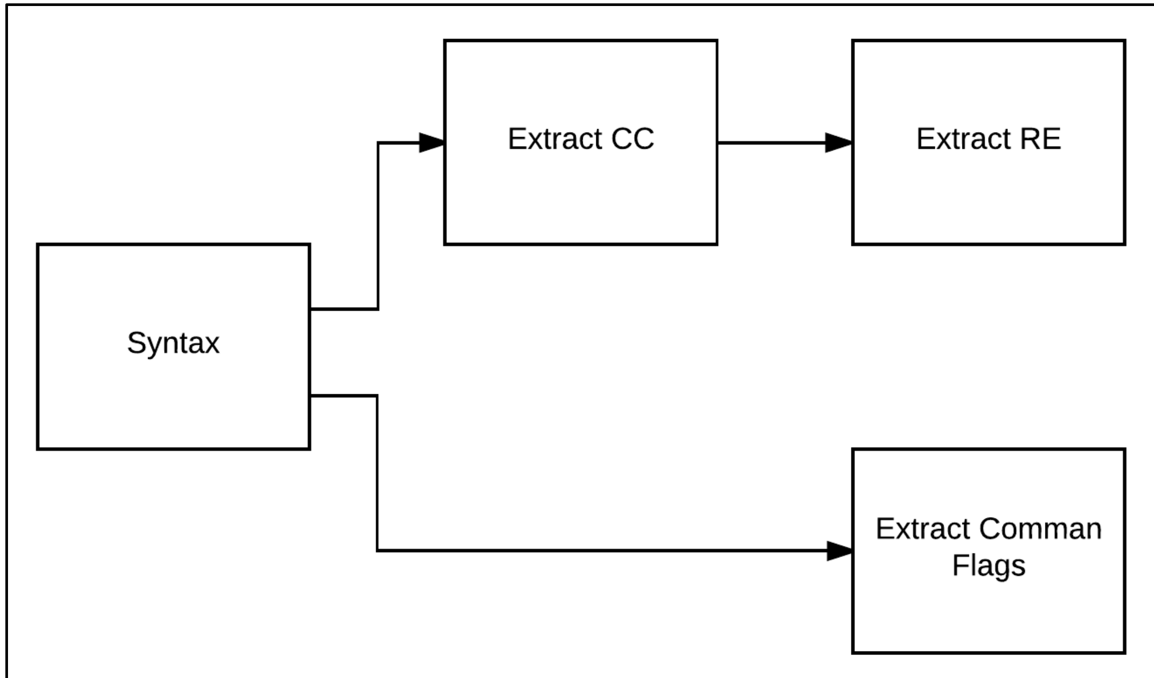

Figure 3.3. Regular Expression Generator Flow Chart

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.

The GNU extended regular expression syntax supports the basic metacharacters as well as a few other metacharacters. The word character parameter transforms to “\w” and its complement “\W”. the start and end parameters transform to “\<” and “\>” respectively. For any other metacharacter that are not supported by GNU basic and extended regular expression syntax, we add a literal character to CC to increase the size of the set. Table 3.3 shows each parameter and its transformation based on the regular expression syntax.

Parameter	GNU Basic	GNU Extended	Unicode ICU
Any	“.”	“.”	“.”
Posix Bracket Expression	“p”	“p”	“[[:value:]]” where value is a POSIX name.
Word Character	“w”	“\w”	“\w”
Not Word Character	“W”	“\W”	“\W”
Whitespace	“\s”	“\s”	“\s”
Not whitespace	“\S”	“\S”	“\S”
Tab	“t”	“t”	“\t”
Digit	“d”	“d”	“\D”
Not Digit	“D”	“D”	“\D”
Property	“p”	“p”	“\p{property}”
Not Property	“P”	“P”	“\P{property}”
Name Property	“N”	“N”	“\N{property name}”
Unicode Codepoint	“u”	“u”	“\u{FFFF}”

Table 3.3. Parameter to metacharacter Transformation Based on Regular Expression Syntax

The Unicode ICU regular expression syntax supports several more metacharacters. The Tab parameter transforms to “\t”. The Digit and Not Digit parameters transform to “\d” and “\D”. The POSIX Bracket Expression transforms to “[[:value:]]” where “value” is the value of the parameter (alnum, alpha, blank, digit, graph, lower, upper, print, punct and xdigit).

Unicode Metacharacters

The Unicode ICU regular expression syntax supports a number of Unicode related metacharacters that need to be handled different from the previous metacharacters to test their functionality to the maximum. The Unicode Property metacharacter “\p” supports many properties that represent different character classes. We categorized these properties based on their types (binary, enum, numeric, string). After that, the regular expression generator randomly selects a property from one of the lists based on the type specified in the parameter value. For example, if the parameter had the value “Property=Binary”, the regular expression generator would randomly select one of the binary properties and transform the parameter to, for example, “\p{Alpha}”.

For other types of properties, they need to have a value and come in the syntax "\p{property=value}". We first randomly select a property matching the type specified in the parameter. After that, if the type is "enum", the regular expression generator randomly selects one of the predefined property values for that property. If the type is "numeric", the regular expression generator uses a random number as a property value. If the property type is "string", the regular expression generator randomly generates a string value for the selected property. The negated Unicode property is treated the same as the Unicode property but with "\P" instead of "\p".

Another Unicode metacharacter is the name property "\N{name}" where "name" is a standard Unicode character name. It is equivalent to "\p{Name=name}" but we decided to have it as a separate parameter since it is treated differently in our regular expression generator. Every Unicode Character has a standard name in the Unicode standard. The name property returns the Unicode character associated with the provided name. icGrep supports "\N{regex}" where it matches regex to the list of Unicode Character Names. The regular expression generator randomly selects one of character names from the standard Unicode data file and add the property, i.e. "\N{ LATIN CAPITAL LETTER A}", to the set CC.

The final Unicode metacharacter supported by icGrep is the Unicode code point. When given "\u{FFFF}" where FFFF are four hexadecimal digits, icGrep matches a single code point encoded U+FFFF. If the value of the Unicode Code Point parameter is "true", the regular expression generator randomly selects a four hexadecimal Unicode encoding and adds the met character along with the encoding to the character class set (CC). We also use Unicode code point representation for ranges in ICU syntax. Figure.3.4 shows the pseudocode for collecting character classes.

```
FOR ALL parameter in P:
        IF Value of parameter is not False THEN
                IF parameter is "Any" THEN
                        generateRandom(Any, Syntax)
                        ADD generated string to CC set
                ELSE IF parameter is "Property" THEN
                        generateRandom(Property, Value, Syntax)
                        ADD generated string to CC set
                ELSE IF ...
                …
RETURN the set CC
```

Figure 3.4. Pseudocode to extract the Character Class set

After filling the character class set using the algorithm in Figure 3.4, the regular expression generator iterates through the rest of the metacharacters that function as either operators on the character class set or are assertion metacharacters. Each of the operator metacharacters use one or more elements from the character class set. The result of the transformation is stored in another set (RE). RE contains elements of a regular expression after applying the operators to the character class set. The pseudocode for generating elements of RE is described in Figure 3.5.

```
FOR ALL parameter in P:
        IF Value of parameter is not False THEN
                IF parameter is "Repetition {n}" THEN
                        SELECT random c from CC
                        generateRandom(RepetitionN, c,  Syntax)
                        ADD generated string to RE set
                ELSE IF parameter is "List" THEN
                        SELECT random c1, c2 and c3 from CC
                        generateRandom(List, c1, c2, c3, Syntax)
                        ADD generated string to RE set
                ELSE IF…
                …
SHUFFLE RE
IF value of START is True THEN
        ADD start Character to RE
IF value of END is True THEN
        ADD end character to RE

RETURN the set RE
```

Figure 3.5. Pseudocode to extract the Regular Expression set

Table 3.4 shows how each of the operator parameters is transformed depending on the regular expression syntax.

<table>
<tr><th>Parameter</th><th>GNU Basic</th><th>GNU Extended</th><th>Unicode ICU</th></tr>
<tr><td>Zero or One</td><td>“cc\?”</td><td>“cc?”</td><td>“cc?”</td></tr>
<tr><td>Zero or More</td><td>“cc*”</td><td>“cc*”</td><td>“cc*”</td></tr>
<tr><td>One or More</td><td>“cc\+”</td><td>“cc+”</td><td>“cc+”</td></tr>
<tr><td>Repetition {n}</td><td>“cc\{n\}”</td><td>“cc{n}”</td><td>“cc{n}”</td></tr>
<tr><td>Repetition {n,m}</td><td>“cc\{n,m\}”</td><td>“cc{n,m}”</td><td>“cc{n,m}”</td></tr>
<tr><td>Repetition {n,}</td><td>“cc\{n,\}”</td><td>“cc{n,}”</td><td>“cc{n,}”</td></tr>
<tr><td>Repetition {,m}</td><td>“cc\{,m\}”</td><td>“cc{,m}”</td><td>“cc{,m}”</td></tr>
<tr><td>Alternation</td><td>“cc1\|cc2”</td><td>“cc1|cc2”</td><td>“cc1|cc2”</td></tr>
<tr><td>List</td><td>“l”</td><td>“[cc1cc2cc3]”</td><td>“[cc1cc2cc3]”</td></tr>
<tr><td>Not List</td><td>“L”</td><td>“[^cc1cc2cc3]”</td><td>“[^cc1cc2cc3]”</td></tr>
<tr><td>Range</td><td>“r”</td><td>{"[a-zA-Z0-9]","[A-Za-z]","[0-9]"}</td><td>“[\u{FFFF}-\u{FFFF}]” where FFFF are four hexadecimal digits.</td></tr>
<tr><td>Boundary</td><td>“\b”</td><td>“\b”</td><td>“b”</td></tr>
<tr><td>Not Boundary</td><td>“\B”</td><td>“\B”</td><td>“\B”</td></tr>
<tr><td>Look Ahead</td><td>“la”</td><td>“la”</td><td>“(?=cc)”</td></tr>
<tr><td>Negative Look Ahead</td><td>“nla”</td><td>“nla”</td><td>“(?!cc)”</td></tr>
<tr><td>Look Behind</td><td>“lb”</td><td>“lb”</td><td>“(?<=cc)”</td></tr>
<tr><td>Negative Look Behind</td><td>“nlb”</td><td>“nlb”</td><td>“(?<!cc)”</td></tr>
<tr><td>Back Referencing</td><td>“\(re\)...\1”</td><td>“(re)...\1”</td><td>“(re)...\1”</td></tr>
<tr><td>Start</td><td>“^”</td><td>“^”</td><td>“^”</td></tr>
<tr><td>End</td><td>“$”</td><td>“$”</td><td>“$”</td></tr>
</table>

Table 3.4. Transformation of ACTS operator and assertion parameters to regular expressions for icGrep supported syntaxes

For Repetition parameters with n and m, the regular expression generator randomly selects a number between 0 and 10 if their value is “small”, 0 and 100 if their value is “medium” and 0 and 1000 for “large” values. These numbers are set arbitrarily and can be changed if needed. On the other hand, if the Range parameter is set to “true”, the regular expression generator randomly generates two Unicode code points and adds the range between the two code points.

The regular expression generator shuffles the set (RE) for random positioning of each element. After that, if the Back-Referencing parameter is set to “true”, then the regular expression generator randomly selects an element from RE, adds brackets to the element and returns the element to RE and “\1” is appended to the RE set. We do not test for multiple back referencing metacharacters.

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, therefor 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). Since the order of the flags in a command matter in some grep tools, we reserve the order of flags to use for differential testing in the next phase.

Example

Table 3.5 shows an example of a raw test case from ACTS. We will show how the regular expression generator might transform this entity into a concrete test case.

First, the regular expression generator detects the regular expression syntax. If it is set to “off” like in this example, the regular expression has the default Unicode ICU syntax. Next, the regular expression generator extracts the character class set taking in consideration the regular expression syntax. As described previously, the Unicode property character classes are randomly generated.

CC = {“.”, “[[:alnum:]]”, “\W”, “\t”, “\d”, “\D”, “\p{nv=38}”, “\p{QMark}”, “\n{ CANADIAN SYLLABICS Y-CREE LOO }".

RE = {“.{457}”, “\N{CANADIAN SYLLABICS Y-CREE LOO}”, “[\t\d\W]”, “\D+” “\P{QMark}?”, “a(?![[:alnum:]])”, “\p{nv=38}{5,}”}.

Character classes may be used more than once if there are more operators than character classes. On the other hand, if the number of character classes exceed the number of operators in the test frame, the character classes that were not added to the RE set get added to the RE set without an operator. After shuffling the RE set and

adding assertions, Start and End metacharacters, the regular expression adds the appropriate flags to the flag set F.

F = {“-c”, “-i”, “-w”, “-x”, “-t=2”, “-BlockSize=256”}

Finally, the regular expression generator returns the regular expression, i.e.

“^\N{^CANADIAN SYLLABICS Y-CREE LOO$}
[\t\d\W].{8}\P{QMark}?a(?![[:alnum:]])\D+\p{nv=38}{5,}$”.

It also returns the set F.

Parameter	Value	Parameter	Value	Parameter	Value	Parameter	Value
Any	True	Name Property	True	Not List	False	-C	True
Posix Bracket Expression	Alnum	Unicode Codepoint	False	Range	False	-I	True
Word Character	False	Zero Or One	True	Boundary	False	-E	False
Not Word Character	True	Zero Or More	Off	Not Boundary	True	-F	False
Whitespace	False	One Or More	True	Look Ahead	False	-W	True
Not Whitespace	False	Repetition {N}	Large	Negative Look Ahead	True	-X	False
Tab	Ture	Repetition {N,M}	Off	Look Behind	False	Regular Expression Syntax	Off
Digit	True	Repetition {N,}	Small	Negative Look Behind	False	Threads	2
Not Digit	True	Repetition {,M}	False	Back Referencing	False	Block Size	256
Property	Numeric	Alternation	False	Start	True		
Not Property	Binary	List	True	End	True		

Table 3.5. Example of one test case from ACTS for icGrep

3.2.2. String Generator

The next step to creating ready to run test cases is creating input files that create matches to the created regular expression. Borazjany manually wrote the input files for 910 test cases [9]. The idea behind the string generator is to parse the regular expression and create a file containing a match to the regular expression. This step is crucial to a fully automated testing solution. This gives us the ability to not only test for failure producing bugs but also to test for correctness. Figure 3.6 shows the pseudocode for the string generator.

```
S denotes the string generated.
COMPUTE re AST from RE.
FUNCTION GenerateString(AST):
        SWITCH AST:
        Character Class CC THEN:
                APPEND Random character from CC to S.
        Sequence THEN:
                FOR each sub-AST in the sequence:
                        APPEND GenerateString(sub-AST) to S
        Difference THEN:
                RH ← GenerateString(right sub-AST)
                LH ← GenerateString(left sub-AST)
                APPEND RH ∧ ¬ LH to S
        Intersect THEN:
                RH ← GenerateString(right sub-AST)
                LH ← GenerateString(left sub-AST)

                APPEND RH ∩ LH to S
        Alternation THEN:
                RH ← GenerateString(right sub-AST)
                LH ← GenerateString(left sub-AST)

                APPEND RH ∪ LH to S

        Repetition THEN:
                LB ← Lower Repetition Bound.
                IF Bounded Repetition THEN:
                        UB ← Upper Repetition Bound.
                ELSE:
                        UB ← LB + 1000.
                R ← Random integer LB <= R <= UB.
                ITERATE through R:
                        APPEND GenerateString(sub-AST)
        Return list S.
```

Figure 3.6. Pseudocode for the string generator

We included a parameter in ACTS for the input file size. If the size is “small”, the string generator generates a single line matching the regular expression. If the file size is “medium”, Then a file of 10 lines would be generated. Finally, if the size is “large”, the string generator generates a 100-line input file containing 100 matches. These thresholds can be changed if needed depending on the interest of the tester.

The string generator relies on icGrep’s regular expression parser to generate a regular expression syntax tree AST. It is biased to test icGrep by generating files using icGrep regular expression parser but since we are performing differential testing in the end, we minimize the bias factor. If icGrep finds a match in a file and the comparing software does not find a match, then a potential bug is reported.

Now that we have a full concrete test case with an input file to test against, we perform the differential testing on the test case.

3.3. Differential Testing

CoRE implements differential testing by running the generated test cases on icGrep as well as other comparing grep tools to find semantic or logic bugs that do not exhibit explicit erroneous behaviour like crashes or assertion failures.

Why Differential Testing?

At first, it may seem that CoRE did not need the differential testing step. Since we can generate a regular expression and generate an input file, a bug is found if grep fails to find the match.

We mentioned while describing the string generator that we perform differential testing to minimize bias from using icGrep regular expression parser during the string generation. Another reason for differential testing is that some of the generated regular expression may be unmatchable. Unmatchable regular expressions are regular expressions that contain contradictory metacharacters making it impossible to match any string. Looking at one of the shortest unmatchable regular expression like “(?!x)x”, this regular expression contains a contradiction. It says to look for a character “x” that is not “x”. The regular expression generator can be modified to handle unmatchable regular expressions but having unmatchable regular expressions turns out to have some

benefits. There are bugs that cause grep to find a match where it should not have. So, having these tests may help us detect some of these bugs.

The first step of differential testing is identifying the regular expression syntax. icGrep is compared with different grep tools depending on the regular expression syntax. If the syntax is GNU basic regular expression, the comparing grep tool would be GNU grep. If the syntax is GNU Extended regular expression, differential testing is performed on icGrep and GNU egrep, i.e. GNU grep with "-E" flag on. Finally, if the syntax is the default Unicode ICU syntax, icGrep is compared with ICU4C RegexMatcher [33].

After identifying the regular expression syntax, the input for the comparing grep tool may be slightly modified if there are any minor syntax differences. For example, the Unicode name property "\N{name}" is handled differently between icGrep and Unicode ICU grep. icGrep treat the name in the brackets as a regular expression and matches it to the lines of Unicode character names then returns a character class of all characters with a name that matches the regular expression. On the other hand, ICU4C RegexMatcher handles the name as a string and returns a syntax error if no character is found. A regular expression like "\N{SPACE}" match the space character in ICU4C RegexMatcher while it would match 85 characters in icGrep. We alter the regular expression for icGrep by adding the Start "^" and End "$" metacharacters before and after the name. The previous example would look like this "\N{^SPACE$}".

After CoRE performs the differential testing. It reports potential bugs if one of the following situations happen:

- If icGrep or the comparing grep tool crashes.
- If there is a difference in the number of matches found by icGrep and the number of matches found by the comparing grep tool.
- If either icGrep or the comparing grep tool report a syntax error in the provided regular expression.

The first case is straight forward. CoRE runs the test case on icGrep and the comparing grep tool. If any of the processes returns an error code, CoRE reports the test case as a potential bug.

If one of the other two scenarios occurs, it indicates that one of the two tools under differential testing may not be matching what it is supposed to match or matching something it is not supposed to match. CoRE can set a time limit to execute tests and we can decide what to do in case the process was ended because the time limit was reached. In our evaluation process, we simply eliminated tests that made icGrep or the comparing grep reach a time limit. This puts us at a risk of missing bugs that cause grep to hang and having more time would enable us to explore this area to detect hang producing bugs.

Chapter 4. Results and Evaluation

In order to evaluate CoRE, we first run CoRE over different t-way combinatorial configurations and calculate the statement and function coverage rates for each run. We use code coverage to evaluate CoRE following Borazjany's evaluation for testing grep using combinatorial testing [9]. Although several studies debate the strength of the correlation between code coverage and fault detection effectiveness [34, 35, 36], code coverage can give us an indication of a potential quality level of the tests being created. We also measure the fail rate by the formula:

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

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.

There are other evaluation techniques like *mutation adequacy* [37] that might be give a better representation of the effectiveness of CoRE against AFL and AFLFast. But when comparing CoRE to AFL and AFLFast, we would have to generate a large number of mutants and run each testing tool on each of the generated mutants and with each tool taking 24 hours to complete the testing cycle, it was not feasible to incorporate mutation testing.

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. We set 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 %	Elapsed Time
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 4.1. Code coverage and bug rate for different levels of combinatorial testing on CoRE

Table 4.1 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 4.1, 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.

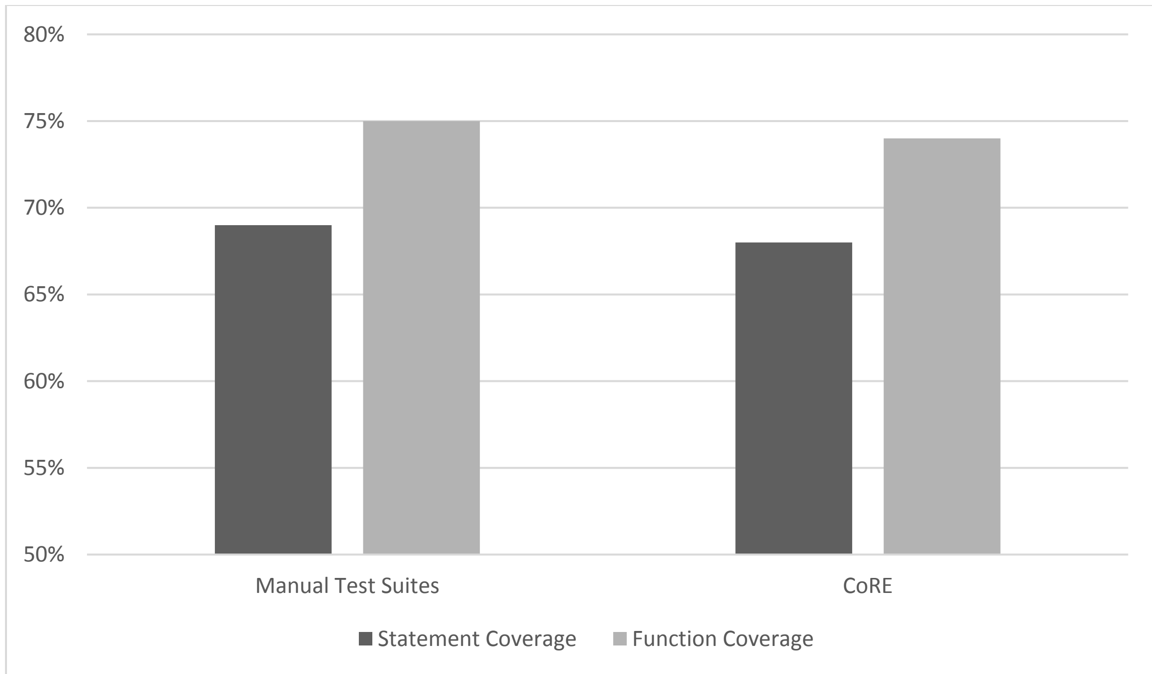

Figure 4.1. Code coverage for manual test suites and CoRE on 5-way combinatorial level

From Figure 4.1, 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.

To evaluate CoRE against AFL and AFLFast, we ran each testing tool 10 times for a period of 24 hours on each run. AFL and AFLFast take a base test suite as input and track the code coverage and alter the input in an attempt to reach new branches [10, 11]. The difference between the two fuzzers is the way the input is mutated to form new tests. We set the empty set as the base test suite for AFL and AFLFast since performance is better when the test suite is minimal according to AFL manual. We ran 5-way combinatorial testing using CoRE for 24 hours iterating through the CSV file as many as needed during that period.

Figure 4.2 Shows code coverage and bug detection over a 24-hour period for CoRE, AFL and AFLFast.

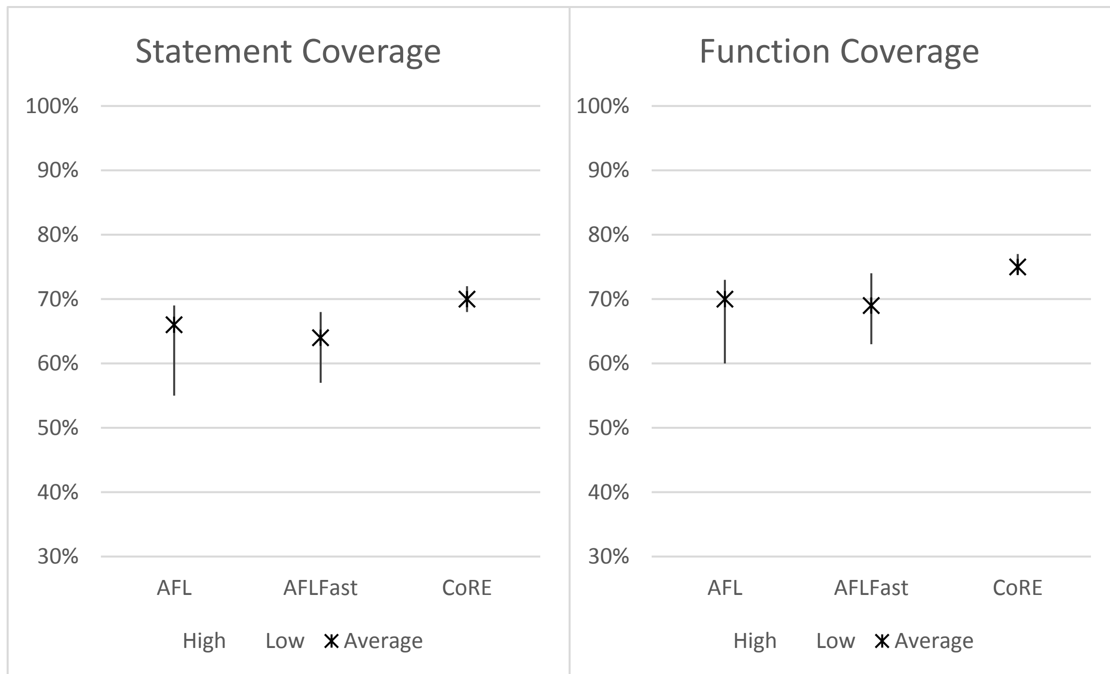

Figure 4.2. Code coverage for AFL, AFLFast and CoRE testing icGrep

Looking at Figure 4.2, we see that CoRE performs better than both AFL and AFLFast in statement coverage and function coverage. CoRE reaches 70% and 75% for statement and function coverage rate respectively while AFLFast scored 64% for statement coverage and 69% for function coverage. AFLFast was not able to find any real bugs on icGrep. AFL on the other hand performed better than AFLFast yet it failed to reach the level of coverage CoRE scored. AFL found 2 unique bugs in icGrep CoRE missed. These bugs occurred when dealing with empty sub-patterns like the alternation between an empty sub-pattern and a non-empty pattern, i.e. “|b”, and matching an empty regular expression in a file. We extended CoRE to incorporate empty sub-patterns for metacharacters that support such patterns.

We ran CoRE on GNU grep comparing test results with the FreeBSD version of grep. We chose GNU grep and FreeBSD grep because they are both very well maintained and finding a bug using CoRE would demonstrate how powerful our approach is. Figure 4.3 shows the statement and function coverage of the manual test suites, AFL, AFLFast and CoRE testing GNU grep. The two graphs show a greater variance in code coverage between the different testing techniques than in icGrep. Although manual tests had a much better score than CoRE, CoRE still managed to score better than AFL and AFLFast in both statement and function coverage.

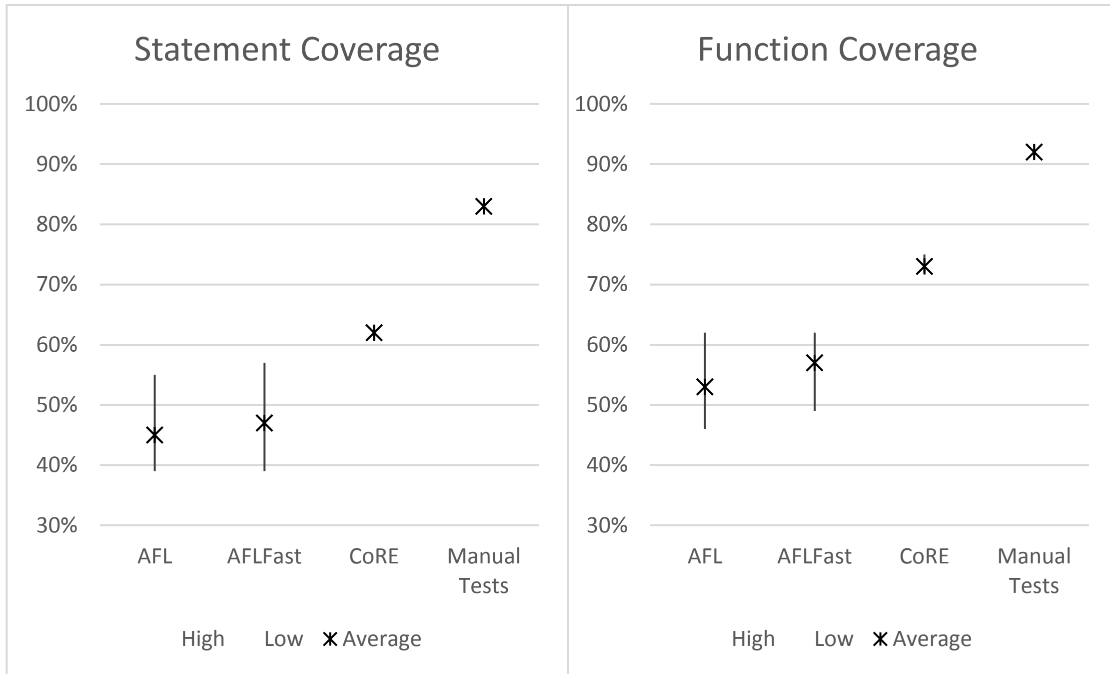

Figure 4.3. Code coverage for AFL, AFLFast, CoRE and Manual Tests testing GNU Grep

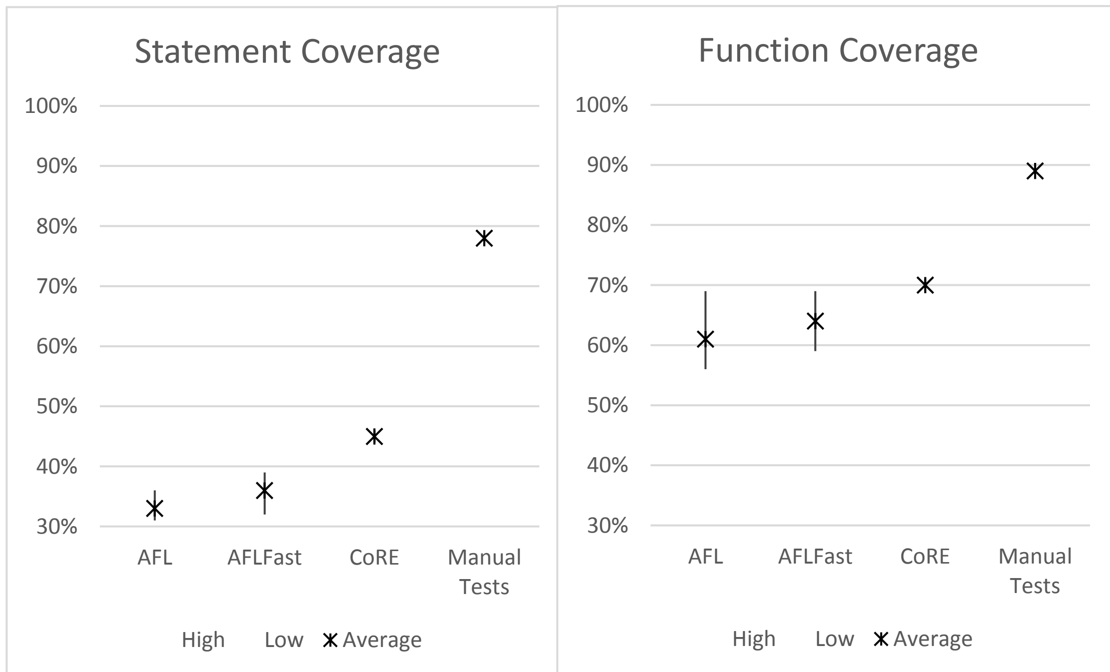

Figure 4.4. Code coverage for AFL, AFLFast, CoRE and Manual Tests testing PCRE Grep

Finally, we evaluated CoRE against PCRE grep's manual test suite as well as AFL and AFLFast. Figure 4.4 shows the results of code coverage for all testing techniques. The two graphs show a similar distribution to the ones in Figure 4.3. The reason why the coverage results for icGrep are different from GNU grep and PCRE grep may be because we measure the coverage of static code and not the JIT code that icGrep relies on. On the other hand, an interesting observation common in Figures 4.2, 4.3 and 4.4 is the difference in the variance in coverage scores over the 10 runs. Both AFL and AFLFast had varied scores over the 10 runs compared to CoRE and manual tests.

Examples of bugs found exclusively by CoRE.

Table 4.2 shows a bug found in FreeBSD grep running on Mac OSX 10.13.1. The three commands were run on GNU grep 3.1 and FreeBSD grep 2.5. The bug occurs whenever there is an optional grouped sub-pattern that does not match anything followed by a back reference.

Command	GNU Grep Output	FreeBSD Grep Output
echo abc | grep –E –color 'a(d?)b?'	abc	abc
echo abc | grep –E –color 'a(d?)b?\1'	abc	abc
echo abc | grep –E –color 'a(d?)b?\1c'	abc	

Table 4.2. Bug in FreeBSD Grep found by CoRE

Another interesting finding was the regular expression "n*(n{113,150}) n+\1" took 220 seconds to match a single line of 591 characters on GNU grep while it takes a fraction of a second on icGrep and ICU RegexMatcher. Since regular expression matching is widely used in web applications for field validation, a performance defect like this could be taken advantage of by hackers to perform a denial-of-service attacks.

For icGrep, CoRE found a bug in icGrep's carry manager. The carry manager controls the carry bits of bitwise operations done by icGrep to find matches. The bug occurs when three of ACTS parameters interact. When setting the block size command flag to either "64" or "128", icGrep throws a failure when matching a regular expression containing a bounded repetition of Unicode properties. Eliminating this bug reduced the fail rate of CoRE by around 50% over different combinatorial levels. Figure 4.5 shows

the decrease in fail rate for each combinatorial level after eliminating the carry manager bug.

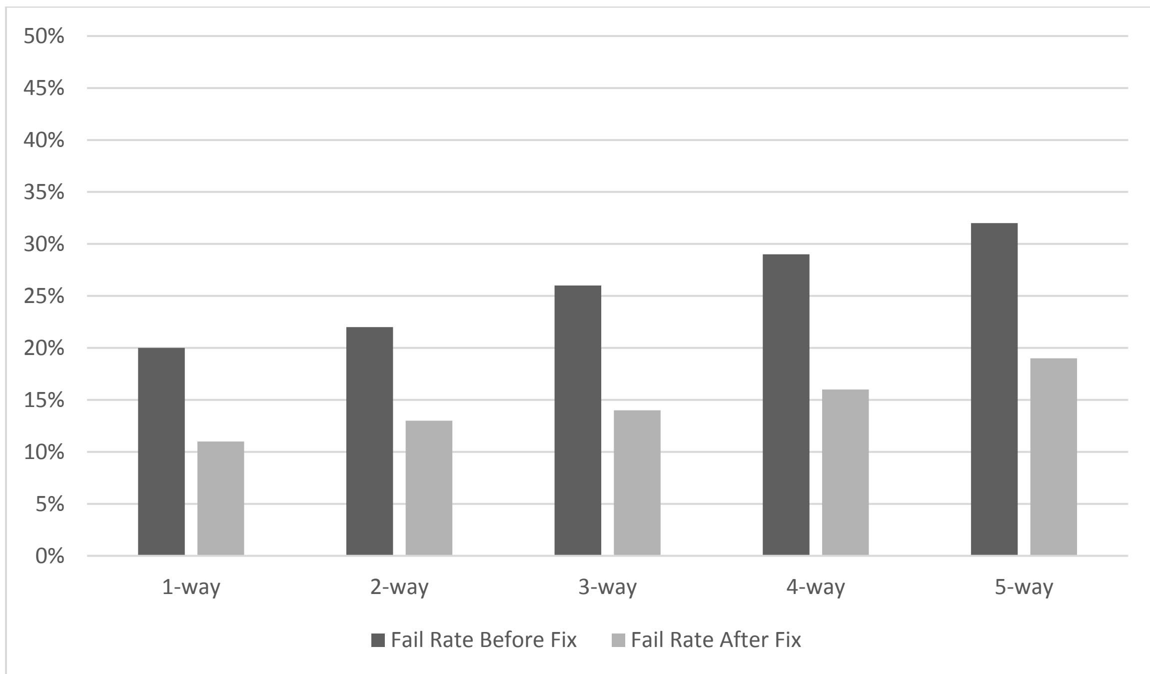

Figure 4.5. Fail rate of CoRE with different combinatorial levels before and after eliminating the carry manager bug in icGrep

Another bug was found in icGrep cache. In order to improve its performance, icGrep stores kernel metadata and compiled object files for reuse. It done is by hashing compiled code for commands with a signature that is the regular expression AST. The bug was in the naming of captured groups. A regular expression like “a(bc)d” would print the AST “(Seq[Name "CC_61" ,Name "\1",Name "CC_64"])”. This meant that the signature for the command would be the same as the signature for matching the regular expression “a(123)d”. To fix this, icGrep now adds a unique name to each capturing group depending on the captured pattern. So for our example “a(bc)d”, icGrep now prints the AST as “(Seq[Name "CC_61" ,Name "\1" =((Seq[Name "CC_62" ,Name "CC_63"])),Name "CC_64"]).

CoRE detected another bug in one of the Unicode properties supported by icGrep. As mentioned earlier, unlike in ICU RegexMatcher, icGrep treats the name “\N{name}” as a regular expression matching all characters matching the name regular expression whereas ICU RegexMatcher treats the name as a fixed string matching a single character. This means if we want to match the Latin digit zero “0”, ICU uses the

syntax “\N”{DIGIT ZERO}” while icGrep’s syntax would be “\N{^DIGIT ZERO$}” to constrain it from matching other zero digits containing “DIGIT ZERO” as part of their name. The bug was in using the Start metacharacter “^” in the name. It was making the property fail to match any character.

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 if a 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.

Chapter 5. 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.

Additionally, we would like to enhance the string generator in CoRE to generate non-matching lines. Currently, the string generator generates lines that match a regular expression. Having non-matching lines would enhance CoRE's performance by detecting bugs that cause grep to find matches where it is not supposed to. We would have to find a design to generate lines that are close to the ones matching the regular expression without actually matching the regular expression. Since we perform differential testing, even if the generated line turned out to match the regular expression, it would not be a problem unless the line matches one grep tool but not the other during the differential testing phase.

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 systems with complex input spaces other than regular expression matching tools.

Chapter 6. Conclusion

In this dissertation, we presented a methodology to reach fully automated combinatorial 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 managed to find bugs that resulted from interactions between metacharacters that were not found by AFL and AFLFast. CoRE does not replace automated testing tools like AFL but is rather an addition to test a different aspect of grep tools. In conclusion, automated combinatorial testing for regular expression matching tools can improve the quality of regular expression matching tools.

Bibliography

[1] K. Thompson, "Programming Techniques: Regular Expression Search Algorithm," vol. 11, no. 6, pp. 419-422, 1968.

[2] B. Kernighan, "A Regular Expressions Matcher," in *Beautiful Code*, O'Reilly Media, 2007.

[3] R. D. Cameron, T. C. Shermer, A. Shrirman, K. S. Herdy, D. Lin, B. R. Hull and D. Lin, "Bitwise data parallelism in regular expression matching," in *The 23rd international conference on Parallel architectures and compilation*, Edmonton, 2014.

[4] T. Stubblebine, Regular Expression Pocket Reference, Sebastopol, CA: O'Reilly & Associates, Inc, 2003.

[5] J. Goyvaerts and S. Levinthan, Regular Expressions Cookbook, O'Reilly Media, 2012, pp. 1-2.

[6] D. R. Kuhn, R. N. Kacker and Y. Lei, Introduction to Compinatorial Testing, Chapman and Hall/CRC, 2013.

[7] M. N. Borazjany, G. Laleh, Y. Lei, R. Kacker and R. Kuhn, "An Input Space Modeling MEthodology for Combinatorial Testing," in *2nd International Workshop on Combinatorial Testing*, Luxembourg, 2013.

[8] D. R. Kuhn and M. J. Reilly, "An investigation of the Applicability of Design of Experiments to Software Testing," in *27th Annual Nasa Goddard/IEEE*, 2012.

[9] M. N. Borazjany, "Applying Combinatorial Testing to Systems with A Complex Input Space," *Thesis (Ph. D),* 2013.

[10] M. Zalewski, "American Fuzzy Lop," [Online]. Available: http://lcamtuf.coredump.cx/afl/README.txt. [Accessed 2 11 2017].

[11] M. Bohme, V. Pham and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *SIGSAC Conference on Computer and Communications Security*, 2016.

[12] 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].

[13] R. D. Cameron, N. Medforth, D. Lin, D. Denis and W. N. Sumner, "Bitwise Data Parallelism with LLVM: The ICgrep Case Study," Zhangjiajie, 2015.

[14] P. Hazel, "Exim and PCRE: How free software hijacked my life," 12 1999. [Online]. Available: http://www.ukuug.org/events/winter99/proc/PH.ps. [Accessed 3 11 2017].

[15] LLVM, "The LLVM Compiler Infrastructure," [Online]. Available: http://llvm.org/. [Accessed 3 11 2017].

[16] D. M. Cohen, S. R. Dalal, J. Parelius and C. Patton, "The Combinatorial Design to Automatic Test Generation," vol. 13, no. 5, pp. 83-88, 9 1996.

[17] M. Grindal, J. Offutt and S. F. Andler, "Combination Testing Strategies: A Survey," vol. 15, no. 3, pp. 167-199, 9 2005.

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

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

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

[21] W. M. McKeeman, "Differential Testing for Software," vol. 10, no. 1, pp. 100-107, 1998.

[22] C. Brubaker, S. Jana, B. Ray, S. Khurshid and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *IEEE Symposium on Security and Privacy*, 2014.

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

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

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

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

[27] G. Argyros, I. Stais, S. Jana, A. D. Keromytis and A. Kiaylias, "SFADiff: Automated Evasion Attacks and Fingerprinting using Black-box Differential Automata Learning," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

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

[29] S. Jana and V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in *IEEE Symposium on Security and Privacy*, 2012.

[30] D. Brumley, J. Caballero, Z. Liang, J. Newsome and D. Song, "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint generation," in *The 16th USENIX Security Symposium*, 2007.

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

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

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

[34] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Transactions on Software Engineering,* vol. 32, no. 8, pp. 608-624, 2006.

[35] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," in *Int'l Workshop on Advances in Model-Based Testing*, 2005.

[36] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *The 36th International Conference on Software Engineering*, 2014.

[37] D. Shin, S. Yoo and D. H. Bae, "Diversity-Aware Mutation Adequacy Criterion for Improving Fault Detection Capability," Chicago, 2016.

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