Comparing White-box and Black-box Test Prioritization

Christopher Henard
University of Luxembourg
christopher.henard@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Mark Harman
University College London
mark.harman@ucl.ac.uk

Yue Jia
University College London
yue.jia@ucl.ac.uk

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

ABSTRACT

Although white-box regression test prioritization has been well-studied, the more recently introduced black-box prioritization approaches have neither been compared against each other nor against more well-established white-box techniques. We present a comprehensive experimental comparison of several test prioritization techniques, including well-established white-box strategies and more recently introduced black-box approaches. We found that Combinatorial Interaction Testing and diversity-based techniques (Input Model Diversity and Input Test Set Diameter) perform best among the black-box approaches. Perhaps surprisingly, we found little difference between black-box and white-box performance (at most 4% fault detection rate difference). We also found the overlap between black- and white-box faults to be high: the first 10% of the prioritized test suites already agree on at least 60% of the faults found. These are positive findings for practicing regression testers who may not have source code available, thereby making white-box techniques inapplicable. We also found evidence that both black-box and white-box prioritization remain robust over multiple system releases.

CCS Concepts

•**Software and its engineering → Software testing and debugging;**

Keywords

Regression Testing, White-box, Black-box

1. INTRODUCTION

Prioritizing regression test suites is important to ensure that testing is effective at revealing faults early in the testing process [25, 26, 33, 57]. Many different test case prioritization techniques have been proposed in the literature [30, 65], yet hitherto, there has been no study that compares white-box and black-box prioritization approaches.

Although white-box techniques have been extensively studied over two decades of research on regression test optimization [25, 30, 47, 65], black-box approaches have been less well studied [35, 36, 46]. Recent advances in black-box techniques have focused on promoting diversity among the test cases, with results reported for test case generation [9, 16, 18, 50] and for regression test prioritization [14, 35, 56, 69]. However, these approaches have neither been compared against each other, nor against more traditional white-box techniques in a thorough experimental study. Therefore, it is currently unknown how the black-box approaches perform, compared to each other, and also compared to the more traditionally-studied white-box techniques.

Black-box testing has the advantage of not requiring source code, thereby obviating the need for instrumentation and source code availability. Conversely, one might hypothesize that accessing source code information would allow white-box testing to increase source code coverage and, thereby, to increase early fault revelation. It has also been claimed that white-box techniques can be expensive [49] and that the use of coverage information from previous versions might degrade prioritization effectiveness over multiple releases [59]. These hypotheses and claims call out for a thorough comparative experimental study over the wide range of possible test prioritization strategies, both white- and black-box.

This paper addresses these questions with an experimental study, reporting results for 20 regression test prioritization techniques, including 10 well-established white-box approaches, which have been thoroughly studied in the past, but also 10 black-box regression techniques that have not. We include, not only methods developed specifically for regression testing, but additionally we ‘port’ techniques, originally invented for test case generation, to regression test prioritization where there is an obvious strategy to achieve such a port. We chose the 10 white-box approaches primarily to provide a baseline for comparison against the relatively newer, and less well-studied, black-box techniques. We evaluate the test effectiveness and efficiency of these 20 regression test prioritization approaches on six versions of five different C programs, widely-used in previous work.

Our study reports upon the relative efficiency and effectiveness of black-box and white-box techniques in terms of execution time and fault detection rate. As well as efficiency and effectiveness, we investigate the similarity between the sets of faults found by the white- and black-box approaches. We also report on the degree to which regression test effectiveness, based on an initial prioritization, degrades over multiple releases of software systems.

We believe that this is the most extensive and inclusive comparative experimental study of test case prioritization techniques yet reported in the literature. Such a study, encompassing multiple black- and white-box approaches, is timely, because recent evidence [35, 46] has suggested that black-box techniques may be promising (yet this claim remains under evaluated).

There is also recent evidence [33] that current regression testing practices involve manual optimization, which is slow, expensive and suboptimal. Furthermore, there is evidence [23, 44] that time available for regression testing is highly limited, making it essential to optimize in order to achieve maximum fault revelation opportunities as early as possible — the purpose of the test prioritization. We found that, on average, white-box techniques prioritize test suites slightly faster than black-box techniques, once the set up costs have been paid. However, the effect is relatively small, and all approaches completed prioritization within a few minutes on standard equipment.

We find that so-called 'additional' approaches outperform 'total' techniques. This is unsurprising, because it replicates and confirms previous studies [47, 59, 68]. We found that Combinatorial Interaction Testing (CIT) and diversity-based techniques (Input Model Diversity and Input Test Set Diameter) perform best among the black-box approaches. Perhaps more surprisingly, we find that the difference between the best performing black- and white-box strategies is relatively slight, with at most 4% difference in the rate of fault detection. Furthermore, we found a high degree of overlap between the faults found by black- and white-box techniques; after only 10% of the test suite has been executed, the techniques already agree on at least 60% of the faults found (rising to 80% after half the test suite has been executed).

These findings suggest that access to source code affords white-box techniques relatively modest advantages and, encouragingly, suggest that prioritization can be effective even in the absence of source code. We also found little evidence that either black- or white-box approaches suffer from degradation over multiple releases; at most 2% degradation was observed for the three best performing black- and white-box techniques, over all 30 program versions studied. This is a further positive finding for practicing regression testing engineers, because it suggests that an initial prioritization can remain robust over multiple releases of the system under test (for both black- and white-box techniques).

The remainder of this paper is organized as follows. Section 2 poses the investigated Research Questions (RQs). Section 3 describes the methodology and the settings used for the experiments. Section 4 analyzes the results of the experiments, answers RQs and discusses threats to validity. Finally, Section 5 examines the related work and Section 6 concludes the paper.

2. RESEARCH QUESTIONS

Arguably the most important issue for any regression test prioritization technique is the rate of fault-revelation; all testing is primarily concerned with the revelation of faults. Therefore, our first three questions concern the fault detection rates of the white-box and black-box approaches.

RQ1. How well do the 10 white-box prioritization methods studied perform in terms of fault detection rate?

Answering RQ1 will help developers know which white-box technique is the most effective. Since this question has been addressed in several previous studies [47, 59, 68], this question can be thought of as a replication study, establishing a baseline for comparison in the remaining questions.

Of course white-box test case prioritization techniques may be inapplicable, for example, where source code is unavailable, or instrumentation is impossible, so a tester may have no choice but to use black-box strategies. It is therefore useful to know how the different black-box techniques perform against each other, motivating our second RQ:

RQ2. How well do the 10 black-box prioritization methods studied perform in terms of fault detection rate?

Where a tester is able to choose both black-box and white-box techniques, he or she will be interested to know which has the best overall performance among all 20 approaches previously investigated, motivating our third RQ:

RQ3. How well do the black-box techniques compare with the white-box ones in terms of fault detection rate?

Knowing the best overall approach is useful, if the tester seeks to use only one technique. However, should it turn out that different approaches find different sets of faults, then it may be useful to use a combination of techniques, to maximize fault revelation overall. Alternatively, should white- and black-box strategies find very similar faults, then a tester can have greater confidence in using one in place of the other. This motivates our fourth RQ:

RQ4. How different are the faults found by the white-box approaches from those found by the black-box techniques?

As a system undergoes change, the initial test ordering might degrade; each change makes the prioritized test suite less suited to the newer system versions. It will be useful for the tester to know the 'robustness' (maintenance of fault revealing potential) of a test suite prioritization technique over multiple releases of the system. It is believed that white-box approaches suffer from degradation [59], and there is no reason to suppose it will be different for black-box techniques; changes will tend to degrade the performance of both prioritization approaches. However, the effect size is unknown, particularly in the case of black-box approaches. This motivates our fifth RQ:

RQ5. How do the white-box and black-box approaches degrade over multiple releases of the system under test?

It has also been recently argued that regression testing scenarios exist where very little time is available for the overall regression testing process [22, 33, 44]. In such situations, the time required for the prioritization process itself becomes paramount; if prioritization takes too long, then it eats into the time available to run the prioritized test suite. In other situations, there is a great deal of time occupied by regression testing, and so this is less important [29, 33]. In order to investigate which techniques would be favorable in situations where limited regression test time is available, we studied the execution time of each algorithm, reporting the results in a final RQ:

RQ6. How do black-box and white-box techniques compare in terms of the required execution time for the prioritization process?

Table 1: The subject programs used in the experiments. For each of them, the number of tests cases, the 6 considered versions together with their size in lines of code and number of embedded faults are presented.

Program	Test Cases	Information	V0	V1	V2	V3	V4	V5
		Version	2.0 *(1996)*	2.2 *(1998)*	2.3 *(1999)*	2.4 *(1999)*	2.5 *(2002)*	2.7 *(2010)*
GREP	440	*Size*	8,163	11,988	12,724	12,826	20,838	58,344
		Faults	-	56	58	54	59	59
		Version	3.01 *(1998)*	3.02 *(1998)*	4.0.6 *(2003)*	4.0.8 (2003)	4.1.1 *(2004)*	4.2 *(2009)*
SED	324	*Size*	7,790	7,793	18,545	18,687	21,743	26,466
		Faults	-	16	18	18	19	22
		Version	2.4.3 *(1993)*	2.4.7 *(1994)*	2.5.1 *(1995)*	2.5.2 *(1996)*	2.5.3 *(1996)*	2.5.4 *(1997)*
FLEX	500	*Size*	8,959	9,470	12,231	12,249	12,379	12,366
		Faults	-	32	32	20	33	32
		Version	3.75 *(1996)*	3.76.1 *(1997)*	3.77 *(1998)*	3.78.1 *(1999)*	3.79 *(2000)*	3.80 *(2002)*
MAKE	111	*Size*	17,463	18,568	19,663	20,461	23,125	23,400
		Faults	-	37	29	28	29	28
		Version	1.0.7 *(1993)*	1.1.2 *(1993)*	1.2.2 *(1993)*	1.2.3 *(1993)*	1.2.4 *(1993)*	1.3 *(1999)*
GZIP	156	*Size*	4,324	4,521	5,048	5,059	5,178	5,682
		Faults	-	8	8	7	7	7

3. METHODOLOGY

3.1 Subject Programs

We use 5 open-source programs written in C, all of which are available from the GNU FTP server [1]: GREP, SED, FLEX, MAKE and GZIP. Test suites for the five programs are available from the Software Infrastructure Repository (SIR) [22]. GREP and SED are popular command-line tools for searching and processing text matching regular expressions. FLEX is a lexical analysis generator, while MAKE controls the compile and build process and GZIP is a widely-used compression utility. This set of programs has been widely used to evaluate test techniques used by researchers in several studies [21, 24, 42, 55, 67].

For each one of these programs, a regression test suite and 6 different versions are used. Table 1 presents, for each version, its identifier and the year that it was released, the size in lines of code calculated using the `cloc` tool [3] and the number of faults contained in each version. The table also records the number of test cases for each program.

3.2 Fault Seeding Process

The faults contained in each version of the programs (see Table 1) were introduced based on mutation analysis [40]. Although faulty versions of the programs are available from SIR, mutation analysis provides more reliable faults than hand seeded ones [11]. Indeed, by analyzing the fault matrices of SIR, we found that these faults are killed by 61% of the test cases in average, making them easy to detect. Besides, the SIR versions are too old to compile and run correctly (we need to execute the test cases). We thus took reliable current versions [1]. These current versions contain configuration scripts that make the program comply with the machine's configuration.

Thus, for each version Vi, $1 \leq i \leq 5$, of each program, a set of carefully selected operators was employed[1], producing faulty programs called mutants. We applied Trivial Compiler Equivalence (TCE) [54] to eliminate equivalent and duplicated mutants. TCE has been found to remove about one third of equivalent mutants, so it is simple and effective.

Naturally, since equivalence is undecidable, this technique cannot remove all equivalent mutants. However, by deploying TCE to remove some equivalent and duplicate mutants, we seek to reduce this threat to validity. We also removed all the mutants that are not killed by the regression test suite (as is common practice [11, 58, 68]).

Although mutation faults have been shown to couple with real faults [11, 43], using it for studying fault revelation in a controlled experimental environment requires precautions to remove potential sources of bias. Several mutation testers, e.g., Andrews *et al.* [11], Ammann *et al.* [10] and Kintis *et al.* [45], claimed that when using mutants to support experimentation, there is a need to filter out those mutants that are easily distinguished from the original program. Such easy-to-kill mutants introduce 'noise' to the mutation score measurement, potentially inflating the value of this metric (and perhaps doing so in an uneven manner, thereby introducing bias). To remove this potential threat to validity, we identified all the subsuming mutants[2] [10, 39, 45] which formed our fault set.

3.3 The 20 Prioritization Approaches Studied

Table 2 depicts an overview of the 20 approaches investigated. For each technique, its acronym, name, prioritization objective and relevant references are listed in the table. The remainder subsection gives details regarding each of these strategies. Although they may use different criteria for prioritizing the test cases, each technique has the unifying overall objective that it aims to order a set $S = \{tc_1, ..., tc_n\}$ of n test cases into an ordered list $L = tc_1, ..., tc_n$.

3.3.1 White-box Techniques

The system under test is instrumented to obtain, for each white-box prioritization technique, the statements, branches and methods executed. The white-box techniques can be distinguished by the source code elements they seek to cover: statements (S), branches (B) or methods (M), and by whether they are total (T) or additional (A). A 'total' technique seeks to maximize the total number of source code elements covered, while an 'additional' technique seeks to cover the greatest number of, hitherto uncovered, source code elements.

[1]We employed the mutant operators set used by Andrews *et al.* [11], i.e., relational, logical, arithmetic, bitwise, statement deletion, unary insertion and constant replacement.

[2]Subsuming mutants are also called 'minimum' mutants [10] and 'disjoint' mutants [45] in the mutation testing literature.

Table 2: The white-box and black-box prioritization techniques considered in the experiments. For each of them, their acronym, name, prioritization objective and main relevant references are presented.

	No.	Acronym	Name	Prioritization Objective	References
White-box	1.	TS	*Total Statement*	Cover the maximum number of statements	[28, 29, 58]
	2.	AS	*Additional Statement*	Cover the maximum number of uncovered statements	[28, 29, 58]
	3.	TB	*Total Branch*	Cover the maximum number of branches	[28, 29, 58]
	4.	AB	*Additional Branch*	Cover the maximum number of uncovered branches	[28, 29, 58]
	5.	TM	*Total Method*	Cover the maximum number of methods	[28, 29, 58]
	6.	AM	*Additional Method*	Cover the maximum number of uncovered methods	[28, 29, 58]
	7.	ASS	*Additional Spanning Statements*	Cover the maximum uncovered dominating statements	[48]
	8.	ASB	*Additional Spanning Branches*	Cover the maximum uncovered dominating branches	[48]
	9.	SD	*Statement Diversity*	Maximize the Jaccard distance between statements	[16, 69]
	10.	BD	*Branch Diversity*	Maximize the Jaccard distance between branches	[16, 69]
Black-box	1.	t-W	*t-wise*	Cover the maximum interactions between t model inputs	[14, 15, 55]
	2.	IMD	*Input Model Diversity*	Maximize the Jaccard distance between model inputs	[36, 37]
	3.	TIMM	*Total Input Model Mutation*	Maximize the number of killed model mutants	[37, 53]
	4.	AIMM	*Additional Input Model Mutation*	Maximize the number of newly killed model mutants	[37, 53]
	5.	MiOD	*Min. Output Diversity*	Minimize the NCD distance between outputs	[56]
	6.	MaOD	*Max. Output Diversity*	Maximize the NCD distance between outputs	[56]
	7.	ID-NCD	*Input Diversity w/ NCD*	Maximize the NCD distance between inputs	[56]
	8.	ID-Lev	*Input Diversity w/ Levenshtein*	Maximize the Levenshtein distance between inputs	[35, 46]
	9.	I-TSD	*Input Test Set Diameter*	Maximize the NCD distance between multisets of inputs	[32]
	10.	O-TSD	*Output Test Set Diameter*	Maximize the NCD distance between multisets of outputs	[32]

These three choices of source code elements (S, B and M) and two different targeting strategies (T and A) yield six different alternative white-box test case prioritization techniques (TS, TB, TM, AS, AB and AM).

A branch or statement, x_1, dominates another, x_2, if (and only if) all executions that involve execution of x_2 also, subsequently involve execution of x_1. The set of non-dominated statements (or branches) is called a 'spanning' set [48], and leads to two approaches: Additional Spanning Statements (ASS) and Additional Spanning Branches (ASB).

Finally, we also considered the diversity of source code elements covered at the statement and branch level. Using the Global Maximum Distances algorithm [36], we order the test cases to maximize the Jaccard distance [36, 38] between the two sets of source code elements covered by consecutive test cases in the sequence. Depending upon whether we measure Jaccard difference between statements covered or branches covered, we obtain two additional (diversity-maximizing) test case prioritization techniques: Statement Diversity (SD) and Branch Diversity (BD).

3.3.2 Black-box Techniques

The first 4 black-box techniques of Table 2 use a model of the program inputs. This is the model typically employed by CIT [51, 55]. The remaining six approaches use either the program inputs or outputs to prioritize the test suite according to diversity measures. More specifically, we define the 10 black-box techniques as follows:

1. t-wise (t-W): The test cases are ordered to maximize the interactions between any t model inputs using a greedy algorithm [55].
2. Input Model Diversity (IMD): Like the white-box technique SD, introduced earlier, but using the set of model inputs instead of the set of statements to calculate the distances.
3. Total Input Model Mutation (TIMM): the constraints of the input model used by CIT are mutated with the operators proposed by Papadakis *et al.* [53]. The test cases are then ordered so that each test case kills the maximum (total) number of model mutants.
4. Additional Input Model Mutation (AIMM): AIMM is the 'additional' version of TIMM (which is 'total'). Each test case at position i in the sequence kills the maximum number of mutants left 'unkilled' by test cases in locations preceding i in the sequence.
5. Min. Output Diversity (MiOD): MiOD computes diversity using the same idea used for the white-box techniques (SD and BD), but using program outputs instead of the set of statements or branches when calculating distances and using the Normalized Compression Distance (NCD) [32, 56] in place of Jaccard distance, with the distances to be minimized. This approach is included as a sanity check only: minimising diversity should hurt prioritization performance if diversity is, indeed, valuable.
6. Max. Output Diversity (MaOD): Like MiOD, but with maximizing the NCD distances.
7. Input Diversity w/ NCD (ID-NCD): Like MaOD but considering the test case *inputs* instead of *outputs*. Although input diversity has not been considered before in the literature of regression testing, output diversity has, so it makes sense to also include, for completeness, input diversity as a black-box test case prioritization criterion. In addition, input diversity has been studied and proved to be effective in model-based testing [35].
8. Input Diversity w/ Levenshtein (ID-Lev): Like ID-NCD, but with a Levenshtein distance [35, 46] instead of NCD.
9. Input Test Set Diameter (I-TSD): Like MaOD, but considering inputs instead of outputs and the NCD metric for multisets [19, 32].
10. Output Test Set Diameter (O-TSD): Like I-TSD, but considering outputs instead of inputs.

3.4 Experimental Settings

The experimental process is depicted in Figure 1. First, the test suite is executed on the initial version of the system under test (V0). This produces the input data for all 20 prioritization techniques presented in the previous section. Each approach produces an ordering of the regression test suite, based on the input, output and coverage information obtained from execution of the test suite on V0. The input models used by the black-box approaches were taken from the previous work on CIT by Petke *et al.* [55].

The resulting prioritized (ordered) test suite is then evaluated on the five subsequent versions (V1 to V5), into which faults have been seeded using mutation testing. None of the test techniques has any information about the faults seeded, nor do any of the approaches evaluated obtain any other information from versions V1 to V5; all information used in prioritization by all techniques is obtained from the initial version, V0. To assess each prioritized test suite, we use the standard measurement for assessing the rate of fault revelation: the Average Percentage of Faults Detected (APFD), which is calculated according to the following formula [68]:

$$\mathrm{APFD}(\pi) = \frac{\sum_{i=1}^{n-1} FT_i}{nm} + \frac{1}{2n},$$

where π is an ordering of the regression test suite, n and m are respectively the number of test cases and the number of faults in the program's version and FT_i the total number faults exposed after executing the ith test case of π.

The process of executing an approach on V0, evaluating it on V1-V5 and calculating the APFD is repeated 100 times to account for the stochastic nature of the prioritization algorithms considered. Stochastic behavior is observed where the technique has to break a tie (where test cases in the ordering have identical values for the guiding objective used to prioritize the sequence). To break such ties, a random choice is made. As a result of this tie breaking schema, different results can be obtained on each execution. By repeating the experiments 100 times, we collect a set of different outcomes for each prioritization approach and each system under test. This sample of all possible executions allows us to use standard inferential statistical techniques to investigate both the significance and the effect size of the differences observed between the algorithms' performance (see Section 3.5).

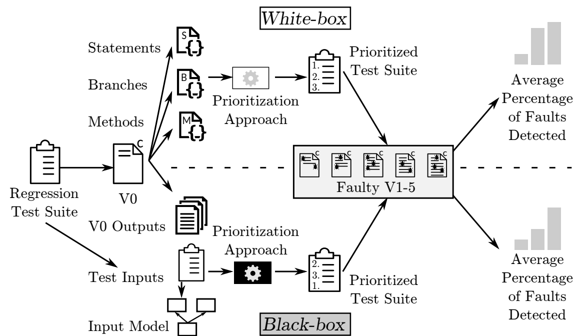


Figure 1: Evaluation of a prioritization approach on a program. First, the test suite is executed on the initial version V0 to produce code coverage information and outputs. Then, the test suite is prioritized, executed on V1 to V5, and the mean percentage of faults detected is calculated.

To evaluate the similarity in the faults found by the white- and black-box approaches (in answer to RQ4), we calculate the Jaccard coefficient [36, 38] between the sets of faults revealed by each approach after executing a given fraction of the test suite. More formally, let FT_i^W and FT_i^B respectively be the set of faults exposed by a white- (W) and black-box (B) prioritization approach, after executing the ith test case in the prioritized order. The similarity between the faults found by the two approaches, W and B, after executing i test cases is calculated as follows:

$$J(FT_i^W, FT_i^B) = \frac{|FT_i^W \cap FT_i^B|}{|FT_i^W \cup FT_i^B|}.$$

Note that $\forall x, y, J(x,y) \in [0,1]$ and that $J(x,y) = 0$ indicates that the two sets x and y are completely different, while $J(x,y) = 1$ indicates that the two sets (of faults, in our case) are identical.

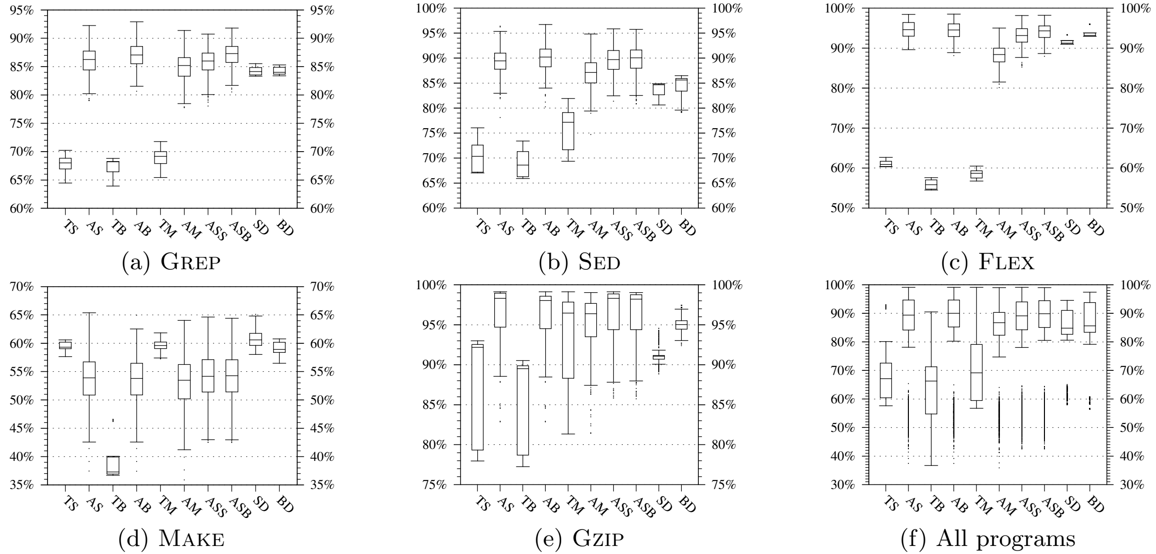
All experiments were executed on a Linux 3.11.0-18-generic laptop with an Intel i7-2720QM Quad Core 2.40GHz CPU and 4GBs of RAM. The programs were compiled with `gcc` [4] 4.8.1 and coverage information (used by the white-box approaches) was obtained by instrumenting V0 using the `gcov` tool [5], which is part of the `gcc` utilities. The prioritization techniques were implemented in Java and we used the `bzip2` compressor [2] for NCD, as it is a 'good real-world compressor' [63]. For RQ6, we recorded the execution times for each of the 20 algorithms investigated using the `time` software [7] and considering the 'real' elapsed time between invocation and termination. Finally, we performed statistical analysis using `R` [6], as detailed in the following section.

3.5 Inferential Statistical Analysis

Following the guidelines on inferential statistical methods for handling randomized algorithms [12, 34], we assess the statistical significance of the differences between the APFD values recorded for each prioritization approach using the unpaired two-tailed Wilcoxon-Mann-Whitney test. We use this test since we have no evidence to support the belief that the results exhibit a Gaussian (normal) distribution, and therefore, Wilcoxon-Mann-Whitney is more appropriate than a parametric test, since it makes fewer assumptions about the distribution. We use an unpaired test because there is no relationship between each of the 100 runs (they simply randomly choose between different tie-breaking choices), and we use a two-tailed test because we make no assumptions about which technique outperforms the other.

To cater for the fact that we use multiple statistical testing techniques, we report the p-values (uncorrected). This facilitates correction using either Bonferroni or some other, less conservative, correction procedure. However, we observe that since we found extremely small p-values, even the Bonferroni correction, with all its conservatism, would not alter conclusions about statistical significance (at either the 5% or the 1% significance levels). In any case, as has been observed elsewhere [34], comparative executions of two different algorithms are likely to produce arbitrarily small p-values for an arbitrarily large number of executions; so long as N is sufficiently large, p will be sufficiently small, given that there are differences between the two algorithms.

Therefore, the more important (and meaningful) statistic for comparing the two different algorithms lies in the effect size, which is captured using the non-parametric Vargha and Delaney effect size measure [62], $\hat{\mathrm{A}}_{12}$.



(a) Grep (b) Sed (c) Flex (d) Make (e) Gzip (f) All programs

Figure 2: RQ1: Average percentage of faults detected for each white-box approach on V1 to V5.

The Vargha and Delaney measure is recommended by Arcuri and Briand and by Wohlin *et al.* [12, 64]. It is a simple and intuitive measure of effect size: it denotes the probability that one technique will outperform another; the greater the probability, the greater the effect size. $\hat{A}_{12}(x, y) = 1.0$ means that, in the sample, algorithm x always outperforms algorithm y (and so the expected probability that x outperforms y is 1.0). $\hat{A}_{12}(x, y) = 0.0$ means that y outperforms x (for every member of the sample and, by inference, with 1.0 probability in the population from which the sample was taken). Similarly, values between 0.0 and 1.0 can be interpreted as probabilities, inferred for the population (of all possible executions of the two algorithms), based on the sample (of executions of the two algorithms).

4. RESULTS

4.1 RQ1: White-box Approaches

The APFD values observed for each technique are presented in Figure 2. For each program (Figure 2(a) to Figure 2(e)), the box plots show the distribution of the 500 APFDs (100 orderings × 5 versions) obtained by each white-box algorithm, listed horizontally across the figure. Figure 2(f) summarizes by showing the distribution of the APFDs for each approach over all the versions of all programs.

Regarding Grep, Figure 2(a), the three best approaches are AS, AB and ASB with a median APFD approximately equal to 87%. The approaches featuring a 'total' strategy perform the worst with APFDs lower than 70% while the two last approaches, those exploiting diversity, are the most robust (exhibiting variance) with an APFD close to 85%. Similar observations can be made for Sed, Flex, and Gzip, but for Make, the best approaches are TS, TM and SD, contradicting the previous results. Here it should be noted that usually additional strategies are more effective [68], because they cover more code at a given time, but total ones could be better in some cases, because they traverse longer paths and cover the same statements with different values.

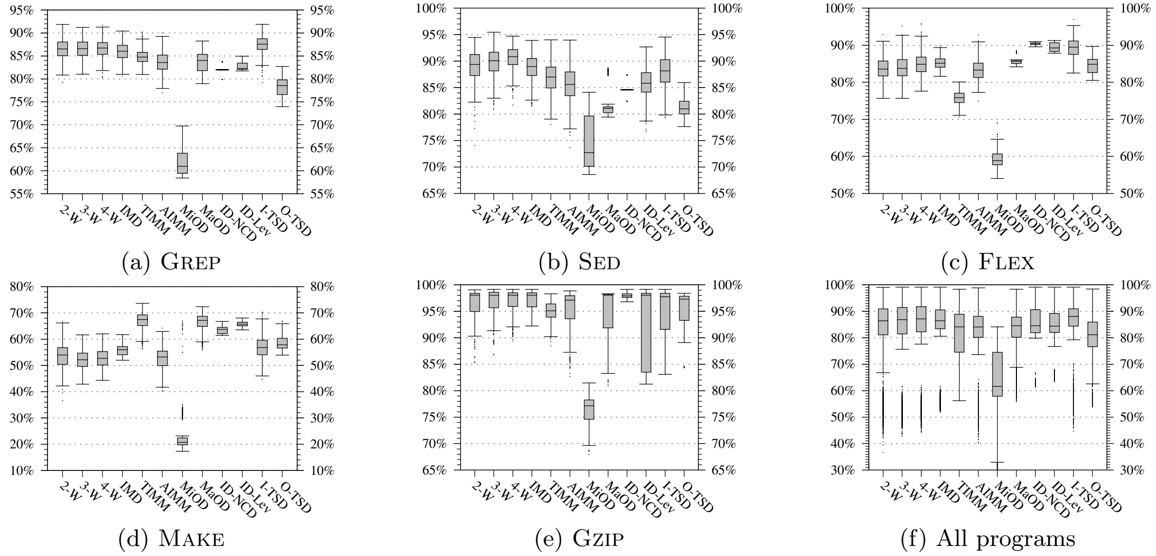
The lower of the two triangular tables in Table 3 presents the results of inferential statistical analysis. Focusing on the white cells for AS, i.e., those of coordinate (2, y) or (x, 2), we can observe that all the p-values are highly significant, except against the approaches No. 4, 7 and 8, i.e., AB, ASS and ASB. Similarly, the effect size measure $\hat{A}_{12}$(line, column) when comparing any white-box approach against AS, i.e., coordinates $(x, 2)$, is lower than 0.5 except for AB, and ASB, meaning that AS perform better in more than 50% of the cases. Overall, the white-box techniques AB, ASS and ASB produce the best fault detection rates.

4.2 RQ2: Black-box Approaches

Figure 3 records the distribution of the APFDs obtained by the black-box approaches for each program (Figure 3(a) to 3(e)) and for all of them together (Figure 3(f)). For Grep, I-TSD is the approach yielding the highest APFD, with a median of 88%. The second best technique is t-W, for which the 4-W version gives the best result (as expected, higher strength combinatorial testing outperforms lower strength). The third best approach on this program is IMD.

All the approaches range from 77% to 92%, except for the sanity check MiOD which provides poor fault detection capability (providing further evidence for the importance of diversity). For Sed, the same three approaches are found to lead. For Flex, the best technique is ID-NCD, with an APFD of 91% and low variance. ID-NCD is closely followed by ID-Lev and I-TSD, indicating that input diversity appears to play an important role on this program. For Make, TIMM, MaOD and ID-Lev are the three best-performing approaches, but t-W, the combinatorial testing approach, is the worst approach. Finally, for Gzip, all the approaches except TIMM and MiOD appear to provide similar performance.

The gray cells of Table 3 record the p-values/$\hat{A}_{12}$ measures for all pairwise black-box approach comparisons. Note that the t-W approach gathers results for $t = 2$ to 4 merged together, but the companion website (see Section 4.7) records the values for each approach separated.



(a) Grep (b) Sed (c) Flex

(d) Make (e) Gzip (f) All programs

Figure 3: RQ2: Average percentage of faults detected for each black-box approach on V1 to V5.

Table 3: RQ1 and RQ2: This table comprises two entirely separate triangular tables of results. The lower triangle presents results comparing white-box techniques, while the upper triangle shows results comparing black-box approaches. Shading is used to help indicate the difference between the two triangles. In each triangular table, each cell contains a p-value and a $\hat{\mathbf{A}}_{12}$(Line, Column) effect size measurement. The key at the bottom of the table translates numeric cell positions into algorithm names. For example, Line 2, Column 1 is in the lower triangle (which concerns the white-box techniques). It records the p-value/$\hat{\mathbf{A}}_{12}$ (of $0^*/0.75$) for AS vs TS. This indicates that white-box technique Additional Statement (AS) has an inferred 0.75 probability of outperforming white-box technique Total Statement (TS). By contrast, Line 5 Column 9 is in the upper triangle, and therefore reports the results for black-box techniques (in this case MiOD vs I-TSD). Further results can be found on the companion website at: `http://henard.net/research/regression/ICSE_2016/`.

Approach No.	1	2	3	4	5	6	7	8	9	10	
1	-	0.07/0.51	0.0*/0.59	0.0*/0.61	0.0*/0.84	0.0*/0.60	0.0*/0.53	8.1E-15/0.56	0.0*/0.47	0.0*/0.64	**1**
2	0.0*/0.75	-	0.0*/0.57	0.0*/0.61	0.0*/0.85	0.0*/0.58	0.10/0.51	5.7E-08/0.54	6.3E-12/0.44	0.0*/0.64	**2**
3	0.0*/0.40	0.0*/0.18	-	0.69/0.50	0.0*/0.87	0.04/0.48	1.6E-12/0.44	4.9E-05/0.47	0.0*/0.39	2.5E-08/0.55	**3**
4	0.0*/0.75	0.25/0.51	0.0*/0.82	-	0.0*/0.83	0.11/0.49	0.0*/0.43	4.2E-11/0.45	0.0*/0.36	2.8E-16/0.57	**4**
5	0.50/0.51	0.0*/0.29	0.0*/0.64	0.0*/0.29	-	0.0*/0.10	0.0*/0.10	0.0*/0.09	0.0*/0.14	0.0*/0.17	**5**
6	0.0*/0.73	0.0*/0.40	0.0*/0.79	0.0*/0.38	0.0*/0.69	-	5.0E-08/0.46	1.1E-07/0.46	0.0*/0.37	0.0*/0.60	**6**
7	0.0*/0.75	0.10/0.49	0.0*/0.81	0.0*/0.48	0.0*/0.71	0.0*/0.59	-	0.44/0.51	1.8E-06/0.46	0.0*/0.63	**7**
8	0.0*/0.75	0.39/0.51	0.0*/0.82	0.78/0.50	0.0*/0.72	0.0*/0.62	0.01/0.52	-	0.0*/0.41	0.0*/0.63	**8**
9	0.0*/0.76	0.0*/0.38	0.0*/0.83	0.0*/0.36	0.0*/0.74	0.07/0.49	0.0*/0.39	0.0*/0.37	-	0.0*/0.66	**9**
10	0.0*/0.77	7.8E-14/0.44	0.0*/0.83	0.0*/0.43	0.0*/0.72	0.03/0.52	4.5E-09/0.45	0.0*/0.43	0.0*/0.58	-	**10**

*p-value lower than 2.2E-16.
White-box: 1. TS 2. AS 3.TB 4. AB 5. TM 6. AM 7. ASS 8. ASB 9. SD 10. BD
Black-box: 1. t-W 2. IMD 3. TIMM 4. AIMM 5. MiOD 6. MaOD 7. ID-NCD 8. ID-Lev 9. I-TSD 10. O-TSD

Overall, statistical analysis confirms the box plots observations: by comparison with the white-box techniques, there is a greater degree of variation in the performance of the different approaches over different programs. Nevertheless, based on the subjects we have studied, our results suggest that I-TSD, t-W (for $t = 4$) and IMD offer the highest fault detection rates among the black-box techniques.

4.3 RQ3: White-box vs Black-box

In this section, the three best white-box and black-box approaches, i.e., AB, ASS, ASB, 4-W, IMD and I-TSD are compared against each other.

In this respect, Figure 4 shows the distribution of the APFDs for these techniques on all the programs. The line with squared points shows the mean APFDs. In terms of median values, the highest difference occurs between ASD and IMD and represents approximately 4%.

In terms of mean values, this maximum difference between the techniques is only 2%. I-TSD is the black-box approach that best competes with the white-box techniques. We observe less than 1% difference between I-TSD and ASS, both in terms of median and mean values, while a difference of less than 2% is observed when comparing I-TSD with ASD. Black box techniques are, thus, surprisingly effective.

Table 4: RQ3: white-box vs black-box. Statistical comparison (p-values/$\hat{A}_{12}(\{4,7,8\},\{1,2,9\})$) between the 3 best black-box and white-box approaches.

		Black-box		
		1. 4-W	**2. IMD**	**9. I-TSD**
White-box	**4. AB**	0.0*/0.58	0.50/0.60	0.0*/0.57
	7. ASS	0.25/0.56	0.0*/0.58	0.0*/0.55
	8. ASB	0.0*/0.58	0.0*/0.60	0.0*/0.57

*p-value lower than 2.2E-16.

The statistical results for the comparison between these 6 approaches are presented in Table 4. Regarding the p-values, all the comparisons white-box vs black-box are highly significant, i.e., lower than 1%, except when comparing AB against IMD and ASS against 4-W. It means that the resulting APFDs are quite different when comparing any of these two but (AB, IMD) and (ASS, 4-W). The effect size measure evaluates white-box against black-box techniques, i.e., $\hat{A}_{12}(x, y)$ with $x \in \{4,7,8\}$ and $y \in \{1,2,9\}$. With respect to these values, the white-box techniques perform better against the 3 black-box techniques from only 56% to 60% of the time.

To conclude, the differences between the APFDs of the white- and black-box techniques range from 2% to 4% and white-box approaches perform better than black-box ones in 56 to 60% of the cases. If the tester has to choose a single technique out of the 20 studied, then the results indicate that this should be ASB, but the differences are small, compared to the variance over the programs studied. Therefore, overall, we find that the black-box approaches are surprisingly competitive with the white-box ones, given that they have less information available, i.e., no structural information upon which to prioritize test suites.

4.4 RQ4: Similarity in the Faults Found

Figure 5 shows the Jaccard coefficients for all the versions of all the programs after executing 0, 10, ... 100% of the test suite. Each box plot represents, for a given percentage of the test suite executed, the distribution of the 22,500 similarity coefficients (5 programs × 5 versions × 9 comparisons (white vs black) × 100 runs). The line with squared points shows the mean Jaccard coefficient. Figure 5 reveals that both mean and median similarity have already risen above 0.6 after executing only 10% of the test suite.

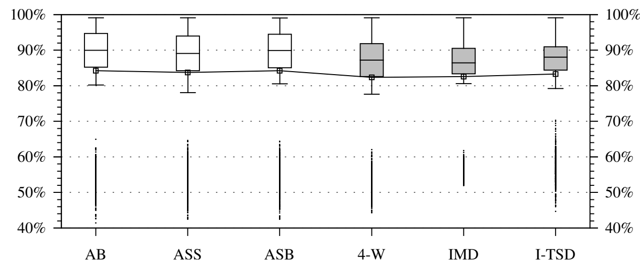


Figure 4: RQ3: Average percentage of faults detected for the three best white- and black-box techniques on all the programs. The line connecting all box plots depicts the mean value of APFD (for comparison with the median value shown in the box).

That is, after executing the first 10% of the test cases, there is already 60% agreement on the faults found by black-box versus white-box testing. This rises to 80% after executing half the test suite.

Table 5 records the mean Jaccard similarity coefficient, over the 100 runs, for each program, version and considered approach at an execution fraction of 10% of the test suite. For Grep, Sed and Make the similarity between black-box and white-box techniques follows the overall trend, while for Gzip the similarity results are considerably 'above trend', ranging from 86% to 99% similarity. With respect to Sed, the results are considerably below trend, ranging from 0.10 to 0.31. Overall we find evidence to suggest that both black-box and white-box techniques find similar sets of faults, over most of the program studied.

4.5 RQ5 : Degradation over Versions

Figure 6 shows, for each of the top 3 white-box and black-box approaches, the degradation (over the 5 versions) of the programs. Thus each box plot for a particular version records 1,500 APFDs (5 programs × 3 approaches × 100 runs). From this figure, we can see that the APFD is surprisingly robust over the 5 versions. In the worst case it increases only slightly (by approximately 2% from V1 to V5) for I-TSD (see Figure 6(f)). This surprising robustness holds for both the white-box and black-box techniques.

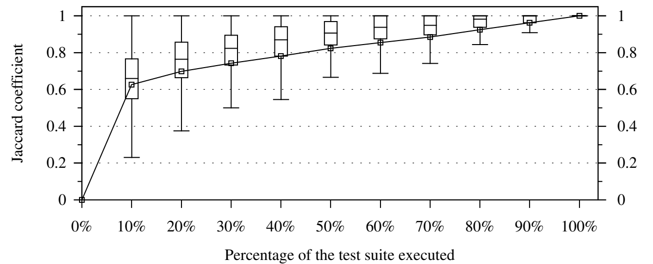


Figure 5: RQ4: Similarities in the faults found by the white- and black-box approaches on all the programs and for all the versions. The line is the mean.

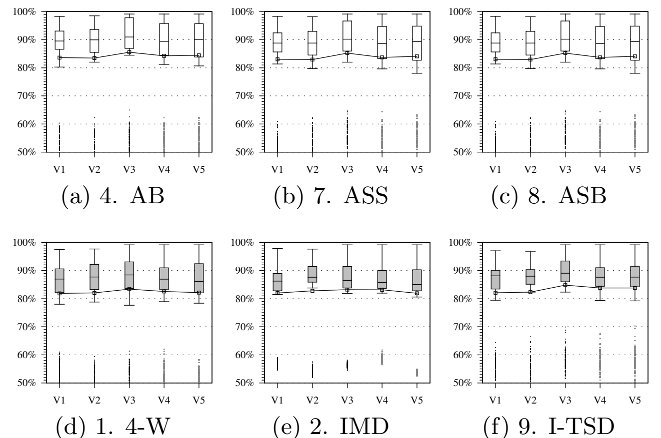


Figure 6: RQ5: Degradation of the average percentage of faults detected over versions for each top 3 white-/black-box approaches on all the programs.

Table 5: RQ4: Mean Jaccard coefficients comparing the similarities between the faults found by the white- and black-box approaches when executing 10% of the test suite over the 100 runs.

		1. 4-W					2. IMD					9. I-TSD				
		Grep	Sed	Make	Flex	Gzip	Grep	Sed	Make	Flex	Gzip	Grep	Sed	Make	Flex	Gzip
4. AB	V1	0.69	0.75	0.66	0.20	0.87	0.67	0.68	0.58	0.21	0.87	0.67	0.72	0.72	0.19	0.92
	V2	0.67	0.69	0.69	0.11	0.86	0.65	0.64	0.62	0.13	0.87	0.65	0.66	0.76	0.12	0.91
	V3	0.70	0.69	0.72	0.31	0.99	0.69	0.64	0.62	0.31	0.99	0.68	0.66	0.78	0.29	0.98
	V4	0.65	0.61	0.68	0.22	0.99	0.65	0.58	0.60	0.24	0.99	0.65	0.60	0.74	0.21	0.98
	V5	0.62	0.69	0.67	0.23	0.99	0.63	0.66	0.59	0.25	0.99	0.63	0.68	0.73	0.22	0.98
7. ASS	V1	0.68	0.73	0.65	0.18	0.88	0.67	0.69	0.57	0.17	0.87	0.67	0.71	0.70	0.18	0.90
	V2	0.66	0.64	0.68	0.11	0.87	0.65	0.64	0.60	0.12	0.87	0.65	0.63	0.73	0.13	0.89
	V3	0.70	0.64	0.70	0.25	0.99	0.69	0.64	0.60	0.24	0.99	0.68	0.63	0.76	0.25	0.98
	V4	0.64	0.60	0.66	0.21	0.99	0.64	0.61	0.58	0.20	0.99	0.64	0.60	0.71	0.20	0.98
	V5	0.61	0.67	0.65	0.21	0.99	0.61	0.67	0.57	0.20	0.99	0.62	0.68	0.70	0.20	0.98
8. ASB	V1	0.67	0.74	0.65	0.20	0.88	0.67	0.69	0.57	0.20	0.87	0.66	0.71	0.72	0.18	0.90
	V2	0.66	0.69	0.69	0.10	0.88	0.64	0.66	0.61	0.11	0.87	0.64	0.66	0.75	0.12	0.89
	V3	0.69	0.69	0.72	0.27	0.99	0.68	0.66	0.62	0.26	0.99	0.67	0.66	0.77	0.26	0.98
	V4	0.66	0.59	0.67	0.23	0.99	0.64	0.58	0.59	0.22	0.99	0.65	0.59	0.73	0.21	0.98
	V5	0.62	0.68	0.66	0.22	0.99	0.62	0.66	0.58	0.22	0.99	0.63	0.67	0.73	0.21	0.98

White-box: 4. AB 7. ASS 8. ASB *Black-box:* 1. 4-W 2. IMD 9. I-TSD

Table 6: RQ6: Execution time in seconds for the three best white-box and black-box prioritization approaches on the 5 programs. The gcc, gcov and Span. S/B columns respectively denote the compilation time, instrumentation time and spanning statements/branches calculation time. The prioritization times (P.) are the mean μ/standard deviation σ over the 100 runs. MDL is the time required to make the input model and Dist. is the time required to calculate the Jaccard distances used by IMD. Finally, Tot. represents the total execution time per approach (its calculation per approach is detailed in the key below the table).

	White-box									Black-box							
	Setup			4. AB		7. ASS		8. ASB		Setup		1. 4-W		2. IMD		9. I-TSD	
	gcc	gcov	Span. S/B	P. μ/σ	Tot.[1]	P. μ/σ	Tot.[2]	P. μ/σ	Tot.[3]	MDL	Dist.	P. μ/σ	Tot.[4]	P. μ/σ	Tot.[5]	P. μ/σ	Tot.[6]
Grep	3.1	32.8	181.6/65.7	20.8/1.0	56.7	0.1/0.3	217.6	0.1/0.3	101.7	t_1	2.0	364.8/2.8	t_1+364.8	1.5/0.2	t_1+3.5	486.1/27.5	486.1
Sed	1.1	12.6	30.9/5.2	3.8/0.4	17.5	0.1/0.3	44.7	0.1/0.3	19.0	t_2	1.3	314.9/21.7	t_2+314.9	0.7/0.1	t_2+2.0	290.2/9.7	290.2
Flex	0.5	38.4	680.2/143.7	35.6/0.8	74.5	0.2/0.4	419.3	0.1/0.3	172.7	t_3	3.4	83.6/2.1	t_3+83.6	1.9/0.3	t_3+5.3	450.2/45.2	450.2
Make	3.6	17.8	418.5/177.7	2.5/0.5	23.9	0.1/0.1	440.0	0.1/0.1	199.2	t_4	0.4	6.1/0.5	t_4+6.1	5.3/0.5	t_4+5.7	42.7/41.9	42.7
Gzip	0.3	7.6	36.8/13.5	1.0/0.1	8.91	0.1/0.1	44.81	0.1/0.1	21.51	t_5	0.2	113.4/14.0	t_5+113.4	0.3/0.5	t_5+0.5	35.0/8.6	35.0
Σ	8.6	109.2	1,348	63.7/-	181.51	0.6/-	1,166.4	0.5/-	514.1	Σt_k	7.3	882.8/-	Σt_k+882.80	8.7/-	Σt_k+17.00	1,304.2/-	1,304.2

[1] gcc + gcov + P. μ
[2] gcc + gcov + Span. S + P. μ
[3] gcc + gcov + Span B + P. μ
[4] MDL + P. μ
[5] MDL + Dist. + P. μ
[6] P. μ

4.6 RQ6 : Performance

Table 6 records the execution time in seconds for the top 3 white- and black-box methods on V0 of the programs. It presents the mean execution time (P. μ) and the standard deviation (P. σ) over the 100 independent runs performed per approach. As well as the prioritization time itself, there is also the time to calculate the input for some of these approaches, denoted as setup time. Indeed, for the white-box, V0 is instrumented to collect the coverage information per test case. For ASS and ASB, the statements and branches need to be processed to keep only the spanning ones.

Regarding the black-box approaches, IMD takes as input a distance matrix between any two model inputs. Thus, Table 6 also records the setup time. Regarding the white-box ones, gcc, gcov and Span. S/B columns respectively denote the compilation time, instrumentation time and spanning statements/branches calculation time. For the black-box approaches, MDL is the time required to make the input model and Dist. is the time to calculate the Jaccard distances used by IMD. Note that preparing the input of the prioritization approaches is done only once. Finally, the columns Tot. of the table represent the total execution time per approach.

The mean white-box prioritization times (P. μ) range from less than 1 second to 36. AB requires more time than ASS and ASB since it has to consider more branches, i.e., code coverage information as input. The setup time is negligible with respect to compilation and instrumentation, from less than a second to 40, but is slightly more important when spanning information has to be calculated, with about 11 minutes for calculating the spanning statements of Flex.

The black-box approaches, overall, take more time than the white-box ones to prioritize the test suite. The mean prioritization times (P. μ) range from less than 1 second for IMD to 486.10 seconds, which represents about 8 minutes for I-TSD on Grep. 4-W and I-ITSD require more time than IMD to prioritize the test suite, since they (respectively) have to calculate the combinations between any 4 model inputs and the multiset NCD metric on the inputs. Regarding the setup, the time to make the model is undetermined, and the calculation of the distances is almost null.

We also calculated the mean prioritization time to the test suite execution time. For AB, ASS, ASB, 4-W, IMD and I-TSD, these are 83.5%, 0.9%, 0.8%, 1,484.8%, 12.6% and 1,686.6%. More details are given on the companion website.

4.7 Threats to Validity

In order to minimize the threat from randomized computation, we ran all algorithms 100 times, using inferential statistics to compare results. In order to facilitate investigation of these potential threats due to implementation details, and to support replication, we make all of our code and data sets from this paper together with some additional results available on the companion website:

`http://henard.net/research/regression/ICSE_2016/`.

We use mutation testing in to assess the rate at which faults are detected by each prioritization technique. As noted in Section 3.2, mutation faults have been shown to be coupled with real faults, making this a reasonable approach to controlled experiment. We used Trivial Compiler Equivalence (TCE) [54] to reduce the impact of both equivalent and duplicate mutants, and also removed all subsumed mutants, to cater for the potential bias due to trivial mutants. We also believe that the 30 versions studied here (six versions of each of the five programs) are a sufficient basis to draw comparative conclusions. Nevertheless, further experimentation with other programs and sets of faults would help to increase the generalizability of the conclusions.

5. RELATED WORK

There is a large body of research on regression-testing techniques [30, 65], covering test suite selection [13, 31, 52], augmentation and regeneration [8, 60, 66], minimization [33, 65] and prioritization [28, 68]. In this related work section, we focus on test case prioritization, which can be divided into two categories: white-box and black-box approaches.

White-box prioritization: The most widely studied techniques are those using dynamic coverage information [49]. Rothermel *et al.* [58] and Elbaum *et al.* [28] introduced the two main white-box strategies: the 'Total' and 'Additional' approaches we studied in this paper. Elbaum *et al.* [27] leverage information regarding the cost of test cases and the severity of faults into a prioritization approach. Jiang *et al.* [41] investigated the use of adaptive random prioritization, showing that additional approaches outperform total ones, while, more recently, Zhang *et al.* [68] proposed probabilistic models that combine varying degrees of the total and additional strategies. Li *et al.* [47] investigated the use of greedy, hill climbing and genetic algorithms, finding greedy techniques to be more effective.

Marre and Bertolino [48] used dominance relations between coverage elements in order to construct spanning sets and used them for prioritization. Zhou *et al.* [69] and Cao *et al.* [16] used the similarity of execution traces to prioritize test suites. Despite being promising, these more recent approaches have not previously been evaluated or compared with the rest of white-box and black-box ones; one of the motivations for the present paper.

Since dynamic coverage information might be unavailable or prohibitively expensive to collect [49], researchers have used static analysis to guide the test prioritization. Mei *et al.* [49] proposed to use call graphs of test cases of object-oriented programs in order to simulate test coverage.

Recently, Ripon *et al.* [59] proposed a more advanced technique, named REPiR, based on information retrieval. REPiR aims at test-class prioritization and can simulate coverage accurately. However, their results show that this approach is not better than the additional one.

The above two approaches are not comparable with ours since they target units, i.e., classes, of object oriented programs while our approach targets system level tests.

Black-box prioritization: Black-box prioritization was initially proposed in the CIT context. Bryce and Colbourn [14] proposed prioritizing test cases for CIT using a greedy heuristic. Bryce and Memon [15] used t-wise interaction coverage to prioritize GUI test suites. Cohen *et al.* [20] also used CIT coverage and proposed prioritizing test suites in the context of highly configurable systems. Henard *et al.* [36, 37] used similarity in order to bypass the combinatorial explosion of CIT. Recently, Petke *et al.* [55] compared higher CIT strengths with lower ones and found that in practice higher strengths can be achieved. However, all these previous studies consider CIT alone, and neither compare with white-box techniques nor other black-box approaches; another motivation for the present study.

Black-box test selection has been studied for model-based testing using similarity functions: Cartaxo *et al.* [17] and Hemmati *et al.* [35] demonstrated that similarity functions can effectively measure diversity and thereby prioritize model-based test suites. Lendru *et al.* [46] used string distance to measure test case diversity, while Rogstad *et al.* [56] applied similarity to database applications. None of these approaches have previously been compared with one another.

Schroeder [61] suggested using automated input-output analysis to perform test suite reduction, by determining the inputs that can affect the program outputs. This approach forms the basis of our output-driven prioritization strategies. Recently, Alshahwan *et al.* [9] used output diversity (which they call 'output uniqueness') to improve application testing. Testing with the use of input model mutants was proposed by Papadakis *et al.* [53]. These approaches have been previously proposed (and proven to be promising) for test case generation (rather than test suite prioritization). Nevertheless, as we have shown in the present paper, these approaches to test case generation can be reused as criteria to guide test suite prioritization and so we included them in the experimental study reported upon in the present paper.

More recently Feldt *et al.* [32] proposed the 'Test Set Diameter' (TSD) concept, which generalizes the use of NCD to multisets. Instead of approximating pairwise diversity, TSD measures the diversity of the entire test suite as a whole [19]. However, like other diversity measures, TSD has not previously been compared with other black-box or white-box approaches, motivating its inclusion in our work.

6. CONCLUSION

Our study has revealed a high degree of overlap between both the faults found by black- and white-box regression test prioritization techniques and also the performance of these approaches. Additionally, we have found that initial prioritization remains robust over multiple releases. Although further research would naturally be advisable to replicate these findings, we believe that they may provide welcome positive news to regression testers, particularly to those who do not have the luxury of source code access.

Acknowledgments: Mike Papadakis is supported by the National Research Fund, Luxembourg, INTER/MOBILITY/14/7562175. Mark Harman and Yue Jia are supported by EPSRC grant Dynamic Adaptive Automated Software Engineering (DAASE: EP/J017515).

7. REFERENCES

[1] GNU FTP Server. `http://ftp.gnu.org/`.

[2] `bzip2`: A freely available, patent free, high-quality data compressor. `http://www.bzip.org/`.

[3] `cloc`: Count Lines of Code. `http://cloc.sourceforge.net/`.

[4] `gcc`: The GNU Compiler Collection. `https://gcc.gnu.org/`.

[5] `gcov` - A Test coverage program. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[6] `R`: The R project for statistical computing. `https://www.r-project.org/`.

[7] `time`: Run programs and summarize system resource usage. `http://linux.die.net/man/1/time`.

[8] N. Alshahwan and M. Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *ISSTA*, pages 45–55, 2012.

[9] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *ISSTA*, pages 181–192, 2014.

[10] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *ICST*, pages 21–30, 2014.

[11] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624, 2006.

[12] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[13] T. Ball. On the limit of control flow analysis for regression test selection. In *ISSTA*, pages 134–142, 1998.

[14] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Info. & Softw. Tech.*, 48(10):960–970, 2006.

[15] R. C. Bryce and A. M. Memon. Test suite prioritization by interaction coverage. In *DOSTA*, pages 1–7, 2007.

[16] Y. Cao, Z. Zhou, and T. Y. Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *QSIC*, pages 153–162, 2013.

[17] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Softw. Test., Verif. Reliab.*, 21(2):75–100, 2011.

[18] T. Y. Chen, F. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Jrnl. Syst. Softw.*, 83(1):60–66, 2010.

[19] A. R. Cohen and P. M. B. Vitányi. Normalized compression distance of multisets with applications. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(8):1602–1614, 2015.

[20] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Softw. Eng.*, 34(5):633–650, 2008.

[21] D. Cotroneo, R. Pietrantuono, and S. Russo. A learning-based method for combining testing techniques. In *ICSE*, pages 142–151, 2013.

[22] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.*, 10(4):405–435, Oct. 2005.

[23] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *FSE*, pages 141–151, 2006.

[24] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Softw. Test., Verif. Reliab.*, 13(2):65–83, 2003.

[25] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, pages 235–245, 2014.

[26] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[27] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*, pages 329–338, 2001.

[28] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[29] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Softw. Qual. Jrnl.*, 12(3):185–210, 2004.

[30] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Info. & Softw. Tech.*, 52(1):14–30, 2010.

[31] E. Engström, M. Skoglund, and P. Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In *ESEM*, pages 22–31, 2008.

[32] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. *CoRR*, abs/1506.03482, 2015.

[33] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *ASE*, pages 361–372, 2014.

[34] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. 2012.

[35] H. Hemmati, A. Arcuri, and L. C. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1):6, 2013.

[36] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Softw. Eng.*, 40(7):650–670, July 2014.

[37] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Assessing software product line testing

via model-based mutation: An application to similarity testing. In *A-MOST*, pages 188–197, 2013.

[38] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[39] Y. Jia and M. Harman. Higher order mutation testing. *Info. & Softw. Tech.*, 51(10):1379–1393, 2009.

[40] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649 – 678, 2011.

[41] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244, 2009.

[42] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE*, pages 474–484, 2012.

[43] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

[44] J. Kim and A. A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[45] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *APSEC*, pages 300–309, 2010.

[46] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Autom. Softw. Eng.*, 19(1):65–95, 2012.

[47] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.*, 33(4):225–237, 2007.

[48] M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Softw. Eng.*, 29(11):974–984, Nov. 2003.

[49] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *IEEE Trans. Softw. Eng.*, 38(6):1258–1275, 2012.

[50] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *ISSTA*, pages 100–110, 2012.

[51] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11, 2011.

[52] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, 2004.

[53] M. Papadakis, C. Henard, and Y. L. Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *ICST*, pages 1–10, 2014.

[54] M. Papadakis, Y. Jia, M. Harman, and Y. LeTraon. Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In *ICSE*, pages 936–946, 2015.

[55] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *FSE*, pages 26–36, 2013.

[56] E. Rogstad, L. C. Briand, and R. Torkar. Test case selection for black-box regression testing of database applications. *Info. & Softw. Tech.*, 55(10):1781–1795, 2013.

[57] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.

[58] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[59] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *ICSE*, pages 268–279, 2015.

[60] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.

[61] P. J. Schroeder and B. Korel. Black-box test reduction using input-output analysis. In *ISSTA*, pages 173–177, 2000.

[62] A. Vargha and H. D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Jrnl. Educ. Behav. Stat.*, 25(2):101–132, 2000.

[63] P. Vitányi, F. Balbach, R. Cilibrasi, and M. Li. Normalized information distance. In *Information Theory and Statistical Learning*, pages 45–82. 2009.

[64] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. 2000.

[65] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[66] S. Yoo and M. Harman. Test data regeneration: Generating new test data from existing test data. *Softw. Test., Verif. Reliab.*, 22(3):171–201, May 2012.

[67] C. Zhang, A. Groce, and M. A. Alipour. Using test case reduction and prioritization to improve symbolic execution. In *ISSTA*, pages 160–170, 2014.

[68] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*, pages 192–201, 2013.

[69] Z. Q. Zhou, A. Sinaga, and W. Susilo. On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites. In *HICSS*, pages 5584–5593, 2012.