This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

An Empirical Comparison of Fixed-strength and Mixed-strength for Interaction Coverage Based Prioritization

RUBING HUANG[1], (Member, IEEE), QUANJUN ZHANG[1], TSONG YUEH CHEN[2], (Member, IEEE), JAMES HAMLYN-HARRIS[2], DAVE TOWEY[3], (Member, IEEE), JINFU CHEN[1], (Member, IEEE)

[1]School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, Jiangsu 212013, China
[2]Department of Computer Science and Software Engineering, Swinburne University of Technology, VIC 3122, Australia
[3]School of Computer Science, University of Nottingham Ningbo China, Ningbo, Zhejiang 315100, China

Corresponding author: Jinfu Chen (e-mail: jinfuchen@ujs.edu.cn).

This work is supported by the National Natural Science Foundation of China under grant nos. 61202110, 71471092, 61502205, and 61872167, the Ningbo Science and Technology Bureau under grant no. 2014A35006, and the Senior Personnel Scientific Research Foundation of Jiangsu University under grant no. 14JDG039. This work is also in part supported by the Young Backbone Teacher Cultivation Project of Jiangsu University, and the sponsorship of Jiangsu Overseas Visiting Scholar Program for University Prominent Young & Middle-aged Teachers and Presidents.

ABSTRACT Test case prioritization (TCP) plays an important role in identifying, characterizing, diagnosing and correcting faults quickly. TCP has been widely used to order test cases of different types, including model inputs (also called *abstract test cases*). Model inputs are constructed by modeling the program according to its input parameters, values, and constraints, and has been used in different testing methods, such as combinatorial interaction testing, and software product line testing. *Interaction coverage-based test case prioritization* (ICTCP) uses interaction coverage information derived from the model input to order inputs. Previous studies have focused generally on the *fixed-strength* ICTCP, which adopts a fixed strength (*i.e.*, the level of parameter interactions) to support the ICTCP process. It is generally accepted that using more strengths for ICTCP, *i.e.*, *mixed-strength* ICTCP, may give better ordering than fixed-strength. To confirm whether mixed-strength is better than fixed-strength, in this paper we report on an extensive empirical study using five real-world programs (written in C), each of which has six versions. The results of the empirical studies show that mixed-strength has better rates of interaction coverage overall than fixed-strength, but they have very similar rates of fault detection. Our results also show that fixed-strength should be used instead of the mixed-strength at the later stage of software testing. Finally, we offer some practical guidelines for testers when using interaction coverage information to prioritize model inputs, under different testing scenarios and resources.

INDEX TERMS Test case prioritization, model input, interaction coverage, mixed-strength, fixed-strength.

Due to limited testing resources, when conducting testing in practice (for example in regression testing), the execution order of test cases can be critical, and more important test cases in a test set should be executed as early as possible. A well-ordered test case execution sequence may be able to identify faults faster than a poorly-ordered sequence, thus allowing activities such as fault characterization, diagnosis, and correction, to be started as soon as possible. The process of determining the order of test cases in a test set is called *test case prioritization* (TCP) [1], and has been used extensively in many testing situations, such as regression testing [2].

Many TCP algorithms have been proposed to guide the prioritization of different types of test case, including code coverage-based prioritization [1], [3], search-based prioritization [4], [5], and adaptive random prioritization [6]–[9]. A *model input* [10] (also called an *abstract test case* [11]) is an important type of test case [12], and can be obtained based on a model of the *software under test* (SUT): It

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

consists of a fixed number of parameters that influence the SUT, a finite set of values for each parameter, and a set of constraints on parameter values. Each model input is constructed by assigning a value to each of the parameters. Model inputs have been widely used in testing, including for highly-configurable systems testing [13], [14], the category-partition testing method [12], and combinatorial interaction testing [15]. Model input prioritization has also been studied extensively in recent years, especially in the field of combinatorial interaction testing [16]–[18] and software product line testing [19], [20].

The interaction coverage is the information derived from the model input itself, represented by the parameter-value combinations covered by the model input. It has been widely used to guide the model input prioritization, and is called *interaction coverage-based test case prioritization* (ICTCP) [17], [18], [21]–[23]. Previous studies have mainly focused on *fixed-strength ICTCP*, which adopts a fixed value for strength (*i.e.*, the level of parameter interactions) to support the whole ICTCP process [18], [21]–[23]. It is expected that ICTCP using more strengths may provide better ordering of test cases than using fixed-strength TCP, but it can be more time-consuming, because more information is required to be considered.

To determine whether mixed-strength is better than fixed-strength, we conducted empirical studies on five real-world programs (written in C), each of which contains six versions, according to some quality evaluation metrics. Based on the experimental results, we present some empirical findings, and provide some practical guidelines for testers when facing the prioritization problem of model inputs. In summary, the main contributions of this work are as described as follows:

1) We investigated 63 ICTCP techniques, involving 6 fixed-strength techniques, and 57 mixed-strength techniques, and compared mixed-strength against fixed-strength for the same maximum prioritization strength.

2) We conducted empirical studies to investigate the testing effectiveness and efficiency of mixed-strength and fixed-strength, from the perspective of the rate of interaction coverage, the rate of fault detection, and prioritization cost.

3) We present empirical findings and analysis comparing mixed-strength and fixed-strength.

4) We provide some practical guidelines for testers about how to choose mixed-strength and fixed-strength techniques, when prioritizing the model inputs under different testing scenarios and resources.

The rest of this paper is organized as follows: Section I introduces some background information about model inputs, and test case prioritization. Section II describes the research questions, and Section III presents the experimental setup. Section IV reports on the empirical studies, analyzes the results, and answers the research questions. In addition, it provides some practical guidelines for testers, and also discusses the limitations of this work. Section V reviews some related work about combinatorial interaction testing, and test case prioritization. Finally, Section VI concludes the paper and proposes future work.

I. BACKGROUND

In this section, we introduce the topic of model input and test case prioritization (TCP).

A. MODEL INPUT

The *software under test* (SUT) is generally influenced by a number of *parameters* or *factors* (such as configurations, features, components). Typically, each parameter can have a fixed number of possible *values*, or *levels*. Generally, there may be constraints on parameter values, with some value combinations being infeasible.

We define the *input parameter model* [11] used to model the SUT as follows:

Definition 1. *An input parameter model, Model*$(P = \{p_1, p_2, \cdots, p_k\}, V = \{V_1, V_2, \cdots, V_k\}, \mathcal{C})$*, represents the information about the test object —* $p_1, p_2, \cdots, p_k$*, are the* k *parameters; each* V_i *is the set of possible values for the* i*-th parameter* (p_i)*; and* $\mathcal{C}$ *is the set of value combination constraints.*

For example, Table 1 gives an input parameter model with two constraints for an application of Partition and Volume Creation, where four parameters are included, of which the first two parameters have two values, the third parameter has three values, and the last parameter has four values. Since the file system "FAT" is limited to the size less than 7096, and the file system "FAT32" is limited to the size less than 32000, two value combination constraints are obtained. To simplify the problem, each parameter is denoted by p_i $(i = 1, 2, 3, 4)$, and each value is labelled by an integer, beginning with 0 and incrementing by 1, from p_1 to p_4 (see Table 1).

After that, we have the following input parameter models for this example: Model$(\{p_1, p_2, p_3, p_4\}, \{\{\text{"0"}, \text{"1"}\}, \{\text{"2"}, \text{"3"}\}, \{\text{"4"}, \text{"5"}, \text{"6"}\}, \{\text{"7"}, \text{"8"}, \text{"9"}, \text{"10"}\}\}, \mathcal{C} = \{p_3 = \text{"4"} \rightarrow p_4 = \text{"7"}, p_3 = \text{"5"} \rightarrow p_4 \neq \text{"10"}\}$. Because specific values of each parameter have no impact on the model, without loss of generality, we can use the following abbreviated version: Model$(|V_1||V_2|\cdots|V_k|, \mathcal{C})$. In addition, we adopt the description method of the constraint set $\mathcal{C}$, previously used in [24], the example above can therefore be represented as: Model$(2^2 3^1 4^1, \mathcal{C} = \{\text{"4"} \rightarrow \text{"7"}, \text{"5"} \rightarrow \neg\text{"10"}\})$.

An input parameter model (if available) can be used to construct *model inputs* [10] (also called *abstract test cases* [11]) for testing the test object. A definition of the model input is given as follows:

TABLE 1: An example for input parameter model.

P	p_1: Format	p_2: Compression	p_3: File System	p_4: Size
V	QUICK(0)	ON(2)	FAT (4)	1000 (7)
	SLOW(1)	OFF(3)	FAT32 (5)	5000 (8)
	–	–	NTFS (6)	10000 (9)
	–	–	–	50000 (10)

File System = "FAT" → **Size** ≤ 4096, *i.e.*, $p_3 = \text{"4"} \rightarrow p_4 = \text{"7"}$.
File System = "FAT32" → **Size** ≤ 32000, *i.e.*, $p_3 = \text{"5"} \rightarrow p_4 \neq \text{"10"}$.

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

Definition 2. *A model input, denoted* $(v_1, v_2, \cdots, v_k)$*, is a* k*-tuple, where* $v_i \in V_i$ $(i = 1, 2, \cdots, k)$.

If all the value constraints in $\mathcal{C}$ are satisfied, then a model input is said to be *valid*, otherwise *invalid*. An example of a valid model input for the previous model is ("1", "3", "6", "7"); and an example of an invalid one is ("0", "2", "5", "10"), which violates the constraint ("5" $\rightarrow \neg$"10").

Definition 3. *A* λ*-wise value combination is a* k*-tuple* $(\widehat{v}_1, \widehat{v}_2, \cdots, \widehat{v}_k)$*, involving* λ *parameters with fixed values (named fixed parameters) and* $(k - \lambda)$ *parameters with arbitrary allowable values (named free parameters), where* $0 \leq \lambda \leq k$ *and*

$$\widehat{v}_i = \begin{cases} v_i \in V_i, & \text{if } p_i \text{ is a fixed parameter} \\ \text{"}-\text{"}, & \text{if } p_i \text{ is a free parameter} \end{cases} \quad (1)$$

The λ-wise value combination is also called λ-*wise schema* [15]. In order to describe the problem clearly (without loss of generality), the free parameters are not considered. In other words, a λ-wise value combination is actually a λ-tuple. Each model input could cover some λ-wise value combinations. For example, a model input ("0", "2", "5", "9") covers four 1-wise value combinations, *i.e.*, ("0"), ("2"), ("5"), and ("9"); and four 3-wise value combinations, *i.e.*, ("0", "2", "5"), ("0", "2", "9"), ("0", "5", "9"), and ("2", "5", "9"). Similar to model inputs, the λ-wise value combination may also be valid or invalid, for example, a 2-wise value combination ("0", "2") is valid; while another one ("5", "10") is invalid. A valid model input covers valid λ-wise value combinations that are valid, regardless of λ values.

To simplify the notation, we define a function $\psi(\lambda, tc)$ for a model input tc that returns the set of all λ-wise value combinations covered by tc, *i.e.*,

$$\psi(\lambda, tc) = \{(v_{j_1}, v_{j_2}, \cdots, v_{j_x}) | 1 \leq j_1 < j_2 < \cdots < j_\lambda \leq k\} \quad (2)$$

Similarly, a function $\psi(\lambda, T)$ for a set T of model inputs is defined to return the set of all λ-wise value combinations covered by all model inputs in T, *i.e.*,

$$\psi(\lambda, T) = \bigcup_{tc \in T} \psi(\lambda, tc) \quad (3)$$

The size of $\psi(\lambda, tc)$, *i.e.*, $|\psi(\lambda, tc)|$, is equal to $C(k, \lambda)$ (*i.e.*, the number of λ-combinations from k elements).

B. TEST CASE PRIORITIZATION

Test case prioritization (TCP) schedules test cases so that those with higher priority, according to some criteria, are executed earlier than those with lower priority. When the execution of all test cases in a test suite is not possible, a well-designed execution order can be very important. The problem of test case prioritization can be defined as follows [1]:

Definition 4. *Given a tuple* (T, Ω, f)*, where* T *is a test suite,* Ω *is the set of all possible permutations of* T*, and* f *is a fitness function from* Ω *to real numbers, the test case prioritization problem is to find a prioritized test set* $S \in \Omega$ *such that:*

$$(\forall S') \; (S' \in \Omega) \; (S' \neq S) \; [f(S) \geq f(S')] \quad (4)$$

There are many fitness functions to support the TCP process, for example fault detection [1], and code coverage [4].

II. RESEARCH QUESTIONS

As we know, mixed-strength ICTCP (abbreviated as MICTCP) uses more strengths (*i.e.*, more information) than fixed-strength (FICTCP for short) to guide the prioritization of model inputs. Therefore, it is expected that MICTCP may provide higher speed to cover value combinations than FICTCP. This leads our first research question:

RQ1: How well does MICTCP compare with FICTCP in terms of interaction coverage rate?

Similarly, it seems likely that MICTCP could generate prioritized model inputs which trigger faults earlier in a test than FICTCP (since more information has been used in MICTCP). This leads to the next research question:

RQ2: How well does MICTCP compare with FICTCP in terms of fault detection rate?

Answers to **RQ1** and **RQ2** would establish whether MICTCP or FICTCP is more effective. In addition, since MICTCP makes use of more information than FICTCP, it is likely to require more prioritization time. Therefore, it is useful to check which technique is better able to balance the tradeoff between testing effectiveness (measured rates of interaction coverage and fault detection) and efficiency (measured by the prioritization cost), leading to our third research question:

RQ3: Which one is more cost-effective between MICTCP and FICTCP?

Finally, we would like to know, when facing different testing scenarios, for example limited testing resources, which ICTCP to choose. Our main aim therefore, is to answer the following question:

RQ4: Which approach should be chosen under different circumstances? MICTCP or FICTCP?

By answering these research questions, we aim to compare MICTCP and FICTCP, from the perspective of testing effectiveness and efficiency; and also present guidelines to method selection when facing different testing environments.

III. EXPERIMENTAL SETUP

Figure 1 presents the experimental process of empirical studies. At the begging, the set of model inputs is ordered by ICTCP, to obtained the prioritized model inputs. First, during the prioritization process, the prioritization time is collected. Second, the ordered set of model inputs is calculated, according to the interaction coverage rate. Finally, by transferring the model inputs into real test cases for each

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

TABLE 2: Studied Programs.

Object	Input Parameter Model	Tests	Information	V0	V1	V2	V3	V4	V5
Flex	$\text{Model}(9, 2^6 3^2 5^1, \mathcal{C}), \lvert\mathcal{C}\rvert = 12$	500	*Version (Year)*	2.4.3 *(1993)*	2.4.7 *(1994)*	2.5.1 *(1995)*	2.5.2 *(1996)*	2.5.3 *(1996)*	2.5.4 *(1997)*
			LOC	8,959	9,470	12,231	12,249	12,370	12,366
			#Faults	–	32	32	20	33	32
Grep	$\text{Model}(9, 2^1 3^3 4^2 5^1 6^1 8^1, \mathcal{C}), \lvert\mathcal{C}\rvert = 83$	440	*Version (Year)*	2.0 *(1996)*	2.2 *(1998)*	2.3 *(1999)*	2.4 *(1999)*	2.5 *(2002)*	2.7 *(2010)*
			LOC	8,163	11,988	12,724	12,826	20,838	58,344
			#Faults	–	56	58	54	58	59
Gzip	$\text{Model}(14, 2^{13} 3^1, \mathcal{C}), \lvert\mathcal{C}\rvert = 61$	156	*Version (Year)*	1.0.7 *(1993)*	1.1.2 *(1993)*	1.2.2 *(1993)*	1.2.3 *(1993)*	1.2.4 *(1993)*	1.3 *(1999)*
			LOC	4,324	4,521	5,048	5,059	5,178	5,682
			#Faults	–	16	18	18	19	22
Make	$\text{Model}(10, 2^{10}, \mathcal{C}), \lvert\mathcal{C}\rvert = 1$	111	*Version (Year)*	3.75 *(1996)*	3.76.1 *(1997)*	3.77 *(1998)*	3.78.1 *(1999)*	3.79 *(2000)*	3.80 *(2002)*
			LOC	17,463	18,568	19,663	20,461	23,125	23,400
			#Faults	–	37	29	28	29	28
Sed	$\text{Model}(11, 2^7 3^1 4^1 6^1 10^1, \mathcal{C}), \lvert\mathcal{C}\rvert = 50$	324	*Version (Year)*	3.0.1 *(1998)*	3.0.2 *(1998)*	4.0.6 *(2003)*	4.0.8 *(2003)*	4.1.1 *(2004)*	4.2 *(2009)*
			LOC	7,790	7,793	18,545	18,687	21,743	26,466
			#Faults	–	8	8	7	7	7

program, the resulting ordered test suite is evaluated on the five versions (V1 to V5) using mutation testing, in terms of the fault detection rate.

A. SUBJECT PROGRAMS

We used five open-source C programs (Flex, Grep, Gzip, Make, and Sed) that were selected from the GNU FTP server [25]. Flex is a lexical analysis generation, while Grep and Sed are widely-used command-line tools for searching and processing text matching regular expressions. Make is a popular utility used to control the compile and build process of programs, while Gzip is a compression utility. These programs have been widely used in the field of test case prioritization [1], [7], [10], [18], [22], [26], [27].

For each of the programs, Table 2 presents its version number and the year that it was released, its size in uncommented lines of code measured by `cloc` [28], and the number of seeded faults. The table also describes the input parameter model for each program modeled by Petke *et al.* [18], [27], and the size of test pool. These test pools are available from the Software Infrastructure Repository (SIR) [29].

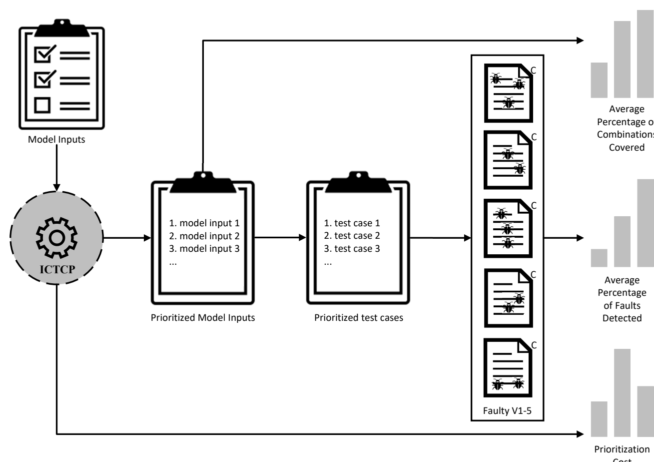


FIGURE 1: Experimental process of empirical studies.

B. THE 63 ICTCP TECHNIQUES STUDIED

Previous investigations have shown that nearly all faults are caused by the interaction among no more than six parameters [30], [31], therefore, we chose the maximum strength value, d, ranging from 1 to 6. Therefore, we consider all possible cases of strength selection with a total of $(2^6 - 1 = 63)$ choices, *i.e.*, 6 techniques with fixed-strength (FICTCP); and 57 techniques with mixed-strength (MICTCP).

According to previous investigations [18], [26], [27], when adopting FICTCP, different strengths provide different levels of performance. Therefore, in this study when comparing FICTCP and MICTCP, we will use the same maximum strength d. More specifically, when FICTCP uses the prior-

TABLE 3: The 63 ICTCP techniques.

d	L_d	Strengths	d	L_d	Strengths
$d=1$	'1'	1	$d=6$	'100000'	6
$d=2$	10	2		'100001'	1, and 6
	'11'	1, and 2		'100010'	2, and 6
$d=3$	'100'	3		'100100'	3, and 6
	'101'	1, and 3		'101000'	4, and 6
	'110'	2, and 3		'110000'	5, and 6
	'111'	1, 2, and 3		'100011'	1, 2, and 6
$d=4$	'1000'	4		'100101'	1, 3, and 6
	'1001'	1, and 4		'100110'	2, 3, and 6
	'1010'	2, and 4		'101001'	1, 4, and 6
	'1100'	3, and 4		'101010'	2, 4, and 6
	'1011'	1, 2, and 4		'101100'	3, 4, and 6
	'1101'	1, 3, and 4		'110001'	1, 5, and 6
	'1110'	2, 3, and 4		'110010'	2, 5, and 6
	'1111'	1, 2, 3, and 4		'110100'	3, 5, and 6
$d=5$	'10000'	5		'111000'	4, 5, and 6
	'10001'	1, and 5		'100111'	1, 2, 3, and 6
	'10010'	2, and 5		'101011'	1, 2, 4, and 6
	'10100'	3, and 5		'101101'	1, 3, 4, and 6
	'11000'	4, and 5		'101110'	2, 3, 4, and 6
	'10011'	1, 2, and 5		'110011'	1, 2, 5, and 6
	'10101'	1, 3, and 5		'110101'	1, 3, 5, and 6
	'11001'	1, 4, and 5		'110110'	2, 3, 5, and 6
	'10110'	2, 3, and 5		'111001'	1, 4, 5, and 6
	'11010'	2, 4, and 5		'111010'	2, 4, 5, and 6
	'11100'	3, 4, and 5		'111100'	3, 4, 5, and 6
	'10111'	1, 2, 3, and 5		'101111'	1, 2, 3, 4, and 6
	'11011'	1, 2, 4, and 5		'110111'	1, 2, 3, 5, and 6
	'11101'	1, 3, 4, and 5		'111011'	1, 2, 4, 5, and 6
	'11110'	2, 3, 4, and 5		'111101'	1, 3, 4, 5, and 6
	'11111'	1, 2, 3, 4, and 5		'111110'	2, 3, 4, 5, and 6
				'111111'	1, 2, 3, 4, 5, and 6

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

itization strength d, MICTCP adopts d and other strengths less than d, which means that the number of strengths used in MICTCP is limited to d, *i.e.*, $1, 2, \cdots, d$.

To simplify the notation, we define the term L_d to represent '$x_d x_{d-1} \cdots x_1$' that is a d-digit binary number, *i.e.*, $x_i \in \{0, 1\}$. If $x_i = 1$ $(1 \leq i \leq d)$, the strength i is included; and if $x_i = 0$, the strength i is excluded. Obviously, when x_d is equal to 1 and other bits are equal to 0, it is FICTCP; otherwise, it is MICTCP. For example, L_3 = '100' represents FICTCP with strength 3; while L_4 = '1101' represents MICTCP with strengths 1, 3, and 4. Table 3 gives an overview of the 63 ICTCP techniques investigated, from which d, L_d, and strengths used, are presented.

Algorithm 1 shows the details of the ICTCP, which builds on previous algorithms to prioritize model inputs using interaction coverage [17], [18], [21]–[23], *i.e.*, it calculates the fitness of each candidate that has not been chosen, and selects one candidate such that it has the maximum fitness as the next test case in the prioritized set. As for the fitness function fitness(L_d, tc, S), it can be defined as following:

$$fitness(L_d, tc, T) = \sum_{i=1}^{d} \frac{\left|\{\xi | \xi \in (\psi(i, tc) \setminus \psi(i, T)), x_i = 1\}\right|}{C_k^i} \quad (5)$$

When facing the tie case (*i.e.*, there exists more than one candidate achieving the maximum fitness), the algorithm adopts *random tie-breaking* [32] to randomly choose one. After that, the algorithm removes the selected test case from the candidates. This process is repeated until all model inputs are chosen.

Since ICTCP involves randomization (due to the random tie-breaking technique [32]), we ran each experiment 100 times, and collected a set of different outcomes for each prioritization technique: This could help us further analyze performance differences between different prioritization techniques.

C. FAULT SEEDING

For each of the subject programs, the original version contains no seeded-in faults. In this paper, we have used mutation analysis [33]. As discussed in previous studies [34], [35], mutation analysis can provide more realistic faults than hand-seeding, and may be more appropriate for studying test case prioritization.

For the five subject programs, we used the same mutation faults as used by Henard *et al.* [10], *i.e.*, we employed the mutant operators set used by Andrews *et al.* [34], such as statement deletion, constant replacement, unary insertion, arithmetic operator replacement, logical operator replacement, relational operator replacement, and bitwise logical operator replacement. Among all mutants, we removed the duplicated and equivalent mutants as possible, and also removed all mutants that are not killed by any model input by following previous practices [1], [34], [36]. In addition, all the subsuming mutants [37] (also called minimum mutants [38] or disjoint mutants [39]) that are easily killed from the original program were removed, because these mutants may affect the value of the mutation score measurement [34], [38]–[40]. A mutation fault is said to be detected by a test case when execution outputs are different for the original and fault-seeded versions. Table 2 shows the number of faults adopted in this study.

Algorithm 1 ICTCP Procedure

Input: T ▷ An unordered set of model inputs
L_d ▷ A d-digit $(1 \leq d \leq k)$ binary number
Output: S ▷ A prioritized set of model inputs
1: $S \leftarrow []$
2: **while** $|T| > 0$
3: **if** $|T| > 1$
4: Select $tc \in T$, where $max\big(\text{fitness}(L_d, tc, S)\big)$ ▷ take the random one in case of equality
5: $S.add(tc)$
6: $T \leftarrow T \setminus \{tc\}$
7: **else** ▷ T contains only one element
8: $S.add(tc)$, where $tc \in T$
9: $T \leftarrow \emptyset$
10: **end if**
11: **end while**
12: **return** S

D. EVALUATION METRICS

To evaluate different ICTCP strategies, in this study we focused on the following three aspects: (a) rate of interaction coverage, to measure how quickly each prioritized set of model inputs covered value combinations; (b) rate of fault detection, to measure how well each prioritized test set identified faults; and (c) prioritization cost, to measure how quickly each prioritized test set was obtained.

1) Interaction Coverage Rate

The *average percentage of λ-wise combinations covered* (APCC) [41], also named *average percentage of combinatorial coverage* [42] or *average percentage of covering-array coverage* [18], is used to measure the rate of interaction coverage of strength λ achieved by a prioritized test set of model inputs. Its definition is given as follows.

Definition 5. *Suppose $T = \{t_1, t_2, \cdots, t_n\}$ is a set of model inputs with size n, the APCC definition of S at strength λ $(1 \leq \lambda \leq k)$ is:*

$$APCC(\lambda, S) = \frac{\sum_{i=1}^{n} \left|\psi(\lambda, \bigcup_{j=1}^{i} \{t_j\})\right|}{n \times |\psi(\lambda, T)|} - \frac{1}{2n} \quad (6)$$

The APCC metric values range from 0.0 to 1.0, with higher values meaning better rates of interaction coverage at a specific strength λ. In this paper, we considered APCC with $\lambda = 1, 2, 3, 4, 5$, and 6, by following previous studies [18].

However, previous APCC calculations only evaluate a given prioritized set of model inputs at a specific strength λ, resulting in the case that different strengths may draw different conclusions. In this paper, therefore we adopt the average APCC metric value based on the strengths 1, 2, 3, 4,

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

5, and 6, *i.e.*,

$$AvgAPCC(S) = \frac{1}{6}\sum_{\lambda=1}^{6} APCC(\lambda, S) \quad (7)$$

2) Fault Detection Rate

The *average percentage of faults detected* (APFD) was used to assess different prioritization techniques [1]. Its definition is given as follows (from [1]):

Definition 6. *Suppose T is a test suite containing n test cases, and F is a set of m faults revealed by T. Let SF_i be the number of test cases in the prioritized test set S of T that are executed until detecting fault $F_i \in F$. The APFD for test sequence S is given by the following equation:*

$$APFD(S) = 1 - \frac{SF_1 + SF_2 + \cdots + SF_m}{n \times m} + \frac{1}{2n} \quad (8)$$

3) Prioritization Cost

The prioritization cost measures the prioritization time required for each ICTCP technique, and represents the efficiency of the technique. Lower prioritization costs mean better performance.

E. STATISTICAL ANALYSIS

When assessing the statistical significance of the differences between the APCC or APFD values (used to evaluate each ICTCP technique), it is reasonable to use an unpaired test because there was no relationship between any of the 100 runs. Therefore, following previous studies dealing with randomized algorithms [43], [44], we used the unpaired two-tailed Wilcoxon-Mann-Whitney test of statistical significance (set at a 5% level of significance).

Because multiple statistical prioritization techniques were employed, we report the p-values — as the number of the executions increases, p becomes sufficiently small [10], which means that there are differences between the two algorithms. We used the non-parametric Vargha and Delaney effect size measure [45], $\hat{A}_{12}$, which presents the probability that one technique is better than another — with a higher effect size (value) indicating higher probability. For example, $\hat{A}_{12}(x, y) = 1.0$ indicates that, based on the sample, algorithm x always performs better than algorithm y; and $\hat{A}_{12}(x, y) = 0.0$ means that x always has worse performance.

IV. RESULTS

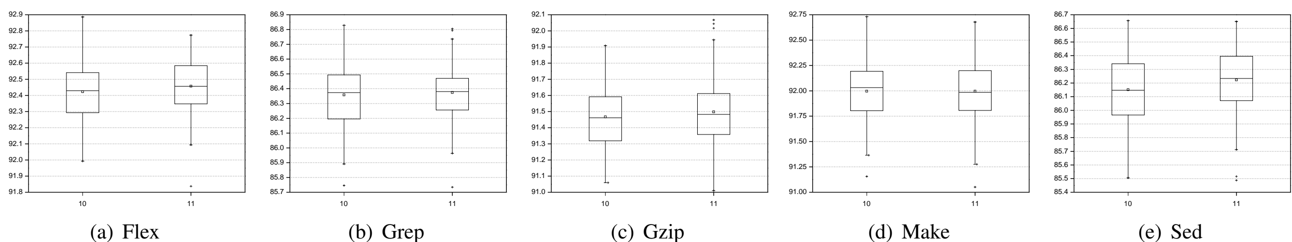
In this section, the results of empirical studies are presented comparing MICTCP and FICTCP techniques. In the plots in each figure in this section (Figures 2 to 11), the X-axis lists the prioritization techniques compared, and the Y-axis shows the AvgAPCC or APFD values for that technique. Each box plot shows the mean (square in the box), median (line in the box), upper and lower quartile, and min/max AvgAPCC or APFD values for the prioritization technique. In addition, Tables 4 and 6 give the statistical pairwise comparisons of AvgAPCC and APFD between MICTCP and FICTCP, from which each cell provides the p-value/$\hat{A}_{12}$ measure.

A. RQ1: INTERACTION COVERAGE EXPERIMENTS

Figures 2 to 6 present the AvgAPCC results of different d values, each of which contains five sub-figures for subject program Flex, Grep, Gzip, Make, and Sed, respectively. Each plot shows the distribution of the 100 AvgAPCC values (*i.e.*, 100 orderings). Table 4 records the statistical pairwise AvgAPCC comparison between MICTCP and FICTCP.

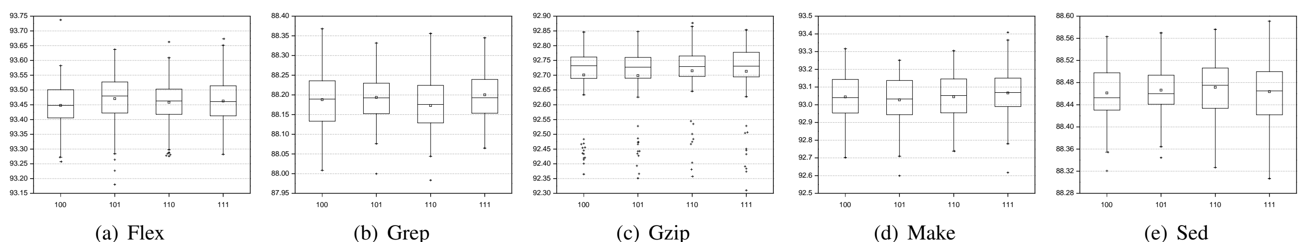
1) Observations

Based on the experimental data, we have the following observations:



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

FIGURE 2: AvgAPCC metric values for each program when $d = 2$.

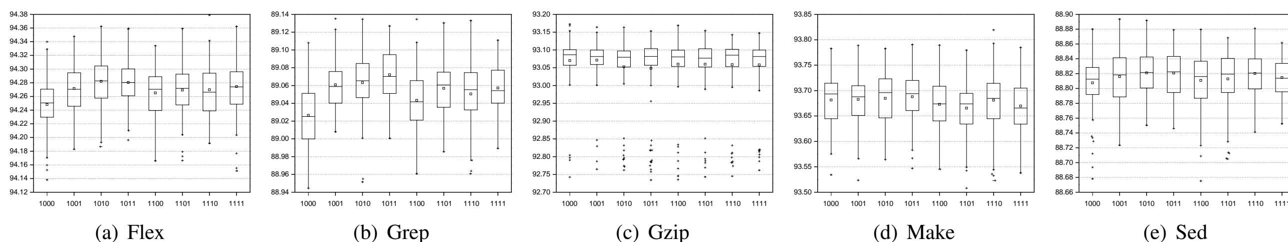


(a) Flex (b) Grep (c) Gzip (d) Make (e) Sed

FIGURE 3: AvgAPCC metric values for each program when $d = 3$.

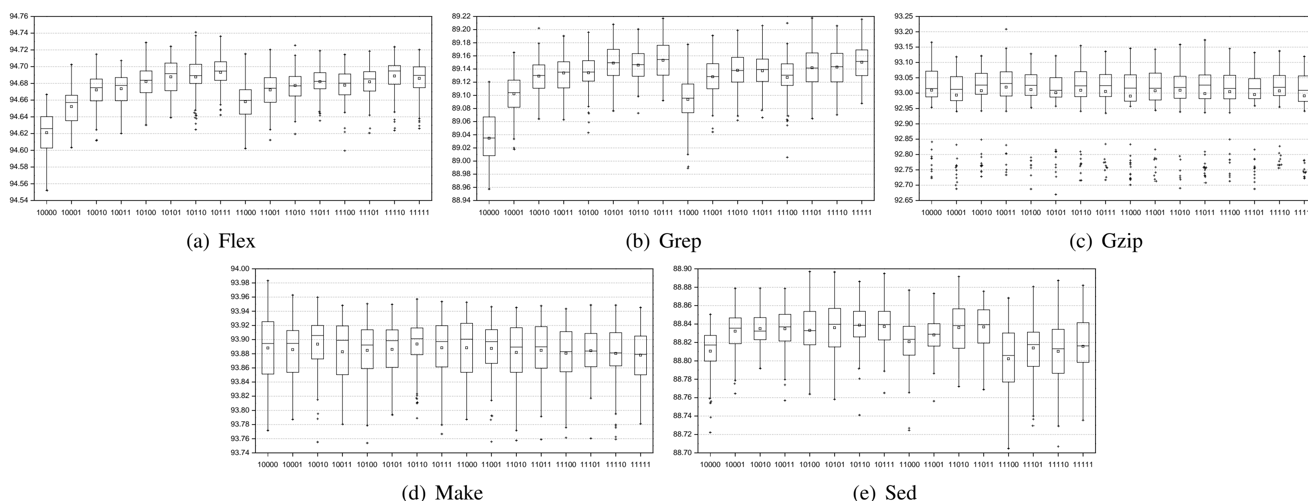
2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access



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

FIGURE 4: AvgAPCC metric values for each program when $d = 4$.



(a) Flex (b) Grep (c) Gzip

(d) Make (e) Sed

FIGURE 5: AvgAPCC metric values for each program when $d = 5$.

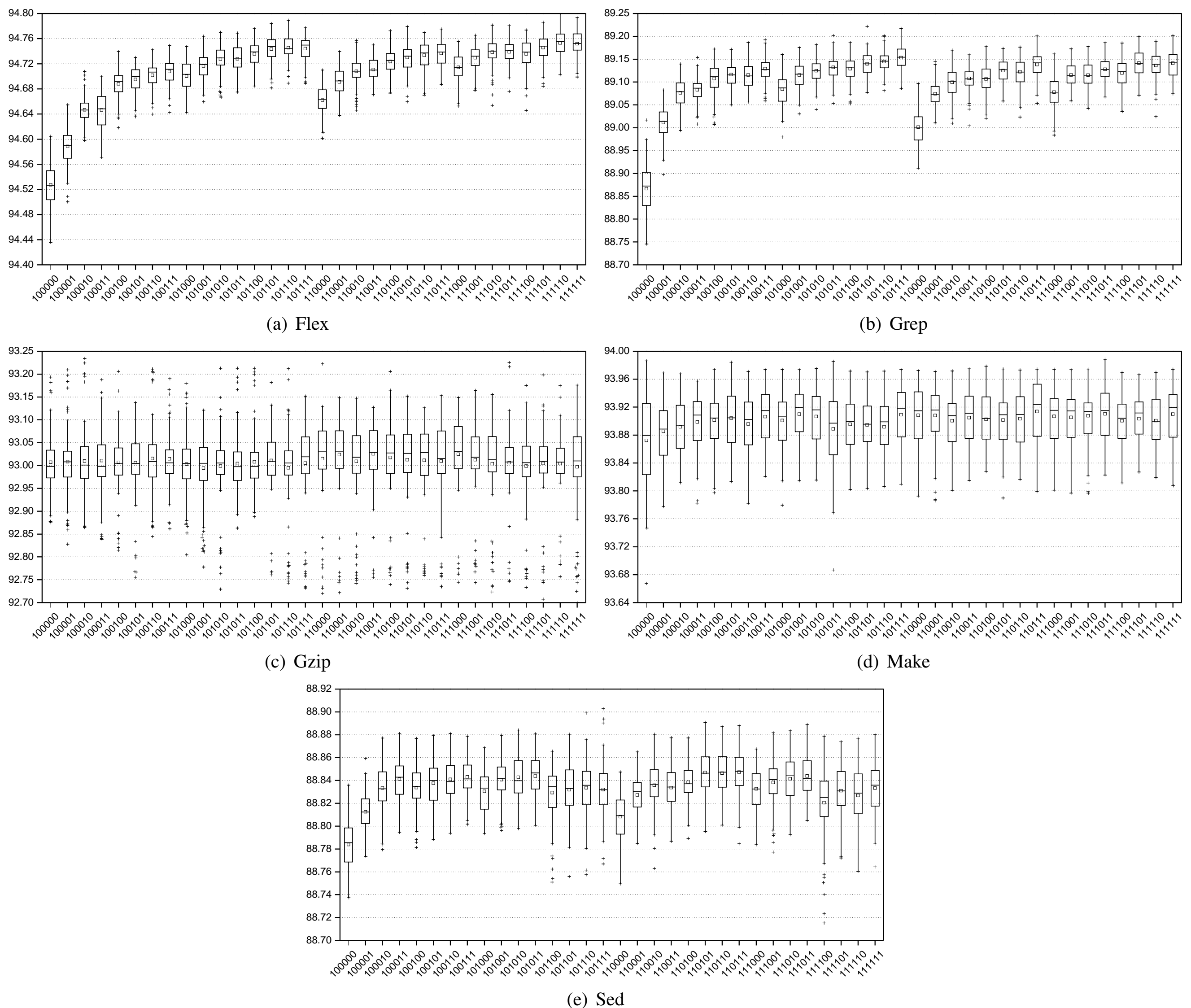
1) When $d = 2$ or $d = 3$, *i.e.*, the maximum strength used in FICTCP and MICTCP is equal to 2 or 3, FICTCP performs similarly to MICTCP. In nearly all cases, MICTCP has slightly higher rates of interaction coverage than FICTCP, both in terms of mean and median AvgAPCC values. Nevertheless, sometimes FICTCP achieves slightly better AvgAPCCs than MICTCP, for example, '110' vs '100' for program Grep. In other words, MICTCP does not always perform better than FICTCP. The statistical analysis generally confirms the box plot results, because all p-values are higher than 0.05; and all effect size $\hat{A}_{12}$ values are around 0.50. It can be also observed that the $\hat{A}_{12}$ values of MICTCP compared to FICTCP are higher than or equal to 0.50 (except '110' vs '100' for program Grep, as its values is 0.44).

2) When d is high (*i.e.*, $d = 4$, 5, and 6), the comparisons between FICTCP and MICTCP have different levels of performance for different subject programs. More specifically,

- For programs Flex and Grep, each MICTCP technique constructs prioritized model inputs with much higher AvgAPCC values than FICTCP, irrespective of d values. Both in terms of mean and median AvgAPCC values, the minimum differences of AvgAPCC between MICTCP and FICTCP are approximately 0.02, 0.03, and 0.06 for program Flex with d equalling 4, 5, and 6, respectively; while the maximum differences reach about 0.03, 0.06, and 0.23. The case of program Grep is similar to that of program Flex, *i.e.*, the minimum AvgAPCC differences are close to 0.02, 0.06, and 0.14 for d being 4, 5, and 6; while the maximum differences approach 0.04, 0.12, and 0.29, respectively. According to the statistical comparisons, all p-values are much less than 0.05, which means that the differences between FICTCP and MICTCP are highly significant; while all effect size $\hat{A}_{12}$ values are much higher than 0.50, ranging in [0.67, 1.00] for Flex and [0.76, 1.00] for Grep, which indicates that MICTCP has better performance than FICTCP at least 67% of the time.
- As for programs Gzip and Make, we observe the followings: a) When d is equal to 4 or 5, FICTCP and MICTCP have very similar AvgAPCC values, regardless of both mean and median values. In some cases, however, FICTCP has slightly better performance than some MICTCP techniques such as '1100', '1101', '1110', '1111', '11100', '11101', '11110', and '11111'. The statistical analysis generally validates the box plot observations: all p-values are higher than 0.05 (except the case of '1101' vs '1000' with 7.0E-03), which means that FICTCP and MICTCP do not have highly significant differences; while the effect size $\hat{A}_{12}$ values are less than 0.50 in most cases, which indicates that MICTCP performs better than FICTCP in less than 50% of the cases. b) When d is equal to 6, the

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access



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

FIGURE 6: AvgAPCC metric values for each program when $d = 6$.

case seems opposite. In detail, although there exist few MICTCP techniques that have similar AvgAPCC values to FICTCP, in many cases MICTCP achieves much higher rates of interaction coverage. Considering program Make for example, all MICTCP techniques have higher AvgAPCC values than FICTCP with the maximum mean and median differences being approximately 0.04. Moreover, the majority of p-values for comparing MICTCP and FICTCP are much less than 0.05, which means that the AvgAPCC differences between them are highly significant in the majority of cases. Meanwhile, all effect size $\hat{A}_{12}$ values for comparing MICTCP against FICTCP are greater than 0.50. Specifically, the $\hat{A}_{12}$ values range from 0.52 to 0.65 for program Gzip, indicating that MICTCP performs better than FICTCP from 52% to 65% of the time; while ranging from 0.50 to 0.64 for program Make, which means that MICTCP has better performance than FICTCP in 50% to 60% of the cases.

- For Sed, different d values result in different observations: a) When $d = 4$, MICTCP has slightly better performance than FICTCP, both in terms of median and mean values. This observation can be confirmed by the statistical analysis, *i.e.*, the half of p-values are less than 0.05, which means that the AvgAPCC differences between MICTCP and FICTCP are not highly significant in half of the cases. However, all effect size $\hat{A}_{12}$ values are greater than 0.50, (0.53 0.61), which indicates that MICTCP outperforms FICTCP 53% to 61% of the time. b) When $d = 5$, MICTCP techniques are similar or better than FICTCP, except '11100' and '11110', both in terms of mean and median AvgAPCC values. The maximum difference of the mean AvgAPCC between FICTCP and MICTCP reaches more than 0.02.

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

TABLE 4: Statistical pairwise AvgAPCC comparison between MICTCP and FICTCP.

d	MICTCP	FICTCP	Flex	Grep	Gzip	Make	Sed
2	'11'	'10'	0.17/0.56	0.60/0.52	0.42/0.53	0.86/0.51	0.03/0.59
	'101'		0.03/0.59	0.51/0.53	0.52/0.47	0.48/0.47	0.41/0.53
3	'110'	'100'	0.27/0.55	0.16/0.44	0.73/0.51	0.94/0.50	0.15/0.56
	'111'		0.29/0.54	0.27/0.54	0.51/0.53	0.22/0.55	0.74/0.51
	'1001'		2.8E-05/0.67	1.0E-10/0.76	0.91/0.50	0.86/0.49	0.14/0.56
	'1010'		6.2E-10/0.75	2.9E-12/0.79	0.28/0.46	0.67/0.52	0.01/0.61
	'1011'		5.2E-09/0.74	8.5E-17/0.84	0.66/0.48	0.42/0.53	9.6E-03/0.61
4	'1100'	'1000'	6.9E-04/0.64	2.1E-03/0.63	0.72/0.49	0.18/0.45	0.45/0.53
	'1101'		8.4E-05/0.66	1.9E-08/0.73	0.40/0.47	7.0E-03/0.39	0.13/0.56
	'1110'		4.6E-04/0.64	2.9E-06/0.69	0.89/0.49	0.83/0.49	0.02/0.60
	'1111'		3.2E-06/0.69	3.0E-09/0.74	0.57/0.48	0.08/0.43	0.23/0.55
	'10001'		3.2E-15/0.82	8.6E-25/0.92	0.21/0.45	0.61/0.48	3.1E-10/0.76
	'10010'		5.1E-26/0.93	2.3E-32/0.98	0.78/0.51	0.44/0.53	5.5E-12/0.78
	'10011'		8.0E-28/0.95	1.1E-32/0.99	0.57/0.52	0.46/0.47	1.5E-11/0.78
	'10100'		1.9E-30/0.97	3.2E-32/0.98	0.88/0.51	0.51/0.47	1.4E-08/0.73
	'10101'		1.4E-31/0.98	8.7E-34/1.00	0.37/0.46	0.74/0.49	3.4E-10/0.76
	'10110'		1.1E-30/0.97	1.0E-33/1.00	0.80/0.51	0.52/0.53	9.8E-15/0.82
	'10111'		4.0E-33/0.99	7.1E-34/1.00	0.75/0.49	0.99/0.50	8.8E-13/0.79
5	'11000'	'10000'	1.3E-19/0.87	3.9E-20/0.88	0.40/0.47	0.92/0.50	3.3E-03/0.62
	'11001'		5.7E-26/0.93	1.6E-31/0.98	0.80/0.49	0.83/0.49	3.1E-07/0.71
	'11010'		4.8E-31/0.97	9.9E-33/0.99	0.65/0.48	0.43/0.47	2.6E-09/0.74
	'11011'		4.6E-32/0.98	5.8E-33/0.99	0.87/0.49	0.60/0.48	1.6E-12/0.79
	'11100'		1.0E-29/0.96	2.3E-30/0.97	0.45/0.47	0.23/0.45	0.12/0.44
	'11101'		2.8E-31/0.98	8.8E-33/0.99	0.34/0.46	0.43/0.47	0.52/0.53
	'11110'		6.5E-31/0.97	1.8E-33/0.99	0.72/0.49	0.22/0.45	0.88/0.51
	'11111'		1.9E-30/0.97	5.3E-34/1.00	0.13/0.44	0.09/0.43	0.28/0.54
	'100001'		4.8E-24/0.91	2.0E-32/0.99	0.65/0.52	0.69/0.52	2.9E-17/0.85
	'100010'		3.2E-34/1.00	2.7E-34/1.00	0.05/0.58	0.64/0.52	1.1E-28/0.95
	'100011'		5.8E-34/1.00	2.6E-34/1.00	0.33/0.54	0.99/0.50	1.2E-31/0.98
	'100100'		2.6E-34/1.00	2.7E-34/1.00	0.02/0.60	5.6E-03/0.61	9.1E-29/0.96
	'100101'		2.6E-34/1.00	2.6E-34/1.00	3.0E-03/0.62	6.8E-03/0.61	3.4E-30/0.97
	'100110'		2.6E-34/1.00	2.6E-34/1.00	1.7E-04/0.65	7.8E-03/0.61	2.8E-31/0.98
	'100111'		2.6E-34/1.00	2.6E-34/1.00	0.02/0.60	0.15/0.56	1.8E-32/0.99
	'101000'		2.6E-34/1.00	2.8E-34/1.00	7.1E-04/0.64	0.04/0.58	2.5E-28/0.95
	'101001'		2.6E-34/1.00	2.6E-34/1.00	3.8E-03/0.62	0.12/0.56	2.5E-31/0.98
	'101010'		2.6E-34/1.00	2.6E-34/1.00	7.1E-03/0.61	0.15/0.56	6.7E-32/0.98
	'101011'		2.6E-34/1.00	2.6E-34/1.00	6.1E-03/0.61	0.08/0.57	6.5E-32/0.98
	'101100'		2.6E-34/1.00	2.6E-34/1.00	0.01/0.60	5.2E-04/0.64	2.4E-24/0.92
	'101101'		2.6E-34/1.00	2.6E-34/1.00	1.9E-04/0.65	0.02/0.60	1.4E-25/0.93
	'101110'		2.6E-34/1.00	2.6E-34/1.00	6.1E-03/0.61	0.04/0.59	2.1E-26/0.94
	'101111'		2.6E-34/1.00	2.6E-34/1.00	0.05/0.58	0.02/0.60	3.7E-26/0.93
6	'110000'	'100000'	2.7E-34/1.00	8.3E-32/0.98	0.12/0.56	5.3E-03/0.61	8.0E-13/0.79
	'110001'		2.6E-34/1.00	2.7E-34/1.00	0.39/0.54	0.24/0.55	2.5E-26/0.93
	'110010'		2.6E-34/1.00	2.6E-34/1.00	0.07/0.58	0.11/0.56	1.6E-28/0.95
	'110011'		2.6E-34/1.00	2.6E-34/1.00	0.08/0.57	0.20/0.55	1.8E-28/0.95
	'110100'		2.6E-34/1.00	2.6E-34/1.00	0.15/0.56	3.7E-03/0.62	2.8E-31/0.98
	'110101'		2.6E-34/1.00	2.6E-34/1.00	5.5E-03/0.61	0.02/0.59	2.2E-32/0.99
	'110110'		2.6E-34/1.00	2.6E-34/1.00	2.0E-03/0.63	0.01/0.60	2.7E-32/0.98
	'110111'		2.6E-34/1.00	2.6E-34/1.00	0.02/0.59	0.05/0.58	3.1E-32/0.98
	'111000'		2.6E-34/1.00	2.9E-34/1.00	0.11/0.56	0.46/0.53	2.3E-28/0.95
	'111001'		2.6E-34/1.00	2.6E-34/1.00	3.5E-03/0.62	0.04/0.59	2.4E-29/0.96
	'111010'		2.6E-34/1.00	2.6E-34/1.00	1.9E-03/0.63	0.02/0.59	1.2E-30/0.97
	'111011'		2.6E-34/1.00	2.6E-34/1.00	3.9E-03/0.62	4.7E-03/0.62	2.4E-32/0.98
	'111100'		2.6E-34/1.00	2.6E-34/1.00	2.3E-03/0.62	0.06/0.58	5.0E-18/0.85
	'111101'		2.6E-34/1.00	2.6E-34/1.00	4.2E-02/0.58	3.7E-03/0.62	1.9E-26/0.94
	'111110'		2.6E-34/1.00	2.6E-34/1.00	8.5E-03/0.61	4.0E-03/0.62	2.0E-22/0.90
	'111111'		2.6E-34/1.00	2.6E-34/1.00	6.0E-03/0.61	1.3E-03/0.63	1.7E-26/0.94

As observed in the statistical results, apart from the last four MICTCP techniques (*i.e.*, '11100', '11101', '11110', and '11111'), all others techniques have highly significant differences compared to FICTCP, because p-values are much less than 0.05. However, all MICTCP techniques (except '11100') have effect size $\hat{A}_{12}$ values which range from 0.51 to 0.82 compared to FICTCP, which means that MICTCP outperforms FICTCP in 51% to 82% of the cases. c) When $d = 6$, all MICTCP techniques have much higher AvgAPCCs than FICTCP, where the minimum difference is about 0.02; and the maximum difference reaches 0.05, both in terms of median and mean values. The p-values of MICTCP compared to FICTCP are much lower than 0.05, which means that their differences are highly significant; while the effect size $\hat{A}_{12}$ values are much higher than 0.50,

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

ranging from 0.79 to 0.99, which means that MICTCP outperforms in 79% to 99% of the cases.

To sum up, when d is high, in most cases MICTCP has similar or better performance than FICTCP, although there are exceptions. Additionally, the statistical analysis supports the box plot observations.

3) With the increase of d, in most cases MICTCP achieves higher differences against FICTCP for all programs, although there exist some fluctuations. In other words, when d is higher, the differences between MICTCP and FICTCP becomes higher. Consider the effect size values for comparing MICTCP against FICTCP, the 5-tuple of $\hat{A}_{12}$ intervals for program Flex is ([0.56, 0.56], [0.54, 0.59], [0.64, 0.75], [0.82, 0.99], [0.91, 1.00]) for five d values, respectively, *i.e.*, $d = 2$, 3, 4, 5, and 6; ([0.52, 0.52], [0.44, 0.54], [0.63, 0.79], [0.88, 1.00], [0.99, 1.00]) for program Grep; ([0.53, 0.53], [0.47, 0.53], [0.46, 0.50], [0.44, 0.51], [0.52, 0.65]) for program Gzip; ([0.51, 0.51], [0.47, 0.55], [0.39, 0.53], [0.43, 0.53], [0.52, 0.64]) for program Make; and ([0.59, 0.59], [0.51, 0.56], [0.53, 0.61], [0.44, 0.82], [0.39, 0.99]) for program Sed. It can be seen that the percentage of the cases where MICTCP outperforms FICTCP generally increases, along with the increase of d.

To sum up, in terms of interaction coverage rates, MICTCP generally performs better than FICTCP in most cases, especially when d is high. However, the better performance of MICTCP compared to FICTCP is not always observed, which means that sometimes FICTCP could have better rates of interaction coverage.

TABLE 5: Mean APCC at each λ for FICTCP and MICTCP.

Object	Technique	Strength Value (λ) 1	2	3	4	5	6	*AvgAPCC*
Flex	'10'	99.64	98.99	96.58	92.61	87.01	79.71	92.42
	'11'	99.66	98.99	96.60	92.64	87.07	79.80	92.46
	100'	99.62	98.93	97.71	94.50	88.86	81.06	93.45
	'111'	99.65	98.97	97.71	94.49	88.87	81.09	93.46
	'1000'	99.58	98.82	97.61	95.40	90.83	83.25	94.25
	'1111'	99.64	98.91	97.68	95.38	90.79	83.25	94.28
	'10000'	99.53	98.69	97.40	95.28	91.65	85.18	94.62
	'11111'	99.64	98.85	97.56	95.36	91.61	85.09	94.69
	'100000'	99.45	98.47	97.08	94.93	91.45	85.78	94.53
	'111111'	99.64	98.81	97.47	95.25	91.61	85.74	94.75
Grep	'10'	99.29	97.61	91.73	84.24	76.42	68.86	86.36
	'11'	99.32	97.61	91.75	84.25	76.43	68.87	86.37
	'100'	99.25	97.52	93.93	87.56	79.60	71.26	88.19
	'111'	99.32	97.59	93.93	87.53	79.58	71.26	88.20
	'1000'	99.19	97.35	93.85	88.56	81.60	73.62	89.03
	'1111'	99.32	97.52	93.91	88.53	81.53	73.55	89.06
	'10000'	99.06	97.16	93.64	88.50	81.80	74.05	89.04
	'11111'	99.32	97.46	93.84	88.55	81.75	73.98	89.15
	'100000'	98.79	96.83	93.34	88.30	81.76	74.18	88.87
	'111111'	99.32	97.40	93.75	88.49	81.79	74.10	89.14
Gzip	'10'	98.71	97.31	94.46	90.58	86.17	81.57	91.47
	'11'	98.70	97.30	94.47	90.62	86.24	81.67	91.50
	'100'	98.69	97.29	95.14	92.15	88.51	84.42	92.70
	'111'	98.70	97.30	95.15	92.17	88.52	84.44	92.71
	'1000'	98.71	97.32	95.17	92.41	89.20	85.61	93.07
	'1111'	98.70	97.30	95.15	92.40	89.19	85.61	93.06
	'10000'	98.70	97.30	95.13	92.33	89.08	85.52	93.01
	'11111'	98.70	97.30	95.12	92.31	89.05	85.48	92.99
	'100000'	98.70	97.29	95.13	92.34	89.08	85.49	93.01
	'111111'	98.71	97.31	95.14	92.34	89.09	85.53	93.02
Make	'10'	99.08	98.10	95.83	91.97	86.69	80.32	92.00
	'11'	99.08	98.11	95.82	91.96	86.69	80.33	92.00
	'100'	99.07	98.10	96.56	93.55	88.71	82.26	93.04
	'111'	99.08	98.11	96.57	93.57	88.75	82.31	93.07
	'1000'	99.07	98.09	96.55	94.12	90.09	84.18	93.68
	'1111'	99.06	98.09	96.55	94.10	90.07	84.15	93.67
	'10000'	99.03	98.08	96.55	94.12	90.43	85.11	93.89
	'11111'	99.07	98.09	96.55	94.11	90.40	85.04	93.88
	'100000'	98.99	98.07	96.53	94.07	90.41	85.27	93.89
	'111111'	99.07	98.09	96.53	94.10	90.43	85.25	93.91
Sed	'10'	99.13	96.97	90.25	83.20	76.64	70.73	86.15
	'11'	99.16	96.96	90.34	83.31	76.74	70.82	86.22
	'100'	99.11	96.92	92.71	86.99	80.66	74.36	88.46
	'111'	99.14	96.94	92.71	86.98	80.66	74.36	88.47
	'1000'	99.08	96.87	92.70	87.30	81.40	75.49	88.81
	'1111'	99.13	96.92	92.71	87.29	81.38	75.46	88.82
	'10000'	99.04	96.80	92.65	87.30	81.47	75.60	88.81
	'11111'	99.13	96.89	92.70	87.29	81.40	75.49	88.82
	'100000'	98.99	96.73	92.58	87.27	81.49	75.65	88.79
	'111111'	99.13	96.86	92.67	87.30	81.46	75.59	88.84

2) Analysis

In this section, we briefly analyze the above AvgAPCC and APCC observations. For ease of description, let the strength λ be used in the calculation of APCC.

On the one hand, FICTCP adopts a fixed strength d $(1 \leq d \leq 6)$ to prioritize model inputs, which means that it attempts to cover d-wise value combinations as quickly as possible. On the other hand, MICTCP adopts more than one strength, less than or equal to d (from which the prioritization strength d is bound to be chosen), to guide the prioritization, which means that there exists a balance among different strengths. In other words, MICTCP may satisfy more strengths to sacrifice the strength d.

Our results are consistent with three cases, listed as follows:

- **Case 1:** When d (used in FICTCP and MICTCP) is equal to λ used in APCC, FICTCP with the strength d achieves higher or similar d-wise APCCs to MICTCP, because FICTCP uses the number of uncovered λ-wise value combinations as the prioritization criterion (although there are a few exceptions due to *local optimization* being used instead of *global optimization* in FICTCP).
- **Case 2:** When d is higher than λ, MICTCP would have prioritized model inputs with higher APCCs than FICTCP, because MICTCP considers the strengths lower than d during the prioritization process, which may deliver better λ-wise APCC values.
- **Case 3:** When d is less than λ, it is difficult to distinguish which one is better between FICTCP and MICTCP for obtaining prioritized model inputs with higher APCCs, but they could perform similarly. The main reason for this is that: Considering a small d (for example, $d = 2$ or 3), when all d-wise value combinations have been covered by already selected model inputs (indicating that all value combinations at the strengths less than d have also been covered), the remaining candidate model inputs could be prioritized randomly for both MICTCP and FICTCP.

Table 5 describes a λ-wise $(1 \leq \lambda \leq 6)$ APCC comparison of FICTCP and d-strength MICTCP for each subject program, from which the above three cases could be validated. As shown in Eq.(7), the AvgAPCC metric is the average result of six APCC values at strengths from 1 to 6. As a consequence, when d is small (such as $d = 2, 3$), the AvgAPCC difference between FICTCP and MICTCP could be small, due to **Case 1**, and **Case 3**. However, when d is high (such as $d = 4, 5, 6$), the difference generally seems highly significant, due to **Case 1**, and **Case 2**.

In a few cases of Gzip and Make, such as $d = 4$ and $d = 5$, their AvgAPCC observations slightly conflict with the

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

above explanations. The main reason may be the special input parameter model of each program. As shown in Table 2, the model of program Gizp is Model$(14, 2^{13}3^1, \mathcal{C})$, $|\mathcal{C}| = 69$, and the model of program Make is Model$(10, 2^{10}, \mathcal{C})$, $|\mathcal{C}| = 28$. Both of them have nearly all parameters with binary values, and many constraints among parameter values, resulting in the case that FICTCP with $d = 4$ or $d = 5$ could achieves considerable (even better) λ-wise ($1 \leq \lambda < d$) APCCs compared to MICTCP. As shown in Table 5, for program Gzip, FICTCP with $d = 4$ (*i.e.*, '1000') and $d = 5$ (*i.e.*, '10000') can have higher or equal λ-wise APCC values compared to d-strengths MICTCP (*i.e.*, '1111' for $d = 4$, and '11111' for $d = 5$). Similarly, for program Make, the '1000' is equal to or better than the '1111' in terms of APCCs at each λ value; and the '10000' has lower APCCs at $\lambda = 1, 2$ than the '11111', but it has higher APCCs at other λ values (*i.e.*, $\lambda = 3, 4, 5$).

*To answer **RQ1** therefore, MICTCP does not always achieve better rates of interaction coverage compared with FICTCP, even though it uses more information to support the prioritization of model inputs. Nevertheless, MICTCP performs better than FICTCP in many cases, especially when d is high.*

B. RQ2: FAULT DETECTION EXPERIMENTS

Figures 7 to 11 present the APFD results for different d values, each of which contains five sub-figures for subject programs Flex, Grep, Gzip, Make, and Sed. Each plot shows the distribution of the 500 APFD values (*i.e.*, 100 orderings $\times$ 5 versions). Table 6 records statistical pairwise APFD comparisons of MICTCP and FICTCP.

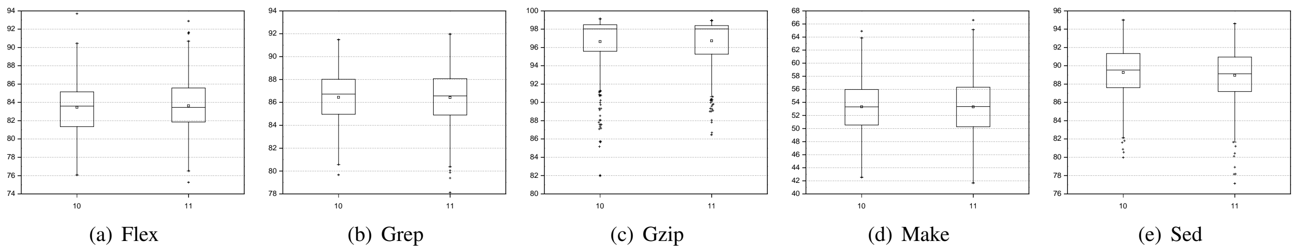
1) Observations

Based on the experimental data, we have made the following observations:

1) For all five programs (Flex, Grep, Gzip, Make, and Sed) with all d values, MICTCP has similar APFDs to FICTCP. From the figures it can be seen that the maximum APFD difference between FICTCP and MICTCP is around 1%, both in terms of mean and median APFD values.

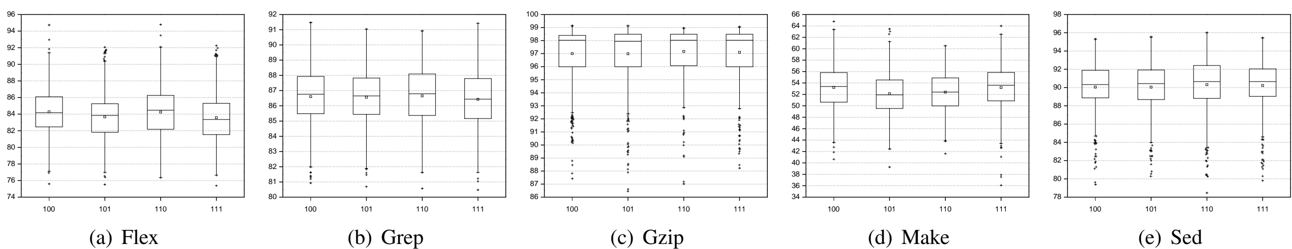
2) For some cases, MICTCP performs slightly better than FICTCP, such as program Grep with $d = 6$. However, FICTCP also achieves slightly higher APFDs in some cases, such as program Make with $d = 5$ and $d = 6$. In other words, there is no technique that is always the best.

3) The statistical analysis overall validates the box plot observations. More specifically, the majority of p-values (220 out of 285 cases being equal to 77%) are greater than 0.05,



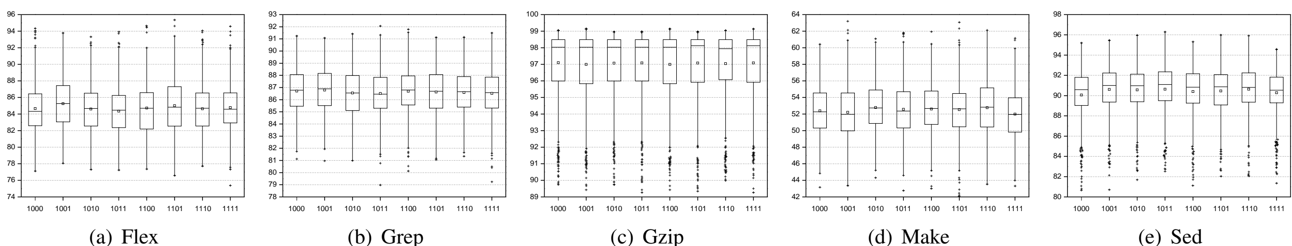
(a) Flex (b) Grep (c) Gzip (d) Make (e) Sed

FIGURE 7: APFD metric values for each program when $d = 2$.



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

FIGURE 8: APFD metric values for each program when $d = 3$.



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

FIGURE 9: APFD metric values for each program when $d = 4$.

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

TABLE 6: Statistical pairwise APFD comparison between MICTCP and FICTCP.

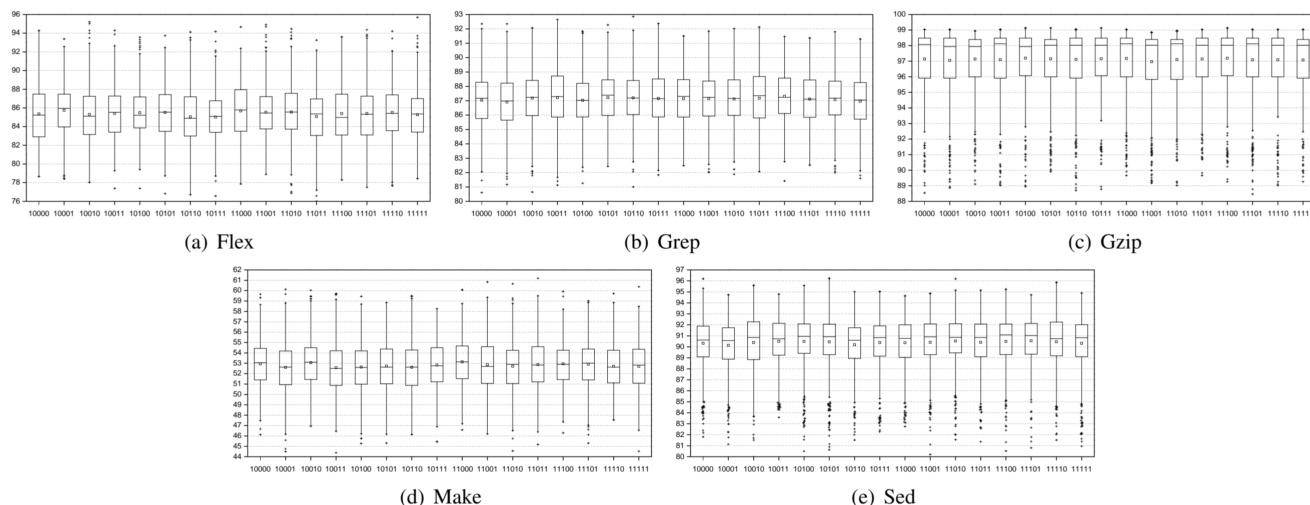
d	MICTCP	FICTCP	Flex	Grep	Gzip	Make	Sed
2	'11'	'10'	0.49/0.51	0.95/0.50	0.95/0.50	0.96/0.50	0.08/0.47
3	'101'	100	7.3E-04/0.44	0.71/0.49	0.97/0.50	3.4E-06/0.42	0.94/0.50
	'110'		0.96/0.50	0.64/0.51	0.76/0.51	4.2E-04/0.44	0.03/0.54
	'111'		1.7E-05/0.42	0.08/0.47	0.46/0.51	0.54/0.51	0.12/0.53
4	'1001	1000	2.2E-04/0.57	0.44/0.51	0.88/0.50	0.29/0.48	5.7E-04/0.56
	'1010'		0.55/0.51	0.21/0.48	0.88/0.50	0.06/0.54	2.4E-03/0.56
	'1011'		0.38/0.48	0.08/0.47	0.96/0.50	0.48/0.51	3.0E-04/0.57
	'1100'		0.57/0.51	0.99/0.50	0.84/0.50	0.12/0.53	0.02/0.54
	'1101'		0.04/0.54	0.62/0.49	0.80/0.50	0.48/0.51	0.03/0.54
	'1110'		0.60/0.51	0.31/0.48	0.70/0.49	0.04/0.54	1.8E-03/0.56
	'1111'		0.24/0.52	0.17/0.48	0.67/0.51	0.04/0.46	0.29/0.52
5	'10001	10000	7.9E-03/0.55	0.04/0.54	0.79/0.50	0.02/0.46	0.30/0.48
	'10010'		0.86/0.50	0.01/0.55	0.73/0.51	0.75/0.51	0.33/0.52
	'10011'		0.58/0.51	0.52/0.51	0.85/0.50	9.0E-03/0.45	0.24/0.52
	'10100'		0.37/0.52	0.01/0.54	0.85/0.50	0.03/0.46	0.07/0.53
	'10101'		0.27/0.52	0.03/0.54	0.65/0.51	0.11/0.47	0.11/0.53
	'10110'		0.16/0.47	0.08/0.53	0.60/0.49	0.02/0.46	0.38/0.48
	'10111'		0.19/0.48	0.08/0.53	0.56/0.49	0.40/0.48	0.51/0.51
	'11000'		0.04/0.54	0.06/0.53	0.59/0.49	0.28/0.52	0.56/0.51
	'11001'		0.24/0.52	0.11/0.53	0.45/0.51	0.35/0.48	0.29/0.52
	'11010'		0.12/0.53	0.03/0.54	0.34/0.52	0.12/0.47	0.11/0.53
	'11011'		0.40/0.48	2.5E-03/0.56	0.63/0.49	0.62/0.49	0.31/0.52
	'11100'		0.98/0.50	0.20/0.52	0.57/0.49	0.66/0.49	0.10/0.53
	'11101'		0.87/0.50	0.12/0.53	0.88/0.50	0.83/0.50	0.07/0.53
	'11110'		0.24/0.52	0.66/0.51	0.64/0.49	0.06/0.47	0.33/0.52
	'11111'		0.99/0.50	0.10/0.53	0.38/0.52	0.18/0.48	0.48/0.51
6	'100001'	100000	0.52/0.51	5.5E-03/0.55	0.34/0.52	0.02/0.46	0.13/0.47
	'100010'		0.29/0.52	2.3E-04/0.57	0.01/0.55	2.2E-03/0.44	0.14/0.53
	'100011'		0.37/0.48	4.2E-06/0.58	0.08/0.53	0.02/0.46	0.08/0.47
	'100100'		0.42/0.51	0.01/0.55	0.40/0.52	0.47/0.49	0.70/0.51
	'100101'		0.47/0.51	4.7E-06/0.58	0.88/0.50	0.91/0.50	6.9E-03/0.45
	'100110'		0.54/0.51	3.1E-04/0.57	0.50/0.49	0.07/0.47	0.28/0.48
	'100111'		0.27/0.52	0.49/0.51	0.11/0.53	0.74/0.49	0.97/0.50
	'101000'		0.65/0.51	1.1E-04/0.57	0.15/0.53	0.08/0.47	0.32/0.48
	'101001'		0.82/0.50	5.0E-03/0.55	0.43/0.51	3.8E-03/0.45	0.09/0.47
	'101010'		0.44/0.51	3.3E-04/0.57	0.87/0.50	0.04/0.46	0.13/0.47
	'101011'		0.34/0.52	0.01/0.55	0.23/0.52	0.07/0.47	0.61/0.51
	'101100'		0.49/0.49	3.9E-03/0.55	0.71/0.49	0.68/0.49	0.55/0.49
	'101101'		0.73/0.51	0.20/0.52	0.47/0.51	5.3E-03/0.45	0.02/0.46
	'101110'		0.38/0.48	9.4E-03/0.55	0.09/0.53	0.04/0.46	0.82/0.50
	'101111'		0.03/0.46	0.06/0.53	0.67/0.51	0.84/0.50	0.37/0.48
	'110000'		1.6E-03/0.56	0.18/0.52	0.04/0.54	0.04/0.46	0.21/0.48
	'110001'		0.04/0.54	0.14/0.53	0.04/0.54	0.04/0.46	9.6E-04/0.44
	'110010'		0.31/0.52	8.1E-03/0.55	0.32/0.52	0.04/0.46	0.33/0.48
	'110011'		0.92/0.50	5.8E-03/0.55	0.29/0.52	0.26/0.48	0.06/0.46
	'110100'		0.58/0.49	0.04/0.54	1.00/0.50	0.66/0.51	0.30/0.48
	'110101'		0.16/0.53	0.04/0.54	0.56/0.49	0.69/0.49	0.45/0.49
	'110110'		0.19/0.48	5.7E-05/0.57	0.68/0.49	0.12/0.47	0.24/0.52
	'110111'		0.84/0.50	0.17/0.53	0.21/0.52	0.07/0.47	0.80/0.50
	'111000'		0.41/0.51	0.02/0.54	0.56/0.51	0.02/0.46	0.33/0.52
	'111001'		0.04/0.54	0.03/0.54	0.58/0.51	0.59/0.49	0.74/0.49
	'111010'		0.06/0.53	0.10/0.53	0.43/0.51	0.67/0.51	0.17/0.48
	'111011'		0.74/0.49	0.03/0.54	0.55/0.51	0.04/0.46	0.73/0.51
	'111100'		0.99/0.50	0.18/0.52	0.87/0.50	0.02/0.46	0.37/0.48
	'111101'		0.80/0.50	5.5E-05/0.57	0.41/0.52	0.50/0.49	0.55/0.49
	'111110'		0.53/0.49	0.03/0.54	0.20/0.52	0.44/0.51	0.87/0.50
	'111111'		0.83/0.50	1.7E-03/0.56	0.19/0.52	0.85/0.50	0.95/0.50

which means that only 23% of cases are less than 0.05. In other words, in most cases the APFD differences between MICTCP and FICTCP are not highly significant. As for the effect size $\hat{A}_{12}$ values, when d is high, *i.e.*, $d = 4$, 5, and 6, the most $\hat{A}_{12}$ values are greater than 0.50 for all programs except program Make. Additionally, the percentages of the cases with $\hat{A}_{12}$ values higher than, equaling to, and less than 0.50 are equal to approximately $150/285 = 53\%$, $39/285 = 14\%$, and $96/285 = 33\%$, respectively. As a consequence, MICTCP performs slightly better than FICTCP in about 53% of the cases; while the opposite situation occurs in 33% of the cases (*i.e.*, FICTCP performs slightly better than MICTCP).

To conclude, MICTCP has very similar rates of fault detection compared with FICTCP, and there is no consistently superior technique for prioritizing model input-

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access



(a) Flex (b) Grep (c) Gzip

(d) Make (e) Sed

FIGURE 10: APFD metric values for each program when $d = 5$.

s. Nevertheless, MICTCP slightly performs better than FICTCP in more than 50% **of the cases.**

2) Analysis

In this section, we briefly present an analysis of above APFD observations.

When FICTCP uses the prioritization strength d to prioritize model inputs, MICTCP uses d and other prioritization strengths lower than d. As discussed in Section IV-A, MICTCP could achieve a balance between d and other strengths during the prioritization process, which could result in the following two cases: 1) MICTCP achieves the lower rates of covering d-wise value combinations than FICTCP; and 2) MICTCP achieves higher rates of covering value combinations at strengths lower than d than FICTCP.

As we know, each program fault could be triggered by a number of parameters, *i.e.*, the *failure-triggering fault interaction* (FTFI) number [31], or the *failure-causing parameter interaction* (FCPI) number [46]. For example, when the FTFI number of a fault is equal to 2, *i.e.*, this fault is caused by two parameters and testing all pairwise value combinations is bound to identify such a fault. Consider a fault F with the FTFI number τ ($1 \leq \tau \leq k$), where k is the number of parameters of the input parameter model, there exist the following three cases:

- **Case 1:** If $d = \tau$, *i.e.*, the fault F could be triggered by d parameters; FICTCP may detect F more quickly than MICTCP, because FICTCP could cover d-wise value combinations sooner.
- **Case 2:** If $d > \tau$, *i.e.*, the fault F could be identified by less than d parameters; MICTCP may perform more effectively than FICTCP to detect F, because MICTCP could cover τ-wise value combinations more quickly.
- **Case 3:** If $d < \tau$, *i.e.*, the fault F could be caused by more than d parameters; it is difficult to distinguish whether MICTCP or FICTCP is better, because they do not aim to cover τ-wise value combinations as soon as possible.

As shown in Table 7, the FTFI number of each τ is given for each program. It can been seen that for subject programs Flex, Grep, Gzip, and Sed, over 50% of faults are with FTFI numbers being equal to less than 3; while for program Make, most of faults are with the FTFI number higher than 6. Therefore, it would be expected that MICTCP could have better rates of fault detection than FICTCP, especially when d is high. As investigated in Section IV-B, the observations do not fully support such as assertion.

There are two possible reasons:

1) The above three cases are satisfied in many cases rather than in all cases, because covering λ-wise value combinations could also achieve a degree of covering value combinations at a strength either higher or lower than λ. For example, an element from candidates selected as the next model input that covers the largest number of uncovered λ-wise value combinations can also cover a number of uncovered λ'-wise value combinations, where $1 \leq \lambda < \lambda' \leq k$. Similarly, this model input may cover a certain number of uncovered λ''-wise value combinations, where $1 \leq \lambda'' < \lambda \leq k$.

2) As we know, two faults have the same FTFI number, yet they may have different properties. For example, the FTFI number of two faults F_1 and F_2 is equal to 2, *i.e.*, these faults are caused by two parameters p_1 and p_2. The parameters p_1 and p_2 have the values from the sets V_1 and V_2, respectively. Suppose that the fault F_1 could be triggered

TABLE 7: The FTFI number of each program.

Object	FTFI Number (τ)							*Sum*
	1	2	3	4	5	6	6+	
Flex	10	33	38	18	8	29	13	149
Grep	8	94	81	55	46	0	1	285
Gzip	23	10	2	0	2	0	0	37
Make	0	0	0	0	5	4	142	151
Sed	4	46	34	9	0	0	0	93

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

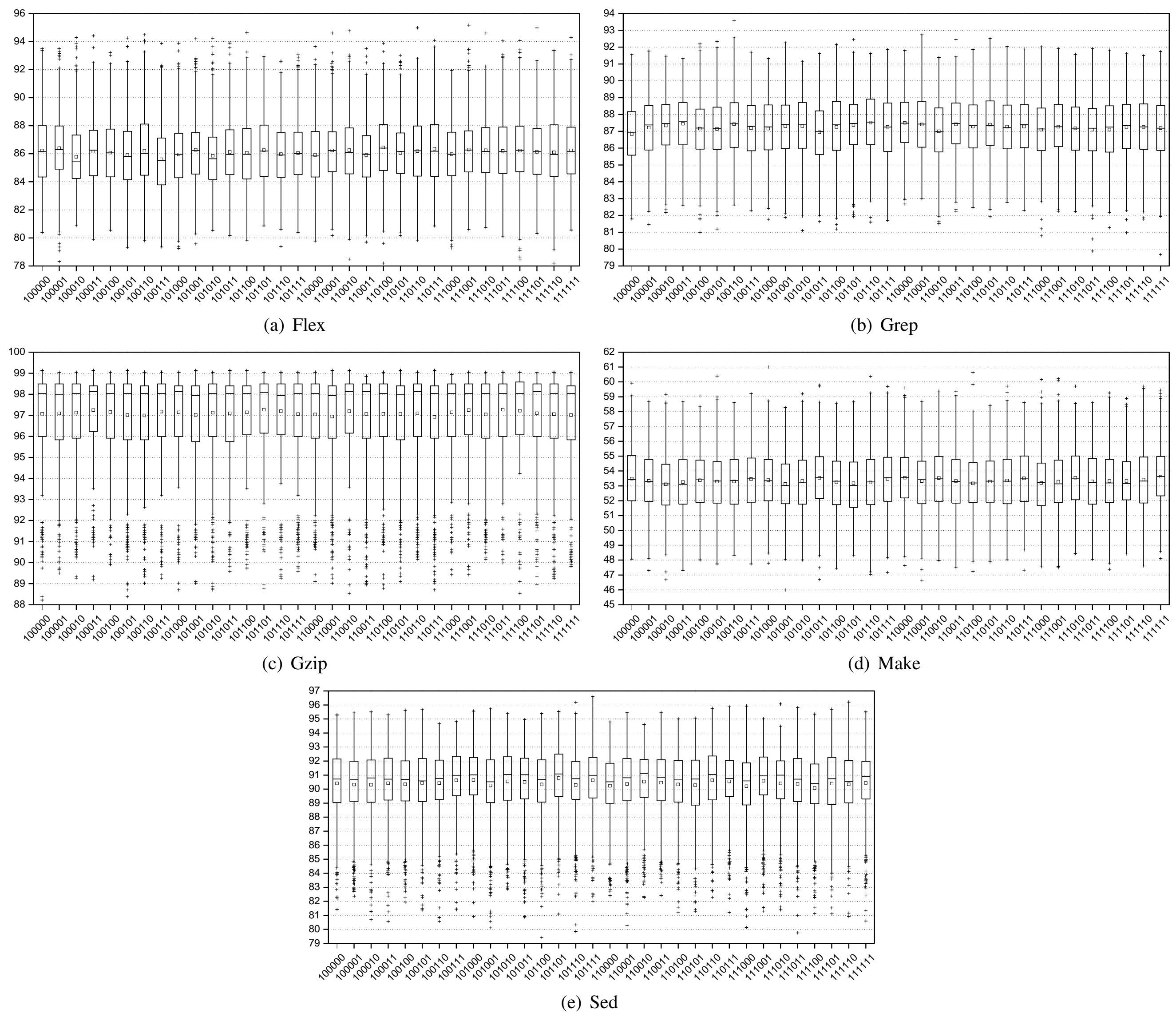


FIGURE 11: APFD metric values for each program when $d = 6$.

by the 2-wise interaction $(p_1 = v_1)\&\&(p_2 = v_2)$; and the fault F_2 could be identified by the 2-wise interaction $(p_1 \neq v_1)\&\&(p_2 \neq v_2)$, where $v_1 \in V_1$ and $v_2 \in V_2$. Therefore, the probability of detecting F_1 (also *failure rate* of F_1) is equal to $\theta_1 = \frac{1}{|V_1| \times |V_2|}$; while the probability of detecting F_2 is $\theta_2 = \frac{(|V_1|-1)\times(|V_2|-1)}{|V_1| \times |V_2|}$. With the increase of $|V_1|$ and $|V_2|$, the failure rates of F_1 and F_2 are significantly different, *i.e.*, θ_1 approaches 0.0; while θ_2 is close to 1.0. In other words, the F_1 becomes more difficult to detect, and the F_2 is easier to identify. As a consequence, the fault with a low FTFI number may not be as easily identified as another fault with high FTFI number, because the former may have a lower failure rate. As discussed in Section III-C, all subsuming faults were removed from the experiments, which means that most of the seeding faults have low failure rates.

*To answer **RQ2**, the APFD differences between FICTCP and MICTCP are approximately 1%, which means that FICTCP and MICTCP have very similar rates of fault detection. In addition, there is no best prioritization technique, which means that MICTCP does not always outperform FICTCP, because in some cases FICTCP can achieve better performance.*

C. RQ3: PRIORITIZATION COST EXPERIMENTS

In this section, we present the prioritization cost of each ICTCP technique for each program. Table 8 shows the prioritization time of each prioritization technique for each program, from which the prioritization time is represented by the mean and standard deviation (μ/σ) over the 100 runs. In this table, we also show the prioritization cost sum of all programs for each prioritization technique.

From the table, it can be observed that (as expected), MICTCP requires more prioritization time than FICTCP,

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

TABLE 8: Prioritization time (μ/σ) in seconds for MICTCP and FICTCP.

d	Method	Flex	Grep	Gzip	Make	Sed	*Sum*
$d=1$	'1'	0.18/0.01	0.15/0.01	0.03/0.01	0.02/0.00	0.10/0.01	0.48/–
$d=2$	'10'	0.60/0.04	0.51/0.03	0.18/0.01	0.05/0.01	0.48/0.06	1.82/–
	'11'	0.91/0.02	0.67/0.02	0.22/0.01	0.06/0.00	0.50/0.02	2.36/–
$d=3$	'100'	1.43/0.02	1.32/0.04	0.79/0.02	0.16/0.01	1.31/0.02	5.01/–
	'101'	1.64/0.05	1.48/0.05	0.80/0.01	0.16/0.01	1.47/0.02	5.55/–
	'110'	2.23/0.04	1.88/0.07	0.95/0.02	0.20/0.01	1.74/0.02	7.00/–
	'111'	2.37/0.11	2.00/0.05	1.01/0.02	0.22/0.01	1.83/0.02	7.43/–
$d=4$	'1000'	2.43/0.03	2.83/0.06	2.23/0.03	0.28/0.01	3.44/0.06	11.21/–
	'1001'	2.63/0.04	3.03/0.07	2.28/0.02	0.30/0.02	3.55/0.04	11.79/–
	'1010'	3.05/0.04	3.40/0.07	2.43/0.05	0.34/0.02	3.81/0.06	13.03/–
	'1011'	3.30/0.10	3.57/0.09	2.48/0.03	0.35/0.03	3.92/0.05	13.62/–
	'1100'	3.91/0.04	4.40/0.10	3.03/0.02	0.44/0.01	4.77/0.09	16.55/–
	'1101'	4.11/0.06	4.59/0.13	3.08/0.03	0.45/0.01	4.91/0.16	17.14/–
	'1110'	4.52/0.05	5.09/0.12	3.25/0.03	0.50/0.04	5.16/0.05	18.52/–
	'1111'	4.86/0.06	5.10/0.10	3.23/0.03	0.48/0.01	5.30/0.05	18.97/–
$d=5$	'10000'	3.09/0.16	4.98/0.08	4.80/0.04	0.37/0.04	6.39/0.07	19.63/–
	'10001'	3.25/0.13	5.18/0.17	4.90/0.03	0.38/0.01	6.50/0.07	20.21/–
	'10010'	3.70/0.14	5.52/0.06	5.05/0.04	0.41/0.01	6.85/0.06	21.53/–
	'10011'	3.99/0.17	5.69/0.09	5.10/0.04	0.42/0.01	6.95/0.07	22.15/–
	'10100'	4.59/0.15	6.51/0.09	5.56/0.05	0.52/0.01	7.78/0.08	24.96/–
	'10101'	4.90/0.17	6.71/0.08	5.64/0.04	0.53/0.01	7.90/0.07	25.68/–
	'10110'	5.28/0.18	7.26/0.12	5.78/0.06	0.56/0.01	8.18/0.11	27.06/–
	'10111'	5.66/0.25	7.30/0.12	5.85/0.05	0.58/0.01	8.31/0.09	27.70/–
	'11000'	5.65/0.20	7.91/0.11	7.06/0.05	0.64/0.01	9.76/0.10	31.02/–
	'11001'	5.83/0.15	8.16/0.09	7.16/0.04	0.65/0.01	9.89/0.08	31.69/–
	'11010'	6.34/0.15	8.51/0.12	7.32/0.05	0.69/0.01	10.19/0.19	33.05/–
	'11011'	6.67/0.18	8.74/0.09	7.37/0.15	0.71/0.01	10.37/0.07	33.86/–
	'11100'	7.20/0.14	9.74/0.13	7.86/0.06	0.80/0.01	11.08/0.10	36.68/–
	'11101'	7.55/0.12	9.79/0.08	7.91/0.05	0.82/0.02	11.38/0.12	37.45/–
	'11110'	8.15/0.14	10.12/0.13	8.04/0.05	0.86/0.01	11.68/0.07	38.85/–
	'11111'	8.65/0.25	10.11/0.08	8.15/0.15	0.88/0.01	11.58/0.08	39.37/–
$d=6$	'100000'	2.16/0.03	12.65/0.31	8.38/0.05	0.33/0.01	12.06/1.35	35.58/–
	'100001'	2.36/0.05	12.77/0.32	8.43/0.10	0.34/0.01	11.77/0.18	35.67/–
	'100010'	2.82/0.06	13.59/0.48	8.57/0.10	0.38/0.01	12.28/0.19	37.64/–
	'100011'	3.11/0.08	13.71/0.28	8.73/0.36	0.42/0.01	12.66/0.23	38.63/–
	'100100'	3.71/0.11	14.93/0.52	9.93/1.31	0.51/0.01	13.82/0.22	42.90/–
	'100101'	3.98/0.12	15.25/0.29	9.24/0.10	0.52/0.01	14.01/0.23	43.00/–
	'100110'	4.42/0.15	15.99/0.31	9.41/0.10	0.56/0.01	14.37/0.37	44.75/–
	'100111'	4.82/0.18	15.97/0.32	9.46/0.11	0.56/0.03	14.61/0.19	45.42/–
	'101000'	4.88/0.12	19.03/0.41	10.75/0.09	0.62/0.01	17.21/0.54	52.49/–
	'101001'	5.08/0.11	19.55/0.33	10.79/0.10	0.65/0.01	17.28/0.20	53.35/–
	'101010'	5.46/0.13	20.13/0.33	10.98/0.09	0.69/0.01	17.57/0.18	54.83/–
	'101011'	5.86/0.18	20.34/0.39	11.03/0.24	0.69/0.01	17.81/0.32	55.73/–
	'101100'	6.35/0.16	21.82/1.27	11.63/0.08	0.78/0.01	18.63/0.18	59.21/–
	'101101'	6.67/0.25	22.01/0.34	11.63/0.09	0.79/0.02	18.98/0.20	60.08/–
	'101110'	7.06/0.31	22.64/2.12	11.78/0.07	0.83/0.02	19.39/0.34	61.70/–
	'101111'	7.59/0.20	22.32/0.36	11.92/0.09	0.87/0.01	19.26/0.16	61.96/–
	'110000'	5.47/0.18	26.10/0.46	13.84/0.12	0.70/0.01	21.55/0.37	67.66/–
	'110001'	5.68/0.17	27.07/0.48	14.01/0.11	0.74/0.01	21.66/0.18	69.16/–
	'110010'	6.18/0.24	28.13/0.59	14.19/0.10	0.78/0.01	22.17/0.19	71.45/–
	'110011'	6.61/0.16	28.22/0.45	14.22/0.12	0.77/0.02	22.34/0.19	72.16/–
	'110100'	6.96/0.14	29.58/0.51	14.67/0.13	0.87/0.01	23.49/0.21	75.57/–
	'110101'	7.19/0.17	29.28/0.48	14.74/0.14	0.86/0.01	23.88/0.32	75.95/–
	'110110'	7.61/0.14	29.89/0.48	14.92/0.20	0.90/0.02	24.17/0.19	77.49/–
	'110111'	8.03/0.18	30.46/0.95	15.05/0.10	0.96/0.02	24.34/0.22	78.84/–
	'111000'	8.30/0.22	34.16/0.65	16.33/0.13	1.00/0.03	26.13/0.16	85.92/–
	'111001'	8.34/0.21	34.23/0.52	16.41/0.16	1.01/0.02	26.44/0.20	86.43/–
	'111010'	9.18/0.42	34.92/0.54	16.57/0.14	1.04/0.02	26.91/0.20	88.62/–
	'111011'	9.32/0.20	35.01/0.45	16.73/0.11	1.08/0.02	26.94/0.35	89.08/–
	'111100'	9.80/0.24	36.97/1.75	17.24/0.16	1.15/0.03	28.30/0.24	93.46/–
	'111101'	10.10/0.28	36.95/0.50	17.38/0.14	1.22/0.03	28.41/0.21	94.06/–
	'111110'	10.69/0.25	37.53/0.49	17.56/0.13	1.26/0.03	28.66/0.26	95.70/–
	'111111'	11.00/0.30	37.28/4.82	17.79/0.22	1.26/0.01	28.89/0.23	96.22/–

because it adopts more strength information to guide the prioritization of model inputs. Additionally, MICTCP with lower strengths (such as 1 and 2) could have comparative prioritization costs to FICTCP.

As we know, when using AvgAPCC as the evaluation metric, we believe that MICTCP with lower strengths (such as 1 and 2) may be more cost-effective than FICTCP, because MICTCP has better rates of interaction coverage than FICTCP, but requires considerable prioritization time. However, when using APFD as the evaluation metric, we believe that FICTCP would be more preferable than MICTCP, because it has very similar rates of fault detection, but needs less prioritization cost. It can be seen that using different evaluation metrics, the results seem contradictory. However, AvgAPCC and APFD have the same mechanism, but they have been used as the metric for different testing scenarios.

More specifically, APCC measures the rate of interaction coverage at a given strength λ, *i.e.*, the speed covering λ-wise value combinations. In effect, APCC guarantees that all λ-wise value combinations have the same probability to be failure-causing, because they provide the same contribution during the APCC calculation. Therefore, APCC has two assumptions, described as following: 1) the FTFI number is equal to λ; and 2) each λ-wise value combination is failure-causing with the same probability. In addition, the AvgAPCC measures the average APCCs based on the strengths ranged from 1 to 6, which means that it considers value combinations coverage at different strengths. In other words, higher speed to cover value combinations at each strength, higher AvgAPCC values. Similar to APCC, AvgAPCC also has the following two assumptions: 1) the FTFI numbers range from 1 to 6; and 2) each value combination at a fixed strength has the same probability to be failure-causing. From this perspective, therefore AvgAPCC has the same mechanism as APFD.

On the one hand, when using AvgAPCC as the evaluation metric, *i.e.*, considering each value combination is failure-causing, there exist many faults with low and high failure rates. Therefore, the AvgAPCC experiments (as shown in Section IV-A) represent an initial stage of software testing. This is because, during the life span of the software, many types of faults (such as low and high failure rates) exist. On the other hand, as discussed in Section IV-B, most of the faults have low failure rates, which means that these faults are difficult to detect. In other words, the APFD experiments represent a later stage of software testing. This is because, usually more and more faults will be detected and removed from the code, *i.e.*, the software is being tested or maintained.

*To answer **RQ3**, we need to discuss this research question according to different metrics of testing effectiveness, i.e., MICTCP with low strengths (such as 1 and 2) will be more cost-effective than FICTCP at an initial stage of software testing; while it will be more cost-effective to use FICTCP instead of MICTCP at a later stage of software testing.*

D. RQ4: SELECTION OF FICTCP AND MICTCP

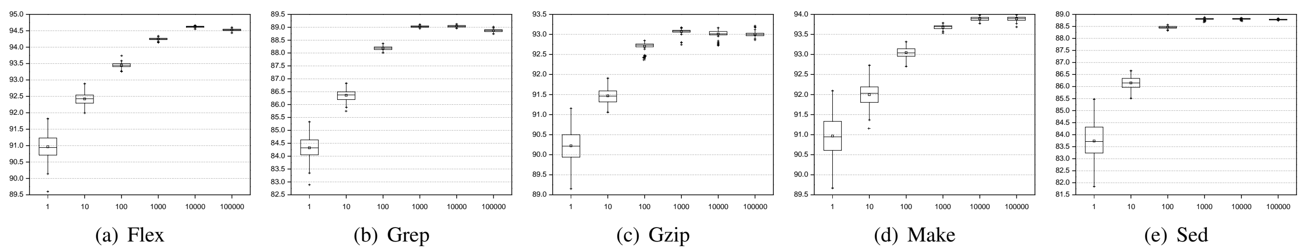
In this section, we attempt to present some guidelines for testers about the best choice of prioritization technique (FICTCP or MICTCP).

Figure 12 shows the AvgAPCC values of FICTCP with six d values, from which we can observe that FICTCP with $d = 1$ is worst, followed by that with $d = 2$ and $d = 3$. However, 4-wise, 5-wise, and 6-wise FICTCP perform similarly. However, FICTCP with $d = 5$ overall performs best. As shown in Table 9, the statistical analysis validates the above observations. However, Figure 13 shows the APFD results of FICTCP with different d values, from which it can be observed that in terms of mean and median APFD values, the maximum APFD difference between them is approximately 5%. However, FICTCP with the high strength generally achieves better rates of fault detection than FICTCP with low strength. The statistical results for the comparison between these 6 approaches are presented in Table 10, which overall confirms the box-plot observations. In addition, as shown in Table 8, FICTCP requires more prioritization time, with the increase of d.

Similarly, as discussed in Section IV-A, MICTCP with more strengths could obtain better AvgAPCC values, and MICTCP with high d could have better performance than that with low d. However, as discussed in Section IV-B, MICTCP has similar fault detection rates to FICTCP when using the

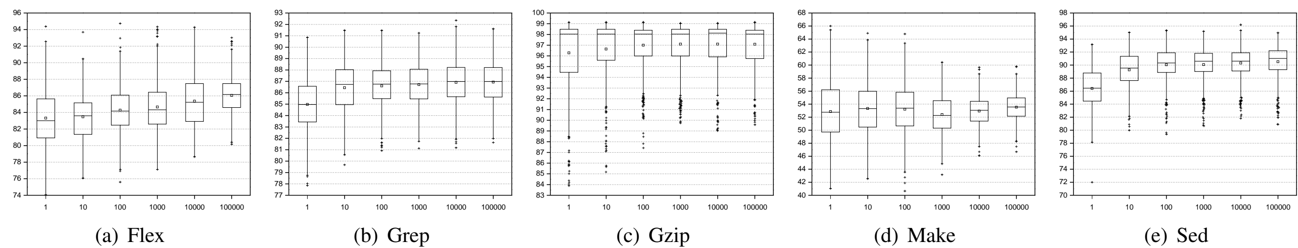
2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access



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

FIGURE 12: AvgAPCC metric values for FICTCP with different d values.



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

FIGURE 13: APFD metric values for FICTCP with different d values.

TABLE 9: Statistical pairwise AvgAPCC comparison between FICTCP techniques ($\mathcal{A}$ and $\mathcal{B}$).

$\mathcal{A}$	$\mathcal{B}$	Flex	Grep	Gzip	Make	Sed
'10'	'1'	2.6E-34/1.00	2.6E-34/1.00	3.1E-34/1.00	5.3E-30/0.97	2.6E-34/1.00
'100'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'1000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'10000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'100000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'100'	'10'	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.7E-34/1.00	2.6E-34/1.00
'1000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'10000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'100000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'1000'	'100'	2.6E-34/1.00	2.6E-34/1.00	2.7E-33/0.99	2.6E-34/1.00	2.6E-34/1.00
'10000'		2.6E-34/1.00	2.6E-34/1.00	5.0E-31/0.97	2.6E-34/1.00	2.6E-34/1.00
'100000'		2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00	2.6E-34/1.00
'10000'	'1000'	2.6E-34/1.00	0.12/0.56	2.6E-11/0.23	2.8E-34/1.00	0.68/0.52
'100000'		2.6E-34/1.00	2.3E-33/0.01	3.8E-16/0.17	1.4E-33/0.99	6.4E-10/0.25
'100000'	'10000'	8.5E-32/0.02	1.1E-33/0.01	0.01/0.40	0.80/0.51	8.5E-14/0.19

same d value; while FICTCP with high d overall performs better than MICTCP with low d. Therefore, MICTCP with high d overall outperforms that with low d. Additionally, as shown in Table 8, MICTCP with higher d and more strengths generally needs more prioritization time.

During the selection of FICTCP and MICTCP, we mainly considered the following two impact factors: testing resources, and testing stage. As discussed in Section IV-C, MICTCP is preferable at an initial stage of software testing; and FICTCP is superior at a later stage of software testing. Therefore, we discuss another impact factor, *i.e.*, how to choose the detailed MICTCP or FICTCP technique from many possible choices (for example, there are 6 FICTCP techniques, and 57 MICTCP techniques), under different testing scenarios. *To answer **RQ4**, Table 11 describes the selection guide for different testing scenarios and testing resources. More specifically, when testing resources are sufficient at an initial stage of software testing, we recommend that MICTCP with '111111' be applied to the prioritization of model inputs; otherwise, MICTCP with a low d is recommended. Similarly, when testing resources are sufficient at a later stage of software testing, we recommend the use of FICTCP with $d = 6$ (i.e., '100000') to prioritize model inputs; otherwise, FICTCP with low d is suggested.*

TABLE 10: Statistical pairwise APFD comparison between FICTCP techniques ($\mathcal{A}$ and $\mathcal{B}$).

$\mathcal{A}$	$\mathcal{B}$	Flex	Grep	Gzip	Make	Sed
'10'	'1'	0.29/0.52	1.9E-23/0.68	0.65/0.51	0.08/0.53	8.4E-43/0.75
'100'		6.2E-07/0.59	7.6E-30/0.71	0.30/0.52	0.11/0.53	1.0E-68/0.82
'1000'		6.9E-10/0.61	9.4E-33/0.72	0.23/0.52	0.10/0.47	4.9E-70/0.82
'10000'		8.2E-21/0.67	5.1E-39/0.74	0.12/0.53	0.47/0.51	2.3E-77/0.84
'100000'		1.2E-42/0.75	2.3E-40/0.74	0.37/0.52	1.0E-03/0.56	6.4E-82/0.85
'100'	'10'	9.8E-06/0.58	0.47/0.51	0.56/0.51	0.86/0.50	1.3E-06/0.59
'1000'		1.7E-08/0.60	0.20/0.52	0.45/0.51	1.2E-04/0.43	1.5E-07/0.60
'10000'		9.6E-21/0.67	3.1E-03/0.55	0.25/0.52	0.13/0.47	2.6E-10/0.62
'100000'		1.8E-48/0.77	2.0E-03/0.56	0.75/0.51	0.21/0.52	5.6E-15/0.64
'1000'	'100'	0.22/0.52	0.60/0.51	0.82/0.50	1.0E-04/0.43	0.66/0.51
'10000'		3.0E-07/0.59	0.02/0.54	0.62/0.51	0.12/0.47	0.10/0.53
'100000'		1.8E-25/0.69	0.01/0.54	0.72/0.49	0.17/0.53	5.3E-04/0.56
'10000'	'1000'	1.4E-04/0.57	0.07/0.53	0.94/0.50	1.1E-03/0.56	0.25/0.52
'100000'		1.3E-19/0.67	0.06/0.53	0.34/0.48	1.9E-11/0.62	1.4E-03/0.56
'100000'	'10000'	6.7E-06/0.58	0.95/0.50	0.29/0.48	7.0E-05/0.57	0.04/0.54

E. LIMITATIONS OF THIS WORK

It is possible that the results of this work may not be valid when applied to the general population of software. The major threats to external validity are the programs and their faults. We used five programs in our experiments, therefore, it may be difficult to generalize the results for all other programs. However, we believe that each program with six versions is sufficient to draw comparative conclusions, and these programs have been widely used in the prioritization field [1], [7], [10], [18], [22], [26], [27]. The faults in each version of each program were based on mutation testing, however these faults were obtained from previous test case

TABLE 11: Guidelines for choosing FICTCP and MICTCP.

		Testing Resource	
		Sufficient	**Insufficient**
Testing Scenario	**Initial Stage**	'111111'	MICTCP with low d
	Later Stage	'100000'	FICTCP with low d

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

prioritization results [10]. Considering this, additional studies including more programs and more sets of faults are required to minimize these threats. Additionally, as discussed before, the input parameter model and candidate model input set may influence the generalization of the conclusions, so we would like to conduct more empirical studies to further investigate our method, especially in larger and more complex systems, and with different sets of model inputs.

In terms of the internal validity, choice of the maximum strength (*i.e.*, d) was set as 6, according to previous investigations [30], [31]. However, no studies attempt to adopt strength higher than 6 to guide the prioritization of model inputs. Additional studies with higher d values may reduce this threat.

V. RELATED WORK

In this section, we present some related work about combinatorial interaction testing, and test case prioritization.

A. COMBINATORIAL INTERACTION TESTING

Combinatorial interaction testing (CIT) [15] is a black-box testing method taht aims to generate an effective test suite (a *covering array* [47]) to identify faults that are caused by the parameter interactions. As discussed in Wu and Nie [48], the research field of CIT can be divided into six areas: *Model*, *Generation*, *Optimization*, *Evaluation*, *Diagnosis*, and *Application*. Here, we briefly introduce the most important work on CIT, since the year 2010 (see the survey reference for details about earlier CIT work [15], [49]).

Model: As discussed in Section I-A, the model for CIT is intended to identify parameters, values, and constraints. Segall *et al.* [50] proposed two methods to construct parameters and values of the model, which considerably reduces the complexity of the modeling task. Satish *et al.* [51], [52] adopted UML activity and sequence diagrams to extract the parameters and values for the model, respectively. Arcaini *et al.* [53] attempted to validate the models by checking that the constraints were consistent; that there was no constraint implied by the other constraints; and that the parameters and their values were really necessary. Gargantini *et al.* [54] used search-based CIT to validate constraints among configuration parameters. Tzoref-Brill *et al.* [55] applied three different forms of visualization (matrices, graphs, and treemaps) to visualize the relationships between the different elements of the model. Satish *et al.* [56] proposed a method to build combinatorial test input model from use case artifacts. Tzoref-Brill and Maoz [57] proposed a syntactic and semantic differencing technique for combinatorial models of test designs that defines a concise and canonical representation for differences between two models.

Generation: This area of research aims to generate as small test suites as possible. It is the most active area of CIT, and there are many popular algorithms or tools, such as *Automatic Efficient Test Generator* (AETG) [58], Pairwise *Independent Combinatorial Testing* (PICT) [[59], *Advanced Combinatorial Testing System* (ACTS) [60], [61], and *Covering Arrays by Simulated Annealing* (CASA) [62]. Recently, there are many algorithms for CIT test suite generation using different search-based techniques, such as Swarm Optimization [63]–[65], Harmony Search [66], Genetic Algorithm [67], [68], and Hyperheuristic Search [69], [70]. Additionally, there are other construction tools using different information, such as coverage inheritance [71], Two-Mode Meta-Heuristic Algorithm [72], unsatisfiable cores [73], interaction trees [74], combinatorial optimization [75], balance between frequencies and fault detection [76], and similarity or distance [20], [77]. More algorithms and tools have been listed in an orchestrated survey [78].

Optimization: This area of research focusses on prioritizing or minimizing the number of test cases in the CIT test suite. Different information can be used to guide the prioritization or minimization of test cases, such as similarity [19], [20], [79], interaction coverage [17], [80]–[83], test case cost [42], [84], switching cost [85], and model mutation [10]. For more details about optimization of CIT, please see our earlier work [86].

Evaluation: This area of research includes the assessment of CIT and metrics to evaluate test suites constructed by different CIT tools. Felbinger *et al.* [87] presented a quality assessment of CIT test suites, according to mutation score, coverage, and model inference. Some work [88]–[91] compared CIT with random testing [92], and adaptive random testing [93], from the perspective of fault detection, code coverage, and interaction coverage. Petke *et al.* [18], [27] conducted empirical studies to investigate three CIT test suite constructors, and compared different prioritization strengths for prioritizing CIT test suites, based on testing effectiveness and efficiency. Choi *et al.* [94] evaluated the testing effectiveness of prioritized CIT through a case study; while Medeiros *et al.* compared 10 sampling algorithms (including 5 CIT algorithms) for configurable systems. Kuhn *et al.* [95] presented many combinatorial coverage metrics for evaluating CIT test suites, Chen and Zhang [96] proposed a metric called *tuple density*, which measures CIT test suites by considering higher-strength interaction coverage. Wang *et al.* [42] proposed some metrics to evaluate prioritized CIT test suites by considering test case cost and weight; while Huang *et al.* [41] proposed a series of metrics for prioritized CIT test suites by taking account of different levels of interaction coverage.

Diagnosis: This area of research centres on locating concrete failure-causing interactions. Zhang and Zhang [46] proposed a fault characterization method, called *faulty interaction characterization*. Many kinds of information have been used to guide the diagnosis, such as failure-inducing combinations [97], [98], test augmentation and classification [99], constraint solving and optimization [100], tuple relation tree [101], partial covering array [102], and logistic regression [103].

Application: This approach is to apply CIT to different testing environments and system applications. On the one hand, there are many testing environments that adopt the

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

principles of CIT, for example, even sequence testing [104]–[106], grammar-based testing [107], security testing [108]–[110], scenario-based testing [111], solution testing [112], and certificate testing [113]. On the other hand, many system applications have been tested by CIT, including MP3 applications [114], [115], concurrent programs [116], [117], cloud environments [118], mobile applications [119]–[121], software products lines [19], [20], big data applications [122], industrial settings [123], web applications [124]–[126], and cyber-physical systems [127].

It should be noted that we only present some representative examples of CIT research rather than all CIT work (for more details and studies, please see the CIT repository [48]).

B. TEST CASE PRIORITIZATION

Test case prioritization (TCP) aims at ordering a set of test cases to achieve an early optimization based on preferred properties [1]. It gives an approach the ability to execute highly important test cases earlier, according to some criteria. Here, we present the main TCP approaches, based on different knowledge to guide the prioritization process.

Coverage-based TCP: This approach uses the code information to support the prioritization process, such as function coverage, branch coverage, and statement coverage [128]. There are two main coverage-based prioritization approaches [2]: *total coverage-based TCP* and *additional coverage-based TCP*. The former selects each element from the candidates as the next test case such that it achieves the highest coverage; while the latter chooses the next test case such that it has the highest coverage of uncovered code or statement by already executed tests.

Search-based TCP: This approach has quite a number of different implementation algorithms such as Greedy [4], [129], Genetic Algorithms (GA) [4], [130], Ant-Colony [131], Adaptive Random Sequences [6]–[9], and others [132], [133]. Experiments by Li [4] showed that GA is worse than a greedy algorithm on computer-generated data. However, the application of a search-based TCP technique may differ based on different factors, such as the selected test suite, fitness function, and the like. The current results showed the major benefit of GA in TCP, but there are some drawbacks, such as the time-consuming nature of the process.

Requirements-based TCP: This approach uses system requirements information to prioritize test cases. Srikanth *et al.* [134] proposed a new approach to prioritize system test cases based on four factors: requirements volatility, customer priority, implementation complexity, and fault-proneness of the requirements. Recently work by Srikanth [135], showed that the combination of two or more factors may provide better testing effectiveness than a single factor. Muthusamy *et al.* [136] proposed a requirement-based TCP approach, based on traceability, completeness, the impact of a fault in requirements, changes in requirements, customer priority, and developers views.

Risk-based TCP: This approach concerns on the potential risks existed the software to be developed. Srikanth *et al.* [135], proposed two risk-based TCP techniques, based on the risk information of the system. Some researchers used the information of requirement risks to prioritize test cases that were expected to distinguish the faults related to the risks of the system [137]. Hettiarachchi *et al.* [138] proposed five steps to use requirement risk values for prioritizing test cases.

Fault-based TCP: This approach attempts to prioritize test cases to identify certain targeted faults that can be detected when executing particular statements. Yu and Lau [139] proposed a fault-based TCP approach, by adopting a new effectiveness metric, the *Fault Adequate Test Size* (FATS), which is used to determine the size of the minimal fault adequate subset. Higher FATS values mean lower chances of detecting all targeted faults.

History-based TCP: This approach uses history data for prioritizing test cases. Kim and Porter [140] proposed a history-based TCP technique by assigning the weight for each test case based on history data such as the count of executions which detected a fault. Khalilian *et al.* [141] proposed an extension of Kim and Porter's history-based TCP approach [140], aiming to improve fault detection rates.

Other TCP approaches using different information, such as Bayesian-Network-based TCP [142], and cost-aware-based TCP [143], also exist [144], [145].

VI. CONCLUSION AND FUTURE WORK

Model input prioritization aims to schedule the model inputs so that the more important elements will be selected to be run earlier. Interaction coverage has been widely used in the prioritization of model inputs, and is called *interaction coverage-based test case prioritization* (ICTCP). Previous studies have focused on fixed-strength interaction coverage for supporting the ICTCP (*fixed-strength ICTCP*, FICTCP). When mixed-strength is used for ICTCP (*mixed-strength ICTCP*, MICTCP), does it have better performance than fixed-strength? To answer this question, we conducted empirical studies to compare the testing effectiveness and efficiency of MICTCP and FICTCP. We have also provided some practical guidelines for testers choosing between MICTCP and FICTCP, when prioritizing model inputs under different testing scenarios. The results of the empirical studies have the following findings:

1) In terms of the rates of interaction coverage, when the maximum prioritization strength d is small, such as $d = 2, 3$, both FICTCP and MICTCP have similar performance; however, when d is high, such as $d = 4, 5, 6$, MICTCP performs much better than FICTCP in many cases, and their differences are highly significant. Nevertheless, sometimes FICTCP has better rates of interaction coverage than MICTCP.

2) In terms of the rates of fault detection, MICTCP has very similar performance compared with FICTCP in many cases. However, overall MICTCP performs slightly better than FICTCP, although the differences are not highly significant. Nevertheless, sometimes FICTCP has slightly better fault detection rates than MICTCP.

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

3) MICTCP with low strengths (such as 1 and 2) is more cost-effective than FICTCP at the initial testing stage; while this is reversed at the later stages of testing.

4) When testing resources are not sufficient, MICTCP and FICTCP with low d are recommended at the initial and later stages of software testing, respectively. However, when testing resources are sufficient, MICTCP with '111111' and FICTCP with '100000' are suggested for the initial and later stages of testing.

We would also like to use other algorithms to implement the ICTCP, such as search-based prioritization [4], to make our conclusions more generally applicable. In addition, since we have only considered interaction coverage as the prioritization criterion in this paper, we would like to consider additional information such as the switching cost and weight of model inputs, to guide the prioritization in future.

ACKNOWLEDGMENTS

We would like to the anonymous reviewers for their many helpful comments, and would also like to thank Christopher Henard for sharing with us the fault data for the five programs.

REFERENCES

[1] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929–948, 2001.
[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2012.
[3] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," Software Testing, Verification and Reliability, vol. 25, no. 4, pp. 371–396, 2015.
[4] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," IEEE Transactions on Software Engineering, vol. 33, no. 4, pp. 225–237, 2007.
[5] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed, "Multi-objective test case prioritization in highly configurable systems: A case study," Journal of Systems and Software, vol. 122, pp. 287–310, 2016.
[6] J. Chen, L. Zhu, T. Y. Chen, D. Towey, F.-C. Kuo, R. Huang, and Y. Guo, "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," Journal of Systems and Software, vol. 135, pp. 107–125, 2018.
[7] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09), 2009, pp. 233–244.
[8] X. Zhang, T. Y. Chen, and H. Liu, "An application of adaptive random sequence in test case prioritization," in Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE'14), 2014, pp. 126–131.
[9] X. Zhang, X. Xie, and T. Y. Chen, "Test case prioritization using adaptive random sequence with category-partition-based distance," in Proceedings of the 16th International Conference on Software Quality, Reliability and Security (QRS'16), 2016, pp. 374–385.
[10] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in Proceedings of the 38th Interaction Conference on Software Engineering (ICSE'16), 2016, pp. 523–534.
[11] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," Empirical Software Engineering, vol. 11, no. 4, pp. 583–611, 2006.
[12] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," Communications of the ACM, vol. 31, no. 6, pp. 676–686, 1988.
[13] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in Proceedings of the 38th International Conference on Software Engineering (ICSE'16), 2016, pp. 643–654.
[14] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. A. Porter, "Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach," IEEE Transactions on Software Engineering, vol. 40, no. 1, pp. 43–66, 2014.
[15] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Computer Survey, vol. 43, no. 2, pp. 11:1–11:29, 2011.
[16] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and contraints," Information and Software Technology, vol. 48, no. 10, pp. 960–970, 2006.
[17] R. Huang, J. Chen, D. Towey, A. Chan, and Y. Lu, "Aggregate-strength interaction test suite prioritization," Journal of Systems and Software, vol. 99, pp. 36–51, 2015.
[18] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," IEEE Transactions on Software Engineering, vol. 41, no. 9, pp. 901–924, 2015.
[19] M. Al-Hajjaji, T. Thum, J. Meinicke, M. Lochau, and G. Saake, "Similarity-based prioritization in software product-line testing," in Proceedings of the 18th International Software Product Line Conference (SPLC'14), 2014, pp. 197–206.
[20] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," IEEE Transactions on Software Engineering, vol. 40, no. 7, pp. 650–670, 2014.
[21] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation (DoSTA'07), 2007, pp. 1–7.
[22] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07), 2007, pp. 255–264.
[23] R. Huang, W. Zong, J. Chen, D. Towey, Y. Zhou, and D. Chen, "Prioritizing interaction test suites using repeated base choice coverage," in Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC'16), 2016, pp. 174–184.
[24] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," Empirical Software Engineering, vol. 16, no. 1, pp. 61–102, 2011.
[25] GNU FTP Server. http://ftp.gnu.org/.
[26] X. Qu, M. B. Cohen, and K. M. Woolf, "A study in prioritization for higher strength combinatorial testing," in Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13), 2013, pp. 285–294.
[27] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in Proceedings of the 12th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), 2013, pp. 26–36.
[28] cloc: Count Lines of Code. http://cloc.sourceforge.net/.
[29] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empirical Software Engineering, vol. 10, no. 4, pp. 405–435, 2005.
[30] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW-27'02), 2002, pp. 91–95.
[31] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," IEEE Transaction on Software Engineering, vol. 30, no. 6, pp. 418–421, 2004.
[32] R. Huang, J. Chen, D. Chen, and R. Wang, "How to do tie-breaking in prioritization of interaction test suites?" in Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE'14), 2014, pp. 121–125.
[33] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678, 2011.
[34] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria,"

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

IEEE Transactions on Software Engineering, vol. 32, no. 8, pp. 608–624, 2006.

[35] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," IEEE Transactions on Software Engineering, vol. 32, no. 9, pp. 733–752, 2006.

[36] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in Proceedings of the 35th Interaction Conference on Software Engineering (ICSE'13), 2013, pp. 192–201.

[37] Y. Jia and M. Harman, "Higher order mutation testing," Information and Software Technology, vol. 51, no. 10, pp. 1379–1393, 2009.

[38] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST'14), 2014, pp. 21–30.

[39] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC'10), 2010, pp. 300–309.

[40] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16), 2016, pp. 354–365.

[41] R. Huang, D. Towey, J. Chen, and Y. Lu, "New metrics for prioritized interaction test suites," IEICE Transactions on Information and Systems, vol. 97-D, no. 4, pp. 830–841, 2014.

[42] Z. Wang, L. Chen, B. Xu, and Y. Huang, "Cost-cognizant combinatorial test case prioritization," International Journal of Software Engineering and Knowledge Engineering, vol. 21, no. 6, pp. 829–854, 2011.

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

[44] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," Software Testing, Verification and Reliability, vol. 24, no. 3, pp. 219–250, 2014.

[45] A. Vargha and H. D. Delaney, "A critique and improvment of the cl common language effect size statistics of mcgraw and wong," Journal of Education and Behavioral Statistics, vol. 25, no. 2, pp. 101–132, 2000.

[46] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11), 2011, pp. 331–341.

[47] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in Proceedings of the 25th International Conference on Software Engineering (ICSE'03), 2003, pp. 38–48.

[48] H. Wu and C. Nie, "Combinatorial testing repository," gist.nju.edu.cn/ct_repository.

[49] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," Journal of Systems and Software, vol. 86, no. 8, pp. 1978–2001, 2013.

[50] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Simplified modeling of combinatorial test spaces," in Proceedings of the 5th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'12), 2012, pp. 573–579.

[51] P. Satish, K. Sheeba, and K. Rangarajan, "Deriving combinatorial test design model from uml activity diagram," in Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13), 2013, pp. 331–337.

[52] P. Satish, A. Paul, and K. Rangarajan, "Extracting the combinatorial test parameters and values from uml sequence diagrams," in Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14), 2014, pp. 88–97.

[53] P. Arcaini, A. Gargantini, and P. Vavassori, "Validation of models and tests for constrained combinatorial interaction testing," in Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14), 2014, pp. 98–107.

[54] A. Gargantini, J. Petke, M. Radavelli, and P. Vavassori, "Validation of constraints among configuration parameters using search-based combinatorial interaction testing," in Proceedings of the 8th International Symposium on Search Based Software Engineering (SSBSE'16), 2016, pp. 49–63.

[55] R. Tzoref-Brill, P. Wojciak, and S. Maoz, "Visualization of combinatorial models and test plans," in Proceedings of the 38th International Conference on Automated Software Engineering (ICSE'16), 2016, pp. 144–154.

[56] P. Satish, M. B., M. S. Narayan, and K. Rangarajan, "Building combinatorial test input model from use case artefacts," in Proceedings of the 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'17), 2017, pp. 220–228.

[57] R. Tzoref-Brill and S. Maoz, "Syntactic and semantic differencing for combinatorial models of test designs," in Proceedings of the 39th International Conference on Software Engineering (ICSE'17), 2017, pp. 621–631.

[58] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 437–444, 1997.

[59] J. Czerwonka, "Pairwise testing in real world: Practical extensions to test case generators," in Proceedings of the 24th Pacific Northwest Software Quality Conference (PNSQC'06), 2006, pp. 419–430.

[60] K. C. Tai and Y. Lei, "A test generation strategy for pairwise testing," IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 109–111, 2002.

[61] Y. Lei, R. Kacker, D. R. Kuhn, and V. Okun, "Ipog/ipod: Efficient test generation for multi-way software testing," Software Testing, Verification, and Reliability, vol. 18, no. 3, pp. 125–148, 2008.

[62] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE'09), 2009, pp. 13–22.

[63] B. S. Ahmed and K. Z. Zamli, "A variable strength interaction test suites generation strategy using particle swarm optimization," Journal of Systems and Software, vol. 84, no. 12, pp. 2171–2185, 2011.

[64] H. Wu, C. Nie, F.-C. Kuo, H. Leung, and C. J. Colbourn, "A discrete particle swarm optimization for covering array generation," IEEE Transactions on Evolutionary Computation, vol. 19, no. 4, pp. 575–591, 2015.

[65] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading," Information and Software Technology, vol. 86, pp. 20–36, 2017.

[66] A. Alsewari and K. Z. Zamli, "Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support," Information and Software Technology, vol. 54, no. 6, pp. 553–568, 2012.

[67] R. Qi, Z. Wang, and S. Li, "A parallel genetic algorithm based on spark for pairwise test suite generation," Journal of Computer Science and Technology, vol. 31, no. 2, pp. 417–427, 2016.

[68] S. Esfandyari and V. Rafe, "A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy," Information and Software Technology, vol. 94, pp. 165–185, 2018.

[69] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in Proceedings of the 37th International Conference on Software Engineering (ICSE'15), 2015, pp. 540–550.

[70] K. Z. Zamli, F. Din, G. Kendall, and B. S. Ahmed, "An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation," Information Sciences, vol. 399, pp. 121–153, 2017.

[71] A. Calvagna and A. Gargantini, "T-wise combinatorial interaction test suites construction based on coverage inheritance," Software Testing, Verification and Reliability, vol. 22, no. 7, pp. 507–526, 2012.

[72] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation," in Proceedings of the 30th International Conference on Automated Software Engineering (ASE'15), 2015, pp. 1–12.

[73] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E.-H. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16), 2016, pp. 614–624.

[74] C. Song, A. Porter, and J. S. Foster, "itree: efficiently discovering high-coverage configurations using interaction trees," IEEE Transactions on Software Engineering, vol. 40, no. 3, pp. 251–265, 2014.

[75] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," Journal of Systems and Software, vol. 98, pp. 191–207, 2014.

[76] S. Gao, J. Lv, B. Du, C. J. Colbourn, and S. Ma, "Balancing frequencies

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

and fault detection in the in-parameter-order algorithm," Journal of Computer Science and Technology, vol. 30, no. 5, pp. 957–968, 2015.

[77] E.-H. Choi, C. Artho, T. Kitamura, O. Mizuno, and A. Yamada, "Distance-integrated combinatorial testing," in Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE'16), 2016, pp. 93–104.

[78] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE'14), 2014, pp. 323–334.

[79] R. Huang, Y. Zhou, W. Zong, D. Towey, and J. Chen, "An empirical comparison of similarity measures for abstract test case prioritization," in Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC'17), 2017, pp. 3–12.

[80] D. Blue, I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Interaction-based test-suite minimization," in Proceedings of the 35th International Conference on Software Engineering (ICSE'13), 2013, pp. 182–191.

[81] R. Huang, X. Xie, D. Towey, T. Y. Chen, Y. Lu, and J. Chen, "Prioritization of combinatorial test cases by incremental interaction coverage," International Journal of Software Engineering and Knowledge Engineering, vol. 23, no. 10, pp. 1427–1458, 2013.

[82] R. Huang, J. Chen, T. Zhang, R. Wang, and Y. Lu, "Prioritizing variable-strength covering array," in Proceedings of the 37th International Computers, Software & Applications Conference (COMPSAC'13), 2013, pp. 502–511.

[83] R. Huang, J. Chen, Z. Li, R. Wang, and Y. Lu, "Adaptive random prioritization for interaction test suites," in Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14), 2014, pp. 1058–1063.

[84] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," International Journal of System Assurance Engineering and Management, vol. 2, no. 2, pp. 126–134, 2011.

[85] H. Wu, C. Nie, and F.-C. Kuo, "The optimal testing order in the presence of switching cost," Information and Software Technology, vol. 80, pp. 57–72, 2016.

[86] R. Huang, W. Zong, D. Towey, Y. Zhou, and J. Chen, "An empirical examination of abstract test case prioritization techniques," in Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C'17), 2017, pp. 141–143.

[87] H. Felbinger, F. Wotawa, and M. Nica, "Mutation score, coverage, model inference: Quality assessment for t-way combinatorial test-suites," in Proceedings of the 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'17), 2017, pp. 171–180.

[88] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. N. Kacker, and D. R. Kuhn, "An empirical comparison of combinatorial and random testing," in Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14), 2014, pp. 68–77.

[89] A. Calvagna, A. Fornaia, and E. Tramontana, "Random versus combinatorial effectiveness in software conformance testing: a case study," in Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15), 2015, pp. 1797–1802.

[90] C. Nie, H. Wu, X. Niu, F.-C. K. andHareton Leung, and C. J. Colbourn, "Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures," Information and Software Technology, vol. 62, pp. 198–213, 2015.

[91] S. Vilkomir, A. Alluri, D. R. Kuhn, and R. N. Kacker, "Combinatorial and mc/dc coverage levels of random testing," in Proceedings of the 17th International Conference on Software Quality, Reliability and Security Companion (QRS-C'17), 2017, pp. 61–68.

[92] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000-2014)," in Proceedings of the 2nd International Workshop on the Future of Software Engineering (FOSE'14), 2014, pp. 117–132.

[93] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," Journal of Systems and Software, vol. 83, no. 1, pp. 60–66, 2010.

[94] E.-H. Choi, S. Kawabata, O. Mizuno, C. Artho, and T. Kitamura, "Test effectiveness evaluation of prioritized combinatorial testing: A case study," in Proceedings of the 16th International Conference on Software Quality, Reliability and Security (QRS'16), 2016, pp. 61–68.

[95] D. R. Kuhn, I. D. Mendoza, R. N. Kacker, and Y. Lei, "Combinatorial coverage measurement concepts and applications," in Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13), 2013, pp. 352–361.

[96] B. Chen and J. Zhang, "Tuple density: a new metric for combinatorial test suites (nier track)," in Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), 2011, pp. 876–879.

[97] L. S. Ghandehari, Y. Lei, T. Xie, D. R. Kuhn, and R. N. Kacker, "Identifying failure-inducing combinations in a combinatorial test set," in Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST'12), 2012, pp. 370–379.

[98] L. S. Ghandehari, Y. Lei, D. Kung, R. N. Kacker, and D. R. Kuhn, "Fault localization based on failure-inducing combinations," in Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE'13), 2013, pp. 168–177.

[99] K. Shakya, T. Xie, N. Li, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification," in Proceedings of the 5th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'12), 2012, pp. 620–623.

[100] J. Zhang, F. Ma, and Z. Zhang, "Faulty interaction identification via constraint solving and optimization," in Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12), 2012, pp. 186–199.

[101] X. Niu, C. Nie, Y. Lei, and A. T. Chan, "Identifying failure-inducing combinations using tuple relationship tree," in Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'13), 2013, pp. 271–280.

[102] Z. Wang, T. Guo, W. Zhou, W. Zhang, and B. Xu, "Generating partial covering array for locating faulty interactions in combinatorial testing," in Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE'13), 2013, pp. 578–583.

[103] K. Nishiura, E.-H. Choi, and O. Mizuno, "Improving faulty interaction localization using logistic regression," in Proceedings of the 17th International Conference on Software Quality, Reliability and Security (QRS'17), 2017, pp. 138–149.

[104] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," IEEE Transactions on Software Engineering, vol. 37, no. 1, pp. 48–64, 2011.

[105] D. R. Kuhn, J. M. Higdon, J. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in Proceeds of the 5th International Conference on Software Testing, Verification and Validation (ICST'12), 2012, pp. 601–609.

[106] S. Sampath and R. C. Bryce, "Improving the effectiveness of test suite reduction for user-session-based testing of web applications," Information and Software Technology, vol. 54, no. 7, pp. 724–738, 2012.

[107] E. Salecker and S. Glesner, "Combinatorial interaction testing for test selection in grammar-based testing," in Proceedings of the 5th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'12), 2012, pp. 610–619.

[108] J. Bozic, B. Garn, D. E. Simos, and F. Wotawa, "Evaluation of the ipo-family algorithms for test case generation in web security testing," in Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15), 2015, pp. 1–10.

[109] J. Bozic, B. Garn, I. Kapsalis, D. E. Simos, S. Winkler, and F. Wotawa, "Attack pattern-based combinatorial testing with constraints for web security testing," in Proceedings of the 15th International Conference on Software Quality, Reliability and Security (QRS'15), 2015, pp. 207–212.

[110] D. E. Simos, D. R. Kuhn, A. G. Voyiatzis, and R. N. Kacker, "Combinatorial methods in security testing," IEEE Computer, vol. 49, no. 10, pp. 80–83, 2016.

[111] L. du Bousquet, M. Delahaye, and C. Oriat, "Applying a pairwise coverage criteria to scenario-based testing," in Proceedings of the 9th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'16), 2016, pp. 83–91.

[112] A. Sherif, "Combinatorial testing implementations in solutions testing," in Proceedings of the 9th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'16), 2016, pp. 59–64.

[113] K. Kleine and D. E. Simos, "Coveringcerts: Combinatorial methods for x.509 certificate testing," in Proceedings of the 11th International Conference on Software Testing, Verification and Validation (ICST'17), 2017, pp. 69–79.

[114] Z. Zhang, X. Liu, and J. Zhang, "Combinatorial testing on id3v2 tags of mp3 files," in Proceedings of the 5th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'12), 2012, pp. 587–590.

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

[115] S. Wang, T. Wu, Y. Yao, B. Jin, and L. Ding, "Combinatorial testing on mp3 for audio players," in Proceedings of the 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'17), 2017, pp. 273–275.

[116] X. Qi, J. He, and P. Wang, "A mixed-way combinatorial testing for concurrent programs," in Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE'13), 2013, pp. 699–02.

[117] X. Qi, J. He, P. Wang, and H. Zhou, "Variable strength combinatorial testing of concurrent programs," Frontiers of Computer Science, vol. 10, no. 4, pp. 631–643, 2016.

[118] W. Wu, W.-T. Tsai, C. Jin, G. Qi, and L. Jie, "Test-algebra execution in a cloud environment," in Proceedings of the 8th International Symposium on Service Oriented System Engineering (SOSE'14), 2014, pp. 59–69.

[119] S. Vilkomir and B. Amstutz, "Using combinatorial approaches for testing mobile applications," in Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14), 2014, pp. 78–83.

[120] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in Proceedings of the 38th International Conference on Software Engineering (ICSE'16), 2016, pp. 559–570.

[121] S. Vilkomir, "Multi-device coverage testing of mobile applications," Software Quality Journal, vol. 26, no. 2, pp. 197–215, 2018.

[122] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16), 2016, pp. 637–647.

[123] X. Li, R. Gao, W. E. Wong, C. Yang, and D. Li, "Applying combinatorial testing in industrial settings," in Proceedings of the 16th International Conference on Software Quality, Reliability and Security (QRS'16), 2016, pp. 53–60.

[124] W. Wang, S. Sampath, Y. Lei, R. N. Kacker, D. R. Kuhn, and J. Lawrence, "Using combinatorial testing to build navigation graphs for dynamic web applications," Software Testing, Verification and Reliability, vol. 26, no. 4, pp. 318–346, 2016.

[125] M. Raunak, D. R. Kuhn, and R. N. Kacker, "Combinatorial testing of full text search in web applications," in Proceedings of the 17th International Conference on Software Quality, Reliability and Security Companion (QRS-C'17), 2017, pp. 100–107.

[126] X. Qi, Z. Wang, J. Mao, and P. Wang, "Automated testing of web applications using combinatorial strategies," Journal of Computer Science and Technology, vol. 32, no. 1, pp. 199–210, 2017.

[127] A. Arrieta, G. Sagardui, L. Etxeberria, and J. Zander, "Automatic generation of test system instances for configurable cyber-physical systems," Software Quality Journal, vol. 25, no. 3, pp. 1041–1083, 2017.

[128] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," IEEE Transactions on Software Engineering, vol. 42, no. 5, pp. 490–505, 2016.

[129] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," Automated Software Engineering, vol. 19, no. 1, pp. 65–95, 2012.

[130] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," IEEE Transactions on Software Engineering, vol. 42, no. 10, pp. 918–940, 2016.

[131] D. Gao, X. Guo, and L. Zhao, "Test case prioritization for regression testing based on ant colony optimization," in Proceedings of the 6th International Conference on Software Engineering and Service Science (ICSESS'15), 2015, pp. 275–279.

[132] B. Jiang and W. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," Journal of Systems and Software, vol. 105, pp. 91–106, 2015.

[133] S. Eghbali and L. Tahvildari, "Test case prioritization using lexicographical ordering," IEEE Transactions on Software Engineering, vol. 42, no. 12, pp. 1178–1195, 2016.

[134] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE'05), 2005, pp. 62–71.

[135] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors: An industrial study," Information and Software Technology, vol. 69, pp. 71–83, 2016.

[136] T. Muthusamy and K. Seetharaman, "A new effective test case prioritization for regression testing based on prioritization algorithm," International Journal of Applied Information Systems, vol. 6, no. 7, pp. 21–26, 2014.

[137] H. Yoon and B. Choi, "A test case prioritization based on degree of risk exposure and its empirical study," International Journal of Software Engineering and Knowledge Engineering, vol. 21, no. 02, pp. 191–209, 2011.

[138] C. Hettiarachchi, H. Do, and B. Choi, "Effective regression testing using requirements and risks," in Proceedings of the 8th International Conference on Software Security and Reliability (SERE'14), 2014, pp. 157–166.

[139] Y. T. Yu and M. F. Lau, "Fault-based test suite prioritization for specification-based testing," Information and Software Technology, vol. 54, no. 2, pp. 179–202, 2012.

[140] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in Proceedings of the 24th International Conference on Software Engineering (ICSE'02), 2002, pp. 119–129.

[141] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data," Science of Computer Programming, vol. 78, no. 1, pp. 93–116, 2012.

[142] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases based on bayesian networks," in Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07), 2007, pp. 276–290.

[143] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," Journal of Systems and Software, vol. 85, no. 3, pp. 626–637, 2012.

[144] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," Software Quality Journal, vol. 21, no. 3, pp. 445–478, 2013.

[145] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," Information and Software Technology, vol. 93, pp. 74–93, 2018.

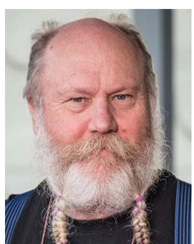
RUBING HUANG (M'12) is an associate professor in the Department of Software Engineering, School of Computer Science and Communication Engineering, Jiangsu University, China. He received the Ph.D. degree from Huazhong University of Science and Technology, China, in computer science and technology. His current research interests include software testing, software security analysis, and software maintenance, especially adaptive random testing, random testing, combinatorial testing, and regression testing. He has more than 40 publications in journals and proceedings, including in ICSE, IEEE-TR, JSS, IST, IET Software, IJSEKE, SCN, COMPSAC, SEKE, and SAC. He has served as the program committee member of SEKE14-19, SAC17-19, and CTA17-19. He is a senior member of the China Computer Federation, and a member of the IEEE and the ACM.

QUANJUN ZHANG is a candidate for the degree of M.Eng in the School of Computer Science and Communication Engineering, Jiangsu University (China). He received the B.Eng. degree in computer science and technology from Jiangsu University. His current research interests include software testing.

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2018.2879638, IEEE Access

TSONG YUEH CHEN (M'03) received the B.Sc. and M.Phil. degrees from The University of Hong Kong, China; the M.Sc. and D.I.C. from the Imperial College of Science and Technology, London, U.K.; and the Ph.D. degree from The University of Melbourne, Australia. He is currently a professor of software engineering in the Department of Computer Science and Software Engineering, Swinburne University of Technology, Australia. He is the inventor of metamorphic testing and adaptive random testing. His main research interest is in software testing.



JAMES HAMLYN-HARRIS is deputy chair of the Department of Computer Science and Software Engineering in the School of Software and Electrical Engineering, Faculty of Science, Engineering and Technology, Swinburne University of Technology, Australia. His qualifications include a Master of IT (Internet Computing) and a Graduate Certificate in eForensics, Ph.D. in Engineering and two Applied Science degrees. He teaches computer programming, data communications and internet security. He supervises research into delay-tolerant networks, network security and software testing.

DAVE TOWEY (M'03) is an associate professor at University of Nottingham Ningbo China (UNNC), in Zhejiang, China, where he serves as the director of teaching and learning, and deputy head of school, for the School of Computer Science. He is also the deputy director of the International Doctoral Innovation Centre at UNNC. He is a member of the UNNC Artificial Intelligence and Optimization research group. His current research interests include software testing and technology enhanced teaching and learning. He received the B.A. and M.A. degrees in computer science, linguistics, and languages from the University of Dublin, Trinity College, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from the University of Hong Kong, China. He co-founded the ICSE International Workshop on Metamorphic Testing in 2016. He is a member of both the ACM and the IEEE.

JINFU CHEN (M'13) received the B.Eng. degree in 2004 from Nanchang Hangkong University, Nanchang, China and the Ph.D. degree in 2009 from Huazhong University of Science and Technology, Wuhan, China, both in computer science and technology. He is currently a full professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include software testing, software analysis, and trusted software.

• • •

2169-3536 (c) 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.