A Theoretical and Empirical Study of Diversity-aware Mutation Adequacy Criterion

Donghwan Shin, *Student Member, IEEE,* Shin Yoo, and Doo-Hwan Bae, *Member, IEEE*

Abstract—Diversity has been widely studied in software testing as a guidance towards effective sampling of test inputs in the vast space of possible program behaviors. However, diversity has received relatively little attention in mutation testing. The traditional mutation adequacy criterion is a one-dimensional measure of the total number of killed mutants. We propose a novel, diversity-aware mutation adequacy criterion called *distinguishing mutation adequacy criterion*, which is fully satisfied when each of the considered mutants can be identified by the set of tests that kill it, thereby encouraging inclusion of more diverse range of tests. This paper presents the formal definition of the distinguishing mutation adequacy and its score. Subsequently, an empirical study investigates the relationship among distinguishing mutation score, fault detection capability, and test suite size. The results show that the distinguishing mutation adequacy criterion detects 1.33 times more unseen faults than the traditional mutation adequacy criterion, at the cost of a 1.56 times increase in test suite size, for adequate test suites that fully satisfies the criteria. The results show a better picture for inadequate test suites; on average, 8.63 times more unseen faults are detected at the cost of a 3.14 times increase in test suite size.

Index Terms—Mutation testing, test adequacy criteria, diversity

1 Introduction

One fundamental limitation of software testing is the fact that, to validate the behavior of the Program Under Test (PUT), we can only ever sample a very small number of test inputs out of the vast input space. Almost all existing testing techniques are, at some level, attempts to answer the following question: *how does one sample a finite number of test inputs to cover as wide a range of program behaviors as possible?*

The concept of diversity has received much attention while answering the above question. For example, Adaptive Random Testing (ART) [1] seeks to increase the diversity of randomly sampled test inputs by choosing an input that is as different from those already sampled as possible. Clustering-based test selection and prioritization [2], [3] assumes that a diverse set of test inputs would explore and validate a wider range of program behaviors. Diversity in test output has been studied as a test adequacy criterion for black box testing of web applications [4]. Information theoretic measures of diversity have also been studied as test selection criteria [5], [6].

In contrast, improving the test effectiveness based on the diversity has received little attention in mutation testing; the emphasis has been instead on the reduction of mutation testing cost. As classified by Jia and Harman [7], a good many studies attempt to reduce the cost by mutant sampling [8], [9], selective mutation [10], higher order mutation [11], [12], mutant clustering [13], [14], and mutant subsumption [15], [16], [17]. However, the foundation of mutation testing (i.e., *mutation adequacy criterion*) remains essentially the same as it was when first proposed in the 1970s [18]. The mutation adequacy criterion is a testing criterion that estimates the real fault detection capability of a test suite by the simple count of the number of artificially generated faulty programs (i.e., *mutants*) distinguished (i.e., *killed*) from its original program. Despite its potential correlation between the diversity of mutants and the real fault detection capability, the mutation-adequate test suite does not fully exploit the diversity. Suppose a pathological case in which a single test can kill all generated mutants. The traditional mutation adequacy criterion simply determines the single test as adequate, although the single test does not consider the diversity of the mutants. If we had a richer mutation adequacy criterion, it would be possible to have more powerful mutation-adequate test suites using the same set of mutants. Such a case calls for a richer mutation adequacy criterion.

To tackle this problem, a novel mutation adequacy criterion called the *distinguishing mutation adequacy criterion* was proposed in our previous paper [19]. At the core of the new criterion lies the idea that mutants can be "*distinguished*" from each other by the set of tests that kill them. Our mutation adequacy criterion aims not only to kill, but also to distinguish as many mutants as possible, leading to a more diverse set of tests based on the same set of mutants. The empirical results on real faults showed that test suites adequate to the distinguishing mutation adequacy criterion can increase the fault detection rate by up to 76.8 percentage points in comparison to the traditional mutation adequate criterion [19]. However, since we considered only 100% adequate test suites for the mutation adequacy criteria, the relationship between the percentage of the mutation adequacy (i.e., *mutation score*) and the fault detection effectiveness was not fully investigated.

In this paper, we significantly extend our previous work in a manner that is both theoretical and empirical. Theoretically, to capture the diversity of mutants in terms of a set of tests, we establish a firm definition of the *mutant distinguishment* as the foundation of the distinguishing mutation

• *D. Shin, S. Yoo, and DH. Bae are with the School of Computing, KAIST, Daejeon, Republic of Korea.*
E-mail: donghwan@se.kaist.ac.kr, shin.yoo@kaist.ac.kr, bae@se.kaist.ac.kr

Manuscript received 10 Aug. 2016.

adequacy criterion. We also define a novel mutation score for the distinguishing mutation adequacy criterion called *distinguishing mutation score*, which measures the percentage of mutants distinguished by a given test suite. Empirically, we provide a comprehensive investigation of the relationships among mutation scores, fault detection effectiveness, and test suite sizes for the traditional and distinguishing mutation adequacy criterion. Further, we measure the correlation between the two different mutation scores and real fault detection.

The technical contributions of this paper are as follows:

- This paper formally describes the distinguishing mutation adequacy criterion in comparison to the traditional mutation adequacy criterion based on theoretical considerations of the mutant distinguishment.
- This paper introduces a novel mutation adequacy score called the *distinguishing mutation score* that extends the distinguishing mutation adequacy criterion. This score represents how large a percentage of mutants are distinguished by a given set of tests.
- The relationships between mutation scores, fault detection effectiveness, and test suite sizes for the traditional and distinguishing mutation adequacy are empirically investigated using real-world faults in the `Defects4J` database [20]. The results show that, on average for all adequate and inadequate test suites, the distinguishing mutation adequacy criterion statistically increases the real fault detection effectiveness for 74.8% of all faults in comparison to the traditional mutation adequacy criterion. In terms of effect size, the distinguishing adequacy is 8.26 times more effective than the traditional adequacy, whereas it requires 3.07 times more tests.
- The correlation between the mutation scores (both traditional mutation score and distinguishing mutation score) and the fault detection is investigated. The results show that the distinguishing mutation score is statistically more correlated with the fault detection than is the traditional mutation score (p-value=2.00e-12), and the $\hat{A}_{12}$ effect size is 0.652.

The rest of the paper is organized as follows. Section 2 presents a formal definition of the existing mutation adequacy criterion. Section 3 introduces the distinguishing mutants adequacy criterion and the distinguishing mutation score using the same formal notations. Section 4 describes the design of our empirical evaluation, the results of which are presented and analyzed in Section 5. Section 6 discusses related work, and Section 7 concludes.

2 Background

Mutation testing is a well-known technique for improving the ability of test suites to detect real faults using mutants (i.e., artificially generated faults). Many mutants are automatically generated from the Program Under Test (PUT) as an original program using a set of *mutation operators* (i.e., predefined rules for changes). A mutant is *killed* by a test when the mutant and the original program show different behaviors for the test. A test suite is assessed by the *mutation adequacy criterion*, which counts the number of killed mutants. If all mutants are killed by a test suite, the test suite is called *mutation-adequate*. It is widely known that mutation-adequate test suites are effective at detecting real faults [21], [22], [23], [24].

2.1 Theoretical Framework for Mutation Testing

To formally represent the notion of mutation adequacy criteria considered in this paper, we summarize the essential elements of the theoretical framework for the mutation-based testing methods. Detailed descriptions for the framework are presented in our previous work [25].

Let P be a set of programs that includes the program under test. In mutation testing, there are three essential programs in P: an original program $p_o \in P$, a mutant m in a set of mutants $M \subset P$ generated from p_o, and a correct program $p_s \in P$ that represents the true requirements[1] about p_o. Let T be a set of all tests. For a test $t \in T$, if the behaviors of p_o and p_s are different, it is said that t *detects* a fault in p_o. Similarly, if the behaviors of p_o and m are different for t, it is said that t *kills* m. Note that the notion of behavioral difference is an abstract concept. It is formalized by a testing factor, called a *test differentiator*, which is defined as follows:

Definition 1 (Test differentiator).
A *test differentiator* $d : T \times P \times P \to \{0, 1\}$ is a function such that

$$d(t, p_x, p_y) = \begin{cases} 1\ (true), & \text{if the behaviors of } p_x \text{ and} \\ & p_y \text{ are } \textit{different} \text{ for } t \\ 0\ (false), & \text{otherwise} \end{cases}$$

for a test $t \in T$ and programs $p_x, p_y \in P$.

By definition, a test differentiator concisely represents whether the behaviors of $p_x \in P$ and $p_y \in P$ are different for $t \in T$. In other words, a test differentiator encapsulates the two abstract concepts: the behavior and the difference. To keep things general, we consider a set of test differentiators D that includes all possible test differentiators for P. However, it is easy to think a specific test differentiator whenever needed. Regarding the behavior, Morell [26] stated that the behavior may include any of the test execution results, for example its output, its internal variables, its execution time, or its space consumption. The specific type of behaviors to be observed can be defined as the observability of a test differentiator. For example, in order to observe only the external results of programs, a test differentiator that observes the external results can be considered. Similarly, the specific definition of the difference can be decided in context. For example, in order to distinguish 1/3 and 0.333, a mathematically strict test differentiator can be considered.

A test differentiator, or simply a differentiator, can formally describe the notion of differences in mutation testing.

1. While p_s is not a real program, this is not a serious assumption, because we only require the behavior of p_s for a given set of tests. In practice, a human may play the role of p_s, acting as a human oracle.

For example, when a test t detects a fault in a program p_o, it is clearly formalized as follows[2]:

$$d(t, p_o, p_s) \neq 0.$$

On the other hand, when t kills a mutant m, it is also clearly formalized as follows:

$$d(t, p_o, m) \neq 0.$$

Note that p_o, p_s, and m are general entities, and largely separated from any specifics such as programming languages or mutation methods.

In addition to a differentiator that formalizes the difference of two programs for a single test, it will be helpful to consider whether the two programs are different for a set of tests. A *d-vector* is defined to represent such difference of the programs as follows:

Definition 2 (d-vector).
A *d-vector* $\mathbf{d} : T^n \times P \times P \to \{0, 1\}^n$ is an n-dimensional vector, such that

$$\mathbf{d}(TS, p_x, p_y) = \langle d(t_1, p_x, p_y), ..., d(t_n, p_x, p_y)\rangle$$

for all $TS = \{t_1, \cdots, t_n\} \subseteq T^n$, $d \in D$, and $p_x, p_y \in P$.

In other words, a differentiator d returns a Boolean value (i.e., 0 or 1) from a single test, whereas a d-vector $\mathbf{d}$ returns n-dimensional Boolean vector from n tests. For example, when a test suite TS detects a fault in a program p_o, a d-vector clearly represents it as follows:

$$\mathbf{d}(TS, p_o, p_s) \neq \mathbf{0}$$

where $\mathbf{0}$ implies the zero vector (i.e., all components equal to zero). Similarly, when TS kills a mutant m, a d-vector also formalizes it as follows:

$$\mathbf{d}(TS, p_o, m) \neq \mathbf{0}.$$

In other words, m is killed by TS when its d-vector is not equal to the zero vector. This theoretical framework with concise representations provides the foundation of the theoretical considerations throughout the remainder of this paper.

2.2 Mutation Adequacy Criterion and Mutation Score

Since mutation testing was first proposed in the 1970s [18], it has been widely studied in the aspects of both theory and practice, and the mutation adequacy criterion has played a key role in the studies of mutation testing. Using a differentiator, the mutation adequacy criterion can be clearly and concisely formalized as follows:

$$\forall m \in M, \mathbf{d}(TS, p_o, m) \neq \mathbf{0}. \tag{1}$$

This means that a test suite TS is mutation-adequate if all mutants $m \in M$ are killed by at least one test $t \in TS$. In other words, if the d-vectors of all mutants for TS are non-zero, the TS is mutation-adequate. To be precise, some of the mutants may not be killed by any test, and their d-vectors are equal to zero for all tests. Such mutants are called *equivalent* mutants. While this concept is not within the main scope of this paper, we will partially discuss equivalent mutants in Section 3.6. For now, however, we focus on non-equivalent mutants. In the rest of this paper, we refer to (1) as the traditional mutation adequacy criterion, in contrast to the diversity-aware mutation adequacy criterion defined in Section 3.4.

Equation (1) is general enough for use in various mutation testing approaches. For example, there is a spectrum of mutation approaches from a strong mutation [18] to a weak mutation [27]. This spectrum depends on the observability of d. If d observes the external execution results of a mutant to decide whether the mutant is killed or not, it is a strong mutation analysis. On the other hand, if d observes the internal states of a mutant to decide whether the mutant is killed or not, it is a weak mutation analysis.

Meanwhile, the percentage of killed mutants is used to quantitatively measure the quality of a test suite. This is called the *mutation score*, or simply the *mScore*, and can be formalized as follows:

$$mScore(TS, M, p_o) = \frac{|\{m \in M \mid \mathbf{d}(TS, p_o, m) \neq \mathbf{0}\}|}{|M|} \tag{2}$$

In other words, the number of mutants killed by TS over all mutants implies the quantitative adequacy of TS in terms of fault detection capability. Thanks to both theoretical and empirical studies that have stated that the fault detection effectiveness of a test suite can be reasonably measured by a mutation score [21], [24], [28], [29], the mutation score is very widely used as an alternative measure of real fault detection in various testing studies [30], [31].

3 Diversity-Aware Mutation Adequacy

In this section, we first consider the limitations of the traditional mutation adequacy, and describe the notion of mutant distinguishment to make room for improvement behind the limitation. We also provide a spatial interpretation to aid in understanding of the mutant distinguishment. Based on the formal definition of the mutant distinguishment, we define the diversity-aware mutation adequacy criterion and the diversity-aware mutation score. We discuss an important issue in the distinguishing mutation adequacy criterion at the end.

3.1 Limitations of Traditional Mutation Adequacy

To find room for improvement in the traditional mutation adequacy criterion, we provide a working example in Fig. 1. Let us assume that we have four different mutants and three different tests. Each of the values represents whether a test kills a mutant. For example, $d(t_1, p_o, m_1)$ is 1, which means that t_1 kills m_1. For $TS = \{t_1, t_2, t_3\}$, the d-vector $\mathbf{d}$ for m_1 is $\mathbf{d}(TS, p_o, m_1) = \langle 1, 0, 0\rangle$.

In the working example, a test suite $TS_1 = \{t_1\}$ is adequate to the traditional mutation adequacy criterion because all four mutants are killed by t_1. However, all the mutants are not distinguished from each other in terms of the kill results for the adequate test suite TS_1. We remind the reader that all mutants by default exhibit different changes to each other, whereas such diversity of mutants does not contribute to the assessment of the quality of a test suite. This is

2. In experiments, when the correct version of a program for a fault is known in advance, the correct version can be used as p_o. In this case, the corresponding faulty version should be used as p_s so that the difference between p_o and p_s implies the fault.

Test	$d(t_i, p_o, m_1)$	$d(t_i, p_o, m_2)$	$d(t_i, p_o, m_3)$	$d(t_i, p_o, m_4)$
t_1	1	1	1	1
t_2	0	0	1	1
t_3	0	1	0	1

Fig. 1. Working example for demonstrating the limitation of the traditional mutation adequacy criterion. The table represents whether a test kills a mutant. For example, $d(t_1, p_o, m_1)$ is 1 which means that t_1 kills m_1.

because the traditional mutation adequacy criterion simply counts the number of killed mutants without considering the diversity of those mutants. In other words, we already have diverse mutants, but the diversity has been wasted in assessing the quality of a test suite. If an adequacy criterion can utilize the diversity of mutants to assess the quality of test suites, the test suites adequate to the criterion will be more effective at detecting faults than other test suites that are adequate to only the simple criterion, which does not consider the diversity. We hypothesize that a more diverse set of tests will show higher fault detection effectiveness, and here we find room for improvement in the diversity of tests by using the diversity of mutants.

The limitation of the traditional mutation score lies in a similar area. In the working example, the three test suites $TS_1 = \{t_1\}$, $TS_2 = \{t_1, t_2\}$, and $TS_3 = \{t_1, t_2, t_3\}$ are clearly different, whereas their traditional mutation scores are all equal to 1 (i.e., 100% of mutants are killed by each of the test suites). This calls for a new mutation score capable of measuring the diversity of mutants in terms of a test suite.

3.2 Mutant Distinguishment

To consider the diversity of mutants in a mutation adequacy criterion, we should define how mutants are distinguished by a test. This is the notion of *mutant distinguishment*. For two mutants m_x and m_y generated from p_o and a test t, we can consider the two different formal descriptions for the mutant distinguishment, as follows:

$$d(t, m_x, m_y) \neq 0, \tag{3}$$

$$d(t, p_o, m_x) \neq d(t, p_o, m_y). \tag{4}$$

This means that the mutant distinguishment can be defined in two different ways: based on direct mutant behaviors for the test, as shown in (3), or based on their kill results for the test, as shown in (4).

Interestingly, there is a nice mathematical relationship between the two descriptions as follows:

$$d(t, p_o, m_x) \neq d(t, p_o, m_y) \implies d(t, m_x, m_y) \neq 0. \tag{5}$$

This means that (3) always holds whenever (4) holds. In other words, if two mutants are distinguished by a test based on their kill results, then the behaviors of the two mutants for the test must be different. The proof is simple: if one of the two mutants m_x and m_y is killed by a test t, then another mutant must not be killed by t when (4) holds. This implies that the two mutants have different behavior for t. However, (3) does not guarantee (4). For example, if each of m_x, m_y, and p_o has different behaviors from the others for t, then (3) holds, whereas (4) does not hold because $d(t, m_x, m_y) \neq 0$ but $d(t, p_o, m_x) = d(t, p_o, m_y) = 1$.

The mathematical relationship between (3) and (4) clearly shows that (4) is a stronger formal description of the notion of mutant distinguishment than (3). We now formally define the *mutant distinguishment* with respect to a single test as follows:

Definition 3 (Mutant distinguishment by a test).

Two mutants m_x and m_y generated from p_o are *distinguished by a test t* if and only if the following condition holds:

$$d(t, p_o, m_x) \neq d(t, p_o, m_y)$$

for a differentiator d.

By definition, two mutants are distinguished by a test when the two mutants' kill results are different for the test. In the working example, the four mutants are undistinguished from each other by t_1 because $d(t_1, p_o, m_i) = 1$ for all $i \in \{1, \cdots, 4\}$. Using t_2, m_1 and m_3 are distinguished, and m_1 and m_4 are distinguished, but m_1 and m_2 are not distinguished. Using t_3, m_1 and m_2 are distinguished, and m_1 and m_4 are distinguished, but m_1 and m_3 are not distinguished.

In terms of a set of tests, we extend the concept of the mutant distinguishment with the aid of a d-vector, as follows:

Definition 4 (Mutant distinguishment by a set of tests).

Two mutants m_x and m_y generated from p_o are *distinguished by a set of tests TS* if and only if the following condition holds:

$$\mathbf{d}(TS, p_o, m_x) \neq \mathbf{d}(TS, p_o, m_y)$$

for a d-vector $\mathbf{d}$ based on a differentiator d.

This means that two mutants are *distinguished by a set of tests* when the mutants' d-vectors (i.e., kill patterns) are different for the set of tests. In other words, two mutants are distinguished by a set of tests if there is at least one test that distinguishes the two mutants. In the working example, if we consider a test suite $TS_2 = \{t_1, t_2\}$, m_1 and m_3 are distinguished, and m_1 and m_4 are distinguished, but m_1 and m_2 are not distinguished. Using another test suite $TS_3 = \{t_1, t_2, t_3\}$, all of the four mutants are distinguished from each other.

We also note that Ammann et al. [16] recently discussed the notion of undistinguished mutants. They stated that if two mutants are killed by precisely the same set of tests, the mutants are undistinguished, even if the mutants may involve different syntactic changes to an original program. This concept is consistent with our definition of the mutant distinguishment. However, there was no attempt to consider the diversity of mutants based on this concept.

3.3 Spatial Interpretation of Mutant Distinguishment

To make the concept of the mutant distinguishment clearer, we provide a spatial interpretation of the mutant distinguishment based on the theoretical framework of mutation-based testing [25].

The basic idea is simple: an n-dimensional vector can represent the position of a point in a n-dimensional space. In

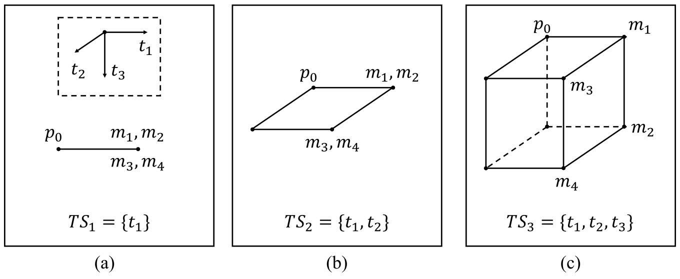


Fig. 2. Positions of mutants for each test suite in the working example. More positions (and their mutants) are distinguished as more tests are added.

the same way, the d-vector $\mathbf{d}(TS, p_o, m)$ of a mutant m for a test suite TS can be regarded as representing the position of m in a multidimensional space composed of TS (i.e., the dimensions) and p_o (i.e., the origin). Considering the space, we can locate mutants at certain positions represented by their d-vectors. In the working example, the space is composed of the three tests (t_1, t_2, t_3) with p_o at the origin. In that space, m_1 is at the point (1,0,0), m_2 is at (1,0,1), m_3 is at (1,1,0), and m_4 is at (1,1,1).

Fig. 2 shows the positions of mutants depending on their test suites corresponding to the working example. In Fig. 2-(a), the four mutants m_1 through m_4 are distinguished from p_o by t_1. As more tests are added to the test suite, the dimensions of the space are extended, and the positions are distinguished. Fig. 2-(c) shows that the four mutants' positions are distinguished by TS_3.

We should note that the spatial interpretation of the mutant distinguishment is related to the mutant subsumption graph introduced by Kurtz et al. [32]. For example, Fig. 2-(c) shows the 3-dimensional space that represents the possible positions of mutants for $TS = \{t_1, t_2, t_3\}$ and p_o. If we remove all positions (i.e., nodes) that do not have any mutants, the remaining graph is equivalent to the mutant subsumption graph. This is because the notion of the mutant distinguishment is closely related to the notion of the mutant subsumption. For more details, please refer to our previous work for a theoretical framework on mutation-based testing [25].

3.4 Distinguishing Mutation Adequacy Criterion

We now define a new mutation adequacy criterion called the *distinguishing mutation adequacy criterion*, as follows:

Definition 5 (Distinguishing mutation adequacy criterion). For a set of mutants M generated from an original program p_o, a test suite TS is *distinguishing mutation-adequate* when the following condition holds:

$$\forall m_x, m_y \in M', \mathbf{d}(TS, p_o, m_x) \neq \mathbf{d}(TS, p_o, m_y)$$

where $m_x \neq m_y$ and $M' = M \cup \{p_o\}$.

In other words, a test suite TS is distinguishing mutation-adequate if all possible pair of different mutants m_x and m_y in M' are distinguished by TS. In the working example, $TS_3 = \{t_1, t_2, t_3\}$ distinguishes all mutants in $M' = \{p_o, m_1, \cdots, m_4\}$ because all mutants have unique d-vectors for TS_3. For the sake of simplicity, we let the d-criterion hereafter refer to the distinguishing mutation adequacy criterion (i.e., diversity-aware) and, similarly, the k-criterion to refer to the traditional mutation adequacy criterion (i.e., kill-only).

It is important to appreciate the role of $p_o \in M'$ in Definition 5. Considering $m_y = p_o$, the d-criterion can be simplified as follows:

$$\forall m_x \in M, \mathbf{d}(TS, p_o, m_x) \neq \mathbf{d}(TS, p_o, p_o). \tag{6}$$

Since it is trivial that $\mathbf{d}(TS, p_o, p_o) = \mathbf{0}$, (6) is exactly the same as (1) (i.e., the k-criterion). This means that the d-criterion *subsumes* the k-criterion. In other words, if a test suite is adequate to the distinguishing mutation adequacy criterion, the test suite is guaranteed to be adequate to the traditional mutation adequacy criterion. This subsumption relationship is especially important because it is closely related to the fault detection effectiveness. Zhu [33] proved that the subsumption relationship guarantees a better fault detection effectiveness when the testing adequacy criteria are used to guide the generation of test suites, not used to generate the test data. For example, it is common that a tester generates tests one by one and adds them to a test suite until a certain testing criterion is satisfied. In this testing scenario, the d-criterion adequate test suites are more effective at detecting faults than the k-criterion adequate test suites.

The difference between the k-criterion and the d-criterion is clearly shown by the difference between Fig. 2-(a) and Fig. 2-(c). As shown in Fig. 2-(a), the k-criterion only distinguishes between the original program (i.e., p_o) and the generated mutants (i.e., $m_1, \cdots, m_4$) and does not distinguish among the generated mutants. On the other hand, Fig. 2-(c) shows that the d-criterion not only distinguishes the original program and the generated mutants but also distinguishes among the generated mutants.

3.5 Distinguishing Mutation Score

As explained in Section 2.2, the mutation score is used to measure the quality of a test suite based on the k-criterion. Similarly, we can define another mutation score called the *distinguishing mutation score*, or simply the $dScore$, based on the d-criterion. Briefly speaking, the $mScore$ implies how many mutants are killed by a test suite, whereas the $dScore$ implies how many mutants are distinguished by a test suite. In terms of d-vectors, the $mScore$ considers the number of non-zero d-vectors for a test suite, whereas the $dScore$ considers the number of unique d-vectors for a test suite.

We remind the reader that two mutants are distinguished by a set of tests when the d-vectors of the mutants are different for the set of tests. In other words, distinguished mutants have unique d-vectors, whereas undistinguished mutants (i.e., mutants that are not distinguished from each other) have the same d-vector. Thus, the number of unique d-vectors is equal to the number of distinguished mutants. Using the number of unique d-vectors, we define the distinguishing mutation score as follows:

Definition 6 (Distinguishing mutation score). For a test suite TS, the *distinguishing mutation score*, or

Test suite	Undistinguished mutant sets	$mScore$	$dScore$
$TS_1 = \{t_1\}$	$\{p_0\}$, $\{m_1, m_2, m_3, m_4\}$	4/4 = 1.0	2/5 = 0.4
$TS_2 = \{t_1, t_2\}$	$\{p_0\}$, $\{m_1, m_2\}$, $\{m_3, m_4\}$	4/4 = 1.0	3/5 = 0.6
$TS_3 = \{t_1, t_2, t_3\}$	$\{p_0\}$, $\{m_1\}$, $\{m_2\}$, $\{m_3\}$, $\{m_4\}$	4/4 = 1.0	5/5 = 1.0

Fig. 3. $mScore$ and $dScore$ for each test suite corresponding to the working example. According to the $mScore$, all the test suites are equal. However, their $dScore$s are different.

the $dScore$, using a set of mutants M generated from an original program p_o is

$$dScore(TS, M, p_o) = \frac{|\{\mathbf{d}(TS, p_o, m) \mid m \in M'\}|}{|M'|}$$

where $M' = M \cup \{p_o\}$.

In other words, the $dScore$ is the number of unique d-vectors given by a test suite over the maximum number of unique d-vectors when all mutants are distinguished from each other. Again, some of mutants may not be distinguishable by any test, and we now focus on the distinguishable mutants. We will discuss the universally indistinguishable mutants in Section 3.6.

On the other hand, a unique d-vector implies a set of undistinguished mutants (i.e., a set of mutants that are not distinguished from each other), because all mutants that are not distinguished from each other have the same d-vector. Thus, the number of unique d-vectors is equal to the number of sets of undistinguished mutants. This means that the $dScore$ represents the number of mutant sets left undistinguished by a given test suite over the maximum number of undistinguished mutant sets (i.e., M' when all sets of undistinguished mutants are singleton).

In the working example, the $mScore$ and the $dScore$ are measured for each test suite as shown in Fig. 3. At first, all mutants are killed by $TS_1 = \{t_1\}$ so that the $mScore$ = 4/4 = 1.0. At the same time, TS_1 separates the undistinguished mutant sets into two: one for the live mutants and the other for the killed mutants. The number of undistinguished mutant sets (or the number of unique d-vectors) is 2 and the $dScore$ = 2/5 = 0.4. After t_2 is added, the $mScore$ of $TS_2 = \{t_1, t_2\}$ does not change, whereas the $dScore$ changes because the newly added test t_2 distinguishes m_1 and m_2 (and m_3 and m_4). The number of undistinguished mutant sets by TS_2 is 3 which means $dScore$ = 3/5 = 0.6. By adding t_3 to the test suite, all the five mutants are distinguished, each mutant has a unique d-vector, all sets of undistinguished mutants are singleton, and the $dScore$ becomes 5/5 = 1.

The difference between the $mScore$ and the $dScore$ is clearly shown in Fig. 2-(a) and Fig. 2-(c). Based on the spatial interpretation, it is very intuitive that the $mScore$ implies how many mutants are distinguished from the original program, whereas the $dScore$ implies how many mutants are distinguished to each other.

3.6 Universally Indistinguishable Mutants

In mutation testing, it may be the case that a mutant and an original program are syntactically different but semantically equivalent, so that there is no test to kill the mutant. Formally, there is a mutant $m_e \in M$ generated from an original program p_o such that

$$\mathbf{d}(T, p_o, m_e) = \mathbf{0} \tag{7}$$

for all tests T. The mutant m_e is called an *equivalent* mutant. Unfortunately, whether a mutant is equivalent or not is undecidable [34]. However, many researchers have attempted to tackle this problem with practical approximation [35], [36], [37].

Under the d-criterion, it may be the case that two mutants are syntactically different but semantically equivalent, so that there is no test to distinguish the mutants. In other words, there are two mutants $m_x, m_y \in M'$ such that

$$\mathbf{d}(T, p_o, m_x) = \mathbf{d}(T, p_o, m_y). \tag{8}$$

We say that m_x and m_y are *universally indistinguishable* mutants. Similar to the case of equivalent mutant detection, deciding whether two mutants are universally indistinguishable or not is clearly undecidable. While attempts to solve this problem are not within the scope of this paper, we want to show how this problem is closely related to the traditional equivalent mutant detection problem.

When a mutant is equivalent, the mutant and its original program are universally indistinguishable. The proof is quite simple. Since p_o is in M', we can set $m_y = p_o$ in (8). This leads to $\mathbf{d}(T, p_o, m_y) = \mathbf{d}(T, p_o, p_o) = \mathbf{0}$, so that (8) becomes equal to (7). This shows that both the equivalent mutant detection (i.e., whether a mutant is equivalent to its original program or not) and the universally indistinguishable mutants detection (i.e., whether two mutants are universally indistinguishable or not) are a problem of the equivalence between the two programs for all tests. To be precise, at most $|M|$ equivalency problems (i.e., per mutant) should be answered for the equivalent mutant detection, whereas at most $|M| \times (|M| + 1)/2$ equivalency problems (i.e., per pair of mutants) should be answered for the universally indistinguishable mutants detection. Practically, this may be a huge burden for testing engineers when they have to solve the equivalency problems manually. However, meaningful results to solve the equivalency problem automatically are being researched [35], [36], [37], and we believe that this area will develop further in the future.

We should note that the notion of *duplicated* mutants proposed in [36] is similar to that of universally indistinguishable mutants. Two mutants are duplicated when they are not equivalent to the original program but equivalent to each other. The key difference between the notion of duplicated mutants and universally indistinguishable mutants comes from the sets of mutants they consider: the extended mutant set $M' = M \cup \{p_o\}$ is considered in universally indistinguishable mutants, whereas only the generated mutants $m \in M$ is considered in duplicated mutants. To be precise, the set of universally indistinguishable mutants is the union of the set of duplicated mutants and the set of equivalent mutants. In other words, by adapting M' instead of M, not only equivalent mutants but also duplicated mutants can be considered together in the notion of universally indistinguishable mutants.

3.7 Effort compared to Traditional Mutation Adequacy

The most critical effort required by the mutation approaches is that needed to relieve the equivalency problems, which have already been discussed in Section 3.6. This section discusses the additional effort required to use the d-criterion in comparison to the effort required to use the k-criterion.

Given a test suite TS and a set of mutants M generated from an original program p_o, the k-criterion evaluates whether there exists a test $t \in TS$ that kills a mutant $m \in M$. Formally, the k-criterion evaluates $\exists t \in TS, d(t, p_o, m) \neq 0$ for each mutant $m \in M$. On the other hand, the d-criterion evaluates whether there exists a test $t \in TS$ that distinguishes a pair of mutants $m_x, m_y \in M'$ where $M' = M \cup \{p_o\}$. Formally, the d-criterion evaluates $\exists t \in TS, d(t, p_o, m_x) \neq d(t, p_o, m_y)$ for each pair of mutants $m_x, m_y \in M'$.

Note that both criteria require the same information: whether t kills m or not (i.e., $d(t, p_o, m)$ is 1 or 0) for all $t \in T$ and $m \in M$. The only difference is the evaluations: the k-criterion requires at most $|M|$ evaluations of $\exists t \in TS, d(t, p_o, m) \neq 0$ per $m \in M$, whereas the d-criterion requires at most $|M| \times (|M| + 1)/2$ evaluations of $\exists t \in TS, d(t, p_o, m_x) \neq d(t, p_o, m_y)$ per pair $m_x, m_y \in M'$.

Since each evaluation of the mutant distinguishment in the d-criterion is a simple comparison of the kill results of a pair of mutants, the time to evaluate the mutant distinguishment is negligible compared to the time to obtain the kill results. For example, in test suite selection, generating a d-criterion adequate test suite from 6,956 tests with 1,924 mutants using a greedy algorithm requires 0.186 seconds, whereas obtaining the kill results for the mutants and tests requires 6,307 seconds [19]. As a result, if the kill results of all mutants for every test are given, the additional effort required to use the d-criterion in comparison to use the k-criterion is not a big deal.

We note that satisfying the d-criterion may be considerably expensive than the k-criterion in practice, because the d-criterion requires the execution of every test on every mutant, while the k-criterion only requires one test that kills each mutant. A number of sound run-time optimization techniques for test execution and test generation based on the k-criterion [38], [39] are not directly applicable for the d-criterion. However, studies on reducing the execution cost of tests on mutants, such as parallelization [40] and split-steam [41], will help make the d-criterion more practical.

4 Experimental Description

4.1 Research Questions

In the experiments, we investigate the following four main research questions:

- RQ1: How does the fault detection effectiveness of each mutation adequacy criterion vary according to score?
- RQ2: How does the test suite size of each mutation adequacy criterion vary according to score?
- RQ3: Which mutation score is more correlated with real fault detection?
- RQ4: Does the test pool affect the fault detection effectiveness of each mutation adequacy criterion?

RQ1 focuses on the fault detection effectiveness of the mutation adequacy criteria in achieving given levels of score. We not only investigate the relationship between the score and the fault detection for each mutation adequacy criterion, but also compare the fault detection effectiveness among the mutation adequacy criteria.

RQ2 deals with the test suite size for achieving the given score levels of the mutation adequacy criteria. Since the d-criterion is stronger than the k-criterion, it is likely that d-criterion will require more tests than the k-criterion. We compare the sizes of the d-suites (i.e., the d-criterion adequate test suites) and the k-suites (i.e., the k-criterion adequate test suites) according to the adequacy score.

RQ3 considers correlations between the mutation adequacy score and real fault detection. Whereas the $mScore$ is widely accepted as a good metric to compare the fault detection effectiveness of arbitrary test suites, it may be the case that the $dScore$ is more correlated to real fault detection effectiveness. We compare the correlation between the $dScore$ and the real fault detection and the correlation between the $mScore$ and the real fault detection.

RQ4 addresses the effect of test pools in the evaluation. As we generate adequate test suites from a large test pool, the type of the test pool may distort the analysis results. For example, the difference in the fault detection effectiveness between the d-criterion and the k-criterion may significantly vary when a large random test pool is utilized, whereas the difference may not significantly vary when a coverage-directed specific test pool is utilized. We perform the same experiments using different test pools and observe whether the analysis results vary with the different test pools.

To answer the above questions, we design the experiment as illustrated in Fig. 4. We use developer-fixed and manually-verified faults of the real applications in the database of `Defects4J` [20] (v1.1.0). For each fault, we use a large number of developer-written tests given by `Defects4J` as the test pool. The test pool is used to execute many mutants generated by the mutation analysis tool `Major` [42] (v1.3.0); the kill information that records the mutants killed by each test of the test pool is constructed. We then generate test suites adequate to a certain adequacy criterion with a specific score level (interval) for RQ1, RQ2, and RQ4. For RQ3, we generate purely random test suites. For RQ4, with the aid of `Randoop` [43] (v3.0.6) and `EvoSuite` [44] (v0.2.0), we additionally generate a large number of automatically-generated tests as a different type of test pool and repeat the main experiments. Meanwhile, thanks to the fact that `Defects4J` provides both faulty and fixed versions of programs for each fault, it is possible to measure whether each of the generated test suites detects the fault or not. In the following subsections, we explain each part of our experiment in more detail.

4.2 Subject Faults

We conduct our experiments on real applications provided by `Defects4J`. There are 357 developer-fixed and manually-verified real faults and corresponding fixes from five Java applications (JFreeChart, Closure compiler, Commons Math, Joda-Time, and Commons Lang). For each fault, the faulty version and the corresponding fixed version of the

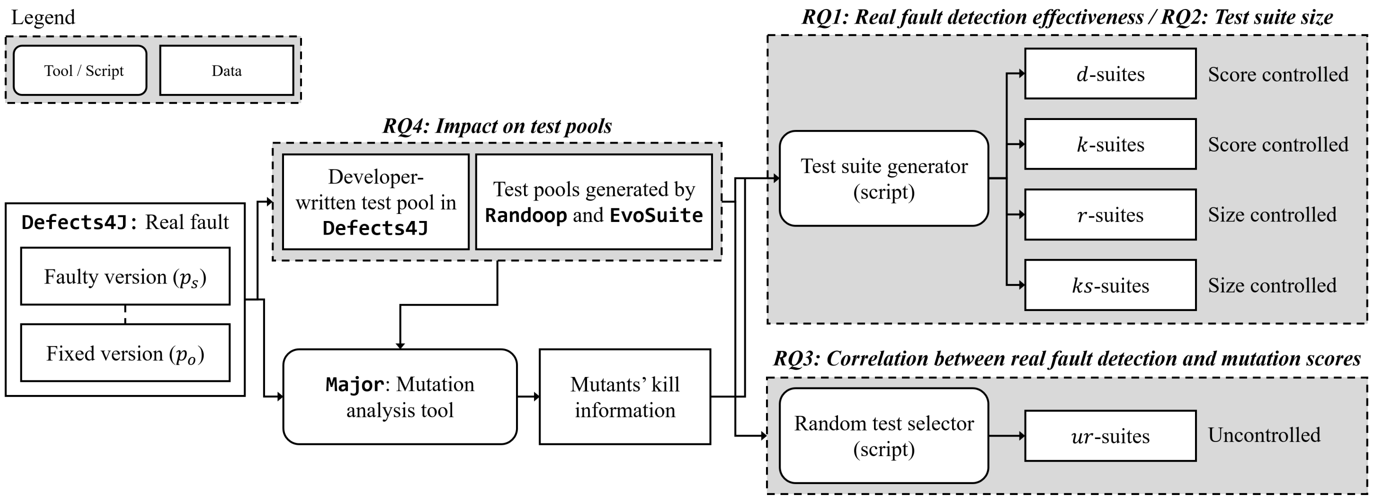

Legend
Tool / Script
Data
RQ4: Impact on test pools
Developer-written test pool in Defects4J
Test pools generated by Randoop and EvoSuite
Defects4J: Real fault
Faulty version (p_s)
Fixed version (p_o)
Major: Mutation analysis tool
Mutants' kill information
RQ1: Real fault detection effectiveness / RQ2: Test suite size
Test suite generator (script)
d-suites
Score controlled
k-suites
Score controlled
r-suites
Size controlled
ks-suites
Size controlled
RQ3: Correlation between real fault detection and mutation scores
Random test selector (script)
ur-suites
Uncontrolled

Fig. 4. Experimental setup: overview

fault are given. The difference between the faulty and fixed version of a fault does not include unrelated changes such as refactorings. Since each fault is given as an independent fault-fix pair of program versions, we treat each fault as a separate subject program. Hereafter, we use the terms “fault” and “faulty program” interchangeably.

We exclude 5 out of 357 faults because they are not able to produce mutation analysis results within a practical time limit (i.e., one hour per test). As a result, 352 faulty programs remain. Table 1 summarizes the 352 faulty programs for each application. Detailed information for each subject fault is available from our webpage at http://se.kaist.ac.kr/donghwan/downloads. From here on, P refers the set of 352 subject faulty programs and $p \in P$ refers to one of the faulty programs.

4.3 Test Pools

In order to compare two test adequacy criteria for a given program, it is necessary to carry out a statistical comparison of the fault detection capabilities of a large number of independent test suites generated using each of the adequacy criteria [45]. To achieve this goal, we need to prepare a test pool for each faulty program and generate many test suites by selecting tests from the test pool with the aid of the adequacy criteria. Fortunately, `Defects4J` provides plenty of relevant developer-written tests that touch the modified classes between the faulty version and the fixed version for each fault. We use the relevant developer-written tests as a large test pool for each fault. In Table 1, the column Dev. tests (sum) shows the sum of the number of developer-written tests (i.e., the size of the test pool) for each fault. For example, there are total 72,005 tests for the 27 Time faults. A total of 553,477 tests are used as the test pools for the 352 studied faults.

On the other hand, for RQ4, which attempts to address the effect of different types of test pools, we need the tests other than the developer-written tests. To obtain automatically-generated tests, we consider two widely studied automatic test generation tools, `Randoop` [43] and `EvoSuite` [44]. `Randoop` is used to generate a large number of random tests, whereas `EvoSuite` is used to generate a set of tests with the objective of maximizing branch coverage. For each fault, we run both of the tools with the 300 seconds time budget, and limit the maximum number of generated tests to 2,000 for `Randoop`. Since automatic test generation tools may generate problematic tests that can cause compile errors, runtime errors, and sporadically fails [20], [46], we automatically removed those tests using the script given in `Defects4J`.

Fig. 5 visualizes the distributions of sizes (i.e., number of tests), code coverage ratios, and mutation scores of all the studied test pools. Statement coverage over the classes modified by the fault fix is computed by `Cobertura`[3]; the mutation score over the all mutants generated from the same classes is computed by `Major` [42]. Note that the set of subject faults for each test pool varies, because not all of the 352 studied faults in P are detected by the EvoSuite pool, nor are they all detected by the Randoop pool. Let $P_E \subseteq P$ and $P_R \subseteq P$ refers the set of subject faults for the EvoSuite pool and the Randoop pool, respectively. Then, $|P_E| = 85$ and $|P_R| = 58$. Thus, Fig. 5 shows not only the characteristics of the test pools but also that of the subject faulty programs. For example, the high mutation scores of the `EvoSuite` test pools are not just because of the characteristics of the test pools but mainly because of the characteristics of the 85 faulty programs.

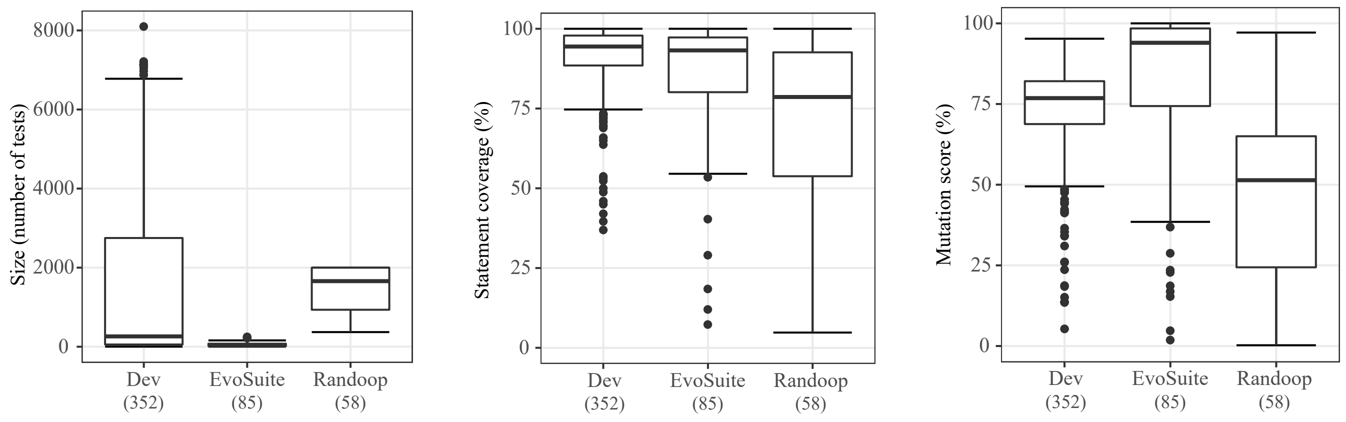
4.4 Mutation Analysis

We use the `Major` [42] mutation analysis tool to generate and executing all mutants for the test pool for each fault. By default, `Major` provides a set of commonly used sufficient mutation operators [17], [47] including the AOR (Arithmetic Operator Replacement), LOR (Logical Operator Replacement), COR (Conditional Operator Replacement), ROR (Relational Operator Replacement), ORU (Operator Replacement Unary), STD (STatement Deletion), and LVR (Literal Value Replacement). We applied all of the mutation operators to generate as many mutants as possible. Since the

3. http://cobertura.github.io/cobertura (v2.0.3)

TABLE 1
Summary for subject faults, developer-written tests, all generated mutants, killed mutants, and distinguished mutants by the test pool

Application	Faults	Dev. tests (sum)	All mutants (sum)	Killed mutants (sum)	Distinguished mutants (sum)
Chart	25	5,806	21,611	8,614	1,462
Closure	133	443,596	109,727	82,676	34,685
Lang	65	11,409	81,524	63,551	5,467
Math	106	20,661	101,978	73,931	14,591
Time	27	72,005	19,996	13,665	3,838
Total	352	553,477	334,836	242,437	60,043



(a) Size (b) Statement coverage (c) Mutation score

Fig. 5. Distribution of sizes, code coverage ratios, and mutation scores of different test pools. Note that the number of subject faults for each test pool varies (i.e., Dev = 352, EvoSuite = 85, Randoop = 58), and the graphs show not only the characteristics of the test pools but also that of the subject faults.

use of sufficient mutation operators may affect the experimental results, we will discuss this issue in Section 5.5.

In Table 1, the columns All mutants, Killed mutants, and Distinguished mutants represent the mutation-related information. The number of mutants killed and distinguished by the test pool are the maximum number of kill-able and distinguishable mutants in the test suite selection, respectively. For example, for the 25 Chart faults, 8,614 mutants and 1,462 mutants among 21,611 mutants are killed and distinguished by the test pool, respectively.

4.5 Test Suite Generation

For each fault, we generate a large number of test suites from the test pool for various experiments. Depending on the use of mutation adequacy criteria, we classify our experiments into two types. The first one is called *controlled* experiments, which means that mutation scores are controlled to generate many mutation adequate test suites with various score levels. The second one is called *uncontrolled* experiments, which means that test suites are randomly generated without any control, and the mutation scores of the random test suites are measured after the generation. The controlled experiments are for RQ1, RQ2, and RQ4, whereas the uncontrolled experiments are for RQ3.

4.5.1 Controlled Test Suite Generation

In the *controlled* experiments, we aim to generate evenly distributed test suites achieving various mutation score levels in a balanced manner. We control the score s in a range of 0.05 to 1.0 in steps of 0.05. In other words, there are 20 score intervals, greater than or equal to s and less than s+0.5, for each $s \in \{0.05, 0.1, \cdots, 0.95, 1.0\}$. From here on, let c-suite(s) refer to the c-criterion adequate test suite whose score is between s and s+0.05, where $c \in \{d, k\}$. For each score interval, we generate 1,000 independent test suites for both the d-criterion and the k-criterion. In other words, we generate 1,000 c-suites(s) for all $c \in \{d, k\}$ and $s \in \{0.05, \cdots, 1.0\}$.

To generate mutation-adequate test suites achieving various score levels, we use the simple greedy algorithm provided by Andrews et al. [21]. This algorithm is designed to simulate the behavior of a human tester who will add a new test that improves the score to achieve a higher score level. For the given values of c and s, the algorithm iteratively and randomly selects a test t from the test pool. The selected t is added to the c-suite(s) only if t improves the score of the c-suite(s). The algorithm halts when the score of the c-suite(s) is between s and s+0.05. If the score of the c-suite(s) goes over the interval, then the algorithm repeats from the beginning.

In addition to d-suites(s) and k-suites(s), to investigate the influence of test suite size, we also consider size-controlled test suites whose size are the same as the size of d-suite(s). We first consider an r-suite(s), a random test suite with the same size as a d-suite(s), to investigate whether the effectiveness of the d-suite(s) is simply due to its size. Second, we attempt to control the size of a k-suite(s) to investigate whether the effectiveness difference between the d-suite(s) and the k-suite(s) is simply due to their size difference. One easy way to control the size of a k-suite(s) is to add random tests so that the test suite have the same size as a d-suite(s). However, adding random tests to a k-suite(s) to increase its size may introduce a bias and underestimate the effectiveness of the larger test suite, since random test selection performs poorly in general. As a more conservative way to control the size of a k-suite(s), we apply stacking [48]: given the size of a d-suite(s), we generate and stack k-suites(s) until the number of tests of their union reaches the size of the d-suite(s). Note that the score of an r-suite(s) and a ks-suite(s) is not necessarily between s and s+0.05.

In summary, for a given score level s, we investigate the following score-controlled test suites:

- d-suite(s): the d-criterion adequate test suite whose score is between s and s+0.05.
- k-suite(s): the k-criterion adequate test suite whose score is between s and s+0.05.
- ks-suite(s): stacked k-suite(s) whose size is the same as the d-suite(s).
- r-suite(s): random test suite whose size is the same as the d-suite(s).

We attempt to generate 1,000 c-suites(s) for each criterion $c \in \{d, k, ks, r\}$, score level $s \in \{0.05, \cdots, 1.0\}$, and faulty program $p \in P$, which leads to a total of 1,000 $\times$ 4 $\times$ 20 $\times$ 352 = 28,160,000 controlled test suites. However, for certain values of c, s and p, it is almost impossible to generate a c-suite(s) because some mutants are always killed together so that the mutation score is not discretized into as many as 20 intervals. We skip the generation of a c-suite(s) for p if the generation process fails over 100 times in a row. As a result, we generate 24,488,000 test suites for the controlled experiments.

4.5.2 Uncontrolled Test Suite Generation

In the *uncontrolled* experiments, test suite generation is relatively simple: we generate 1,000 uncontrolled random test suites, or simply ur-suites, for each faulty program. Since we have the 352 faulty programs, we generate 1,000 $\times$ 352 = 352,000 ur-suites for the uncontrolled experiments in total.

For each ur-suite, to investigate the correlation between the mutation scores and the fault detection, we measure the $mScore$, the $dScore$, and the fault detection (i.e., whether the test suite detects the fault or not). Section 4.6.3 provides details on how to investigate the correlations.

4.6 Variables and Measures

For each RQ, there are different independent and dependent variables and measures.

4.6.1 RQ1: Fault Detection Effectiveness

We define the fault detection effectiveness of an adequacy criterion as the probability that a test suite selected to be adequate to the criterion will detect a fault [49]. We measure the fault detection effectiveness of a criterion c achieving a score level s for a fault p as the proportion of the fault-detecting test suites among 1,000 c-suites(s) for p, or simply $rate(c, s, p)$, in RQ1. For example, let x be the number of fault-detecting test suites among 1,000 c-suites(s) for a faulty program p. Then $rate(c, s, p) = x/1000$ implies the fault detection effectiveness of c that achieves s for p. To measure the effectiveness of c with s for p, we observe whether a c-suite(s) detects the fault or not for all c and s for each p. In other words, c and s are the independent variables, whereas the fault detection (i.e., either $true$ or $false$) of each c-suite(s) for p is the dependent variable.

To compare the fault detection effectiveness of the d-criterion and the k-criterion using statistical methods, we follow the guideline provided by Arcuri and Briand [50]. Since our dependent variable (i.e., real fault detection) is dichotomous, we choose the non-parametric proportion test and *Risk Ratio* (RR)[4] for the statistical methods. For each s and p, we first measure the p-value of the non-parametric proportion test to investigate whether the fault detection effectiveness of the d-suites(s) is statistically greater (or less) than that of the k-suites(s). Since the alternative hypotheses are directional (i.e., greater than or less than), we perform one side proportion tests for each alternative hypothesis. Further, to describe the effect size of the difference, we measure RR which represents how many times the number of fault-detecting d-suites(s) is greater than the number of fault-detecting k-suites(s). To prevent the denominator from becoming zero, the RR value is calculated with the following formula:

$$RR(s, p) = \frac{detects(d, s, p) + 0.5}{detects(k, s, p) + 0.5}$$

where $detects(c, s, p)$ refers to the number of fault-detecting test suites among the 1,000 c-suites(s) for a fault p.

4.6.2 RQ2: Test Suite Size

In RQ2, we define the size of c-suites(s) as the average number of tests in each of 1,000 c-suites(s) for each p. We observe the number of tests in each c-suite(s) for all c, s, and p. In other words, for each p, c and s are the independent variables, whereas the number of tests in each c-suite(s) for p is the dependent variable. Since the sizes of the ks-suites(s) and r-suites(s) are by definition equal to the size of corresponding d-suites(s), we only consider d-suites(s) and k-suites(s) in RQ2.

We measure the relative size of d-suites(s) over k-suites(s) for all s and p to investigate how many times the size of the d-suites(s) is greater than the size of the k-suites(s).

4.6.3 RQ3: Correlation with Real Fault Detection

In RQ3, we observe the $mScore$, the $dScore$, and the fault detection of a ur-suite for each p. In other words, for each p, there is no independent variable, whereas the $mScore$, the $dScore$, and the fault detection are the dependent variables.

We measure two correlations among the three dependent variables: (1) the correlation between the $mScore$ and the fault detection and (2) the correlation between the $dScore$ and the fault detection. Since the fault detection is dichotomous, we use the rank-biserial correlation coefficient to measure the correlations.

4. We adapt risk ratio instead of odds ratio because it is more intuitive to interpret in our experimental setting.

4.6.4 RQ4: Impact of Test Pools

As explained in Section 4.3, we additionally investigate the impact of test pools considering two different types of automatically-generated tests: random tests generated by `Randoop` and branch coverage maximizing tests generated by `EvoSuite`. For each test pool, we do the same experiments with RQ1. We observe whether a c-suite(s) detects the real fault or not for all c and s for each p. In other words, c and s are the independent variables, whereas the real fault detection (i.e., either $true$ or $false$) of each c-suite(s) for p is the dependent variable. We then compare the results from each test pool.

5 Analysis Results

5.1 RQ1: Fault detection effectiveness

5.1.1 RQ1-1: Relationship between Fault Detection Effectiveness and Mutation Scores

One of our main objectives is to investigate the fault detection effectiveness of the mutation adequacy criteria (i.e., the k-criterion and the d-criterion) for various score levels. Fig. 6 summarizes the relationships between the fault detection effectiveness and the score level for each adequacy criterion. Each point represents the fault detection effectiveness $rate(c, s, p)$ for a criterion $c \in \{d, k, ks, r\}$, a score interval $s \in \{0.05, \cdots, 1.0\}$, and a faulty program $p \in P$. Each box-plot represents the distribution of $rate(c, s, p)$ over p for given values of c and s.

For the d-criterion, $rate(d, s, p)$ increases with increasing s up to 1 for all p. This means that achieving a higher $dScore$ tends to lead to the detection of more real faults for all score levels. This tendency is very important when we generate test suites using the d-criterion. Since the fault detection effectiveness of the d-criterion monotonically increases with increasing score levels, achieving a higher $dScore$ is preferable in terms of fault detection. Of course, testing efforts also increase with increasing score levels. We will discuss the testing efforts in Section 5.2.

For the k-criterion, $rate(k, s, p)$ also increases with increasing s up to 1 for all p similar to the d-criterion. However, it is worthwhile to consider the correlation between the score level and the fault detection rate. In comparison to the $kScore$, the $dScore$ achieves more linear correlation with the fault detection rate. Fig. 6 suggests that, particularly for the inadequate test suites, the d-criterion is better correlated with the fault detection effectiveness than the k-criterion. The fact that the d-criterion shows more linear correlation is encouraging, as it allows more intuitive interpretation of the $dScore$ of inadequate test suites with respect to the fault detection effectiveness.

While the fault detection effectiveness of both the d- and k-criteria increase along with the corresponding mutation scores, the $dScore$ shows more linear correlation with the fault detection effectiveness.

5.1.2 RQ1-2: Comparison between the d-criterion and the k-criterion

To compare the fault detection effectiveness of the mutation adequacy criteria, we summarize the average fault detection effectiveness $rate(c, s, p)$ of each criterion c for each score interval s in Fig. 7. For example, as an average for the 352 faulty programs p, $rate(d, 1.0, p)$ is 0.943, $rate(k, 1.0, p)$ is 0.877, $rate(ks, 1.0, p)$ is 0.902, and $rate(r, 1.0, p)$ is 0.355. Overall, Fig. 7 clearly shows that the d-criterion outperforms the other criteria for all score levels.

Fig. 7 shows again that the d-criterion is better correlated with the fault detection rate in comparison to the k-criterion. Interestingly, a similar phenomenon appears in the ks-criterion. Recall that a ks-suite has the same size as a d-suite. This implies that the improvement of the fault detection effectiveness of the d-criterion in comparison to the k-criterion is largely related to test suite size. However, on average for all s and p, $rate(d, s, p)$ is 5.73 percentage points higher than $rate(ks, s, p)$. This signifies that the improvement is not just because of the size effect. Also, it is encouraging that the d-criterion is parametric-free in contrast to the ks-criterion; the d-criterion does not require a predefined test suite size.

For deeper investigations into the d-criterion and the k-criterion, we narrow our focus from the average of all faults to each fault. For each fault, we perform a non-parametric proportion test to analyze whether the fault detection effectiveness of the d-criterion is statistically greater (or less) than that of the k-criterion. Specifically, we perform two independent one side proportion tests for each value of s and p: one to determine whether $rate(d, s, p)$ is statistically greater than $rate(k, s, p)$, the other to determine whether $rate(d, s, p)$ is statistically less than $rate(k, s, p)$. In other words, the null hypothesis is $H_0 : rate(d, s, p) = rate(k, s, p)$ for both proportion tests, while the alternative hypotheses for each proportion test are $H_{1,GT} : rate(d, s, p) > rate(k, s, p)$ and $H_{1,LT} : rate(d, s, p) < rate(k, s, p)$, respectively. The Bonferroni correction [50] is applied to reduce the probability of Type 1 errors, so that both proportion tests use $\alpha = 0.05/1000 = 0.00005$ where $N = 1000$. Based on the two proportion tests, all the 352 faults are classified into three types: *GT*, *LT*, and *EQ*. If $rate(d, s, p)$ is statistically greater (or less) than $rate(k, s, p)$, then the fault is classified as *GT* (or *LT*) type. If the fault is of neither *GT* nor *LT* type, the fault is classified as *EQ* type.

Fig. 8 summarizes the distribution of fault types for each score interval. For example, for the adequate test suites (i.e., $s = 1.0$), 15.1% of the subject faults show a statistical increase in their fault detection effectiveness when using the d-criterion in comparison to the k-criterion, whereas the remaining 84.9% show no statistical difference. On average for all score levels, 74.8% are *GT* type, 21.6% are *EQ* type, and only 3.6% are *LT* type. Note that there are no *LT* type faults when $s = 1.0$. This means that the d-criterion is *always better than or equal to* the k-criterion in terms of fault detection effectiveness when each of the adequacy criteria is fully satisfied. This is intuitive considering the subsumption relationship between the d-criterion and the k-criterion, as explained in Section 3.4.

Then why does the fault detection effectiveness of *EQ* type faults not increase with the d-criterion in comparison to the k-criterion, even when $s = 1.0$? There are 299 *EQ* type faults (i.e., 84.9% of all the studied faults) when $s = 1.0$. Among the 299 faults, 286 faults are "maximized" by

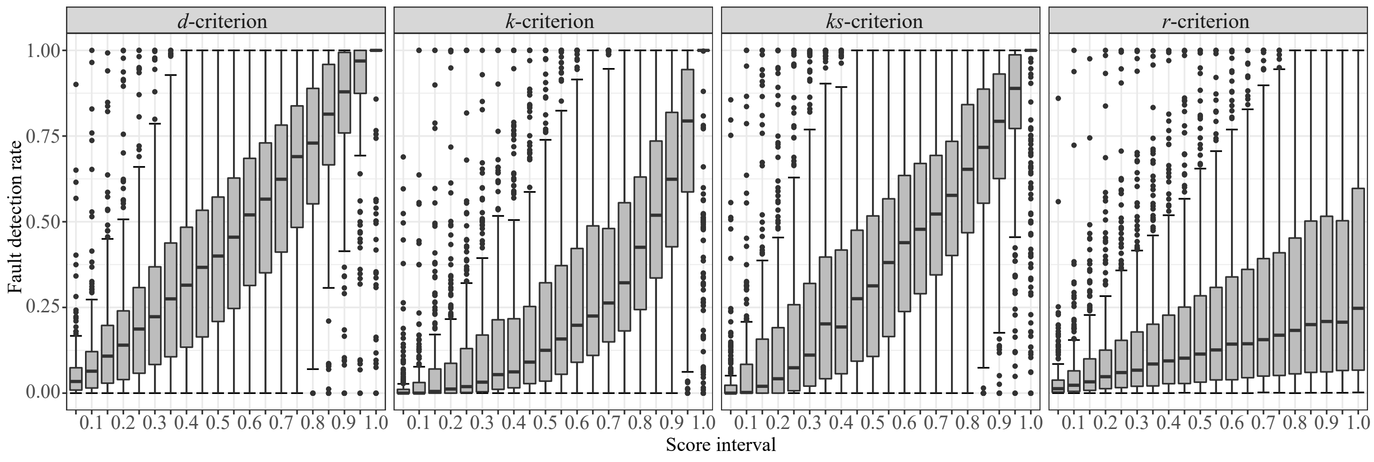


Fig. 6. Relationships between fault detection and score for each criterion

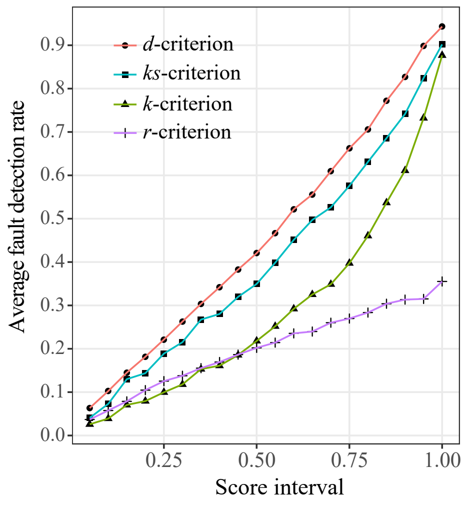


Fig. 7. Comparison of fault detection effectiveness among the criteria

the k-criterion (i.e., $rate(k, 1.0, p) = 1$ or the $dScore$ of k-suite(1.0) is 1), 10 faults are more likely to be detected by the d-criterion than by the k-criterion but not statistically significant; the remaining 3 faults are detected by neither the d-criterion nor the k-criterion, because there are no mutants killed by the fault-detecting tests for the 3 faults.

In addition to the statistical significance of the difference of the fault detection effectiveness between the d-criterion and the k-criterion, we analyze the effect size of the difference. We measure the Risk Ratio (RR) values that represent the ratio of $rate(d, s, p)$ over $rate(k, s, p)$ for each s and p. Fig. 9 summarizes the results. Each point represents the logarithm of RR for a fault p in a given score interval s. Since the maximum RR value reaches 200, we use a logarithmic scale. Each box-plot represents the distribution of the logarithm of RR for all $p \in P$ for a given s. Table 2 summarizes the minimum, average, and maximum RR value for each score interval. For example, for all score intervals, the average RR is 8.26, which means that the fault detection effectiveness of the d-criterion is 8.26 times higher than that of the k-criterion.

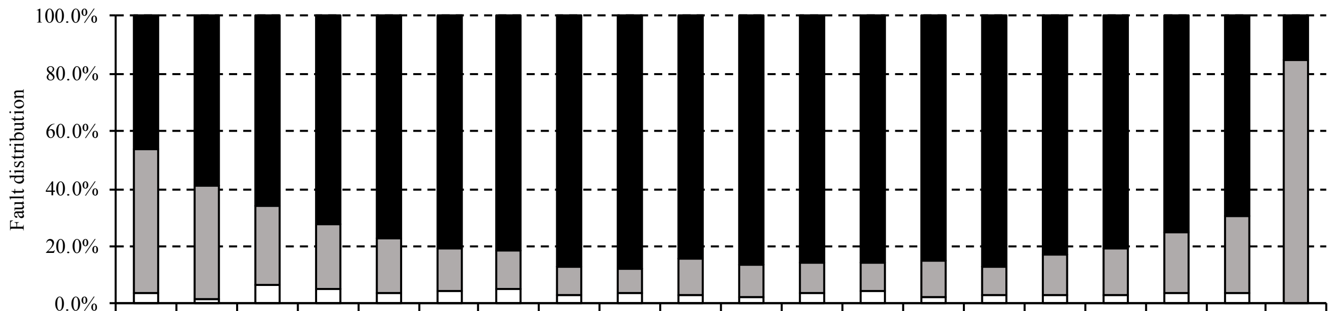
The maximum RR is very large especially for the inadequate test suites (i.e., $s \leq 0.9$). These large RR values are due to cases in which the k-criterion almost fails to detect a fault, whereas the d-criterion detects the fault very well. On average, the RR values for the inadequate test suites are particularly large, which means that the d-criterion provides considerable improvements for the inadequate test suites in comparison to the k-criterion. On the other hand, for the adequate test suites (i.e., $s = 1.0$), the minimum RR is 1.00, which means that the d-criterion is at least as effective as the k-criterion. The average RR value for $s = 1.0$ is 1.33, which means that the d-criterion is 1.33 times more effective than the k-criterion for the adequate test suites. The maximum RR value for $s = 1.0$ is 14.29, which means that the d-criterion is at most 14.29 times more effective than the k-criterion for the adequate test suites.

For more information, we summarize the comparison results between the d-criterion and the ks-criterion in Table 3. Similar to Table 2, it provides the minimum, average, and maximum RR value for each score interval. Overall, the RR values in Table 3 (i.e., compared to the ks-criterion) are similar to the values in Table 2 (i.e., compared to the k-criterion), while the values are slightly smaller than when compared to the k-criterion. For example, for all score intervals, the average RR value of the d-criterion over the ks-criterion is 6.89, whereas that of the d-criterion over the k-criterion is 8.26. This is intuitive in that the size of the ks-suites is increased to be equal to the d-suites in comparison to the k-suites. Further, the results shows that the fault detection effectiveness of the diversity-aware mutation adequacy criterion is superior to the traditional mutation adequacy criterion, even when the test suite size is considered.

On average for all score intervals, the d-criterion increases the fault detection effectiveness in a statistically significant manner for 74.8% of all subject faults in comparison to the k-criterion, and the effectiveness increases 8.26 times.

5.2 RQ2: Test Suite Size

As explained in Section 4.6, we measure the relative sizes of the d-suites(s) over the k-suites(s) for each s and p. Fig. 10



	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
■ GT	46.6%	59.1%	65.8%	72.1%	77.4%	81.0%	81.5%	86.8%	88.0%	84.0%	86.4%	85.5%	86.0%	85.1%	87.0%	83.1%	80.8%	74.8%	69.5%	15.1%
▨ EQ	49.4%	39.0%	27.6%	22.6%	18.5%	14.6%	13.1%	10.2%	8.0%	13.2%	11.1%	10.4%	9.3%	12.3%	9.9%	13.8%	15.8%	21.6%	26.8%	84.9%
□ LT	4.0%	1.9%	6.5%	5.2%	4.0%	4.4%	5.4%	3.0%	4.0%	2.8%	2.5%	4.1%	4.7%	2.5%	3.1%	3.1%	3.4%	3.6%	3.7%	0.0%

Fig. 8. Types of faults depending on the difference of fault detection effectiveness between the d-criterion and the k-criterion

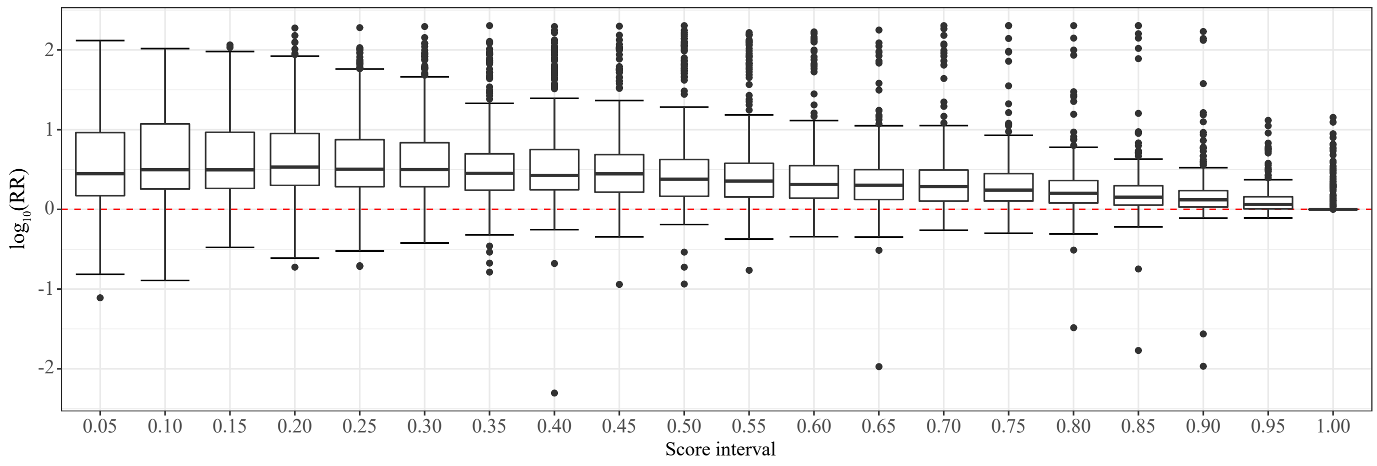


Fig. 9. Risk Ratio (RR) effect sizes for fault detection effectiveness of the d-criterion over the k-criterion for each score interval

summarizes the size ratios of each score interval. Each boxplot represents the relative sizes for all subject faults for a given score interval. Table 4 summarizes the minimum, average, and maximum size ratios for each score interval. For example, on average for all score intervals, the size ratio is 3.07, which means that the d-suites are 3.07 times larger than the k-suites on average.

For all score intervals, the d-suite requires 3.07 times more tests than the k-suite on average. The large relative size means that the number of required tests to achieve the same score level may largely vary between the d-criterion and the k-criterion. This is because the mutants generated from the faulty programs are easy to kill (i.e., distinguish from the original program) but hard to distinguish (i.e., distinguish among mutants). The average size ratio is 1.56 for the adequate test suites (i.e., $s = 1$), which means that the d-criterion requires 1.56 times more tests than the k-criterion for the adequate test suites.

On average for all score intervals, the d-criterion requires 3.07 times more tests than the k-criterion.

5.3 RQ3: Correlation with Fault Detection

Section 5.1 shows that the d-criterion is more effective at detecting faults than is the k-criterion for all score intervals; however, the question remains which score between the distinguishing mutation score (i.e., $dScore$) and the traditional mutation score (i.e., $mScore$) is more correlated with real fault detection.

Fig. 11 summarizes the rank-biserial correlation coefficients between fault detection and the mutation scores (i.e., $dScore$ and $mScore$, respectively). Each point represents the rank-biserial correlation coefficient between fault detection and each of the scores for a fault. On average, the coefficients for the $dScore$ and the $mScore$ are 0.780 and 0.718, respectively. This means that fault detection is slightly more correlated with the distinguishing mutation score than with the traditional mutation score.

To measure the statistical significance, we apply the Mann-Whitney U-test to the coefficients. The p-value is 2.00e-12, which means that the correlation coefficient between the d-criterion and fault detection is statistically different from the correlation between the k-criterion and fault detection. We also measure the Vargha and Delaney's $\hat{A}_{12}$ statistic following the guidelines [50]. The $\hat{A}_{12}$ is 0.652, which means that the correlation between fault detection

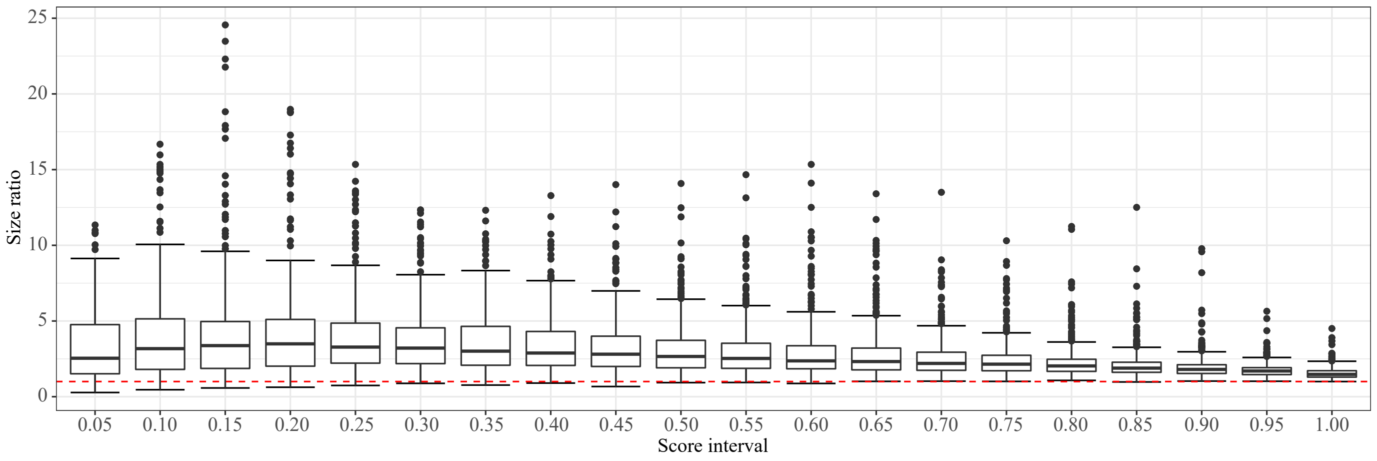


Fig. 10. Relative size of d-suites over k-suites for each score interval

TABLE 2
Effect size for each score interval (compared to the k-criterion)

Score	Effect size (RR)		
interval	minimum	average	maximum
0.05	0.08	8.57	131.20
0.10	0.13	9.82	104.00
0.15	0.33	10.33	115.60
0.20	0.19	12.46	188.40
0.25	0.19	12.03	190.40
0.30	0.38	12.54	196.60
0.35	0.16	11.47	201.00
0.40	0.00	13.64	196.60
0.45	0.11	10.63	198.80
0.50	0.12	12.77	201.00
0.55	0.17	10.93	165.20
0.60	0.45	9.13	168.60
0.65	0.01	5.44	177.40
0.70	0.55	6.31	201.00
0.75	0.50	4.87	201.00
0.80	0.03	3.86	201.00
0.85	0.02	4.51	201.00
0.90	0.01	3.16	169.80
0.95	0.78	1.50	13.11
1.00	1.00	1.33	14.29
All	0.00	8.26	201.00

TABLE 3
Effect size for each score interval (compared to the ks-criterion)

Score	Effect size (RR)		
interval	minimum	average	maximum
0.05	0.08	8.51	131.20
0.10	0.12	9.07	104.00
0.15	0.21	9.20	115.60
0.20	0.19	11.11	188.40
0.25	0.11	10.15	190.40
0.30	0.24	10.82	196.60
0.35	0.16	9.48	201.00
0.40	0.00	11.59	196.60
0.45	0.11	8.71	198.80
0.50	0.12	10.95	201.00
0.55	0.17	9.59	165.20
0.60	0.45	7.62	168.60
0.65	0.01	3.81	177.40
0.70	0.43	4.88	201.00
0.75	0.38	3.78	201.00
0.80	0.01	2.83	201.00
0.85	0.02	3.72	201.00
0.90	0.01	2.63	169.80
0.95	0.59	1.15	5.40
1.00	0.58	1.11	5.49
All	0.00	6.89	201.00

and the $dScore$ is higher than the correlation between fault detection and the $mScore$ with the probability of 0.652.

The $dScore$ is statistically more correlated to real fault detection than the $mScore$ (Mann-Whitney U-test, p-value=2.00e-12). The $\hat{A}_{12}$ is 0.652.

5.4 RQ4: Impact on Test Pool

Based on the two different test pools (i.e., the EvoSuite pool and the Randoop pool), we repeat the experiments to investigate the difference of the fault detection effectiveness between the d-criterion and the k-criterion. Table 5 summarizes the types of faults and the effect size (RR) as analyzed in Section 5.1.2.

In Table 5, the overall patterns are similar between the EvoSuite pool and the Randoop pool. For both test pools, the percentage of *GT* type faults are much higher than the percentage of *LT* type faults. These results shows that, in comparison to satisfying the k-criterion, satisfying the d-criterion is beneficial to increase the fault detection effectiveness regardless of the test pool types. For both test pools, the d-criterion adequate test suites (i.e., $s = 1.0$) has no *LT* type faults.

We have not found a significant correlation between the characteristics of the test pool (e.g., test pool size, code coverage ratios, and number of fault-detecting tests) and the

TABLE 4
Size ratio for each score interval

Score	Size ratio		
interval	minimum	average	maximum
0.05	0.28	3.28	11.34
0.10	0.46	4.06	16.67
0.15	0.58	4.30	24.55
0.20	0.63	4.20	18.99
0.25	0.74	3.99	15.34
0.30	0.88	3.82	12.34
0.35	0.77	3.63	12.31
0.40	0.90	3.48	13.29
0.45	0.67	3.37	14.00
0.50	0.93	3.19	14.09
0.55	0.94	3.10	14.67
0.60	0.87	2.99	15.34
0.65	1.01	2.87	13.41
0.70	1.03	2.68	13.50
0.75	1.01	2.53	10.30
0.80	1.08	2.33	11.24
0.85	0.97	2.14	12.51
0.90	1.03	1.99	9.78
0.95	1.02	1.78	5.64
1.00	1.00	1.56	4.50
All	0.28	3.07	24.55

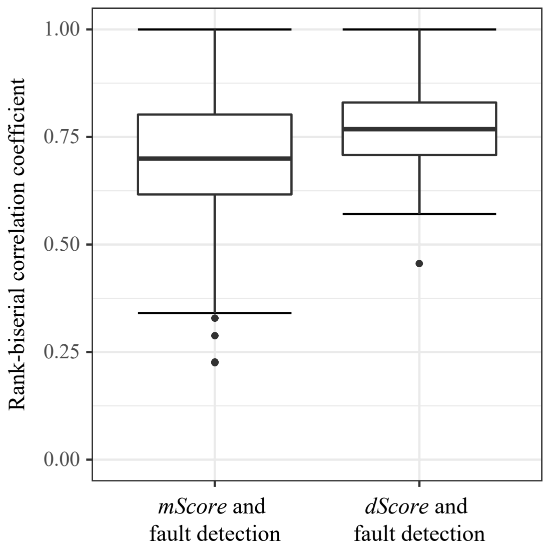


Fig. 11. Rank-biserial correlation coefficients for each mutation adequacy criteria

effectiveness improvement of the d-criterion in comparison to the k-criterion.

Regardless of test pool types, in comparison to satisfying the k-criterion, satisfying the d-criterion is beneficial to increase the fault detection effectiveness.

5.5 Threats to Validity

Since any experiment is subject to threats to validity, careful identification of the threats is important in empirical evaluations. In this subsection, we consider three types of threats to the validity of our experiments: *external*, *internal*, and *construct* threat.

One threat to external validity comes from the representativeness of the subject applications (i.e., five real world applications written in Java) and their faults (i.e., 352 developer-fixed and manually-verified real faults). Whereas this threat can only be properly addressed by further empirical studies, we have tried to use a non-trivial number of real faults collected in the `defect4j` repository.

Our results are also dependent on the test pools, including developer-written tests given by `Defects4J` and test pools generated by `Randoop` and `EvoSuite`. To reduce this threat, we partially confirmed that the results of the real fault detection did not significantly vary depending on the types of test pools in RQ4. Further, the formal subsumption relationship between the d-criterion and the k-criterion guarantees that satisfying the d-criterion never decreases the fault detection effectiveness in comparison to satisfying the k-criterion in test suite selection.

The mutation operators we have used are another threat to external validity. There are many other mutation operators that are applicable in Java. For example, if more various mutants are generated by a richer set of mutation operators, then the gap between the d-criterion and the k-criterion will be bigger in terms of both real fault detection and test suite size. However, a set of sufficient mutation operators is widely accepted as the baseline. To give a fair comparison between the d-criterion and the k-criterion, we have tried to use the sufficient mutation operators by default.

In terms of threats to internal validity, our experiments rely on several tools such as `Major`, `EvoSuite`, and `Randoop`. To confirm the correctness of the tools, we have tried our best to follow up on and resolve up-to-date issues from the tool web pages. On the other hand, we developed several scripts specifically for our experiments (e.g., test suite generation), which could increase threats to internal validity. To reduce such threats, we validated the scripts by testing them on simple programs and mutants several times and confirmed their correctness.

The fault detection effectiveness of a test adequacy criterion may vary depending on the testing scenarios. For example, a test adequacy criterion may be used not for test suite selection but test data generation (e.g., Counter-exampled based test generation [51], [52] and search-based test generation [23], [39], [53]). Further information about the testing scenarios and their impact on the fault detection capability is well-described in Zhu et al. [33]. We have a plan to develop a test generation method to satisfy the d-criterion.

For the mutation scores, $mScore$ and $dScore$, equivalent mutants and universally indistinguishable mutants must be removed. As controlled experiments, similar to the work of Andrews et al. [21], the mutants that are not killed and distinguished by the test pools are deemed to be equivalent and universally indistinguishable, respectively. In other words, we use the test pools to approximate the equivalent mutants and the universally indistinguishable mutants. This may limit the number of subject mutants and affect our results. However, the set of non-equivalent mutants and the set of distinguishable mutants are identical for any test

TABLE 5
Fault types and effect sizes for the fault detection effectiveness for each test pool

Score	EvoSuite pool				Randoop pool			
interval	*GT*	*LT*	*EQ*	RR (avg)	*GT*	*LT*	*EQ*	RR (avg)
0.05	60.7%	8.2%	31.1%	10.64	64.4%	2.2%	33.3%	21.79
0.10	80.0%	6.2%	13.8%	15.69	78.0%	4.0%	18.0%	21.24
0.15	86.8%	1.5%	11.8%	20.03	72.0%	2.0%	26.0%	18.27
0.20	87.5%	8.3%	4.2%	13.81	75.5%	1.9%	22.6%	14.93
0.25	86.8%	5.3%	7.9%	16.12	76.9%	3.8%	19.2%	8.08
0.30	91.8%	0.0%	8.2%	13.65	82.0%	0.0%	18.0%	11.09
0.35	89.7%	4.4%	5.9%	9.52	84.0%	2.0%	14.0%	7.51
0.40	85.3%	4.0%	10.7%	8.40	80.8%	0.0%	19.2%	11.62
0.45	89.2%	4.1%	6.8%	8.50	82.7%	0.0%	17.3%	7.84
0.50	87.0%	3.9%	9.1%	7.58	83.3%	1.9%	14.8%	3.50
0.55	89.5%	1.3%	9.2%	9.83	77.4%	1.9%	20.8%	3.72
0.60	86.5%	5.4%	8.1%	6.12	71.7%	1.9%	26.4%	3.24
0.65	87.2%	1.3%	11.5%	6.76	73.1%	0.0%	26.9%	3.04
0.70	90.5%	0.0%	9.5%	1.93	66.7%	0.0%	33.3%	2.39
0.75	90.8%	2.6%	6.6%	4.42	66.7%	0.0%	33.3%	2.01
0.80	84.4%	3.9%	11.7%	1.71	61.5%	0.0%	38.5%	1.66
0.85	78.9%	1.3%	19.7%	4.35	52.8%	0.0%	47.2%	2.46
0.90	80.3%	0.0%	19.7%	4.08	51.9%	0.0%	48.1%	1.32
0.95	68.1%	0.0%	31.9%	1.35	39.6%	2.1%	58.3%	1.16
1.00	20.0%	0.0%	80.0%	1.22	22.4%	0.0%	77.6%	1.19
Average	81.0%	3.1%	15.9%	8.29	68.2%	1.2%	30.6%	7.40

suites used in a comparison for each fault. Further studies are needed as it may depend on the selection of the studied program.

To allow reproducibility of the results presented in this paper, all the test pools generated by `EvoSuite` and `Randoop`, the mutants generated by `Major`, and the implementation of the distinguishing mutation adequacy criterion are freely available from our webpage at http://se.kaist.ac.kr/donghwan/downloads.

6 Related Work

To the best our knowledge, this is the first study to attempt to improve the real fault detection capability of mutation adequacy criteria by considering the diversity of mutants. However, all mutation testing studies are, at some level, related to this paper because the mutation adequacy criterion is the essence of mutation testing. In this section, we first focus on mutation studies related to the mutant distinguishment, which is the foundation of the d-criterion.

As mentioned before, Ammann et al. [16] defined the notion of mutant distinguishment in terms of mutant set minimization, whereas their main focus was redundant mutants (i.e., those that do not contribute to the quality of a test suite), defined by the *dynamic subsumption* relation between mutants. If a mutant m_x is killed by at least one test in a set of tests TS, and another mutant m_y is always killed whenever m_x is killed, then m_x dynamically subsumes m_y with respect to TS. They asserted that a mutant set M_{min} is minimal with respect to TS when there are no dynamically subsumed mutants in M_{min}. In the working example shown in Fig. 1, m_1 dynamically subsumes m_2, m_3, and m_4 with respect to $TS = \{t_1, t_2, t_3\}$, so that $M_{min} = \{m_1\}$. Note that the dynamically subsumed mutants are "redundant" only in terms of the k-criterion, which does not cater for the diversity of mutants. In the d-criterion, all mutants are "valuable" in terms of the diversity of mutants, except the universally indistinguishable mutants described in Section 3.6. In the working example, all four mutants are distinguished by TS; they are not redundant in terms of the d-criterion. This observation signifies that redundant mutants and inflated mutation scores are mainly due to the k-criterion, and the d-criterion clearly resolves these problems. In general, many mutants can be removed by mutant subsumptions to yield a smaller mutant set in terms of the k-criterion as shown, for example, by Just and Schweiggert [17], whereas more tests can be added to yield a stronger test suite in terms of the d-criterion as shown in this paper.

Prior to the study of Ammann et al. [16], Kinits et al. [54] defined that two mutants are *disjoint* if the test sets that kill them are likely to be disjoint. Whereas the notion of distinguished mutants and disjoint mutants are similar, the definition of disjoint mutants is not as precise as the definition of distinguished mutants. Meanwhile, Kurtz et al. [32] extended the study of Ammann et al. [16] by defining three different types of subsumption graphs (i.e., true, dynamic, and static subsumption graphs), as discussed in Section 3.3. All of these studies are based on mutant subsumption, which attempts to remove redundant mutants in terms of the traditional kill-only mutation adequacy. On the other hand, we define a novel diversity-aware mutation adequacy to improve the fault detection effectiveness of mutation testing.

Just et al. [55] attempted to avoid executing a test t for as many t-equivalent (i.e., test-equivalent) mutants as possible. For a test t, a mutant m is t-equivalent if t cannot detect the difference between m and its original program p_o. Using our formal framework, the t-equivalent mutant is represented as $d(t, p_o, m) = 0$. They also partitioned mutants into equivalence classes using the intermediate results of the mutants for each t. It is equivalent to make every mutant in such an equivalence class indistinguishable by t. Similarly, Ma and Kim [14] proposed an approach for executing fewer mutants by clustering mutants. They defined that a mutant m_x is c-overlapped (i.e., conditionally overlapped) to another mutant m_y for a test t if m_x and m_y return the same output for t. Using our formal framework, the c-overlapped mutants m_x and m_y for t are represented as $d(t, m_x, m_y) \neq 1$, which is the dual of (3) discussed in Section 3.2. While our approach utilizes the diversity of mutants as opposed to reducing the redundancy of mutants, the equivalency problem is important in the distinguishing mutation adequacy criterion, as discussed in Section 3.6. In this regard, both studies may help to efficiently find distinguishable mutants.

There are several studies that have attempted to improve the effectiveness of interest by increasing its diversity. Chen at el. [1] proposed Adaptive Random Testing (ART) to maximize the diversity of tests by selecting the test input that is the farthest away from the tests already executed so far. Recently, Patric and Jia explored the trade-off between diversification (i.e., select a wide range of tests to increase the change of finding new faults) and intensification (i.e., select tests similar to those previously shown to be successful) in ART [56]. Their results showed that the intensification is the most effective strategy for numerical programs, whereas the diversification seems to be more effective for programs with composite inputs.

The notion of diversity is also explored in fault localization (i.e., identifying possible locations of faults in the program under test) by Baudry et al. [57]. They defined the concept of a Dynamic Basic Block (DBB), which is the set of statements that is covered by the same set of tests. Larger DBB implies lower accuracy of fault localization since all statements in the DBB are as suspicious as the faulty statement. Their experiments showed that optimizing a test suite to distinguish statements in a DBB can improve the fault localization accuracy. This is similar to our work in terms of improving the performance of interest by considering its diversity.

7 Conclusion

This paper introduces a novel diversity-aware mutation adequacy criterion called the *distinguishing mutation adequacy criterion* and a corresponding mutation score called the *distinguishing mutation score* based on the formal definition of the *mutant distinguishment*. The new adequacy aims to use the diversity of mutants to encourage the diversity of adequate tests. Based on the formal definitions of the new adequacy, it is proved that the distinguishing mutation adequacy criterion subsumes the traditional kill-only mutation adequacy criterion. This subsume relation provides theoretical evidence that the distinguishing mutation adequacy criterion is more effective at detecting faults than is the traditional mutation adequacy criterion.

We also provide an empirical evaluation of the mutation adequacy criteria in terms of their fault detection effectiveness, test suite sizes, and various score levels. We use 352 real faults to study real world applications. The results show that, on average for all score levels, the distinguishing mutation adequacy criterion statistically increases the fault detection effectiveness for 74.8% of all faults in comparison to the traditional mutation adequacy criterion; the distinguishing adequacy is 8.26 times more effective than the traditional adequacy. On the other hand, the distinguishing mutation adequacy criterion requires 3.07 times more tests than does the traditional mutation adequacy criterion. In terms of mutation scores, while both distinguishing and traditional mutation score are correlated with real fault detection, the distinguishing mutation score is statistically more correlated to real fault detection than the traditional mutation score.

While the expensive cost is a long-standing problem in mutation testing, we should not overlook the fault detection effectiveness. Among various ways of improving the effectiveness, investigations into a stronger mutation adequacy criterion is promising because it is general enough and applicable to all mutation-based testing methods. To balance between improving the effectiveness and reducing the cost of mutation testing, one potential direction for future work is an investigation of the trade-off between reduced cost and improved fault detection capability when various mutation reduction methods are applied along with the distinguishing mutation adequacy criterion. A proper cost-benefit analysis of using the diversity-aware mutation adequacy criterion instead of the traditional mutation adequacy criterion is also needed.

There are many avenues for potential applications. Considering that a mutation adequacy criterion is the essence of mutation-based testing, the distinguishing mutation adequacy criterion is applicable for addressing various testing problems such as test data generation and test case prioritization. In particular, we have extended the studies of Fraser and Arcuri [39], [58] to generate whole test suites that satisfies the distinguishing mutation adequacy criterion while keeping the total size as small as possible. Our preliminary results showed that the test suites generated by the distinguishing mutation adequacy criterion have similar sizes and better fault detection capabilities in comparison to the test suites generated by the traditional mutation adequacy criterion. We plan to continue our research on test data generation. We hope the distinguishing mutation adequacy criterion can be utilized in both theoretical and empirical studies in the future.

References

[1] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*. Springer, 2004, pp. 320–329.

[2] D. Leon, A. Podgurski, and W. Dickinson, "Visualizing similarity between program executions," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2005, pp. 310–321.

[3] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009, pp. 201–212.

[4] N. Alshahwan and M. Harman, “Coverage and fault detection of the output-uniqueness test selection criteria,” in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014, pp. 181–192.
[5] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, “Searching for cognitively diverse tests: Towards universal test diversity metrics,” in *Proceedings of the International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, 2008, pp. 178–186.
[6] R. Feldt, S. Poulding, D. Clark, and S. Yoo, “Test set diameter: Quantifying the diversity of sets of test cases,” in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), to appear*.
[7] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649–678, 2011.
[8] A. T. Acree Jr, “On mutation,” *Ph.D. thesis*, 1980.
[9] T. A. Budd, “Mutation analysis of program test data,” *Ph.D. thesis*, 1980.
[10] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.
[11] Y. Jia and M. Harman, “Higher order mutation testing,” *Information and Software Technology (IST)*, vol. 51, no. 10, pp. 1379–1393, 2009.
[12] P. Reales Mateo, M. Polo Usaola, and J. L. Fernandez Aleman, “Validating second-order mutation at system level,” *Software Engineering, IEEE Transactions on*, vol. 39, no. 4, pp. 570–587, 2013.
[13] S. Hussain, “Mutation clustering,” *MS. Thesis*, 2008.
[14] Y.-S. Ma and S.-W. Kim, “Mutation testing cost reduction by clustering overlapped mutants,” *Journal of Systems and Software*, 2016.
[15] G. Kaminski, P. Ammann, and J. Offutt, “Improving logic-based testing,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.
[16] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing theoretical minimal sets of mutants,” in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2014, pp. 21–30.
[17] R. Just and F. Schweiggert, “Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators,” *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 490–507, 2015.
[18] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
[19] D. Shin, S. Yoo, and D.-H. Bae, “Diversity-aware mutation adequacy criterion for improving fault detection capability,” in *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 122–131.
[20] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440.
[21] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 8, pp. 608–624, 2006.
[22] N. Li, U. Praphamontripong, and J. Offutt, “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage,” in *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2009, pp. 220–229.
[23] M. Harman, Y. Jia, and W. B. Langdon, “Strong higher order mutation-based test data generation,” in *Proceedings of the symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 212–222.
[24] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 654–665.
[25] D. Shin and D.-H. Bae, “A theoretical framework for understanding mutation-based testing methods,” in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 299–308.
[26] L. J. Morell, “A theory of fault-based testing,” *IEEE Transactions on Software Engineering (TSE)*, vol. 16, no. 8, pp. 844–857, 1990.
[27] M. Woodward and K. Halewood, “From weak to strong, dead or alive? an analysis of some mutation testing issues,” in *Proceedings of the International Conference on Software Testing, Verification, and Analysis Workshop (ICSTW)*, 1988, pp. 152–158.
[28] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Theoretical and empirical studies on using program mutation to test the functional correctness of programs,” in *Proceedings of the symposium on Principles of Programming Languages (POPL)*, 1980, pp. 220–233.
[29] A. J. Offutt, “Investigations of the software testing coupling effect,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.
[30] 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.
[31] D. Shin, E. Jee, and D.-H. Bae, “Comprehensive analysis of fbd test coverage criteria using mutants,” *Software & Systems Modeling*, vol. 15, no. 3, pp. 631–645, 2016.
[32] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, “Mutant subsumption graphs,” in *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2014, pp. 176–185.
[33] H. Zhu, “A formal analysis of the subsume relation between software test adequacy criteria,” *IEEE Transactions on Software Engineering (TSE)*, vol. 22, no. 4, pp. 248–255, 1996.
[34] T. A. Budd and D. Angluin, “Two notions of correctness and their relation to testing,” *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.
[35] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, “Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation,” *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 1, pp. 23–42, 2014.
[36] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, “Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015, pp. 936–946.
[37] M. Kintis and N. Malevris, “Medic: A static analysis framework for equivalent mutant identification,” *Information and Software Technology*, vol. 68, pp. 1–17, 2015.
[38] R. Just, G. M. Kapfhammer, and F. Schweiggert, “Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis,” in *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2012, pp. 11–20.
[39] G. Fraser and A. Arcuri, “Achieving scalable mutation-based generation of whole test suites,” *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.
[40] P. R. Mateo and M. P. Usaola, “Parallel mutation testing,” *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 315–350, 2013.
[41] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, “Muvm: Higher order mutation analysis virtual machine for c,” in *Proceedings of the International Conference onSoftware Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 320–329.
[42] R. Just, “The Major mutation framework: Efficient and scalable mutation analysis for Java,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 433–436.
[43] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for Java,” in *Proceedings of the Companion to the Object-oriented programming systems and applications (OOPSLA)*, 2007, pp. 815–816.
[44] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” in *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011, pp. 416–419.
[45] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria,” in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 191–200.
[46] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges,” in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[47] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient mutation operators for measuring test effectiveness,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 351–360.

[48] M. Harder, J. Mellen, and M. D. Ernst, “Improving test suites via operational abstraction,” in *Proceedings of the 25th international conference on Software engineering*. IEEE Computer Society, 2003, pp. 60–71.

[49] P. G. Frankl and S. N. Weiss, “An experimental comparison of the effectiveness of branch testing and data flow testing,” *IEEE Transactions on Software Engineering (TSE)*, vol. 19, no. 8, pp. 774–787, 1993.

[50] A. Arcuri and L. Briand, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2014.

[51] A. Gargantini and C. Heitmeyer, “Using model checking to generate tests from requirements specifications,” in *Proceeding of the European Software Engineering Conference Held Jointly with the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1999, pp. 146–162.

[52] C. Cadar, D. Dunbar, and D. R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI*, vol. 8, 2008, pp. 209–224.

[53] P. McMinn, “Search-based software test data generation: a survey,” *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[54] 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)*. IEEE, 2010, pp. 300–309.

[55] R. Just, M. D. Ernst, and G. Fraser, “Efficient mutation analysis by propagating and partitioning infected execution states,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 315–326.

[56] M. Patrick and Y. Jia, “KD-ART: Should we intensify or diversify tests to kill mutants?” *Information and Software Technology*, 2016.

[57] B. Baudry, F. Fleurey, and Y. Le Traon, “Improving test suites for efficient fault localization,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006, pp. 82–91.

[58] G. Fraser and A. Arcuri, “Whole test suite generation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.