An Empirical Evaluation of Manually Created Equivalent Mutants

Philipp Straubinger
University of Passau
Passau, Germany

Alexander Degenhart
University of Passau
Passau, Germany

Gordon Fraser
University of Passau
Passau, Germany

***Abstract*—Mutation testing consists of evaluating how effective test suites are at detecting artificially seeded defects in the source code, and guiding the improvement of the test suites. Although mutation testing tools are increasingly adopted in practice, equivalent mutants, i.e., mutants that differ only in syntax but not semantics, hamper this process. While prior research investigated how frequently equivalent mutants are produced by mutation testing tools and how effective existing methods of detecting these equivalent mutants are, it remains unclear to what degree humans also create equivalent mutants, and how well they perform at identifying these. We therefore study these questions in the context of *Code Defenders*, a mutation testing game, in which players competitively produce mutants and tests. Using manual inspection as well as automated identification methods we establish that less than 10% of manually created mutants are equivalent. Surprisingly, our findings indicate that a significant portion of developers struggle to accurately identify equivalent mutants, emphasizing the need for improved detection mechanisms and developer training in mutation testing.**

***Index Terms*—Mutation Testing, Equivalent Mutants**

I. Introduction

Mutation testing is a well-established industry practice for identifying weaknesses in test suites and guiding test creation [1]. It involves introducing artificial faults (*mutants*) into the source code and assessing whether the corresponding test suites can detect them, thus classifying mutants into killed (detected by the test suite) or alive (undetected) [2]. Killed mutants provide a quantitative assessment of the test suite quality in terms of a mutation score (i.e., the ratio of killed mutants to mutants overall), and surviving mutants point out to developers where there are test gaps.

However, not all mutants can be killed in the first place: Equivalent mutants are semantically equivalent to the original source code, even though they differ in syntax, such that it is impossible to create tests that distinguish between the mutant and the original program. Equivalent mutants skew mutation scores and are misleading for developers, who need to manually discern whether a live mutant is equivalent or signifies a genuine test gap [3].

Contemporary mutation testing tools, such as PIT,[1] incorporate mechanisms to identify or avoid equivalent mutants. These mechanisms often rely on compiler outputs [4], [5] or specific patterns associated with equivalent mutants [3], [6], effectively identifying the most trivial cases. However, complex equivalent mutants still elude automated detection and require manual intervention by developers [3], [7]. Manual inspection of mutants, however, is time-consuming and prone to false positives and false negatives [7].

Despite existing research on how common equivalent mutants are in practice and how difficult they are to detect [8], [9], the current body of knowledge is based on established mutation tools using standard mutation operators to generate mutants. However, less is known about mutants created manually, such as through mutation testing games such as *Code Defenders* [10]–[12]. Examining such human-generated mutants is important for several reasons. First, developers would need to deal with such mutants when using crowdsourcing for mutation testing [12]. Second, human created mutants are an important element of software testing education, where mutation-based assignments, exercises, and games [11] are common. Finally, a recent trend lies in using deep learning models to mimic human edits and produce more similar mutants [13]–[15]. In all of these cases, a deeper understanding in the tendency of humans to produce equivalent mutants, and their ability to detect this, is important.

This paper aims to address the gap of knowledge on equivalence in human generated mutants by leveraging data accumulated through several years worth of *Code Defenders* usage in the context of a university course. In *Code Defenders*, players can assume the role of an attacker, introducing mutants, or a defender, creating tests to detect mutants. To cope with equivalent mutants, the game includes dedicated features focusing on equivalence, such as equivalence duels, where one player can claim a mutant as equivalent, and the mutant creator must either disprove the equivalence or accept the claim. Using the resulting dataset of mutants and tests for a set of Java classes, we investigate how many mutants can be killed by existing tests, how many of the remaining mutants can be detected as equivalent by automated equivalence detection techniques, and how many are actually equivalent. In detail, the contributions of this paper are as follows:

1) We provide an extensive dataset comprising 18,000 manually written mutants and 11,000 corresponding tests in ten different Java classes.
2) We evaluate how well existing automated methods to identify trivial equivalent mutants perform on manually created equivalent mutants.

This work is supported by the DFG under grant FR 2955/2-1, “QuestWare: Gamifying the Quest for Software Tests”.

[1] https://pitest.org/

3) We manually classify a substantial number of equivalent mutants, and assess the ability of players of *Code Defenders* to identify and classify equivalent mutants.

The results reveal that less than 10% of the manually created mutants are equivalent, and nearly two-thirds of players were unable to accurately identify manually created equivalent mutants. This may have implications beyond education, questioning the validity of equivalent mutation classifications by humans in general.

II. Background

A. Mutation Testing

Mutation testing describes the process of seeding artificial defects (mutants) into source code to identify weaknesses in existing test suites. The use of small, artificial defects is based on the *Coupling Effect* and *Competent Programmer* hypotheses [16]. The former suggests that tests revealing simple errors are effective in also uncovering more complex errors, while the latter posits that programmers create nearly correct programs with small deviations from the correct program [2]. Mutants can be classified based on the result of executing available tests against them: A failing test indicates a killed mutant, while mutants survive if all tests pass. Live mutants imply potential test suite deficiencies, and developers can use this information to strengthen their test suites. The ratio of killed to generated mutants represents the mutation score, and serves as a metric correlated with a test suite's fault-detection ability, surpassing metrics like code coverage [17].

While the theoretical aim would be to achieve a mutation score of 100%, and thus having a strong test suite able to detect all mutants, this goal is usually not achievable, as mutants may be semantically equivalent despite their syntactical differences. An equivalent mutant cannot possibly be killed by a test.

B. Equivalent Mutants

Equivalent mutants are mutated versions of the original code that, despite their altered structure, produce the same output as the unmutated code when subjected to a set of tests [3]. These mutants introduce functionally equivalent changes, making them challenging to distinguish from the original code solely based on traditional testing methods. Since finding equivalent mutants is an undecidable problem [18], it is mostly done using heuristics and partial solutions [3].

Detecting equivalent mutants involves addressing a binary classification task with potential errors: False negatives occur when an equivalent mutant is erroneously labeled as non-equivalent, while false positives represent non-equivalent mutants inaccurately marked as equivalent. In mutation testing, false positives are considered more critical as they lead to the exclusion of potentially important killable mutants [3].

Manual analysis is often necessary to identify equivalent mutants but may require significant time investment and is prone to false positives. For example, Schuler and Zeller [7] report that manual classification took 15 minutes per mutant. Yao et al. [19] reported 6 person-months of manual analysis effort dedicated entirely to classifying 1230 mutants. Automated techniques for detecting equivalent mutants are thus desirable, and fall into two main categories [4]: *Detect Approaches*, which directly determine whether a mutant is equivalent, and *Reduce Approaches*, which provide an order from less likely to more likely equivalent mutants.

Recently, Trivial Compiler Equivalence (TCE [4]) and its extended version TCE+ [5] have been demonstrated to be effective at identifying equivalent mutants. TCE identifies equivalent mutants by comparing output files after the compilation of the source code, assuming that equivalent mutants will produce identical compiled outputs. TCE+ builds upon this approach by incorporating an additional optimization step post-compilation, enhancing its effectiveness, especially in languages like Java, where optimization occurs at runtime. Unlike TCE, TCE+ compares output files after the optimization step.

The increasing adoption of machine learning approaches in software engineering has also resulted in predictive approaches not only to speed up mutation testing [20], but also to classify equivalent mutants [6], [21]–[23]. Such machine learning approaches require datasets of equivalent mutants such as MutantBench [24], but since these datasets require substantial manual labelling effort there are efforts to use automation to automatically augment equivalent mutant datasets [25].

C. Code Defenders

Code Defenders is a web application that gamifies Mutation Testing, where players have to competitively create mutants and writing tests to kill them [10]. *Code Defenders* was devised as a crowdsourcing platform to elicit strong mutants and tests, and also serves as a valuable tool for teaching software testing, addressing the common perception among students is that testing is more tedious than software development itself [26]. By incorporating mutation testing, *Code Defenders* familiarizes aspiring developers with testing concepts, potentially increasing the practical application of mutation testing in real-world scenarios [27]. *Code Defenders* has been positively received by students, enhancing their testing skills during a software testing course [11]. It is publicly accessible,[2] open-source,[3] and therefore used by different universities globally.

1) Game Modes: Code Defenders currently offers three game modes: The *Puzzle Mode* is a single-player experience where players solve predefined tasks using mutants/tests. The *Battleground Mode* represents the default multiplayer experience, where players are divided into teams of attackers and defenders, competing over a Java class under test. Finally, the *Melee Mode* is a multiplayer option where all players compete against others, each player taking on both attacker and defender roles simultaneously.

2) Testing: Depending on the game mode (i.e., *easy* vs. *hard*), defenders can either see only the locations of mutants in the game, or also their diff, and then have to create tests to kill these mutants. Tests are created in a web-based user interface and may use various common test libraries such

[2] https://code-defenders.org

[3] https://github.com/CodeDefenders/CodeDefenders

as JUnit5,[4] Hamcrest,[5] and Google Truth.[6] When tests are submitted, they undergo compilation and validation checks before acceptance into the game. The validation ensures tests are non-flaky, deterministic, concentrate on testing a limited set of functionalities, and pass on the original class under test. Depending on the game mode there are certain restrictions on what code is permitted in the tests to ensure fairness and clean tests. For example, *Code Defenders* checks submitted tests to ensure they do not contain loops, calls to `System.*`, additional methods, conditionals, or exceed a configurable maximum number of test assertions.

3) Mutation: Attackers create mutants by editing the CUT and submitting their modified versions (Fig. 1). *Code Defenders* first validates these mutants based on a configurable *Mutant Validation Level*, which can be categorized as *strict*, *moderate*, or *relaxed*. These levels try to prevent both equivalent and unfair mutants, e.g., relying on random values. In the *relaxed* level, there are minimal restrictions, only disallowing calls to `System.*` and `Random`. The *moderate* level targets mutants that might be hard to kill but offer no value for testers. Restrictions include modifying comments, adding additional logical operators and control structures, as well as ternary operators. The *strict* level prohibits adding bitwise operators, using reflection, and modifying signatures. *Code Defenders* also performs basic equivalence detection by stripping whitespaces and comparing mutant and original CUT. This prevents intentional or accidental submission of equivalent mutants (e.g., submitting after adding only a new line).

In addition to the validation of the restrictions, a hash is computed based on the whitespace-stripped mutant code. This hash is used to identify duplicate mutants within a game. If a mutant with the same hash already exists, the newly submitted mutant is rejected. This serves as a basic duplicate detection approach, preventing players from submitting identical mutants multiple times by accident or for point farming.

4) Intent Collection: While defenders always need to reason about code behavior and existing mutants when creating tests, attackers may, in particular in earlier phases of the game where the coverage achieved by the defenders is still low, arbitrarily mutate code without deeper thought. While this is a strategy that likely backfires later in the game, it is particularly undesirable in an educational context [11]. Therefore, a configurable feature aiming to force players to think more deeply about their actions before submitting them is *intent collection*: If enabled, players are required to provide additional metadata when submitting mutants or tests. In particular, attackers have to specify whether they intended to create a *killable* or *equivalent* mutant, or they can choose *don't know* if they are uncertain about the mutant's equivalence. Defenders must select a line in the CUT they intend to target when intent collection is activated.

5) Equivalence Duels: Regardless of whether created on purpose or by accident, equivalent mutants are a common aspect of games, and therefore integrated using the concept of *Equivalence Duels*. After defenders have attempted to kill a mutant and managed to create at least one test that covers it without killing it, they challenge the attacker who created the mutant to such a duel. *Code Defenders* can also be configured to automatically trigger these duels if a mutant has been covered by a specified number of tests without being killed.

TABLE I
OVERVIEW OF CLASSES USED FOR ANALYSIS

CUT Alias	**Years played**	**Number of Games**
ByteVector	1	8
Complex_V1	2	23
Complex_V2	1	20
Document	3	24
HSLColor	1	8
IntHashMap	5	40
Lift	3	37
Options	3	24
Rational	2	36
SparseIntArray	5	43

When challenged to an equivalence duel for a mutant they created, the attacker is temporarily blocked from submitting more mutants until the duel is resolved (Fig. 2). The duel can be resolved by the attacker by submitting a valid test that kills the mutant, thus proving its non-equivalence and earning the attacker a win. Submitting a valid test that does not kill the mutant results in the defender suspecting the mutant to be equivalent winning the duel. Alternatively, the attacker can also accept the mutant as equivalent, leading the defender to win the duel and assuming the mutant's equivalence. To incentivize attackers to invest time in thoughtful testing, losing an equivalence duel results in the loss of all accumulated points for the mutant, while winning the duel earns the attacker an additional point. Independently of the game difficulty setting, attackers always get to see the diff of the mutant that is part of the duel, since they created it in the first place.

III. Evaluation

To understand the role of equivalent mutants within *Code Defenders*, we aim to answer the following research questions:

- ***RQ 1****: How well does TCE(+) perform on manually written mutants?*
- ***RQ 2****: How many equivalent mutants do players of* Code Defenders *create?*
- ***RQ 3****: How well do players perform at detecting (non-) equivalent mutants?*

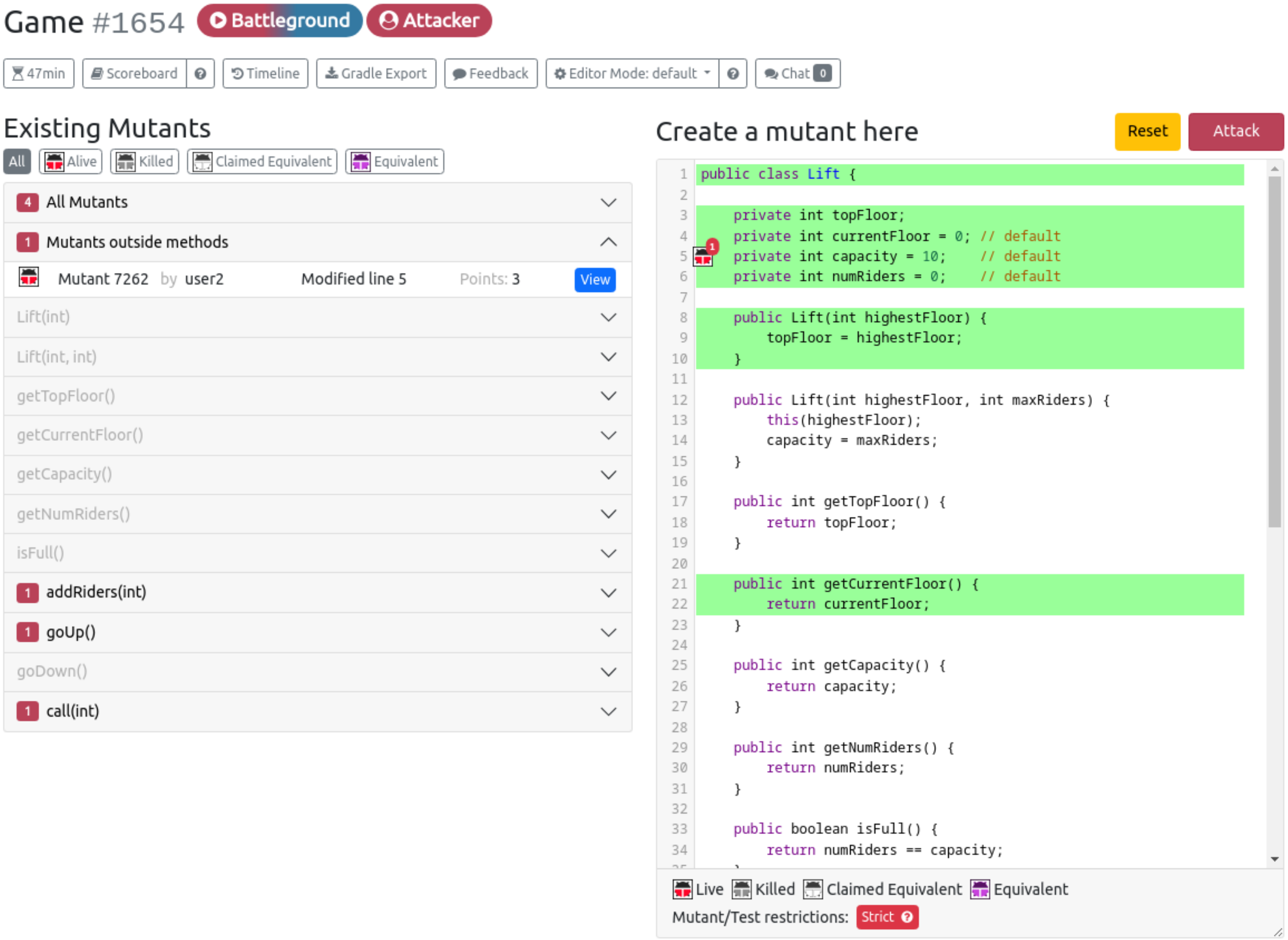
A. Dataset

Our dataset encompasses information gathered from sessions using *Code Defenders* during Software Testing lectures at University of Passau over the last five years (2018–2022). From this data, we extracted the source code for the Classes Under Test (CUTs), along with details about the number and types of games, mutants, and tests associated with each game. Our focus is on battleground games, the most established game type used consistently across all years.

[4] https://junit.org/junit5/

[5] https://hamcrest.org/JavaHamcrest/

[6] https://github.com/google/truth



4 All Mutants

1 Mutants outside methods

Mutant 7262 by user2 Modified line 5 Points: 3 View

Lift(int)

Lift(int, int)

getTopFloor()

getCurrentFloor()

getCapacity()

getNumRiders()

isFull()

1 addRiders(int)

1 goUp()

goDown()

1 call(int)

```
public class Lift {

    private int topFloor;
    private int currentFloor = 0; // default
    private int capacity = 10;    // default
    private int numRiders = 0;    // default

    public Lift(int highestFloor) {
        topFloor = highestFloor;
    }

    public Lift(int highestFloor, int maxRiders) {
        this(highestFloor);
        capacity = maxRiders;
    }

    public int getTopFloor() {
        return topFloor;
    }

    public int getCurrentFloor() {
        return currentFloor;
    }

    public int getCapacity() {
        return capacity;
    }

    public int getNumRiders() {
        return numRiders;
    }

    public boolean isFull() {
        return numRiders == capacity;
```

Mutant/Test restrictions: Strict

Fig. 1. Attacker view of *Code Defenders*

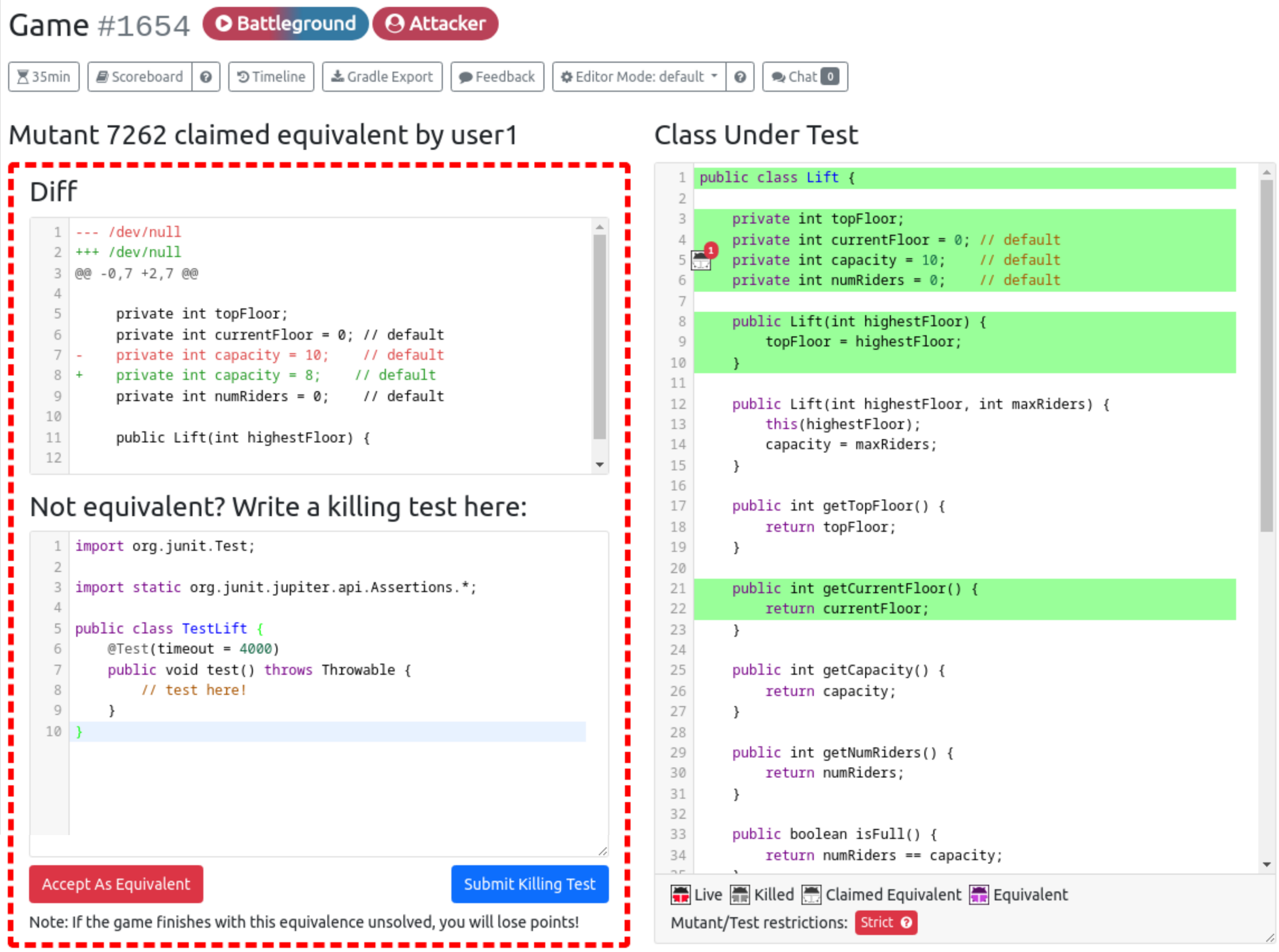
Throughout these years, various CUTs were utilized in *Code Defenders* sessions, predominantly chosen from an available pool [11]. To ensure a sufficient number of mutants and tests for each CUT, we included all CUTs played in at least two years. However, changes to attribute visibilities and additional getters in the `Complex` class led to two different versions of this class, which we count as two distinct CUTs in the dataset. The *Code Defenders* instances not only featured games played seriously but also those created exclusively for testing or demonstration purposes. Consequently, we excluded games with fewer than 15 submitted mutants and tests, respectively. This process resulted in a total of 10 CUTs (Table I).

To prepare the dataset for further evaluation, we compiled and executed all extracted tests against the corresponding CUT, eliminating any tests that either failed to compile or did not pass against the unchanged CUT. In a subsequent step, we compiled the mutants, discarding those that did not compile. The tests from the preceding step were then executed against each mutant. Any mutants for which at least one test fails were then classified as killable, as they cannot be equivalent.

TABLE II
KILLED MUTANTS

CUT Alias	Total	Kill.	Alive	Kill. %	Alive %	Tests
ByteVector	608	477	131	78.45%	21.55%	236
Complex_V1	1447	1341	106	92.67%	7.33%	517
Complex_V2	1374	1281	93	93.23%	6.77%	797
Document	1791	1651	140	92.18%	7.82%	1009
HSLColor	739	553	186	74.83%	25.17%	342
IntHashMap	3472	3253	219	93.69%	6.31%	2935
Lift	1862	1684	178	90.44%	9.56%	1254
Options	1845	1758	87	95.28%	4.72%	684
Rational	1383	1205	178	87.13%	12.87%	681
SparseIntArray	3702	3392	310	91.63%	8.37%	2806
Total	18223	16595	1628	91.07%	8.93%	11261

Table II presents an overview of both the killed and alive mutants after running the tests against them. It also includes the total number of mutants and tests separated by CUT. Notably, more than 90% of all valid mutants in the datasets were killed by tests. For the majority of CUTs, the ratio of killed mutants exceeds 90%. However, `ByteVector` and `HSLColor` stand out as apparent outliers. Several factors could contribute to their lower percentage of killed mutants.



Mutant 7262 claimed equivalent by user1

Diff

```
--- /dev/null
+++ /dev/null
@@ -0,7 +2,7 @@

     private int topFloor;
     private int currentFloor = 0; // default
-    private int capacity = 10;    // default
+    private int capacity = 8;    // default
     private int numRiders = 0;    // default

     public Lift(int highestFloor) {

```

Not equivalent? Write a killing test here:

```java
import org.junit.Test;

import static org.junit.jupiter.api.Assertions.*;

public class TestLift {
    @Test(timeout = 4000)
    public void test() throws Throwable {
        // test here!
    }
}
```

Note: If the game finishes with this equivalence unsolved, you will lose points!

Class Under Test

```java
public class Lift {

    private int topFloor;
    private int currentFloor = 0; // default
    private int capacity = 10;    // default
    private int numRiders = 0;    // default

    public Lift(int highestFloor) {
        topFloor = highestFloor;
    }

    public Lift(int highestFloor, int maxRiders) {
        this(highestFloor);
        capacity = maxRiders;
    }

    public int getTopFloor() {
        return topFloor;
    }

    public int getCurrentFloor() {
        return currentFloor;
    }

    public int getCapacity() {
        return capacity;
    }

    public int getNumRiders() {
        return numRiders;
    }

    public boolean isFull() {
        return numRiders == capacity;
```

Mutant/Test restrictions: Strict

Fig. 2. Equivalence duel during a game of *Code Defenders*

One possibility is that fewer games were played with these two classes, resulting in fewer accumulated tests, particularly for corner cases (about 200 to 300 tests for `ByteVector` and `HSLColor` compared to more than 500 for all other CUTs). Another factor could be that the games involving these two classes utilized the *Intention Collection* feature, potentially influencing how many equivalent mutants attackers create.

B. Analysis Procedure

1) RQ 1: How well does TCE(+) perform on manually written mutants: The first research question aims to evaluate how well the state-of-the-art approaches for identifying equivalent mutants, TCE/TCE+, performs at identifying equivalent mutants created in *Code Defenders* games. Trivial Compiler Equivalence (TCE) [4] identifies mutants as equivalent if their compiled files match the compilation output of the original CUT. However, since Java's compilation involves minimal optimization, particularly compared to languages like C or Fortran, TCE may not be as effective. To address this, we also utilize TCE+, an extension of TCE that incorporates an optimization step after compilation, utilizing the optimized class files for equivalence detection [5].

The required files for TCE are obtained by compiling the sources, while for TCE+ detection, the class files must undergo an optimization after compilation. Following the approach in the original TCE+ paper [5], we employ ProGuard,[7] an open-source shrinker and optimizer designed for Java, primarily aimed at Android apps. Our configuration of ProGuard retains all classes, methods, and attributes regardless of their access modifiers, a crucial consideration as tests may employ reflection to access attributes or methods.

To ensure that ProGuard optimization retains all components used by the tests, the initial step involves optimizing the CUTs, followed by executing the test suites against the optimized versions to confirm the success of all tests. Subsequently, the mutants undergo optimization as well. To assess whether the class files of a mutant (whether normal or optimized) match those of the CUT, we generated `SHA256` hashes for

[7] https://github.com/Guardsquare/proguard

TABLE III
NUMBER OF (RESOLVED) DUELS

CUT Alias	Duels	Mutants killed outside	Resolved duels
ByteVector	66	6	34
Complex_V1	36	2	26
Complex_V2	59	0	43
Document	276	17	186
HSLColor	190	17	129
IntHashMap	1067	113	822
Lift	215	8	137
Options	196	25	83
Rational	129	7	69
SparseIntArray	693	41	554
Total	2927	236	2082

all `.class` files and verified whether the checksums of all mutant files correspond to those of the CUT.

This leaves us with a set of mutants that are neither killed by tests, nor flagged as equivalent by TCE or TCE+. Next, we manually examined a random sample constituting 20% of the remaining mutants in each subgroup: those not eliminated by a test in a duel, mutants marked as equivalent in a duel, and mutants neither involved in a duel nor eliminated. This process yielded an estimated ratio of equivalent mutants in the initial dataset, which we could then compare with the number of equivalent mutants identified by TCE and TCE+.

2) RQ 2: How many equivalent mutants do players of Code Defenders *create?:* The overall dataset combines mutants from multiple games for multiple classes. To understand player behavior in games and answer this research question, we reuse the data containing both the automatically and manually tagged equivalent mutants gathered from RQ1.

3) RQ 3: How well do players perform at detecting (non-) equivalent mutants: To answer this research question, we consider the two mechanisms intended to address equivalent mutants in *Code Defenders*: First, we extract equivalence duels from the database, including the current state of the mutant under consideration. We compare the ratio of equivalent mutants with and without equivalence duels and further analyze mutants with automatically triggered versus manually triggered equivalence duels. Additionally, we examine player actions in equivalence duels based on whether the duel subject is an equivalent mutant or not. For equivalent mutants, this involves whether the player accepted the mutant as equivalent or submitted a killing test. For non-equivalent mutants, we also investigated whether the player accepted the mutant as equivalent, submitted a killing test, and assessed whether that test successfully killed the mutant.

To maintain data consistency and meaningfulness, six mutants from a single `Document` CUT game were excluded from further analysis. In these cases, the attacker provided a test that killed the mutant, yet these mutants were not killed in the analysis, with one even identified as equivalent. This discrepancy may have arisen from mishandling an edge case or a temporary system problem during that specific game.

There are several instances where mutants included in equivalence duels are killed outside of their duel, which occurs when a defender submits a new test after a duel has been triggered (Table III). This can result in the mutant being killed by the defender, causing issues with the duel resolution process. Table III shows that this affects at least 5% of all equivalence duels for nearly all CUTs. To conduct further analysis, we exclude these mutants from the study as their deaths outside of the duel prevent a normal resolution status from being available. Additionally, we also excluded duels that were not resolved by the end of the game, where the resolving rate ranged between 49% and 86% for the different CUTs.

Second, we extract intention information and correlate it with the data collected in Section III-B1 regarding whether mutants were killed or identified as equivalent. For the remaining unknown mutants, we apply the same sampling strategy as in Section III-B1 to manually investigate 20% per subgroup (intended equivalent, intended not equivalent, and not provided), providing an estimate for all mutants. We then analyze how many mutants have a correct, incorrect, or unknown intent, and explore whether players performed equally well in classifying equivalent and non-equivalent mutants or if they were more adept at identifying one over the other.

C. Threats to Validity

a) Threats to external validity: Potential variations in surrounding conditions during game sessions and the dataset's specificity to the Code Defenders session at the university may limit the generalizability of results to other contexts or CUTs. In particular intent information for mutants is available only for two CUTs (`ByteVector` and `HSLColor`). Changes in dependencies, Java versions, and *Code Defenders* over the years may further limit the external validity of the findings. Involving only students in generating mutants and conducting tests could reduce the applicability of the findings and might yield different outcomes compared to professionals working in industry settings. However, it is worth noting that the students who took part in this study were nearing the completion of their Bachelor's degrees, implying they already possessed a certain level of knowledge and experience in testing.

b) Threats to internal validity: The manual analysis of mutants introduces the potential for misclassification, which we tried to minimize using a combination of large test suites dedicated to detecting mutants and automated detection methods. Our sampling procedure for classifying subpopulations of mutants may result in a bias, which we tried to mitigate using a large sample with substantial manual classification effort.

IV. RESULTS

A. RQ 1: How many equivalent mutants are detected automatically?

Table IV provides an overview of the live mutants before automatic detection, excluding mutants killed by tests contained in the dataset, and the number of mutants automatically identified as equivalent by either TCE or TCE+ alongside the remaining unknown mutants, which were neither automatically killed nor flagged as equivalent.

TABLE IV
AUTOMATICALLY DETECTED MUTANTS BASED ON ALIVE MUTANTS

CUT Alias	Alive	Detected		Unknown
		Total	%	
ByteVector	131	26	19.85%	105
Complex_V1	106	39	36.79%	67
Complex_V2	93	28	30.11%	65
Document	140	22	15.71%	118
HSLColor	186	46	24.73%	140
IntHashMap	219	63	28.77%	156
Lift	178	56	31.46%	122
Options	87	3	3.45%	83
Rational	178	60	33.71%	118
SparseIntArray	310	91	29.35%	219
Total	1628	434	26.66%	1193

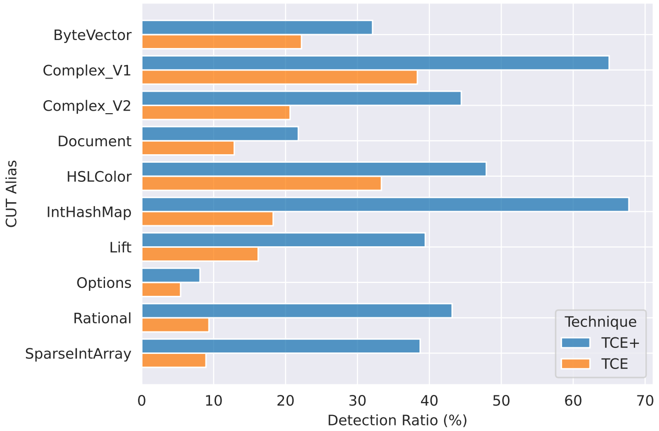


Fig. 3. Equivalent Mutant Detection Ratios for the TCE and TCE+ techniques

Notably, of the approximately 1600 mutants remaining after removing all killed mutants, more than a fourth were detected as equivalent. The ratio of detected equivalent mutants varies across CUTs, ranging from around 3% to nearly 37%. No apparent reasons explain the variations in the ratio of detected equivalent mutants among the remaining live mutants. One minor observation is that the `Options` CUT, with the lowest ratio of detected equivalent mutants (i.e., 3%), also had the highest ratio of killed mutants (i.e., 95%), although this is likely coincidental, as related comparisons do not align (i.e., the highest ratio of detected equivalent mutants do not correspond to the lowest ratio of killed mutants).

The manual classification of 22.5% (268) of the remaining mutants not identified by TCE/TCE+ allows us to estimate the total number of equivalent mutants, and the detection ratio of both techniques. TCE+ achieves an equivalent mutant detection ratio ranging from around 32% to 48% in most cases (see Fig. 3), with the best detection ratio exceeding 65%. On the other hand, TCE detects a maximum of 38% of all equivalent mutants, with detection ratios between 9% and 22%. Overall, TCE detected 16.7% of all equivalent mutants, whereas TCE+ managed to detect 41.5%.

```
   while (it.hasNext()) {
-    IndexableField field = it.next();
+    IndexableField field;
+    field = it.next();
     if (field.name().equals(name)) {
       it.remove();
```

Listing 1. Equivalent mutant in `Document` found by TCE+ but not by TCE

TABLE V
EQUIVALENT MUTANT RATIOS PER CUT FOR ALL MUTANTS

CUT Alias	Detected	Estimated	Total
ByteVector	4.28%	9.05%	13.32%
Complex_V1	2.70%	1.42%	4.12%
Complex_V2	2.04%	2.55%	4.59%
Document	1.23%	4.39%	5.62%
HSLColor	6.22%	6.77%	12.99%
IntHashMap	1.81%	0.87%	2.68%
Lift	3.01%	4.64%	7.65%
Options	0.16%	1.85%	2.01%
Rational	4.34%	5.69%	10.03%
SparseIntArray	2.46%	3.90%	6.36%
Total	2.38%	3.36%	5.75%

All mutants identified as equivalent by TCE were also detected by TCE+, and TCE+ consistently outperforms TCE across all CUTs (Fig. 3). The degree of improvement with TCE+ compared to TCE varies among different CUTs. For some classes (e.g., `Rational` and `SparseIntArray`), TCE+ identifies approximately four times as many equivalent mutants as TCE, while for others (e.g., `ByteVector`, `HSLColor`, and `Options`), the additional optimization step reveals only around 50% more mutants. An example of a mutant detected by TCE+ but not by TCE is illustrated in Listing 1. In this case, a field's declaration and initialization are separated into two lines, a similarity that TCE+ can recognize because it results in the same Java bytecode after optimization.

Summary (*RQ 1*): TCE and TCE+ successfully identified more than a quarter of the remaining alive mutants as equivalent. TCE+ consistently outperformed TCE in detecting equivalent mutants across different CUTs, with a detection rate of 41.5%, while TCE only achieved 16.7%.

B. RQ 2: How many equivalent mutants do players of Code Defenders *create?*

Given the classification of equivalent and non-equivalent mutants allows us to look at player behavior, i.e., how many equivalent mutants are usually created in a game of *Code Defenders*. Table V shows the average ratio of equivalent mutants among all submitted mutants, suggesting a range of 2% to approximately 13% per game with a total of 5.75%.

Notably, the `Options` CUT exhibits the lowest share of equivalent mutants, aligning with its highest percentage of mutants detected as not equivalent and the lowest percentage of mutants automatically identified as equivalent. Conversely, the `ByteVector` and `HSLColor` CUTs have the highest ratio of equivalent mutants. Interestingly, games for these two CUTs utilized the *Intention Collection* feature, where players

TABLE VI
NUMBER OF RESOLVED DUELS AND THEIR OUTCOMES WITH EM = EQUIVALENT MUTANT

CUT Alias	Resolved duels	Resolved duels / game	Mutant killed by test	EM survived test	Non EM survived test	Correctly accepted	Wrongly accepted
ByteVector	34	4.25	8	4	3	17	5
Complex_V1	26	1.13	3	4	1	9	10
Complex_V2	43	2.15	20	8	5	6	9
Document	186	7.75	94	28	21	41	23
HSLColor	129	16.13	60	23	17	38	8
IntHashMap	822	20.55	209	179	162	55	379
Lift	137	3.70	47	27	13	39	23
Options	83	3.46	29	21	17	11	22
Rational	69	1.92	13	15	7	33	7
SparseIntArray	554	12.88	157	132	88	93	172
Total	2082	7.92	640	441	334	342	658

TABLE VII
RESULTS OF THE MUTANT INTENTIONS GIVEN BY THE PLAYERS, AND THEIR EQUIVALENCE STATUS

CUT Alias	**Total**	**Correct %**	**Incorrect %**	**Not provided %**
Intention given by the players				
ByteVector	608	88.32%	4.11%	7.57%
HSLColor	739	88.36%	6.22%	5.41%
When the intention was not equivalent				
ByteVector	527	92.18%	1.53%	6.3%
HSLColor	643	93.07%	1.23%	5.7%
When the intention was equivalent				
ByteVector	81	64.29%	20.24%	15.48%
HSLColor	96	54.44%	42.22%	3.33%

indicated whether they intended to create an equivalent mutant or not. With this feature enabled, players might intentionally create more equivalent mutants because they are aware of the possibility. Conversely, in games featuring other CUTs, people may create most of their equivalent mutants unintentionally.

While manually inspecting the sample of remaining unknown mutants, we observed that some mutants were trivially equivalent. These mutants often employed patterns such as adding equivalent arithmetic (e.g., adding `+<value>-<value>` with the value typically being `+0` and `*1` like in Listing 3), introducing unnecessary calls to methods or field declarations as depicted in Listing 4, or expanding comparisons which would result in the same return value (see Listing 5). We conjecture that these are intentionally created equivalent mutants: If playing in *hard* mode the actual syntactical change is not shown to defenders, and in that case, it does not matter what an equivalent mutant looks like.

There are also non-trivial equivalent mutants, such as shown in Listing 2: The mutant wraps a method call, computing the absolute value around another method that retrieves the index of a key, which is always a positive number; this mutant would also be the result of a traditional "absolute value insertion" mutation, but it can be detected neither by TCE nor by TCE+.

Some mutants in our dataset also proved challenging to detect rather than equivalent. For example, Listing 6 shows a mutant that replaces the return type from a `SingletonList` to an `ArrayList`, a change that can only be identified by

```
  if (requiredOpts.contains(key)) {
-   requiredOpts.remove(
-     requiredOpts.indexOf(key));
+   requiredOpts.remove(
+     Math.abs(requiredOpts.indexOf(key)));
  }
```

Listing 2. Non-trivial equivalent mutant in `Options`

```
  public Complex pow(double power) {
-   double r = abs();
+   double r = abs()+0.0;
    double theta = angle();
```

Listing 3. Trivial equivalent mutant in `Complex` not found bei TCE(+)

```
  @Override
  public Iterator<IndexableField> iterator() {
+   fields.toString();
    return fields.iterator();
  }
```

Listing 4. Trivial equivalent mutant in `Document` not found bei TCE(+)

```
  private int iMax(int a, int b) {
-   if (a > b) return a; else return b;
+   if (a >= b) return a; else return b;
  }
```

Listing 5. Trivial equivalent mutant in `HSLColor` not found bei TCE(+)

```
  if (longOpts.keySet().contains(opt)) {
-   return Collections.singletonList(opt);
+   List<String> list = new ArrayList<>();
+   list.add(opt);
+   return list;
  }
```

Listing 6. Mutant difficult to detect in `Options`

```
- if (str.startsWith("--")) {
+ if (str.contains("--")) {
    return str.substring(2);
  }
```

Listing 7. Mutant difficult to detect in `Options`

TABLE VIII
RATIO OF EQUIVALENT MUTANTS (EM) IN DUELS IN %

CUT Alias	Overall	Manual duels	Automatic duels
ByteVector	42.42	37.74	61.54
Complex_V1	36.11	38.46	40.00
Complex_V2	27.12	27.12	–.–
Document	20.92	19.84	28.57
HSLColor	30.00	28.24	29.52
IntHashMap	7.31	6.27	8.72
Lift	33.80	35.84	37.50
Options	5.61	6.29	8.11
Rational	50.00	50.00	–.–
SparseIntArray	20.92	24.28	12.08
Average	27.42	27.41	28.26

inspecting the type using the `instanceOf` operator. This mutant is hard to detect, but may not be desirable for mutation testing since bugs in return types would be caught by the compiler. On the other hand, Listing 7 shows a mutant that can only be detected if an incorrect `String` input is used, containing a double hyphen somewhere within. This appears to be a strong and useful mutant since it can reveal potentially serious bugs in `String` manipulation.

Summary (*RQ 2*): In total 5.75% of all player-submitted mutants are equivalent, which is 6.94% per CUT on average.

C. RQ 3: How well do players perform at detecting (non-) equivalent mutants?

Of those mutants with involved in equivalent duels, the proportion of equivalent mutants was always 50% or lower (Table VIII), which suggests that defenders frequently gave up, or overestimated the quality of their tests.

On two CUTs (`Complex_V2` and `Rational`) the option to trigger equivalence duels automatically was disabled and therefore no automatic duels were triggered. For all other CUTs, the ratio of equivalent mutants is almost always higher for automatically triggered duels than for manual ones (Table VIII). Defenders, restricted to seeing only the line where the mutant is located rather than the mutated code itself, may use many attempts to reveal a mutant, which may be somewhat unfair if that mutant is equivalent and the defenders are persistent. The higher ratio of equivalent mutants suggests that automatically triggering an equivalence duel after several attempts (10 by default) achieves its purpose of reducing such futile attempts, thus ensuring continuing gameplay.

When submitting a test in a duel, there are two potential outcomes: either the test successfully eliminates the mutant or the mutant survives the tests. About 45% of duels where attackers submitted a test resulted in the successful elimination of the mutant, as shown in Table VI. The remaining 55% of mutants survived, either being equivalent or not. Among these surviving mutants, 57% were equivalent (37.35% of all duels resolved with a test), but the attackers failed to recognize that and tried to write a killing test anyway. Conversely, 43% were not equivalent (23.65% of all duels resolved with a test), yet the players still failed to write a killing test.

When attackers assume a mutant is equivalent, then during a duel they can indicate this by selecting “accept mutant as equivalent”. Table VI illustrates the number of correctly and incorrectly accepted equivalent mutants, revealing that only around 35% were accurately identified as equivalent, while approximately 65% were mutants that were not equivalent. This again indicates that the attackers were not proficient at identifying equivalent mutants, or perhaps they were simply eager to conclude the duel swiftly to resume mutant creation.

Table VII displays the number of mutants and their proportions categorized based on whether their intention information was correct, incorrect, or not provided by the player. Notably, 88% of mutants were correctly classified for both classes, but `HSLColor` had a slightly higher proportion of incorrectly classified mutants compared to `ByteVector`, which had more unclassified mutants. Focusing on non-equivalent mutants within the dataset of mutants with intentions (Table VII), players demonstrated proficiency with over 92% being correctly classified as not equivalent, with an error rate below 2% for both CUTs. However, for equivalent mutants within the dataset of mutants with intentions, the ratio of correct classified intentions by the attackers is 64% and 54% for `ByteVector` and `HSLColor`, respectively, indicating a lower accuracy than the overall correctness ratio suggests, and generally more uncertainty about equivalent mutants.

Summary (*RQ3*): While the majority of players accurately indicated their intention to create an equivalent mutant or not, nearly two-thirds of them were unable to correctly identify equivalent mutants created by other players or themselves.

V. RELATED WORK

Several papers have explored Code Defenders, focusing on mutants and tests. An analysis of 20 games, each featuring a different CUT, reports an average mutation score of 69.48% but does not delve into equivalent mutants [12]. A further study of 12 classes with multiple games per class, showed that 85% of valid mutants could be detected as killable [11], but provided only a surface-level overview of player interactions with equivalence duels. Our study builds on these findings by leveraging tests from multiple years to identify killable mutants, revealing that over 91% of valid mutants, on average, can be detected by these tests. Additionally, a thorough examination of equivalent mutants is conducted, estimating the ratio of equivalent mutants per CUT through automatic equivalence detection and manual investigation of a randomly sampled subset of remaining unknown mutants.

For automatic equivalent mutant detection, we used TCE [4] and TCE+ [5]. TCE, originally designed for C programs, underwent evaluation on a tagged mutant dataset, while TCE+ was evaluated on automatically generated and expert-tagged mutants. In contrast, this paper employs manually created human mutants from Code Defenders, not tagging all mutants but using existing tests to exclude known killable mutants. TCE was found incapable of detecting Java mutants and reported TCE+ detecting 18% to 100% of equivalent mutants [5]. In this study, TCE+ is more effective, but TCE is also capable of detecting equivalent mutants. However, TCE+ does not detect over 70% of equivalent mutants for any CUT in this dataset.

People's proficiency in classifying mutants as equivalent or not was initially examined using four Cobol programs with roughly 40% equivalent mutants [28]. Competent programmers correctly classified 80% of mutants, misclassifying 12% killable mutants and 33% equivalent mutants. In contrast, our intention data involves two programs with a larger number of people who only classified their own mutants. Results from this experiment show differences but align with a similar trend: players classified 88% of mutants correctly while misclassifying 8% of killable and 40% of equivalent mutants.

We studied mutants created manually in *Code Defenders*. Deep learning models have been recently suggested to create mutants that resemble real faults [13]–[15]. It is conceivable that mutants created by deep learning models resemble human-written equivalent mutants more than those created by traditional operators, but further research is required.

VI. Conclusions

Up to 13% of mutants created by humans are equivalent, some intentionally crafted and others created by accident. A substantial share of these equivalent mutants can be found using TCE+, which is important since human classification of equivalent mutants is prone to errors, with incorrect identification occurring in almost two-thirds of cases.

Expanding the capabilities of *Code Defenders* presents opportunities to enhance players' understanding of equivalent mutants. One approach might be to incorporate puzzles specifically designed for resolving equivalence duels, educating players on how to identify equivalent mutants. Additionally, an option could be introduced in equivalence duels where players can indicate that a mutant is not equivalent, but they lack the knowledge to write a test to prove it. A more comprehensive study involving larger projects could yield deeper insights into human proficiency in detecting equivalent mutants.

In order to support experiment replications and further research on mutation testing, our dataset is available at:

https://doi.org/10.6084/m9.figshare.25144313

References

[1] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Trans. Software Eng.*, vol. 48, no. 10, pp. 3900–3912, 2022.

[2] 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.

[3] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Softw. Test. Verification Reliab.*, vol. 4, no. 3, pp. 131–154, 1994.

[4] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pp. 936–946, IEEE Computer Society, 2015.

[5] M. Houshmand and S. Paydar, "TCE+: an extension of the TCE method for detecting equivalent mutants in java programs," in *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017*, vol. 10522 of *LNCS*, pp. 164–179, Springer, 2017.

[6] M. R. Naeem, T. Lin, H. Naeem, and H. Liu, "A machine learning approach for classification of equivalent mutants," *Journal of Software: Evolution and Process*, vol. 32, no. 5, p. e2238, 2020.

[7] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Softw. Test. Verification Reliab.*, vol. 23, no. 5, pp. 353–374, 2013.

[8] I. Marsit, A. Ayad, D. Kim, M. Latif, J. M. Loh, M. N. Omri, and A. Mili, "The ratio of equivalent mutants: A key to analyzing mutation equivalence," *J. Syst. Softw.*, vol. 181, p. 111039, 2021.

[9] R. Pitts, "Random mutant selection and equivalent mutants revisited," in *IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022*, pp. 170–178, IEEE, 2022.

[10] J. M. Rojas and G. Fraser, "Code defenders: A mutation testing game," in *Int. Conference on Software Testing, Verification and Validation Workshops*, pp. 162–167, IEEE Computer Society, 2016.

[11] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas, "Gamifying a software testing course with code defenders," in *ACM Technical Symposium on Computer Science Education, SIGCSE 2019*, pp. 571–577, ACM, 2019.

[12] J. M. Rojas, T. D. White, B. S. Clegg, and G. Fraser, "Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, pp. 677–688, IEEE / ACM, 2017.

[13] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pp. 301–312, IEEE, 2019.

[14] M. Tufano, J. Kimko, S. Wang, C. Watson, G. Bavota, M. D. Penta, and D. Poshyvanyk, "Deepmutation: a neural mutation tool," in *International Conference on Software Engineering*, pp. 29–32, ACM, 2020.

[15] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," in *Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 160–169, IEEE, 2022.

[16] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, "The design of a prototype mutation system for program testing," in *American Federation of Information Processing Societies: 1978 National Computer Conference*, vol. 47, pp. 623–629, AFIPS Press, 1978.

[17] P. J. Walsh, *A measure of test case completeness (software, engineering)*. State University of New York at Binghamton, 1985.

[18] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, pp. 31–45, 1982.

[19] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Int. Conference on Software Engineering (ICSE)*, pp. 919–930, 2014.

[20] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 342–353, 2016.

[21] S. Peacock, L. Deng, J. Dehlinger, and S. Chakraborty, "Automatic equivalent mutants classification using abstract syntax tree neural networks," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 13–18, IEEE, 2021.

[22] C. Brito, V. H. Durelli, R. S. Durelli, S. R. de Souza, A. M. Vincenzi, and M. E. Delamaro, "A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 304–313, IEEE, 2020.

[23] K. Jammalamadaka and N. Parveen, "Equivalent mutant identification using hybrid wavelet convolutional rain optimization," *Software: Practice and Experience*, vol. 52, no. 2, pp. 576–593, 2022.

[24] L. van Hijfte and A. Oprescu, "Mutantbench: an equivalent mutant problem comparison framework," in *Int. Conference on Software esting, Verification and Validation Workshops (ICSTW)*, pp. 7–12, IEEE, 2021.

[25] S. Chung and S. Yoo, "Augmenting equivalent mutant dataset using symbolic execution," in *Int. Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 150–159, IEEE, 2022.

[26] S. G. Elbaum, S. Person, J. Dokulil, and M. Jorde, "Bug hunt: Making early software testing lessons engaging and affordable," in *29th International Conference on Software Engineering (ICSE 2007)*, pp. 688–697, IEEE Computer Society, 2007.

[27] R. A. P. Oliveira, L. B. R. Oliveira, B. B. P. Cafeo, and V. H. S. Durelli, "Evaluation and assessment of effects on exploring mutation testing in programming courses," in *2015 IEEE Frontiers in Education Conference, FIE 2015*, pp. 1–9, IEEE Computer Society, 2015.

[28] A. T. Acree Jr, *On mutation*. Georgia Institute of Technology, 1980.