Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs

Benjamin Kushigian
University of Washington
Seattle, USA
benku@cs.washington.edu

Samuel J. Kaufman
University of Washington
Seattle, USA
kaufmans@cs.washington.edu

Ryan Featherman
University of Washington
Seattle, USA
feathr@cs.washington.edu

Hannah Potter
University of Washington
Seattle, USA
hkpotter@cs.washington.edu

Ardi Madadi
University of Washington
Seattle, USA
ardier@cs.washington.edu

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

Abstract

The presence of equivalent mutants has long been considered a major obstacle to the widespread adoption of mutation analysis and mutation testing. This paper presents a study on the types and prevalence of equivalent mutants in real-world Java programs. We conducted a ground-truth analysis of 1,992 mutants, sampled from 7 open source Java projects. Our analysis identified 215 equivalent mutants, which we grouped based on two criteria that describe *why* the mutants are equivalent and *how challenging* their detection is. From this analysis, we observed that (1) the median equivalent mutant rate across the 7 projects is 2.97%; (2) many equivalent mutants are caused by common programming patterns and their detection is not much more complex than structural pattern matching over an abstract syntax tree.

Based on the findings of our ground-truth analysis, we developed Equivalent Mutant Suppression (EMS), a technique that comprises 10 efficient and targeted analyses. We evaluated EMS on 19 open-source Java projects, comparing the effectiveness and efficiency of EMS to two variants of Trivial Compiler Equivalence (TCE), the current state of the art in equivalent mutant detection. Additionally, we analyzed all 9,047 equivalent mutants reported by any tool to better understand the types and frequencies of equivalent mutants found. Overall, EMS detects 8,776 equivalent mutants within 325 seconds; TCE detects 2,124 equivalent mutants in 2,938 hours.

CCS Concepts

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

Keywords

Mutation Testing, Equivalent Mutants, Static Analysis

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680310

ACM Reference Format:
Benjamin Kushigian, Samuel J. Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680310

1 Introduction

Mutation analysis measures a test suite's ability to detect systematically seeded artificial faults called (program) *mutants*. Mutation testing builds on top of mutation analysis by presenting undetected mutants as test goals to a developer. However, some mutants are *equivalent*—semantically identical to the original program and thus cannot be detected by any test. Equivalent mutants are problematic for both mutation analysis and mutation testing. For mutation analysis, equivalent mutants skew adequacy measures and waste computational resources. For mutation testing, equivalent mutants, when presented as (unsatisfiable) test goals, waste developer time.

Despite a large body of work on equivalent mutant detection approaches (e.g., [3, 18, 21, 24, 26]), systematic studies on interactions between specific mutation operators and the program contexts that lead to their equivalence are missing. This paper closes this gap by investigating how prevalent equivalent mutants are and how challenging their detection is. Additionally, based on a finding that many equivalent mutants can be detected with efficient static analyses, this paper proposes and evaluates Equivalent Mutant Suppression (EMS), a technique that suppresses equivalent mutants at generation time. The evaluation compares EMS to two variants of Kintis et al.'s *Trivial Compiler Equivalence* (TCE) [18], which is the state of the art for Java programs.

This paper investigates the following research questions for the Java programming language:

RQ1 How common are equivalent mutants in the wild?
RQ2 What types of equivalent mutants exist in the wild?

We answer RQ1 and RQ2 with a systematic ground-truth analysis of 1,992 mutants across 7 Java projects (Section 4). Our qualitative analysis of 215 equivalent mutants provides concrete examples and a classification based on two dimensions, measuring *why a mutant is equivalent* and *the reasoning power needed to detect it.*

RQ3 How effective is EMS compared to TCE?
RQ4 What types of equivalent mutants do EMS and TCE find?

RQ5 How efficient is EMS compared to TCE?

We implement EMS on top of the Major mutation framework (Section 5) and answer RQ3–5 by evaluating EMS and TCE on 1,193,633 mutants from 19 Java projects (Section 6).

This paper's key results are:

RA1 We estimate that the median per-project equivalent mutant rate is 2.97% across mutants generated by Major.

RA2 We found that many equivalent mutants are simple: only 30% of mutants require reasoning about state propagation to determine equivalence and only 24% require reasoning about the heap (e.g., alias analysis). These simple equivalent mutants can be detected with efficient static analyses.

RA3 EMS detects about 29% of equivalent mutants, compared to 3.3% and 5.1% for TCE$_{javac}$ and TCE$_{soot}$, respectively. EMS' effectiveness is comparable across all projects.

RA4 EMS' efficacy stems primarily from its ability to reason about value ranges, common patterns in `equals` methods, and Java Standard Library API contracts. The majority of equivalent mutants missed by EMS but detected by TCE$_{soot}$ were mutations of values which are never read.

RA5 EMS detects an equivalent mutant every 0.52 seconds on average, whereas TCE$_{soot}$ detects an equivalent mutant every 4,980 seconds on average.

2 Background and Related Work

Both mutation analysis and mutation testing begin by generating *mutants*, which are syntactic variations of the original program. Usually, these variations are small, such as changing a single `+` operator to a `-` operator. Given a set of tests, a mutant is classified as *killed* if a test passes on the original program but fails on the mutant; otherwise, the mutant is classified as *live*.

Mutation analysis evaluates a test suite according to its ability to kill mutants. The result of mutation analysis is a test adequacy measure. Empirical evidence shows that, although mutants are syntactically simpler than real faults [8], mutant detection is positively correlated with real fault detection [2, 5, 13].

Mutation testing is a testing workflow that presents live mutants to a developer as test goals [1, 16]. Each live mutant represents a potential weakness in the test suite, and presenting them to developers can elicit stronger tests than coverage-guided or ad hoc testing alone. Mutation testing sees increasing adoption in industry, and recent research has shown that mutation testing leads to developers writing more and stronger tests [4, 28, 30].

2.1 Mutant Generation

Traditional mutant generators apply a set of pre-defined *mutation operators*, which are syntactic transformation rules, to the original program to generate a set of mutants. For Java programs, these mutation operators are commonly applied to a program's AST or to its compiled bytecode [1, 6, 15, 22].

More recently, researchers have started using machine-learning-based mutation, with two distinct approaches: (1) learning mutation operators offline (e.g., to mimic real-world faults) and then applying those operators to generate mutants [4, 17], and (2) mutating programs directly by passing them to a (language) model [7, 34].

2.2 Equivalent Mutants

For a mutant to be killable, a test's execution must *reach* the mutated code, *infect* (change) the program state, and *propagate* the infected state to an output observable by the test [11, 36]. An *equivalent mutant* is a mutant which cannot be killed by any test: it is infeasible to create a test that satisfies all three conditions.

Equivalent mutants are problematic for both mutation analysis and mutation testing. In mutation analysis, they waste computational resources (running tests in an attempt to kill equivalent mutants is futile) and skew the mutant detection rate. In mutation testing, they wastes developer time by presenting them with unsatisfiable test goals. Prior work suggests that non-trivial equivalent mutants can lead developers to refactor or otherwise improve their code [30, 31], but many equivalent mutants are trivial.

2.3 Impact of Equivalent Mutants on Testing

Prior work estimates that the time to analyze a single equivalent mutant (and to identify it as such) ranges from 5 to 15 minutes [9, 31, 32]. Additionally, developers have a low tolerance for false positives [10, 28, 33]; presenting an equivalent mutant as a test goal is effectively a false positive, and thus may lead to a lack of adoption of mutation testing in practice. Finally, even though equivalent mutants typically make up a small proportion of all generated mutants, they make up an outsized proportion of the mutants presented to developers. Even a weak initial test suite can kill a large number of generated mutants[1]. For example, if 5% of generated mutants are equivalent and a test suite kills 80% of generated mutants, 25% of the remaining mutants are equivalent.

Prior work also investigated to what extent traditional mutation operators generate equivalent vs. desirable mutants [14, 37]. For example, Yao et al. report on a manual analysis of 1,230 equivalent and stubborn mutants. Stubborn mutants are non-equivalent mutants that remain undetected by a test suite, and are desirable since they indicate shortcomings in the existing test suite. Yao et al.'s results suggest that for some operators generating stubborn mutants is correlated with generating equivalent mutants. The study provides valuable insights into the prevalence and distribution of equivalent mutants across different classes of mutation operators, and it demonstrates a fundamental trade-off (see Section 7.3).

Our work extends Yao et al.'s study by investigating the interactions between specific mutation operators and the context in which they are applied that lead to equivalent mutants.

2.4 Equivalent Mutant Detection Techniques

To mitigate the problems caused by equivalent mutants, prior work has proposed techniques for detecting equivalent mutants [3, 18, 21, 23, 25, 27]. Baldwin [3] proposed six different compiler optimizations as heuristics to detect equivalent mutants but did not implement a detection system. Offutt and Craft [25] built *the Equalizer*, an equivalent mutant detection system for Fortan 77, using these six compiler optimizations, and reported a 10% *equivalent mutant detection rate (EMDR)*. Papadakis et al. [27] introduced TCE, using an optimizing compiler (GCC) to detect equivalent mutants in C programs (EMDR=7.4%). Kintis et al. [18] extended TCE to

[1] Kurtz et al. [19] showed that simulated test completeness of as low as 20% can result in a mutation score of 80% or higher

Table 1: Subject programs and number of generated mutants.

Subject	Loc	Retained		Total		Generated code		Uncomp.
		Classes	Mutants	Classes	Mutants	Classes	Mutants	Mutants
Ant	104,270	685	102,478	685	104,670	0	0	2,192
Bcel	30,258	327	30,221	327	30,694	0	0	473
Chart	96,382	528	120,273	528	121,394	0	0	1,121
Cli	7,070	20	2,874	20	2,888	0	0	14
Codec	7,953	54	24,669	54	24,809	0	0	140
Collections	28,543	263	22,999	263	23,259	0	0	260
Compress	41,838	175	42,570	175	43,178	0	0	608
Csv	1,635	9	2,133	9	2,163	0	0	30
Gson	7,886	41	8,893	42	9,515	0	0	622
H2	140,255	612	182,074	612	184,260	0	0	2,186
JacksonCore	26,004	88	48,808	88	49,141	0	0	333
JacksonDatabind	61,107	356	49,104	356	49,577	0	0	473
JacksonXml	4,927	29	4,198	29	4,247	0	0	49
Jsoup	12,008	57	15,125	57	15,277	0	0	152
JxPath	18,764	136	14,328	143	25,077	7	10,528	221
Lang	21,788	97	38,855	97	39,492	0	0	637
Math	84,324	581	200,147	581	204,506	0	0	4,359
Time	27,801	131	34,935	131	35,255	0	0	320
Tomcat	244,582	1,297	248,949	1,354	268,613	57	16,984	2,680
Total	967,395	5,486	1,193,633	5,551	1,238,015	64	27,512	16,870

Java by using the non-optimizing javac compiler along with the Soot framework [35] to act as an optimization pass (EMDR=5.7%).

The difference in EMDR between the Equalizer and TCE is likely due to differences in both language and mutation operator set. For instance, the Equalizer evaluation used the ABS mutation operator, which replaces an expression e with absolute value $|e|$ and negative absolute value $-|e|$. The ABS operator produces many equivalent mutants (e.g., when an expression is always positive valued), and modern mutation systems do not implement this operator [6, 15, 29]. Similarly, the AOIS operator used in [18] produces a large number of equivalent mutants; we discuss this in depth in Section 7.

TCE represents the state of the art for general equivalent mutant detection in Java programs; other techniques (e.g., solver-based approaches [21, 23]) do not scale to full programs or support only a subset of the Java programming language. TCE leverages the fact that two programs with identical bytecode are equivalent: it compares the compiled bytecode of mutants to that of the original program. TCE is appealing due to its simplicity, requiring only a compiler and `diff`, but its effectiveness depends on the degree to which the compiler canonicalizes the program during optimization.

3 Data Set

We answer our research questions with a data set of 1,193,633 mutants, generated from 19 open-source Java projects. We refer to this data set as the *Full Data Set.*

3.1 Subject Selection

Table 1 summarizes the 19 open-source Java projects (15 from the Defects4J benchmark [12] (v2.0.1) and 4 independent projects) that we selected for analysis based on the following criteria:

(1) **Diversity:** We selected projects with different characteristics such as project domain, project size, and developer community, to improve generalizability of our results.
(2) **Comparability:** We included all projects used by Kintis et al. [18] to enable comparability with prior work.
(3) **Ease of building:** We selected projects that allow us to automatically build, mutate, and analyze them.

We chose to use Defects4J because it is a widely used benchmark, it simplifies reproducibility of our results, and it enables project mutation using the Major mutation framework through a unified interface. Defects4j provides 17 open source Java projects. (Major did not generate mutants for 2/17 projects, which we discarded.) For each of the remaining 15 Defects4J projects, we mutated and analyzed the most recent version available in Defects4J.

Additionally, we chose 4 standalone projects: Apache Tomcat, a webserver application; Apache Ant, a build system for Java; H2, an SQL database engine; and Apache Bcel, a bytecode engineering library. These projects increase diversity and ensure that our data set properly contains the projects used by Kintis et al. Note that the exact project versions and the set of generated mutants differ due to our use of a more recent Java version (Java 8) and the Major mutation framework (Kintis et al. used muJava).

3.2 Mutant Generation

TCE requires source code mutants (as opposed to mutants directly embedded into bytecode). Based on a recent survey of mutation testing tools for Java [1], we chose Major because of its ability to generate source-code mutants and its recency (muJava was not applicable to our data set). We used the most recent version (v2.0.0) of the Major mutation framework [15] for mutant generation and enabled all available mutation operators.

We excluded 64 source files that were automatically generated (e.g., by a parser generator), as well as any mutants derived from those source files. The types of equivalent mutants in generated code may not be representative of those found in developer-written code, and generated code is usually not tested/mutated directly in practice. This excluded about 2.22% of all generated mutants. (Note that when considering mutants in generated code, our conclusions about tool efficacy still hold.)

Major produced between 2,163 and 268,613 mutants per project, for a total of 1,238,015 mutants. Out of all mutants in non-generated code, 1.36% did not compile—these are also excluded from the total in Table 1. One class in Gson did not have any compilable mutants, and is thus excluded from the count of retained classes. Overall, our Full Data Set contains 1,193,633 retained mutants.

4 Equivalent Mutants in the Wild

This section answers RQ1 and RQ2, reporting on our systematic ground-truth analysis that estimates the *equivalent mutant rate (EMR)* for Java programs and characterizes the found equivalent mutants. Specifically, we selected 7 of the 19 projects in our data set for this ground-truth analysis based on the following criteria:

- **Representative sample:** The sampled projects should contribute about 33% of all generated mutants.
- **Diversity of projects:** The sampled projects should exhibit diversity across domain, coverage/kill rates, and size.
- **Ease of Testing:** The sampled projects' testing infrastructure should allow for automated mutation analysis.

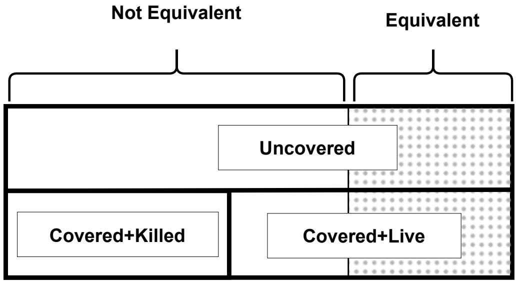

Figure 1: Stratified sampling for estimating EMR. The shaded region represents the unknown set of equivalent mutants. The equivalent mutant rate (EMR) of each strata is the percentage of its shaded area (0% for killed mutants). Stratification reduces sample variance, and thus the sample size required to estimate the overall EMR.

Table 2: Estimated percentage of equivalent mutants (EMR %) and estimated number of equivalent mutants (Eq.) per subject. Mutation analysis and Sampled summarize the data that underlies these estimations.

Subject	Mutants	Estimation		Mutation analysis			Sampled					
		EMR %	Eq.	Killed	Live		All		Cov.		Uncov.	
					Cov.	Uncov.	N	Eq.	N	Eq.	N	Eq.
Chart	120,273	2.97	3,571	33,738	30,860	55,675	429	20	186	16	243	4
Collect.	22,999	1.84	423	14,775	4,453	3,771	360	20	216	18	144	2
Csv	2,133	2.54	54	1,613	347	173	292	31	207	25	85	6
Gson	8,893	5.24	466	6,141	1,764	988	343	70	285	67	58	3
JxPath	14,328	3.07	440	8,981	2,969	2,378	242	21	146	18	96	3
Lang	38,855	3.69	1,435	31,059	6,797	999	251	44	203	41	48	3
Math	200,147	2.97	5,946	153,023	32,403	14,721	75	9	46	7	29	2

4.1 RQ1: How common are equivalent mutants in the wild?

We manually determined the equivalence for 1,992 mutants, randomly sampled based on an *optimal sampling allocation*, and used this data to estimate the EMR for each project.

4.1.1 Methodology. Our goal is to estimate the EMR for each of our 7 sampled projects with 95% confidence and a 2.5% margin of error per project. To estimate EMR, we sample a sufficient number of mutants from each project and manually inspect them.

Mutant sampling To meet our goals for statistical confidence and margin of error, we minimized the number of manual inspections required via stratified sampling and using information from automated mutation analyses. This section lays out the general sampling procedure and reports on the sample composition. Our supplementary material[2] provides an extended statistical analysis of the sampling procedure as well as scripts for reproducibility.

Reducing the sample variance allows us to reach a given confidence interval with fewer samples. We used data collected from running the projects' test suites on the generated mutants to reduce sample variance. This allowed us to divide the mutant population into three strata, as shown in Figure 1: **Uncovered**—mutants in code not covered by any tests; **Covered+Killed**—mutants in covered code that are killed; **Covered+Live**—mutants in covered code that are live. We assume that **Uncovered** and **Covered** have a similar EMR, which we empirically validated. Killing non-equivalent mutants with tests grows the **Covered+Killed** strata, whose EMR is known to be 0%. Conversely, it shrinks the **Covered+Live** strata, thereby increasing its EMR (equivalent mutants cannot be killed, and hence remain in that strata). Since the three strata have different EMRs, sampling uniformly at random is suboptimal.

Mutant equivalence To manually inspect each sampled mutant for equivalence, we assigned each project to one of four coders (some coders were assigned more than one project). For each sampled mutant, a coder determined whether it was equivalent. Since there is some ambiguity around what constitutes an equivalent mutant, we needed to establish a precise definition. For instance, is a mutant equivalent if it produces the same value but takes 10x time to run? Is a mutant equivalent if reflection is required to detect it (e.g., directly accessing a private method)? As an extreme example, a test could inspect every runtime state in the underlying JVM and detect any state infections. In practice, a test should assert on a program's *behavior*, not its syntactic representation on disk or transient states in the JVM, and the definition of an equivalent mutant (infeasible test goal) should be aligned with this expectation. We chose the following definition:

A mutant is equivalent if no test can be written that distinguishes the mutant from the original program by only invoking and inspecting data obtained from the public and package-private API of the program.

In particular, this disallows reflection and timing-based tests, though it does allow access to protected fields through subclassing.

Coders inspected the sampled mutants for equivalence, and labeled each based on the above definition. We allowed for uncertainty in the labeling process: programs are complex, and some mutants required reasoning over complex control flow, class invariants, and sophisticated mathematical properties with little to no documentation. To handle this uncertainty, coders marked mutants that they were not certain about, and resolved ambiguities through discussion. About 2% of all sampled mutants required additional discussion. Overall, the labeling process took over 160 hours, with an average of 5 minutes per mutant, ranging from less than a minute (e.g., very similar mutants) to over one hour for a single mutant. The time-consuming nature of the labeling task reinforces the need for optimal sampling allocation.

4.1.2 Results. Overall, 215 out of 1,992 mutants are labeled as equivalent in the ground-truth dataset. Table 2 shows the EMR estimates for each project as well as the underlying sample composition (Sampled). Collections has the lowest EMR at 1.84% while Gson has the highest EMR at 5.24%. The median EMR is 2.97%. Section 6 uses the ground-truth dataset and the EMR estimates to contextualize the efficacy of the evaluated equivalent-mutant detection approaches. Section 7 discusses the reasons for the relatively low observed EMR, compared to prior work, including the impact of mutation operators and redundant mutants.

[2] https://www.doi.org/10.6084/m9.figshare.26948143.v1

4.2 RQ2: What types of equivalent mutants exist in the wild?

Having established a ground-truth dataset of 215 equivalent mutants, we wished to understand why these mutants are equivalent and how a detection approach could determine equivalence.

4.2.1 Methodology. Coders tagged each equivalent mutant according to two criteria, **RIP** and **ASH**.

The RIP Criterion. The RIP criterion describes *why* a mutant is equivalent. For a mutant to be non-equivalent, the following three properties must hold:

- **R**: The mutated code is *reachable*.
- **I**: Executing mutated code *infects* execution state.
- **P**: The infected state *propagates* to an observable output.

Any equivalent mutant violates at least one of these, and we labeled them according to the weakest property violated (R < I < P).

The ASH Criterion. The ASH criterion describes *how* a tool could detect a mutant's equivalence, broken down by the degree of required reasoning complexity.

- **A**: A tool would need to reason over information available in an attributed *Abstract Syntax Tree*.
- **S**: A tool would need to reason about dataflow across *Local State* (e.g., intra-procedural dataflow analysis).
- **H**: A tool would need to reason about the *Heap* (e.g., perform an alias analysis, reason about class invariants, etc.).

The delineation between ASH, and between S and H in particular, is not always clear. For instance, a (boxed) Integer is technically a heap object, but it is immutable and reasoning about such objects is very similar to reasoning about primitives; `ArrayList.size()` has an (implicit) contract that the result is non-negative; a static getter method may return a literal or final value. While the above examples technically involve a heapy operation, their semantics can be simplified via method summaries or inlining. When differentiating between **S** and **H**, we assumed that a technique belonging to **S** has access to method summaries and inlining. This separates the complexity of reasoning about a program that is factored over several methods (but can be simplified), and the complexity of reasoning about arbitrary class invariants.

When labeling equivalent mutants according to the ASH reasoning power required to detect the equivalence, we assume that all levels of reasoning have access to the following information:

(a) **Constant Folding and Propagation:** Statically known (compile-time) constant value for a variable (e.g., `int seconds = 60 * 60;`). Inlining of constants (e.g., the constant value of `final int ONE = 1;` is known for all uses of `ONE`).

(b) **Method Contracts:** We assume that implicit and explicit method contracts provided by the standard library (e.g., `Collection.size() >= 0` ↦ `true`) are available to all forms of reasoning.

This means that a mutant is labeled A if no other information is required to determine equivalence.

Patterns of Equivalent Mutants. While manually inspecting for mutant equivalence and labeling according to the **RIP-ASH** criteria, coders noted common patterns of equivalent mutants. For instance, coders noticed that many equivalent mutants were formed when a variable declaration's initialization value was mutated but never read before the variable was reassigned:

```
int foo = 1;
if (c) {foo = 2;}
else {foo = 3;}
```

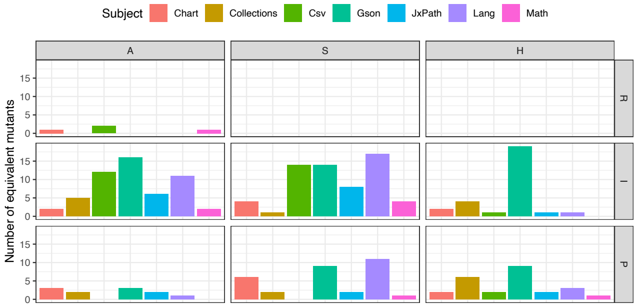

Figure 2: Number of equivalent mutants found in the ground-truth analysis, divided by the RIP-ASH criteria.

We used the labeled data to determine the distribution of equivalent mutants according to RIP-ASH and compile concrete examples.

4.2.2 Results. Figure 2 summarizes the labeling results. The remainder of this section details each of the types of equivalent mutants that we identified during our analysis.

- ***RA** mutants are unreachable and can be detected by inspecting the AST.* For example, `DEBUG` is `false` and the then branch is unreachable.

```
  final boolean DEBUG = false;
  if (DEBUG) {
-   double hGap = 1.0;
+   double hGap = 0.0;
    ...
  }
```

- ***IA** mutants fail to infect state and can be detected by inspecting the AST.* The following mutant replaces a final variable with its constant value. Executing the mutant fails to infect state, making it equivalent. We can discover this by inspecting the value associated with `PEEKED` in the AST.

```
  final int PEEKED = 0;
  ...
- int peeked = PEEKED;
+ int peeked = 0;
```

- ***PA** mutants fail to propagate infected state and can be detected by inspecting the AST.* The following mutant of the `.equals()` doesn't return `true` when `this == other`, and the method continues executing. When `this == other` then the rest of the `.equals()` method should return true, and the infected state will not propagate. We discover this by inspecting the AST: we need to know that the name of the method is `.equals`, that it takes parameter `other`, that there is a reference equality check between `this` and `other` that returns `true` on success[3].

[3] Proving that removing the reference equality check is sound would involve more sophisticated analyses. However, this pattern occurs frequently, is based on an implicit contract of the `.equals` method, and is unlikely to be a source of false positives. This is the only source of potential unsoundness in our labeling scheme (see Section 7).

```
  public boolean equals(Object other) {
    if (this == other) {
-     return true;
+     ;
```

- ***IS** mutants fail to infect state and can be detected with a local dataflow analysis.* The following mutant replaces `!=` with `<`. These operators are equivalent so long as the LHS is less than or equal to the RHS. `strLen` is non-negative (since it is a length), and when `start` is initialized to `0` it is less than or equal to `strLen`. We can perform a dataflow analysis of the loop to discover that `start` is only incremented by 1 and that `strLen` is never altered updated. From this we can conclude that the loop will terminate before the invariant `start <= strLen` fails to hold, and the mutant is equivalent.

```
  int strLen = str.length();
  int start = 0;
- while (start != strLen && ...) {
+ while (start < strLen && ...) {
      start++;
  }
```

- ***PS** mutants fail to propagate infected state and can be detected with a local dataflow analysis.* The following code initializes `code` to a dummy value, but overwrites the dummy value before it is ever used. The mutant infects state but never propagates. We can learn this by performing a def-use analysis of the `code` variable.

```
- int code = 0;
+ int code = 1;
  code = hash();
```

- ***IH** mutants fail to infect state and heap/alias analysis is required to detect it.* The following mutant is equivalent due to a class invariant in which the value `this.lexer` is always non-`null`. To discover this we would need to reason about complex class invariants involving heap state.

```
- if (this.lexer != null) {
+ if (true) {
```

- ***PH** mutants fail to propagate infected state and require a heap/alias analysis to be detected.* This mutant alters execution path (infects) when `len == b.length()`. When `len == b.length()`, the original code calls `b.setLength(len)`, which will not alter state, and the infection never propagates. To discover this we need to reason about heap state to discover that `b.setLength(len)` is a no-op.

```
  void trimTrailingSpaces(final StringBuilder b) {
    int len = b.length();
    while (len > 0 && ..) {
        len = len - 1;
    }
-   if (len != b.length()) {
+   if (len <= b.length()) {
        b.setLength(len);
    }
  }
```

Our findings by the RIP criteria We found that very few (2%) mutants were equivalent due to being unreachable (**R**). The most (67%) equivalent mutants were equivalent due to a failure to infect execution state (**I**). The second most common (31%) reason for equivalence was a failure to propagate infected execution state to an observable output (**P**).

Our findings by the ASH criteria The **A** and **S** tags were most common in our sample, making up 32% and 43% of all tagged mutants. The **H** tag was the least common, making up only 25% of all tagged mutants. Many of these came from the Gson project due to a large number of class invariants.

These are promising findings: these numbers suggest that many equivalent mutants can be detected with efficient analyses.

5 EMS: Equivalent Mutant Suppression

Our manual analysis revealed that many equivalent mutants can be detected with a small set of analyses not much more complex than structural pattern matching over an AST, attributed with types and constant values. For example, many implementations of the `equals` method begin with a reference-equality check:

```
boolean equals(Object other) {
  if (this == other) { return true; }
  // ...
}
```

Matching the above AST, assuming no logic errors in the remainder of the method implementation, is sufficient to determine that the following two mutants are equivalent to the original program:

```
M1: if (false) { return true; }
M2: if (this == other) { /* deleted */; }
```

Listing 1: Equivalent mutants, disabling a reference-equality optimization.

We augmented the Major mutation framework with a set of 10 lightweight suppression rules to investigate whether they could efficiently detect a substantial fraction of observed equivalent mutants. We developed these rules by generalizing from observed equivalent mutants to mutants which are equivalent for the same underlying reason. These rules were designed after manual analysis of only 7 of the 19 projects but are effective for all 19 projects, offering evidence that our rule design does not overfit to the smaller corpus: Section 6 shows that these rules suppress a large fraction of equivalent mutants across projects.

We group our suppression rules according to **RIP-ASH**.

RA

(1) **Unreachable.** EMS suppresses three types of unreachable code: branches that are *conditionally compiled* (e.g., `if (false) {. . . }`), `default` cases of switch statements on `enum` values where all possible values are checked (and no fall-throughs to the `default` case are possible), and do-while loops where the condition is never reached because the body always returns or breaks.

IA

(2) **Standard Library Contracts.** EMS suppresses comparison operand mutations between `>` and `!=` and between `==` and `<=` when the mutated expression is a comparison between 0 an array's `length` field or the return value a standard Java collection's `length()` or `size()` method. Similarly, EMS suppresses mutations for comparisons of −1 with a `String`'s `indexOf` or `lastIndexOf` member.

(3) **Useless Control Flow.** Mutants that remove superfluous control-flow statements, including `return;` as the last statement in a void method, `break;` as the last statement in a `switch`, and `continue;` as the last statement in a loop body.

(4) **Mutate Constant Value.** EMS suppresses mutations to constant-valued expressions which yield the same constant value. Examples include mutating an arithmetic subexpression that is multiplied by `0` and replacing a final field with its statically known value (e.g., mutating `ZERO` to `0`). EMS ensures no side-effect-changing mutants are suppressed (e.g., mutating `0*sideEffect()` to `0*1` will not be suppressed).

(5) **Algebraic Identity.** EMS suppresses mutants that are equivalent due to an algebraic identity. For example, mutating `x*1` to `x/1`, `x+0` to `x-0`, `32>>x` to `32>>>x`, or `x != Integer.MIN_VALUE` to `x > Integer.MIN_VALUE` are all suppressed. Identities are identified by matching types and literals only; this suppression rule does not otherwise reason about values.

IS

(6) **Mutually Exclusive.** Expressions `a || b` and `a != b` operators are equivalent whenever both `a` and `b` cannot both be `true`. Likewise, `a && b` and `a == b` are equivalent when both `a` and `b` cannot both be `false`. EMS suppresses mutations between `||` and `!=` (resp. `&&` and `==`) when it can determine that the operands cannot both be `true` (resp. `false`) The simplest form of this suppression rule is the comparison of single variable to constant values: `a == 1 || a == 2`: both comparisons cannot be true, so EMS suppresses mutating `||` to `!=`. A more complicated example is checking if `char c` is a valid digit: `c < '0' || c > '9'`. EMS uses a range analysis to infer that `c` cannot both be less than `'0'` and greater than `'9'` and suppresses the mutation of `||` to `!=`.

(7) **Loop-Bounds Check.** Mutating the termination condition from `i <` X to `i !=` X, in a for loop of the form `for (int i = 0; i <` X`; i++)`, yields an equivalent mutant if `i` is always less than nor equal to X. EMS suppresses such mutations when it can soundly determine that `i` is guaranteed to be less than or equal to X: `i` is initialized to 0 or a negative value, `i` is monotonically increasing by 1, X is a constant or of the form `x.{size()|length}`, and neither `i` nor X are updated in the loop body.

PA

(8) **equals() Reference Equality.** EMS suppresses mutants that disable an optimization in `equals` (Listing 1).

(9) **Empty Branches.** EMS suppresses mutants to `if` conditions when both branches are empty, so long as EMS can prove that the condition is side-effect free.

PS

(10) **Useless Inititalization.** EMS suppresses the mutation of *unread initializations* of local variables, which is defined to be an initialization of a variable to a value that is never read before being reassigned:

```
// foo's init value 0 is never read
int foo = 0;
if (c) { foo = 1; }
else {foo = 2;}
```

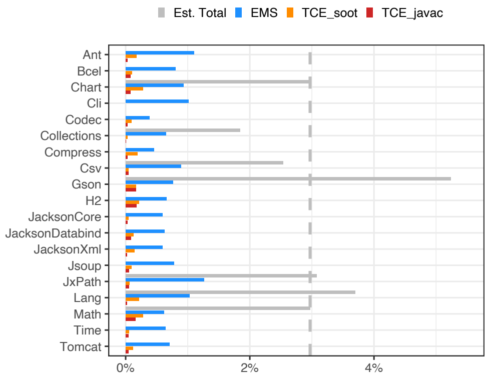

Figure 3: Ratio of equivalent mutants found by each tool, compared to the estimated equivalent mutant ratio.

This is common in practice, and often related to style choices. EMS builds on top of Javac's definite assignment analysis[4]. If a local variable is definitely assigned after deleting its initializer, EMS suppresses mutations to that initializer.

6 Evaluation

To evaluate EMS, we consider how many and which types of equivalent mutants it detects compared to TCE, the current state of the art. We also contrast EMS' efficiency with that of TCE.

For our baseline, we use two variants of TCE: TCE$_{javac}$ and TCE$_{soot}$. The original TCE implementation[5] was written in Jython 2.x, built to target the output structure generated by muJava[6], and not easy to parallelize/decouple. Based on the specification in the original paper and the available code, we reimplemented both TCE variants in Python 3 and Bash.

The mutant compilation phase, which is identical for both variants, runs javac on the individual source code mutants. For efficiency, this step compiles only the mutated Java file. This leads to one or more compiled classfiles (inner and anonymous classes are compiled to their own class files).

TCE$_{soot}$ then runs Soot on each classfile, acting as the optimization phase of compilation, in order to discover equivalences between programs that normal javac compilation would not discover.

Finally, the equivalence analysis compares mutated bytecode to the original bytecode for equality.

6.1 RQ3: How effective is EMS compared to TCE?

6.1.1 Methodology. To answer RQ3, we compare EMS to the two TCE variants in terms of number of equivalent mutants detected.

Having established ground truth for 1,992 equivalent mutants, we computed the recall for all tools (ratio of detected equivalent mutants over the total number of equivalent mutants) for that

[4] https://docs.oracle.com/javase/specs/jls/se8/html/jls-16.html
[5] The published link (https://bitbucket.org/marinosk/ted) is broken, but we were able to locate TCE on GitHub: https://github.com/kintism/ted.
[6] https://cs.gmu.edu/~offutt/mujava/

Table 3: Equivalent mutants found on ground-truth data set.

Subject	Mutants	TCE_{javac}		TCE_{soot}		EMS	
		N	%	N	%	N	%
Chart	20	1	5.00	2	10.00	6	30.00
Collections	20	0	0.00	1	5.00	7	35.00
Csv	31	1	3.23	1	3.23	13	41.94
Gson	70	2	2.86	2	2.86	12	17.14
JxPath	21	1	4.76	2	9.52	11	52.38
Lang	44	0	0.00	1	2.27	11	25.00
Math	9	2	22.22	2	22.22	3	33.33
Total	215	7	3.26	11	5.12	63	29.30

Ground Truth Data Set. We quantify each tool's effectiveness as the number and ratio of equivalent mutants found by that tool.

Additionally, we determined the total number of equivalent mutants found by each tool for the Full Data Set. We use the estimated equivalent mutant rates from the Ground Truth Data Set as a reference point to estimate the tools' recall on the full data set.

6.1.2 Results. Table 3 gives the results for recall on the Ground Truth Data Set. Overall, EMS detects about 29% of the equivalent mutants, compared to 3.3% and 5.1% for TCE_{javac} and TCE_{soot}, respectively. Table 4 and Figure 3 quantify each tool's effectiveness on the Full Data Set. EMS detected 8,776 equivalent mutants, compared to 1,007 and 2,124 for the two TCE variants. Overall, EMS' effectiveness is comparable across all 19 subjects, and it consistently outperforms TCE by a wide margin. EMS detected every equivalent mutant also detected by TCE_{javac}, but it misses 271 equivalent mutants detected by TCE_{soot}.

Soot Nondeterminism While testing our implementation we found that TCE_{soot} is nondeterministic: it detected slightly different sets of equivalent mutants across different runs. This is due to nondeterminism in Soot: the same input classfile can result in different output classfiles. Since TCE_{soot} compares the bytecode generated by Soot for equality, this can produce a false negative.

We performed two analyses to determine the impact of this nondeterminism on our measurements of TCE_{soot}'s effectiveness. (1) We modeled a TCE_{soot} run on an equivalent mutant as a Bernoulli RV and showed that in the worst case, we can be 95% confident that the number of reported equivalent mutants of a single campaign of TCE_{soot} across all mutants would be within 2.2% of the expected number of reported equivalent mutants. (2) We compared the number of equivalent mutants detected in the reported run of TCE_{soot} with the number detected in a previous run. The two runs reported a different set of equivalent mutants, but the *number* of reported equivalent mutants between runs differed only by 1. This suggests that Soot's nondeterminism should lead to a negligible variation in TCE_{soot}'s equivalent mutant detection rate given our sample size.

6.2 RQ4: What types of equivalent mutants do EMS and TCE find?

6.2.1 Methodology. To better understand what types of equivalent mutants EMS and the TCE baselines detect, we manually analyzed all of the 2,124 equivalent mutants detected by TCE_{soot} on the Full Data Set, which includes all 1,007 mutants detected by TCE_{javac}. We

Table 4: Number of equivalent mutants found per subject.

Subject	Mutants	Equivalent mutants found		
		EMS	TCE_{javac}	TCE_{soot}
Ant	102,478	1,127	25	176
Bcel	30,221	242	22	30
Chart	120,273	1,117	91	339
Cli	2,874	29	0	0
Codec	24,669	94	7	22
Collections	22,999	150	1	7
Compress	42,570	194	12	81
Csv	2,133	19	1	1
Gson	8,893	68	15	15
H2	182,074	1,203	321	389
JacksonCore	48,808	290	15	23
JacksonDatabind	49,104	306	41	62
JacksonXml	4,198	25	1	6
Jsoup	15,125	118	8	14
JxPath	14,328	181	7	9
Lang	38,855	399	9	84
Math	200,147	1,230	311	557
Time	34,935	225	15	19
Tomcat	248,949	1,759	105	290
Total	1,193,633	8,776	1,007	2,124

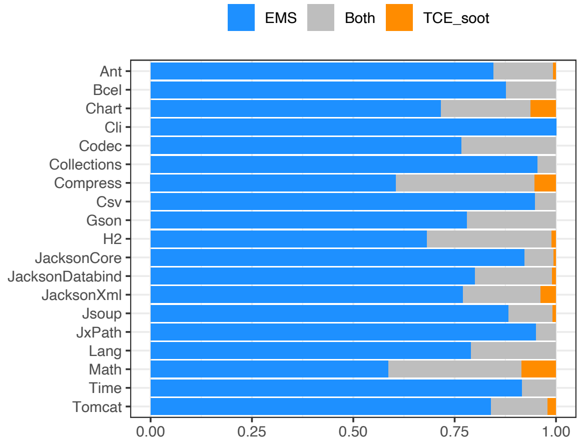

Figure 4: Proportion of equivalent mutants found per tool.

Table 5: Number of equivalent mutants suppressed per rule.

Rule	n
Standard Library Contracts	1,796
Loop-Bounds Check	1,562
Mutually Exclusive	1,488
equals() Reference Equality	1,078
Useless Inititalization	863
Mutate Constant Value	487
Algebraic Identity	484
Unreachable Code	460
Useless Control Flow	364
Empty Branches	194
Total	8,776

Table 6: Tool efficiency results for EMS and both TCE variants, including total runtime across all analyzed mutants, and the time per analyzed mutant (sec/Mut) and time per equivalent mutant detected (sec/EquiMut).

Variant	Mutants		Runtime		
	Analyzed	FoundEqui	Total	sec/Mut	sec/EquiMut
EMS	1,193,633	8,776	1.26h	0.004	0.52
TCE$_{javac}$	1,193,633	1,007	1,549h	4.67	5,538
TCE$_{soot}$	1,193,633	2,124	2,938h	8.86	4,980

additionally mapped EMS suppression rules to types of equivalent mutants suppressed by these rules as well as their RIP-ASH labels, and automatically labeled all 8,776 suppressed mutants.

6.2.2 Results. Table 5 shows the number of equivalent mutant suppressions each rule in EMS is responsible for. Most notably, EMS' efficacy stems from its range analysis and reasoning about common patterns in `equals` methods as well as API contracts defined by the Java Standard Library.

In total, EMS missed 271 equivalent mutants detected by TCE$_{soot}$. These can be broken down into three categories: mutants that *mutate values that are never read* (222), mutants that *mutate constant values to the same value* (40), and mutants that *mutate control flow* in a way that does not change program semantics (9).

EMS does suppress equivalent mutants of each of these forms, but it uses conservative rules, and some of these mutants take forms that are not detected by these conservative rules. For instance, EMS suppresses unread initialization values of variables, but detecting more mutations to values that are written to local variables but never accessed require a more sophisticated analysis.

6.3 RQ5: How efficient is EMS compared to TCE?

6.3.1 Methodology. To answer RQ5, we measure the efficiency of EMS and the two TCE variants, considering two runtimes for the Full Data Set. First, we consider the time it takes to generate the mutants, compared to the default version of Major. For TCE, running Major incurs an overhead because it must export individual source-code mutants. For EMS, running Major incurs an overhead because of enabled mutant suppression. Second, we consider, for TCE, the time it takes to compile and diff the individual source-code mutants. Since TCE$_{soot}$ is an extension of TCE$_{javac}$, we determined its total runtime by adding the additional runtime incurred by running Soot to the runtime of TCE$_{javac}$.

We ran both TCE variants and EMS on a Linux server with 104 CPUs and 1TB of RAM, and timed each process with the `time` utility. We used a ramdisk to store all generated mutants and compiled classfiles to avoid I/O overhead. We restricted parallelization of our analysis to 8 jobs at a time and monitored server workload to ensure sufficient resources are available. Since multithreading makes timing analyses difficult, we report user time plus system time for each process, as this is the total time spent in the process, regardless of the number of threads.

6.3.2 Results. Generating mutants with suppression enabled increases runtime from 1,172 to 1,369 seconds, incurring a 17% time increase. Even with this increase, EMS's mutant generation is almost twice as fast as that of TCE (2,369 secsonds).

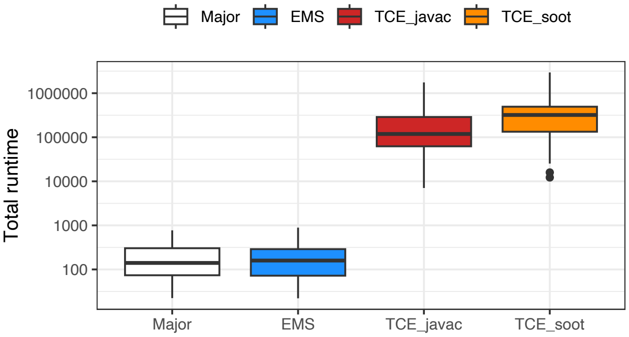

Figure 5: Distribution of total runtime for all subjects.

TCE's true overhead comes from individually compiling each generated mutant. To put this overhead in perspective, we quantify *efficiency* in three ways: (1) total runtime, (2) runtime per mutant, and (3) runtime per equivalent mutant detected.

Table 6 reports the efficiency results. EMS' total runtime was 1.26 hours, compared to 1,549 hours for TCE$_{javac}$ and 2,938 hours for TCE$_{soot}$. Additionally, Figure 5 shows the distribution of total runtime per tool and subject.

7 Discussion

7.1 Time per Detected Equivalent Mutant

Having a low *time per detected equivalent mutant* is crucial for a tool to be viable. Kintis et al.'s data implies that TCE$_{soot}$ found an equivalent mutant on average every 25 seconds, while our experiments suggest that TCE$_{soot}$ takes on average over an hour to discover each equivalent mutant. This subsection attempts to resolve the discrepancy by investigating differences in the reported effectiveness and efficiency between Kintis et al.'s and our data.

7.1.1 Discrepancies in Effectiveness. Our experiments showed that TCE$_{javac}$ and TCE$_{soot}$ respectively reported 0.08% and 0.18% of generated mutants as equivalent. These numbers are smaller than those reported in prior work: Kintis et al. [18] reports that TCE variants respectively found 0.2% and 5.7% of generated mutants to be equivalent, and reported that TCE$_{soot}$ detected 54% of equivalent mutants identified during a ground-truth analysis.

This discrepancy is largely due to the presence of the AOIS operator (not used in Major) in Kintis et al.'s data set, which accounts for 3,770 / 3,904 (94.8%) of the equivalent mutants discovered by TCE$_{soot}$. AOIS replaces a variable `a` with one of the following: `++a`, `--a`, `a++`, or `a--`. When this operator is applied to a variable that is never read again (e.g., in `return a++;`), the infected state cannot propagate, making the mutant equivalent. Detecting these equivalent mutants involves reasoning about (simple) dataflow, and TCE$_{soot}$ is particularly good at detecting this type of reasoning.

Table 14 of Kintis reports that the 3,770 equivalent mutants generated by AOIS represent 15% of the mutants total mutants generated by AOIS, and from this we estimate that $25,133 = 3770/0.15$ mutants in total were generated by AOIS, representing approximately 37% of the total generated mutants.

We estimate that removing the AOIS operator from the data set would result in 134 (3,904 - 3,770) equivalent mutants detected, out of 43,050 (68,183 - 25,133) generated mutants. This corresponds to 0.31% of generated mutants being detected equivalent by TCE$_{soot}$, which is much closer to the 0.18% that we found in our analysis.

7.1.2 Discrepancies in Efficiency. In addition to differences in effectiveness, our efficiency numbers do not align with what was reported in [18]. While we were unable to explain this discrepancy, we explicate our reproduction, which we believe accurately represents the efficiency of the baseline tools, and summarize possible explanations in the supplementary material.

7.2 Undetected Equivalent Mutants

Not all equivalent mutants are created equal: TCE$_{soot}$ is able to detect mutants that require reasoning about complex dataflow. During our manual inspection of mutants reported equivalent by TCE$_{soot}$, we found equivalent mutants involving complex dataflow to be some of the most challenging ones. It is entirely possible that these would not have been detected by a manual analysis without a tool reporting them equivalent, and the utility of finding those equivalent mutants is much higher than some simpler equivalent mutants. While TCE$_{soot}$ may take longer per equivalent mutant discovered, it also detects some more difficult-to-find equivalent mutants.

7.3 A Fundamental Trade-off

Mutation tools face a fundamental trade-off between applying *broad* mutations that are easier to detect but less likely to be equivalent and applying *narrow* mutations that are harder to detect but also more likely to be equivalent. Consider the condition `a < b`. Negating this condition with an ROR mutation operator (`a >= b`) results in a broad change to program semantics (no constraints on `a` or `b`): a test covering this condition *always* alters program execution, providing marginal value over code coverage. Conversely, mutating the condition to `a != b` results in a narrow (subtle) difference in program semantics: a test detecting this mutant must execute the condition in a state where $a > b$. While this is a desirable mutant in general, it is equivalent in some contexts such as a for loop condition (e.g., $i > 10$ is impossible in `for(i=0;i<10;++i)`).

Thus, *narrow mutations produce stronger test goals at the cost of more equivalent mutants* while *broad mutations produce fewer equivalent mutants at the cost of weaker test goals.*

This trade-off is backed up by Yao et al.'s study of stubborn and equivalent mutants [37]. For example, they found that the number of equivalent and stubborn mutants produced by the ROR mutation operators were highly correlated.

Many tools apply broader mutations, presumably to avoid equivalent mutants. For instance, PIT does not produce the narrow mutation `a < b` to `a != b`, but rather the broad mutation `a < b` to `a >= b`. However, narrow mutations are desirable in most contexts and not generating them at all results in a weaker mutant set. If we can precisely identify in what contexts they produce equivalent mutants, we can get the benefits of narrow mutations without incurring the cost of equivalent mutants.

Table 7: Generalizability of EMS rules.

Rule	Major	MuJava	PIT		FB	iBiR	μBERT
			all	*def*			
Standard Library Contracts	✓	✓	✓			✓	✓
Loop-Bounds Check	✓	✓	✓			✓	✓
Mutually Exclusive	✓	✓					
equals() Reference Equality	✓	✓	✓		✓	✓	✓*
Useless Inititalization	✓	✓	✓		✓	✓	✓
Mutate Constant Value	✓	✓			✓	✓	✓
Algebraic Identity	✓	✓	✓	✓		✓	✓
Unreachable Code	✓	✓	✓	✓	✓	✓	✓
Useless Control Flow	✓	✓			✓	✓	
Empty Branches	✓	✓	✓	✓	✓	✓	✓

✓*: Not verified due to μBERT crashing on methods containing the `instanceof` keyword.

7.4 Generalizability of EMS

We implemented EMS for the Major mutation system, but investigated whether EMS generalizes to other mutant generators. We considered two ways in which EMS can generalize:

1. *Rule generalizability* (Section 7.4.1): the exact rules identified for Major generalize to other mutant generators.
2. *Approach generalizability* (Section 7.4.2): new rules can detect equivalent mutants produced by mutant generators other than Major (i.e., equivalent mutants produced by other mutant generators also exhibit clear patterns).

To determine rule and approach generalizability of EMS we considered five other mutant generators, across three different types of mutant-generation approaches:

1. *Traditional.* These generators apply traditional mutation operators. We selected PIT [6] and MuJava [22] to represent traditional mutant generation based on the findings of a recent survey [1].
2. *Learned Operators.* These systems apply learned mutation operators. We considered FB (Facebook) [4] and iBiR [17].
3. *LM-Generated Mutants.* These systems use language models to mutate programs directly. We chose μBERT [7] to represent this approach. We also considered DeepMutation [34], but ultimately discarded this tool because we were unable to build and execute it.

For each representative tool, we read its documentation, reasoned about the systems and, when possible, ran it on example code that could lead to equivalent mutants. We used the results of the tools to determine if the system would generate equivalent mutants. If we could not run a tool (e.g., iBiR due to numerous build environment issues and FB due to being proprietary), we resorted to the published operator specifications when reasoning about whether an operator could lead to predictable patterns of equivalent mutants.

7.4.1 Rule Generalizability. Table 7 summarizes our results: a check indicates that a mutant generator produces an equivalent mutant that the corresponding EMS rule would suppress. The table has two columns for PIT and a single column for all other tools. By default PIT uses a restricted set of mutation operators (*def*) as opposed to all supported operators (*all*). As per Section 7.3, this avoids some equivalent mutants at the cost of generating weaker mutants.

All rules generalize to at least one other system. Only the Mutually Exclusive rule did not generalize to at least one non-traditional mutant generator, which is a consequence of these generators not applying certain narrow mutations. For instance, Major mutates `a && b` to `a == b`, which differs from the original expression only when $\neg a \land \neg b$. Other mutant generators apply much broader mutations such as replacing `a && b` with `true` (which differs whenever $\neg a \lor \neg b$). This again demonstrates the trade-off of avoiding some equivalent mutants at the cost of generating weaker mutants.

Useless Control Flow generalizes to all systems that can easily implement this operator; μBERT only replaces a token with another token, and thus cannot remove a `break`, `continue`, or `return` statement; PIT operates on compiled bytecode where high-level control flow operations are implicit.

Other rules generalize to all or nearly all other mutators.

7.4.2 Approach Generalizability. All other mutant generators produce some mutants that Major does not. We inspected each generator's mutation operators to determine if there are common patterns of equivalent mutants that a new EMS rule could easily suppress. The results below suggest that the EMS approach itself generalizes.

Traditional MuJava's AOIS (inserting pre/post-fix increment/decrement operators) and ABS (replacing an expression with its absolute value) operators both produce many equivalent mutants that can be easily suppressed (e.g., do not insert a postfix operator to a variable going out of scope). While PIT targets bytecode, most of its mutation operators are the same as, or emulate, source-level mutation operators. However, PIT also exhibits patterns of additional, suppressible equivalent mutants. For instance, PIT deletes method calls (Void Method Calls operator) by default, which can lead to equivalent mutants when deleting statements that close streams or file handles in contexts where these are closed implicitly.

Learned Mutation Operators Both iBiR and FB can benefit from EMS. iBiR applies a Move Statements operator—moving a statement to a different position. Without additional constraints, this leads to equivalent mutants whenever the statement order does not matter, as is the case for swapped variable declarations or assignments that do not exhibit a control dependency.

FB applies a `REMOVE_NULL_CHECK` operator, which removes null checks of the form `if (x == null) ...` . If this branch throws an exception, the testable outcome will often be equivalent to removing the check and dereferencing a null pointer.

LM-Generated Mutants It is hard to reason about these systems because they use a black-box model to generate mutants, and this may result in unpredictable behavior. However, we identified several patterns of equivalent mutants produced by μBERT after a cursory inspection of its mutant sets for several programs. Here are two concrete examples: (1) Replacing `c.indexOf(x)` with `c.lastIndexOf(x)` results in an equivalent mutant when used in a contains operation (e.g., `if (c.indexOf(x)>=0)`). (2) Replacing `a && b` with `a & b` results in an equivalent mutant when used with non-side-effecting expressions a and b. A trivial extension of EMS' Mutually Exclusive rule would detect these equivalent mutants.

7.5 Threats to Validity

7.5.1 External Validity. Software projects vary widely in coding style and language features used. This can result in differences among the number of detectable and undetectable equivalences that are generated. Our results may only generalize to subjects that share the characteristics of our subject pool. To combat this, we sampled from a variety of subjects.

7.5.2 Internal Validity. Reasoning about some mutants was very difficult, and this leads to some uncertainty in our data. We minimized this uncertainty by consulting with other authors.

Additionally, some rules rely on method implementations to obey their API contracts. For example, **equals() Reference Equality** assumes that an initial reference equality check in a class' `equals` method can be skipped without affecting the semantics (it is essentially an optimization). It is possible that a valid implementation of `equals` can exist that does not satisfy this assumption. Similarly, suppression rules based on **Standard Library Contracts** can be unsound: there is nothing that prevents a developer from implementing, for example, a `java.util.Collection` with a `size` method that returns negative values. In practice, we did not observe counterexamples to our assumptions for either of these suppression rules, and we believe that these are safe and valuable suppression rules, even if they are technically unsound.

It is also possible that other forms of unsoundness (e.g., faulty implementation) are in some of our rules. We have designed the rules to be sound, but we have not attempted a formal proof of correctness and soundness. To mitigate this, we have written extensive tests of what mutants should and should not be suppressed, and we manually inspected most of the suppressed equivalent mutants. Further, we manually checked all suppressed mutants in the ground-truth dataset and found no false positives.

7.5.3 Construct Validity. We assume the value of detecting an equivalent mutant is constant. We did not consider how different classes of equivalent mutants might be more valuable to detect than others, for instance by virtue of requiring more effort to detect manually or by being more likely to indicate a problem in code.

8 Conclusions

This paper reports on a manual ground-truth analysis of the types of equivalent mutants that exist in the wild and how difficult it is to detect them. With an estimated median per-project equivalent mutant rate of 2.97%, many equivalent mutants are caused by common programming patterns and are detectable with efficient static analyses. We implemented EMS in the Major mutation framework and evaluated it on 19 open source Java projects, comparing it to two TCE variants as baselines. EMS detected 8,776 equivalent mutants within 4,546 seconds, taking only 0.52 seconds per detected equivalent mutant; TCE$_{javac}$ deteted 1,007 equivalent mutants in 1,549 hours, taking 5,538 seconds per equivalent mutant detected, and TCE$_{soot}$ detected 2,124 equivalent mutants in 2,938 hours, taking 4,980 seconds per equivalent mutant detected.

Data-Availability Statement

We provide data files and analysis scripts at: https://www.doi.org/10.6084/m9.figshare.26948143.v1 [20]. Defects4J and Major are publicly available, which supports full reproducibility.

Acknowledgements

This work is supported in part by National Science Foundation grant CCF-1942055.

References

[1] Domenico Amalfitano, Ana C R Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. 2022. How do Java mutation tools differ? *Commun. ACM* 1, 1 (2022), 23.

[2] J.H. Andrews, L.C. Brand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEe, St. Louis, MO, USA, 402–411. https://doi.org/10.1109/ICSE.2005.1553583

[3] Douglas Baldwin and Frederick Sayward. 1979. Heuristics for Determing Equivalence of Program Mutations. (July 1979). https://apps.dtic.mil/sti/pdfs/ADA071795.pdf

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[5] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *Proceedings of the International Conference on Automated Software Engineering (ASE).*

[6] Henry Coles. 2024. Pitest: Real world mutation testing. https://pitest.org (last accessed March 2024).

[7] Renzo Degiovanni and Mike Papadakis. 2022. µBert: Mutation Testing using Pre-Trained Language Models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, Valencia, Spain, 160–169. https://doi.org/10.1109/ICSTW55395.2022.00039

[8] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How Close are they to Real Faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering.* 189–200. https://doi.org/10.1109/ISSRE.2014.40 ISSN: 2332-6549.

[9] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. 2009. The Impact of Equivalent Mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops.* IEEE, Denver, CO, USA, 192–199. https://doi.org/10.1109/ICSTW.2009.37

[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE).* 672–681. https://doi.org/10.1109/ICSE.2013.6606613 ISSN: 1558-1225.

[11] René Just, Michael D. Ernst, and Gordon Fraser. 2013. Using State Infection Conditions to Detect Equivalent Mutants and Speed up Mutation Analysis. arXiv:1303.2784 [cs.SE]

[12] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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).* San Jose, CA, USA, 437–440.

[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014.* ACM Press, Hong Kong, China, 654–665. https://doi.org/10.1145/2635868.2635929

[14] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring mutant utility from program context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 284–294.

[15] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* IEEE, Lawrence, KS, USA, 612–615. https://doi.org/10.1109/ASE.2011.6100138

[16] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing mutants to guide mutation testing. In *Proceedings of the 44th International Conference on Software Engineering.* ACM, Pittsburgh Pennsylvania, 1743–1754. https://doi.org/10.1145/3510003.3510187

[17] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. 2023. iBiR: Bug-report-driven Fault Injection. *ACM Transactions on Software Engineering and Methodology* 32, 2 (April 2023), 1–31. https://doi.org/10.1145/3542946

[18] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering* 44, 4 (April 2018), 308–333. https://doi.org/10.1109/TSE.2017.2684805

[19] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 142–151. https://doi.org/10.1109/ICSTW.2016.41

[20] Benjamin Kushigian, Rene Just, and Samuel Kaufman. 2024. Supplementary material for *Equivalent Mutants in the Wild: Identifying and efficiently suppressing Equivalent Mutants for java programs.* https://doi.org/10.6084/m9.figshare.26948143.v1

[21] Benjamin Kushigian, Amit Rawat, and Rene Just. 2019. Medusa: Mutant Equivalence Detection Using Satisfiability Analysis. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, Xi'an, China, 77–82. https://doi.org/10.1109/ICSTW.2019.00035

[22] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for java. In *Proceedings of the 28th international conference on Software engineering.* ACM, Shanghai China, 827–830. https://doi.org/10.1145/1134285.1134425

[23] Simona Nica and Franz Wotawa. 2012. Using Constraints for Equivalent Mutant Detection. *Electronic Proceedings in Theoretical Computer Science* 86 (July 2012), 1–8. https://doi.org/10.4204/EPTCS.86.1 arXiv: 1207.2234.

[24] A. Jefferson Offutt. 1988. *Automatic Test Data Generation.* Ph. D. Dissertation. Georgia Institute of Technology, Atlanta, GA.

[25] A. Jefferson Offutt and W. Michael Craft. 1994. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 4, 3 (1994), 131–154.

[26] A. Jefferson Offutt and Jie Pan. 1994. *Using Constraints to Detect Equivalent Mutants.* Ph. D. Dissertation. https://pdfs.semanticscholar.org/329d/2f8107679740395bac2cc0525f83adf33a20.pdf?_ga=2.31610622.1559687992.1581913553-1401888376.1581913553

[27] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* Vol. 1. 936–946. https://doi.org/10.1109/ICSE.2015.103 ISSN: 1558-1225.

[28] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. 2021. Does Mutation Testing Improve Testing Practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, Madrid, ES, 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

[29] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (Oct. 2022), 3900–3912. https://doi.org/10.1109/TSE.2021.3107634

[30] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2023. Please fix this mutant: How do developers resolve mutants surfaced during code review?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE, 150–161.

[31] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and Rene Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, Vasteras, 47–53. https://doi.org/10.1109/ICSTW.2018.00027

[32] David Schuler and Andreas Zeller. 2013. Covering and Uncovering Equivalent Mutants. *Software Testing, Verification and Reliability* 23, 5 (2013), 353–374. https://doi.org/10.1002/stvr.1473 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1473.

[33] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Scottsdale Arizona USA, 762–774. https://doi.org/10.1145/2660267.2660339

[34] Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. 2020. DeepMutation: a neural mutation tool. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings.* ACM, Seoul South Korea, 29–32. https://doi.org/10.1145/3377812.3382146

[35] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers.* 214–224.

[36] J. M. Voas. 1992. PIE: A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering* 18, 8 (Aug. 1992), 717–727. https://doi.org/10.1109/32.153381 Num Pages: 11 Place: New York, United States Publisher: IEEE Computer Society.

[37] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, Hyderabad India, 919–930. https://doi.org/10.1145/2568225.2568265

Received 2024-04-12; accepted 2024-07-03