Exploring Pseudo-Testedness: Empirically Evaluating Extreme Mutation Testing at the Statement Level

Megan Maton
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Phil McMinn
University of Sheffield

***Abstract*—Extreme mutation testing (XMT) detects undesirable pseudo-testedness in a program by deleting the method bodies of covered code and observing whether the test suite can detect their absence. Even though XMT may identify test limitations, its coarse granularity means that it may overlook testing inadequacies, particularly at the statement level, that developers may want to address before committing the resources demanded by traditional mutation testing. This paper proposes the use of the statement deletion mutation operator (SDL) to uncover pseudo-tested statements in addition to complete methods. In an experimental evaluation involving four frequently-studied, large, Apache Commons Java projects and 23 projects randomly selected from the Maven Central Repository, we found 722 different cases of pseudo-tested statements. Critically, we discovered that 48% of these statements exist outside of pseudo-tested methods, meaning that the detection of testing deficiencies related to these statements would normally be left to traditional, resource-intensive, mutation testing. Also, we found that a popular Java mutation testing tool would not have mutated some of the statement types involved in the first place, effectively rendering these issues, hitherto, hard to discover. This paper therefore demonstrates that XMT alone is insufficient and should be combined with pseudo-tested statement evaluation to pinpoint subtle, yet important, testing oversights that a developer should tackle before applying traditional mutation testing.**

I. Introduction

Extreme mutation testing (XMT) is an approach to detecting deficiencies in a test suite that individually deletes method bodies in covered program code and observes whether the test suite detects their absence [1]. If each method body deletion is successfully detected — i.e., at least one test now *fails* — the method is said to be "required" for the test suite to pass [2]. However, if a deletion goes unnoticed — i.e., all the tests *still pass* — the method is instead said to be "pseudo-tested". That is, a pseudo-tested method is a method that is executed by a test suite and did not raise an unexpected exception, but its result was not checked by a test's assertions, either directly or indirectly. Since pseudo-tested methods indicate apparent deficiencies in a program's test suite, developers can use knowledge about them to strengthen the program's tests, removing weaknesses related to pseudo-testedness, and ensuring that each method is more effectively tested [1].

Pseudo-tested methods have been shown to achieve lower mutation scores than required methods, further showing how they can be "blind spots" for a test suite [2]. Tools such as Descartes [3] and Reneri [4] can find pseudo-tested methods without the need to do full mutation testing, while also suggesting improvements to developers. The quick feedback that XMT provides thereby enables developers to address issues as their program code evolves. However, XMT's use of a coarse-grained transformation, like method body deletion, may overlook subtler instances of pseudo-testedness. Methods classified as required may not themselves be thoroughly tested, meaning that the test suite still misses some issues that traditional mutation testing may detect. That is, a technique is needed to help developers quickly find smaller units of pseudo-testedness, in addition to that provided for complete methods, without having to resort to a full and costly mutation analysis.

This paper demonstrates that the statement deletion (SDL) mutation operator [5] is an operator that could be used to identify pseudo-testedness in individual program statements and statement blocks. The SDL operator is a versatile mutation operator, capable of mutating most of the code that can appear in a program [6]. Furthermore, Deng et al. [7] found that a test suite that achieves a 100% mutation score using only the SDL operator achieves an average of a 92% mutation score with a full set of operators, but with 81% fewer mutants. Meanwhile, developers at Google use statement block removal to highlight potentially redundant code, also finding that SDL is one of the most productive mutants they use [8], [9]. However, SDL is often left out or only partially implemented in existing widely-used Java mutation tools such as PIT [10] and Major [11], while research has yet to evaluate SDL's usefulness beyond assessing its role as a mutant reduction technique [7], [12].

This paper is the first to use SDL to identify statements that are pseudo-tested by a test suite, finding 722 examples of pseudo-tested statements in both four frequently-studied, large, Apache Commons Java projects as well as 23 projects randomly selected from the Maven Central Repository [13], the largest official software repository for Java. Critically, we found that XMT misses 48% of the pseudo-tested statements as they appear in methods it classifies as required, an oversight that may lead developers to an incorrect assumption that all pseudo-testedness has been addressed — and leaving the detection of testing deficiencies related to these statements to traditional, resource-intensive, mutation testing [14]. We further found that pseudo-tested statements achieved lower mutation scores than required statements and that traditional mutation testing does not typically mutate some Java statement types, such as *continue*, *break* and *throw* statements, leaving gaps in test suite evaluation [15], [16]. Finally, we

PSEUDOSWEEP Classifications
□ Required Statements ■ Pseudo-tested Statements

```
1  /**
2   * Decode bytes encoded with Percent-Encoding based
3   * on RFC 3986. The reverse process is performed in
4   * order to decode the encoded characters to Unicode.
5   */
6  @Override
7  public byte[] decode(final byte[] bytes)
8                               throws DecoderException {
9    if (bytes == null) {
10     return null;
11   }
12   /* ... */
13   try {
14     /* ... */
15     buffer.put((byte) ((u << 4) + l));
16   } catch (final ArrayIndexOutOfBoundsException e) {
17     throw new DecoderException(/* ... */);
18   }
19   /* ... */
20   return buffer.array();
21 }
22
23
24 ✓@Test
25   void testPercentEncoderDecoderWithNullOrEmptyInput()
26                                     throws Exception {
27    /* ... */
28    assertNull(percentCodec.decode(null),/* ... */);
29    /* ... */
30  }
31
32 ✓@Test
33   void testDecodeInvalidEncodedResultDecoding()
34                                     throws Exception {
35    /*...*/
36    try {
37      percentCodec.decode(/* Invalid Encoded Result */);
38    } catch (final Exception e) {
39      assertTrue(
40          DecoderException.class.isInstance(e) &&
41              ArrayIndexOutOfBoundsException.class
42                                .isInstance(e.getCause()));
43    }
44  }
```

Listing 1. An example of a "required" method containing a "pseudo-tested" statement from the *org.apache.commons.codec.net.PercentCodec decode* method, and two tests, including *testDecodeInvalidEncodedResultDecoding*, from the test suite called *org.apache.commons.codec.net.PercentCodecTest*.

manually analysed a sample of 119 pseudo-tested statements to understand their causes, revealing that most pseudo-tested statements are due to discrete test suite inadequacies (e.g., missing tests or partial assertions) that XMT and traditional mutation testing cannot highlight due to their inapplicable operators. The contributions of this paper, then, are as follows:

1) A quantitative evaluation of the frequency of pseudo-testedness at the statement level, specifically those left undetected by XMT (Section V-B).
2) A quantification of the differences between pseudo-tested statements and other covered statements using traditional mutation scores (Sections V-C and V-D).
3) A qualitative manual analysis of the causes of pseudo-tested statements to understand the test suite inadequacies that they uncover (Section V-E).

II. MUTATION TESTING

Mutation testing inserts syntactic changes into the program under test to evaluate a test suite's capability to find real faults if they were present [17], [18]. Although helpful in improving test suite effectiveness, mutation testing presents a significant overhead [19] since the runtime for executing a complete set of mutants can be substantial as every mutant can require a complete run of the test suite. However, not all statements are valuable to test and, therefore, addressing surviving mutants (including pseudo-tested code) can be a waste of resources.

A. The Statement Deletion Operator (SDL)

Deletion mutation operators remove code structures from the program under test [20]. The statement deletion operator (SDL) removes entire statement blocks. The SDL operator removes a statement from the program under test and executes the test suite to see if it causes a failure [7]. The statement deletion mutant is "killed" if the test suite detects the deletion (i.e., a test fails) and it "survives" if it does not. For example, in Listing 1, the SDL operator would remove the *if* statement on Lines 9–11. When it removes this statement, the *bytes* variable can remain *null* beyond this point, causing a *NullPointerException*; therefore, *testPercentEncoderDecoderWithNullOrEmptyInput* fails and kills the mutant. However, suppose the SDL operator removes the *throw* statement on Line 17. The test suite will not fail in that case, as the *testDecodeInvalidEncodedResultDecoding* only checks the exception type if the *decode* method throws an uncaught exception.

Prior empirical studies have shown that SDL independently produces significantly fewer mutants than traditional mutation testing without losing significant effectiveness [6], [7]. Delamaro et al. found that SDL is combinable with other deletion operators, such as operator and variable deletion, to enhance SDL's detection of testing concerns [20]. The SDL operator is limited in its implementation across different Java mutation testing tools, as we now discuss in further detail for three tools.

muJava: Deng et al. implemented their definition of the SDL operator into the muJava mutation tool [7], [21]. This implementation includes deletions for entire statement structures from single-line statements up to entire *for* statements. It also includes a basic set of default values for *return* statements. This involves returning the value "0" for *int*, *char*, *double*, *float*, *long* and *short*, as well as both "true" and "false" for *boolean* types. For the *String* type, muJava returns *null*.

Major: Just et al. identified statement deletion as akin to a genuine fault, yet requiring a stronger mutation operator [16]. For their implementation of the statement deletion operator in Major, they found that deletion of control flow statements can lead to uninitialised variables or unreachable code errors.

PIT: PIT is a state-of-the-art mutation testing tool that inserts and executes mutants in Java programs by using Java bytecode manipulation [10]. This means that its operations do not directly map to the program's source code, making it impossible to depict entire source code statements accurately.

B. Extreme Mutation Testing

Extreme mutation testing (XMT) reduces the number of mutants executed by only removing method bodies from the program under test [1]. Where a return value is necessary for the program to compile, XMT adds default return values using

preset values for different types. After each deletion, XMT runs the tests to determine whether the deletion causes a failure. These definitions classify elements evaluated under XMT:

Definition 1. Not-covered (N) An element in the program under test that is not executed by the test suite.

Definition 2. Required (R) A program element that is executed by the passing test suite and that **cannot** be removed without impacting the test outcomes.

Definition 3. Pseudo-tested (P) A program element that is executed by the test suite and that **can** be removed without impacting test outcomes.

Definition 4. Partially-tested (A) A program element executed by the test suite that **can** be fixed to only a subset of possible values without impacting test outcomes.

If the *decode* method body in Listing 1 was replaced by a single default *return*, at least one of the tests that cover it would fail and thus it is "required" for the tests to pass.

To illustrate the distinction between the last two definitions, consider an integer variable, which may be mutated to "0" and "1". Killing only one mutant leaves the method (or statement) "partially-tested" [22], whereas if both mutants survive, the variable would be "pseudo-tested". For this paper's evaluation, pseudo-tested elements can include those that are partially-tested as both are potential action areas for a developer.

Extreme mutation testing reduces the overhead of mutation testing by applying only a few operators (depending on the method return type) to each method. Previous studies used traditional mutation testing [18] to confirm that pseudo-tested methods are poorly tested compared to other covered methods [2], [22]. These studies calculated the method-level mutation scores for required and pseudo-tested methods, finding that the pseudo-tested ones had lower scores than those that were required by the tests. There are two existing tools that target XMT for Java, as we now discuss in further detail.

Descartes: The Descartes mutation engine for the PIT mutation tool implements XMT for use within PIT [2]. At the Java bytecode level, it is easy for a tool to remove a method and for a developer to translate that back to the method body that the tool deleted. Therefore, it is appropriate to implement XMT through the manipulation of Java program's bytecode.

Reneri: The Reneri tool observes pseudo-tested method executions both with and without a method body's removal, enabling it to suggest potential solutions for pseudo-testedness, such as adding tests and verifying the values of variables [4].

III. Approach

Our automated approach finds relevant pseudo-tested statements by extending the SDL operator to include further transformations and implementing an auxiliary approach to identify pseudo-tested methods. Since identifying pseudo-tested statements within pseudo-tested methods does not present any new information to a developer, it is more helpful to explore the pseudo-tested statements external to pseudo-tested methods, that is, pseudo-tested statements that XMT would not reveal.

For example, for the *decode* method in Listing 1, identifying pseudo-tested methods alone would declare this method as "required" for the test suite to pass. However, a tool can remove the *throw* on Line 17 without causing any test cases to fail. We show the *testDecodeInvalidEncodedResultDecoding* test case as it targets this line. However, despite covering this statement and using an assertion to check the exception type, the structure of the test means that if it does not trigger the program code to throw an exception in the first place, the test does not reach the *catch* statement containing the assertion, meaning the test does not fail. With this information, a developer could add the following JUnit fail assertion:

```
1 fail("Expected exception was not thrown");
```

to the test between lines 37 and 38 to fail when the test has reached this point without catching an exception, ensuring that future maintenance does not accidentally introduce this fault of omission. We implemented this approach in a tool, called PSEUDOSWEEP, as explained in this section's remainder. (We direct the interested reader to reference [23] for further information about the PSEUDOSWEEP tool.)

A. Method Classification

If PSEUDOSWEEP removes a covered method without the test suite failing, the method is pseudo-tested. Where a method requires a *return* statement for compilation, PSEUDOSWEEP uses a pre-defined set of default values as used in the current implementation of Descartes [2]. For example, where a method returns an integer value, PSEUDOSWEEP replaces the method body with *return 1* and then tries again with *return 0*. By checking two return values for each method, PSEUDOSWEEP avoids returning only the value that the tests expect. We include a complete description of the operators in the replication package [24]. By first classifying each method as not covered, required, or pseudo-tested (where the pseudo-tested subset includes partially-tested methods), we can identify the immediate issues of pseudo-testedness, like other XMT tools.

B. Statement Classification

Next, PSEUDOSWEEP locates the pseudo-tested statements and uses the method classifications to identify pseudo-tested statements within the required methods. Note that not-covered methods cannot contain pseudo-tested statements, as the test suite must cover a pseudo-tested statement by definition.

Basing the core deletions on the statement deletion (SDL) operator [7] would mean that PSEUDOSWEEP could only evaluate statements that it can delete in their entirety without causing compilation issues. Therefore, we extended SDL to enable PSEUDOSWEEP to delete variable declarations, *lambda* statements, and labelled loops, as explained in Section III-C.

We also found that SDL and XMT would conflict when classifying some single-statement methods. Given the following method, SDL classifies the *return* statement on Line 2 as "required" because it applies only a single killable mutant.

```
1 public int subtract(int a, int b) {
2     return a - b;
3 }
```

Yet, XMT would apply two mutants, where only one is killable, and classify the method as pseudo-tested. Therefore, we have implemented SDL with the same default operators (i.e., combine SDL with a Return Value Mutation operator), like XMT in Descartes Engine, to improve its effectiveness [3].

C. Metamutant

PSEUDOSWEEP implements statement deletion and method deletion by instrumenting source code to create a metamutant [25]. This metamutant ensures that all statement and method deletions are always compilable. To create the metamutant, PSEUDOSWEEP inserts *if* statements around each statement/method in the source code, enabling it to conditionally execute each element [26]. The condition of the inserted *if* statements calls PSEUDOSWEEP to check if it should run the contained element. The following Java code gives a simplified example of how the tool instruments most program statements.

```
if (PseudoSweep.exec(ID)){
    i++;
}
```

The full instrumentation includes element type and class information. Where change in scope will cause compilation issues, the tool uses default values to enable deletion.

```
String s = "";
if(PseudoSweep.exec(ID)){
    s = "actual";
}
```

In the example above of a variable declaration metamutant, PSEUDOSWEEP enables compilation by assigning a default value and conditionally assigning the original initialisation.

D. Evaluating Deletions

We can use the previously described metamutant to instrument each element. PSEUDOSWEEP runs each deletion with a timeout relative to the original test case runtime to halt any infinite loops introduced by the metamutant. The tool then runs every test three times to identify the covered elements before sequentially "removing" each element and executing the test. If the test passes, the tool runs it twice again to check for flakiness [27]. We skip flaky tests as they may impact pseudo-testedness detection. The results are then internally analysed to identify the elements whose removal does not cause test failures and presented to the user as pseudo-tested elements.

IV. EVALUATION

Since prior work suggests that pseudo-tested methods exist in all projects [1], [2], [22], we first investigate how common pseudo-tested statements are within our project set and their relationship with the "required" methods (RQ1). Next, we determine whether mutation scores for pseudo-tested statements are lower than for required statements (RQ2), as occurs at the method level [2], [22]. Using mutation scores calculated by PIT, we explore whether traditional mutation testing effectively targets pseudo-tested statements (RQ3). Finally, since prior work showed that XMT overlooked concerning issues in required methods [22], we use statement deletion to identify the issues not surfaced by XMT and compare them to the findings of traditional mutation testing, thereby highlighting the gaps between the different techniques (RQ4). Ultimately, this paper answers the following four research questions:

RQ1: How frequent are pseudo-tested elements?
RQ2: Do pseudo-tested elements have low mutation scores?
RQ3: Does PIT's default set of operators effectively highlight deficient testing with respect to pseudo-tested statements?
RQ4: What are the causes of pseudo-tested statements?

A. Projects

This paper investigates pseudo-testedness in diverse, real-world Java projects. Table I shows that we used four large open-source Apache Commons projects and 23 open-source projects randomly chosen from the Maven Central Repository [13]. When randomly selecting the Maven projects, we adopted the following criteria: uses JDK versions 8–11 (due to tooling restrictions from using JavaParser [28]); single module (currently PSEUDOSWEEP only analyses single-module projects); explicit type declarations; uses JUnit 4 or 5; one or more tests that compile and pass before using PSEUDOSWEEP.

To ensure that we studied a diverse set of real-world projects of different sizes and test suite maturities, we followed the project sampling method used by Gruber et al. to identify over 38,000 Maven projects [29]. As the prohibitive time costs of running PIT prevented the use of all these projects, we randomly sampled 180 of them from this list, collecting their source code from GitHub and keeping those that adhered to our inclusion criteria, thereby resulting in a set of 23 projects. We also included four commonly used Apache Commons projects to connect our results about pseudo-tested statements to prior studies on pseudo-tested methods [1], [2].

B. Tooling

Since it is the state-of-the-art mutation testing tool for Java, we used PIT version 1.15.6 with JUnit5 plugin version 1.2.1 and the Gregor mutation engine to calculate the traditional mutation scores [10]. We used PSEUDOSWEEP, as described in Section III, to analyse pseudo-tested statements and methods in the chosen projects. We used PSEUDOSWEEP's implementation of both the SDL mutation operator and XMT because the existing tools (e.g., muJava, Major, and PIT with the Descartes engine) were unsuitable for the following reasons.

For SDL, we could not use muJava [7], [21], [30], [31] because it only supports up to Java 1.6 and was thus not applicable to recent Java projects. Moreover, Major's implementation of SDL only incorporates a subset of the statement deletion operator transformations [11]. Major's source code is unavailable and therefore we could not extend the operator set. Statement deletion is also less useful when implemented at the bytecode level, since the bytecode does not translate directly into Java statements; this means that extending PIT's Gregor Engine to include SDL may not always produce suitable statement deletion mutants. Finally, the most recent version 1.3.2 of the Descartes engine [3] for performing XMT with PIT restricts it to an old 1.7.0 version, thereby limiting compatibility to an out-of-date JUnit5 plugin version 0.16.

TABLE I

Details for each evaluation project, including its GitHub owner and repository name and the numbers of methods (# METHOD), statements (# STMT), tests (# TEST), assertions (# ASSERT), and PIT mutants (# MUTANT), and JaCoCo's Bytecode Instruction Coverage (% BCOV) and Line Coverage (% LCOV) and PSEUDOSWEEP's Statement Coverage (% SCOV). The names of the Apache Commons projects are bold while those of the Maven projects are not. The portion of a project name that is not in italics is not used in the remainder of the paper for brevity. The "–" symbol indicates that it is a non-meaningful total.

Owner / *project*	# METHOD	# STMT	# TEST	# ASSERT	# MUTANT	% BCOV	% LCOV	% SCOV
LindenY / *asw4j*	170	816	42	215	504	48	49	49
authlete / *authlete-jose*	60	570	15	15	324	49	55	53
IvoNet / *beanunit*	44	275	44	56	183	80	75	68
sigpwned / *chardet4j*	80	682	13	13	732	91	68	62
abedra / *chronometrophobia*	65	85	3	5	238	41	55	52
apache / *commons-cli*	140	745	448	688	753	96	96	97
apache / *commons-codec*	625	3933	923	2536	4060	97	95	90
apache / *commons-csv*	145	960	468	1445	718	98	99	99
apache / *commons-math*	6467	46206	6136	12402	45646	92	90	87
coodoo / *coodoo-listing*	81	739	64	84	477	13	16	13
kestreldigital / *data-conjuror*	12	29	2	3	26	54	59	59
fuinorg / *ext4logback*	7	45	4	16	26	51	59	44
vebqa / *f3270*	204	1299	7	6	864	4	4	0.8
mervinkid / *jargser*	10	96	1	6	86	88	84	86
erdtman / java-*json-canonicalization*	65	1102	5	1	993	49	40	33
clerezza / *jena.serializer*	2	32	5	7	17	88	89	88
rbkmoney / *kafka-common-lib*	27	117	6	10	73	29	32	42
Naoghuman / *lib-logger*	36	95	3	0	43	25	23	19
harium / *marine*	14	45	2	2	31	6	7	4
awslabs / *payload-off-loading*-java-common-lib-for-aws	37	188	40	92	100	71	71	68
premiumminds / *pmpersistence*utils	27	228	19	24	106	66	71	68
sergiodeveloper / *sequencepattern*	114	493	16	16	397	41	48	41
andreidore / *smartbill*-java-*client*	16	200	11	9	58	19	21	23
tblsoft / *solr-cmd-utils*	807	7280	179	449	3204	24	24	23
bordertech / *wcomponents*-sass-*compiler*	7	62	1	2	28	70	74	74
wildflyswarm / *wildfly-swarm-spi*	64	268	12	156	180	47	46	40
vincentzurczak / *xml-region-analyzer*	10	222	9	273	196	96	97	97
Total for Apache Commons Projects	7377	51844	7975	17071	51177	–	–	–
Total for Randomly Selected Projects	1959	14968	503	1460	8886	–	–	–
Total for All Projects	9336	66812	8478	18531	60063	–	–	–

C. Methodology

We took the following steps to setup each of the chosen projects. Since every project used Maven it was configured with a "Project Object Model" (POM) file. For each project, we duplicated the POM file and added PSEUDOSWEEP as a dependency to one and PIT with the Gregor Engine as a plugin to the other, thereby ensuring their configurations did not interfere with each other. We used a script to execute both tools and direct each one's output data to a separate directory. As shown in Table I, we also collected the JaCoCo coverage scores to characterise a project's level of testedness. Finally, we implemented and ran scripts that characterise the projects according to the mutants arising from the use of XMT, SDL, and PIT. This enabled us to report metrics such as pseudo-tested statements within required methods and which types of statements contained the most PIT mutants. We used the data produced by these scripts to answer RQ1 through RQ3.

We adopted two approaches to studying pseudo-tested statements. First, we calculated the total number of pseudo-tested statements (P). We also investigated the subset of P that occurs in required methods (PiR). Any pseudo-tested methods identified by extreme mutation testing would already highlight that pseudo-testedness is present, meaning that identifying pseudo-tested statements in pseudo-tested methods does not surface new information to a developer and is thus not a focus of this paper. However, since extreme mutation testing does not reveal pseudo-tested statements within the required methods (PiR), this paper studies their frequency in the project set.

To answer RQ4, we manually analysed a sample of pseudo-tested statements found in the required methods. We investigated these elements because extreme mutation testing would not highlight their potential issues. The first author manually analysed a sample of 119 pseudo-tested statements across 30 required methods to identify their causes. We produced the sample by ordering the required methods according to the number of pseudo-tested statements and then selecting methods of varying counts of pseudo-tested statements (greater than 0) from different projects until we had a sample size of at least 100 pseudo-tested statements within required methods.

First, we manually identified and removed 12 "uninteresting" program statements that PSEUDOSWEEP mutated and ultimately labelled as pseudo-tested. Also called "arid nodes", these statements represent parts of a program's abstract syntax tree (AST) that perform tasks like logging and thus do not implement features that should be subjected to mutation testing [8]. Since the current implementation of our tool does not filter out these statements, they were a subset of the pseudo-tested statements in the required methods. In future work we will enhance PSEUDOSWEEP so that it automatically removes them from consideration. Finally, we removed four

equivalent mutants that PSEUDOSWEEP introduced when an inserted default value was equivalent to the expected value. We manually removed each statement one at a time to confirm that it was pseudo-tested before looking at the tests that covered the method to discern why the statement was removable without impacting the test suite outcomes. While manually reproducing pseudo-testedness, we identified and removed 11 statements in total from the sample that PSEUDOSWEEP had classified as pseudo-tested due to unhandled JUnit annotations and threading. To answer RQ4, we then identified the causes of the remaining 92 (119−12−4−11) pseudo-tested statements, ultimately grouping them into meaningfully labelled categories.

D. Threats to Validity

External Validity: We studied 27 open-source projects, selecting four open-source Apache projects (i.e., *commons-cli*, *commons-codec*, *commons-csv*, and *commons-math*) due to their widespread use in the testing literature and their comprehensive test suites [2], [3], [32], [33]. We randomly selected the remaining 23 projects from the Maven Central Repository according to our project criteria and then forked their source code from the most recent GitHub commit. We used these projects so that we could study pseudo-testedness in projects of various sizes and test suite maturities. Since our findings may not generalise to other projects, further studies should extend this paper's experiments to a larger project set.

Internal Validity: The PSEUDOSWEEP tool and the data analysis scripts are not exempt from defects. We extensively tested the tools, but instrumentation limitations may cause them to fail for code patterns we have yet to encounter. We further ran all of a project's tests three times and recorded execution times to check for flakiness [27]. If a test is flaky, we cannot rely on it to identify pseudo-tested elements [34], so we skip this test. We also ran each test three times against a mutant, mitigating any flakiness sources. We leave an investigation of the impact of flakiness on pseudo-testedness detection for future work. We also checked the results, addressed unexpected values, and created a replication package [24].

We based our categorisation of statement types on JavaParser's and combined sub-categorisations as part of the instrumentation process [28]. Adding our sub-categorisations was necessary to enable PSEUDOSWEEP to instrument source code in a compilable way. For example, in JavaParser, *variable declaration* statements are categorised as *expression* statements. Yet, if we were to delete the statement as we do with other *expression* statements, the instrumentation would introduce compilation errors. Since statements will likely be categorised differently by other tools, future work will investigate how different statement categorisations influence the results. Threaded code originating within tests themselves can interfere with our tool's internal thread control, used for timing out tests if deletions cause infinite loops. We were forced to omit certain tests (see replication package), which may have introduced some small inaccuracies into our results. Finally, the results for RQ1 through RQ3 include arid nodes [8]; future work on PSEUDOSWEEP will remove them from consideration.

We used PIT, with its commonly adopted "default" mutant set, to generate and evaluate the traditional mutants. However, using a different mutation testing tool or a different version or configuration of PIT may yield different results. Finally, the first author performed the manual analysis following a well-documented process to confirm PSEUDOSWEEP's findings and cross-check the results to understand the causes of pseudo-tested statements. Since this manual procedure could have introduced errors, future work will validate these interpretations by discussing them with the developers of each project.

V. RESULTS

A. Project Characterisation

This paper's study differs from prior work as the project set has varying degrees of coverage. We used PSEUDOSWEEP to calculate test coverage and used JaCoCo's [35] line and bytecode coverage scores to confirm them. Table I shows that the statement coverage reported by PSEUDOSWEEP ranges from 0.8% to 99%, which is similarly reflected in the JaCoCo line coverage (4% to 99%) and bytecode coverage (4% to 98%). For the Apache Commons projects used in prior studies, coverage was uniformly high; however, the randomly selected projects exhibited a wide range of coverage scores with *f3270* obtaining the lowest coverage scores at 0.8% for both bytecode instruction and line coverage, and *xml-region-analyzer* obtaining the highest coverage of 97% and 97%, respectively.

The sizes of projects evaluated in this study also ranged from 29 statements up to 46206 statements and from 7 methods to 6467 methods. Using a varied set of projects enables us to explore pseudo-testedness from different perspectives.

B. RQ1: How frequent are pseudo-tested elements?

Table II shows the distribution of "not-covered" (N), "required" (R) and "pseudo-tested" (P) methods and statements for each project. While determining the frequency of pseudo-tested methods enables us to calibrate this paper's results with prior work, calculating the frequency of pseudo-tested statements characterises a phenomenon not heretofore studied. For both methods and statements, the table further reports the total number of elements across a project (N+R+P). For statements, we also identified the subset of pseudo-tested statements that our tool found within the required methods (PiR), thereby highlighting a crucial limitation of XMT.

In the following analysis of Table II's data, we first highlight the overall trends and then respectively examine the frequency data for the Apache Commons and randomly selected projects.

1) Frequency of pseudo-tested methods: Results from prior empirical studies suggest that pseudo-tested methods exist within all projects [1], [2], [22]. However, Table II reveals that this trend does not hold for our chosen projects since seven of the randomly selected ones contained no pseudo-tested methods. With that said, all Apache Commons projects used in prior work contained at least one pseudo-tested method. This result can be explained by the fact that, as mentioned in Section V-A, the chosen project set differs from previous studies, with more varied project types and levels of testedness.

TABLE II

The numbers of not-covered (N), required (R), pseudo-tested (P), and the total number of elements for methods and statements (N+R+P), and pseudo-tested statements in required methods (PiR) for statements, which is a subset of P. Each horizontal bar shows the proportion of a project's elements in N, R, and P.

	Methods					Statements					
Project	# N□	# R▨	# P■	(N+R+P)		# N□	# R▨	# P■	# PiR	(N+R+P)	
asw4j	87	59	24	170		414	385	17	12	816	
authlete-jose	20	31	9	60		273	297	0	0	570	
beanunit	3	26	15	44		87	138	50	31	275	
chardet4j	22	30	28	80		256	426	0	0	682	
chronometrophobia	41	24	0	65		41	44	0	0	85	
commons-cli	4	132	4	140		20	705	20	5	745	
commons-codec	70	523	32	625		385	3285	263	125	3933	
commons-csv	0	144	1	145		12	944	4	2	960	
commons-math	834	5243	390	6467		5841	40105	260	119	46206	
coodoo-listing	72	9	0	81		640	97	2	2	739	
data-conjuror	6	1	5	12		12	14	3	0	29	
ext4logback	4	3	0	7		25	20	0	0	45	
f3270	202	1	1	204		1289	1	9	0	1299	
jargser	2	5	3	10		13	83	0	0	96	
json-canonicalization	22	32	11	65		735	367	0	0	1102	
jena.serializer	0	2	0	2		4	28	0	0	32	
kafka-common-lib	19	6	2	27		68	46	3	2	117	
lib-logger	31	2	3	36		77	1	17	4	95	
marine	12	0	2	14		43	0	2	0	45	
payload-off-loading	9	24	4	37		60	101	27	10	188	
pmpersistence	10	17	0	27		72	154	2	2	228	
sequencepattern	80	29	5	114		293	200	0	0	493	
smartbill-client	14	2	0	16		155	32	13	12	200	
solr-cmd-utils	569	202	36	807		5636	1615	29	24	7280	
wcomponents-compiler	0	7	0	7		16	46	0	0	62	
wildfly-swarm-spi	40	22	2	64		161	106	1	0	268	
xml-region-analyzer	0	9	1	10		6	216	0	0	222	
Total for Apache Commons Projects	908	6042	427	7377		6258	45039	547	251	51844	
Total for Randomly Selected Projects	1265	543	151	1959		10376	4417	175	99	14968	
Total for all Projects	2173	6585	578	9336		16634	49456	722	350	66812	

Some projects exhibit very little coverage, some have only required methods, and others have a more even distribution. By splitting covered methods into required and pseudo-tested methods, the results show that 23 of the 27 projects contain more methods that are required than pseudo-tested, with a maximum of 5243 required methods in *commons-math* and a minimum of zero in *marine*. Overall, pseudo-tested methods were present in 20 of 27 projects, with the smaller randomly selected projects tending to have no pseudo-tested methods.

The maximum number of pseudo-tested methods for the Apache Commons projects was 390 in *commons-math*, while in the randomly selected projects the maximum was 36, which PSEUDOSWEEP found in *solr-cmd-utils*. The median value for pseudo-tested methods across all projects was three, whereas the median number of required methods was 22; overall, projects tended to have fewer pseudo-tested methods than required methods. All projects contained at least one covered method; therefore, PSEUDOSWEEP did not find any projects with both zero required *and* zero pseudo-tested methods.

There were some interesting distributions of method coverage in the randomly chosen projects. For example, *f3270* contained 204 methods of which only two were covered, with one required and one pseudo-tested. The low coverage of this project showed that even with seven test cases covering the two methods, the test suite required only one of the methods to pass. The *lib-logger* project also had low coverage, with 13.8% ((2+3)/36) coverage split into two required and three pseudo-tested methods achieved with only three test cases.

Other project examples include *wcomponents-compiler* and *jena.serializer*, both of which contained less than 10 methods in total, all of which were classified as required. Also, *ext4logback* had fewer than 10 methods but contained four non-covered methods and three required methods. No project with fewer than 10 methods had any pseudo-tested methods.

2) Frequency of pseudo-tested statements: To our knowledge, the frequency of pseudo-testedness at the statement level has yet to be studied. Prior work evaluates SDL as a mutant reduction technique, with limited study of its use beyond this purpose [7], [36]. Understanding the frequency of pseudo-tested statements in different types of methods will enhance our understanding of their importance to developers. Table II shows that the projects contained 66812 statements in total, of which their tests covered only 50178 (49456+722). The set of Apache Commons projects accounts for the majority of statements studied with a total of 51844 statements; the randomly selected projects contributed 14968 statements.

Pseudo-testedness made up a significantly smaller percentage of statements than it does of methods. Pseudo-tested methods comprised 6.1% (578/9336) of the total methods, whereas pseudo-tested statements comprise 1.08% (722/66812) of total statements. For the Apache Commons projects, pseudo-tested statements contributed to 1.05% (547/51844) and for

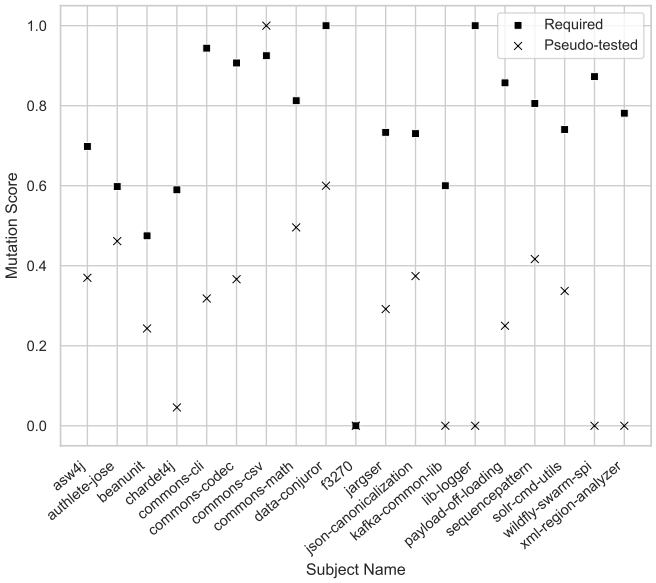

Fig. 1. Mutation scores for pseudo-tested and required methods.

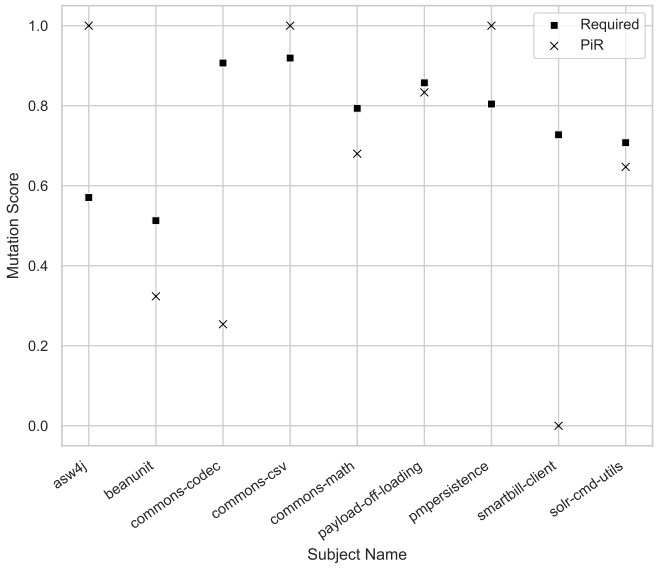

Fig. 2. Mutation scores for PiR and required statements.

randomly selected projects they made up 1.17% (175/14968). The project set also contained zero pseudo-tested statements for ten projects, revealing that pseudo-testedness at the statement level does not exist in all projects. Yet, across all projects, the median of pseudo-tested statements was two.

PSEUDOSWEEP found 350 PiR statements, ranging from zero in 14 projects to 125 in *commons-codec*, a large, well-tested project. The median was also zero, indicating they are uncommon in most projects. However, in the projects where they are present, PiR statements — that would be overlooked by XMT — are 48% (350/722) of pseudo-tested statements.

Conclusion for RQ1. Pseudo-tested elements (P) make up 6.1% (578/9336) of methods and 1.08% (722/66812) of statements. Pseudo-tested statements within required methods (PiR) are 48% (350/722) of pseudo-tested statements.

C. RQ2: Do pseudo-tested elements have low mutation scores?

Figure 1 furnishes the traditional mutation scores for all projects that had both required (R) and pseudo-tested (P) methods. Moreover, Figure 2 gives the traditional mutation scores for those projects with both required (R) and pseudo-tested statements in required methods (PiR). Finally, for each classification of methods and statements, Table III furnishes the number of elements, the total number of mutants, the number of mutants killed by the tests, and the overall mutation score, inclusively ranging between 0.0 and 1.0, as calculated by PIT. In the following analysis of these three data sources, we highlight both the overall and project-specific trends.

1) Method Mutation Scores: Figure 1 shows that the mutation scores for required and pseudo-tested methods follow the trend identified by prior work [2]: those that are pseudo-tested obtain lower mutation scores than those that are required. Table III reveals that the overall mutation score for required methods is 0.82 but only 0.40 for pseudo-tested methods.

TABLE III

For required (R), pseudo-tested (P), and not-covered (N) methods and statements and pseudo-tested in required statements (PiR), the number of program elements (# Elements), number of traditional mutants (# Mutants), number of mutants detected by the test suite (# Killed), and the overall mutation score (Score = # Killed / # Mutants) as calculated by the PIT mutation testing tool.

Elements	# Elements	# Mutants	# Killed	Score
Methods				
Required	6585	45878	37466	0.82
Pseudo-tested	578	2124	839	0.40
Not covered	2173	7115	219	0.03
Statements				
Required	49456	114164	90973	0.80
Pseudo-tested	722	818	406	0.50
PiR	350	322	184	0.57
Not covered	16634	20071	1971	0.10

Figure 1 also shows that, for all projects except *commons-csv*, the mutation scores for the required methods are higher than those of the pseudo-tested methods. We attribute the disparate result for the single Apache Commons project to the fact that it had only one pseudo-tested method with a single statement and a single, killed PIT mutant, ultimately enabling its test suite to obtain a 100% traditional mutation score.

2) Statement Mutation Scores: PSEUDOSWEEP found 14 projects that did not contain any pseudo-tested statements in required methods. Since traditional mutation testing cannot mutate non-existent pseudo-tested statements, Figure 2 does not present traditional mutation scores for them. With that said, Table III shows that the overall mutation score of 0.57 for PiR statements is lower than that of required statements at 0.80, similar to the trend detected at the method level. Figure 2 reveals that for two of the four Apache Commons projects (i.e., *commons-codec* and *commons-math*) the mutation score for required statements is higher than for the PiR statements. Note that for *commons-csv* the mutation score for PiR is higher than the mutation score for the required ones since it contains only one mutant in a PiR statement that was

killed. For *commons-cli*, we found that the PiR statements did not contain any mutants. For the randomly selected projects, the mutation scores for required statements were higher than those for the PiR statements. For *asw4j* and *pmpersistence*, PIT generated only a single, killed mutant in PiR statements, causing the mutation scores for PiR to be 1.0 for both projects and therefore higher than the score for required statements.

Conclusion for RQ2. Pseudo-tested methods (P) obtain a significantly lower mutation score of 0.40 than that of the required methods (R) scoring 0.82. Pseudo-tested statements in required methods (PiR) obtain a mutation score of 0.57 compared to the required statements' mutation score of 0.80.

D. RQ3: Does PIT's default operator set highlight deficient testing with respect to pseudo-tested statements?

Section V-C's answer to RQ2 pointed out that PIT generated very few mutants for the elements that PSEUDOSWEEP classified as pseudo-tested. As shown in Table III, this lower "mutants-per-element" count for pseudo-tested elements was evident at both the method and statement levels. At the method level, required methods contained 6.97 (45878/6585) mutants per method, compared to those that were pseudo-tested, which had an average of 3.67 (2124/578) mutants per method. Required statements also had a higher count of 2.31 (114164/49456) mutants per statement as opposed to PiR statements with 0.92 (322/350) mutants per statement. Importantly, if PIT does not generate a sufficient number of mutants for pseudo-tested elements, then traditional mutation testing will not accurately characterise their well-testedness.

For all of the statement types across all of the projects, Table IV shows that the ones with the fewest number of mutants generated for them were: 588 *break* statements containing 4 mutants between them, 153 *continue* statements containing 0 mutants, and 1997 *throw* statements containing only 82 mutants between them. This result suggests that there are several types of Java statements for which it is not sufficient to measure test adequacy through mutation testing with PIT.

This table also reveals that there are several types of Java statements, such as *do* and *lambda return*, for which there are no pseudo-tested statements in required methods ($\varnothing$). In this case, PIT could not generate any mutants for these statement types ($\uparrow$) and, therefore, the mutation score was undefined ($\perp$). Even when there were PiR statements for certain statement types, like *break* and *continue*, PIT did not generate any mutants and thus the mutation score was also undefined ($\perp$). When PIT did generate mutants for PiR statements, it only yielded 0.92 (322/350) mutants per statement, which is less than the 2.31 (114164/49456) mutants per all statements. Finally, the mutation score for a specific type of PiR statement was always less than the corresponding score for all statements of the same type. For instance, the mutation score for *while* statements was 0.65 for all program statements and only 0.33 for those that are PiR, suggesting that pseudo-tested statements in required methods are less well tested than the others.

Conclusion for RQ3. When using the default operator set from the Gregor engine, PIT generated only 0.92 (322/350) mutants per statement for PiR statements, while generating 2.31 (114164/49456) mutants per statement for required statements. Moreover, statement types such as *break*, *continue*, and *throw* also yield few PIT-generated mutants. Ultimately, the traditional mutation scores for all statements are higher than for those that are PiR, suggesting that PIT's default mutation operator set does not effectively highlight deficient testing for pseudo-tested program statements.

E. RQ4: What are the causes of pseudo-tested statements?

We manually studied pseudo-tested statements to discover their root causes and how developers can handle them. In the remainder of this discussion, we present the results of the manual analysis, furnishing the specific cause of pseudo-tested statements and the number of times that the cause was evident.

No targeting assertion (70): A test case covers statements in this category, but no dedicated test assertion is responsible for checking the pseudo-tested statement. This was the most common category we found, containing 70 statements.

One example was a *void* method call in *solr-cmd-utils* to a silent, error-handling method. The output of the method was silently logging exceptions of a specific type. The test suite did not check these logged exceptions; therefore, the *void* method call statement is pseudo-tested. Any calls to pseudo-tested methods will likely be pseudo-tested. Thus, this pseudo-tested statement presents little information to the developer. In *commons-codec*, we identified multiple statements without assertions checking them. Another example was the *language.Caverphone2.encode* method, containing 82 lines of code (LOC), which encodes an input string into a "CaverPhone 2.0" value. This required method contained 63 statements, of which 23 were pseudo-tested. This encoding process used the *java.lang.String.replace(t, r)* and *java.lang.String.replaceAll(r, r)* method calls to replace specific characters and regular expression patterns within the input string. PIT's default operator set does not mutate these non-*void* method calls, meaning that across all 82 LOC, PIT seeded only three mutants, of which the test suite killed two. This example shows that where a method relies on the returned values of non-*void* method calls, it is easy to miss assertions, as neither code coverage nor PIT's default set would highlight these issues to a developer. The documentation for PIT lists a "Non-Void Method Call" mutator that can be used but is not in the default set. The documentation also notes that this mutator is unstable and may create equivalent mutants. Importantly, statement deletion consistently produces compilable mutants and does not provide equivalent mutants in this context.

Developers could address these issues by adding assertions to evaluate the consequences of the pseudo-tested statements.

Partial assertion (7): We found partial test assertions where an assertion for the statement was present, but it only checked part of program's output. For example, PSEUDOSWEEP identified pseudo-tested statements related to partially checked string variables. Some statements involved method calls that

TABLE IV

For each type of statement, the number of statements for each type (# Statements), the number of traditional mutants generated by PIT (# Mutants), the number of mutants detected by the project's test suite (# Mutants Killed), and the mutation score (Score = # Mutants Killed / # Mutants), for both all statements in a program (All) and only the pseudo-tested statements in the required methods (PiR). In this table, the "∅" symbol means that there were no PiR statements of the specific type, "↑" means that PIT could not generate or kill any mutants since the value for "# Statements" was "∅", and "⊥" indicates that the mutation score was undefined either because "# Mutants" and "# Mutants Killed" were "↑"' or because "# Mutants" was zero. Using these symbols distinguishes between an undefined mutation score when (i) there are no PiR statements of a specific type ("∅") and thus both "# Mutants" and "# Mutants Killed" are "↑" and (ii) there are a non-zero number of PiR statements for which PIT could not generate any mutants, making the mutation score's denominator zero.

	All				PiR			
Statement Type	# Statements	# Mutants	# Mutants killed	Score	# Statements	# Mutants	# Mutants Killed	Score
break	588	4	1	0.25	9	0	0	⊥
continue	153	0	0	⊥	2	0	0	⊥
do	44	1241	720	0.58	∅	↑	↑	⊥
expression	22936	17155	11373	0.66	184	98	63	0.64
for	2968	25528	20748	0.81	13	52	33	0.63
for each	731	1750	846	0.48	∅	↑	↑	⊥
if	16640	58479	37631	0.64	43	91	42	0.46
inner class	39	19	7	0.37	∅	↑	↑	⊥
inner class return	70	60	52	0.87	∅	↑	↑	⊥
lambda	31	69	33	0.48	∅	↑	↑	⊥
lambda return	29	100	61	0.61	∅	↑	↑	⊥
loop condition	3435	6496	5363	0.83	∅	↑	↑	⊥
return	9683	12549	9070	0.72	60	53	36	0.68
switch	102	3170	2209	0.70	1	2	0	0
throw	1997	82	6	0.07	16	0	0	⊥
try	456	1494	539	0.36	∅	↑	↑	⊥
variable declaration	6466	1743	1383	0.79	17	5	3	0.60
while	444	5114	3308	0.65	5	21	7	0.33

returned sections of strings. When tests only partially check these strings using a "contains" assertion, some statements related to the string are not required for the test suite to pass. PIT's default operator set does not mutate this code and would not necessarily reveal the partial assertion issue. We also found this example when the test suite checked only the exception type and no further details, thus showing weak exception handling within the test suites. In *commons-csv*, PSEUDOSWEEP identified a *hashcode* method whose tests check that it produces both equal and unequal hashcodes. The pseudo-tested statement in *hashcode* applied the same operation to all input, and, therefore, one could categorise it as a partial assertion or a redundant statement. However, we can not ascertain this without further knowledge of the project.

No targeting test (9): We found nine pseudo-tested statements caused by the lack of a test aimed to cover the statement. Other tests executed these statements, but the test suite did not include a test specifically designed for the statement. We identified six pseudo-tested *if* statements, caused by no test executing the *if* condition to "true". This left code within the *if* statement body uncovered. While this coverage deficiency could be identified by JaCoCo [35], only addressing coverage may not in turn address the pseudo-testedness. The other two pseudo-tested statements are unchecked *throw* statements that did not contain tests that checked for them. We could have also classified these two statements as ambiguous exceptions, but since there was no test aimed at them in the first place, we concluded that this is the primary category for the statements.

Unintended exception handling (6): These tests used multiple assertions to check different behaviours within a single test, including an expected exception. If any of these assertions trigger the expected exception (even if unintentionally), the JUnit *expected* exception attribute catches it and the test continues to pass, thus making the statement pseudo-tested. In *asw4j*, we found two examples of tests that used assertions to consecutively call the same method with different inputs. The tests also contained a JUnit *expected* exception annotation attribute meant for the last assertion in the tests, which provided the method with an invalid input. Yet, this annotation caught any exception of the same type thrown by the other assertions. This causes the test to pass when the assertions failed unexpectedly, making the statement pseudo-tested.

The categories of *no targeting assertion*, *partial assertion*, *no targeting tests*, and *unintended exception handling* surface actionable insights that a developer gains from studying these pseudo-tested statements found by PSEUDOSWEEP. For example, adding a test for an unchecked behaviour or adding assertions to check more program state are both intuitive outcomes of identifying pseudo-tested statements. They are also concerns that XMT would not raise, meaning that easy-to-address issues will be left for traditional mutation testing to identify at additional expense or, in the cases where traditional mutation testing does not have an appropriate operator for the statement, the developer may overlook the concern altogether.

Conclusion for RQ4. Pseudo-tested statements within required methods can highlight testing issues that XMT, traditional mutation testing, and code coverage do not detect.

VI. RELATED WORK

DeMillo et al. introduced "program mutation" in 1978, from which all Mutation Testing literature has followed [18].

Mutant Reduction Techniques: Reducing the number of mutants produced and executed to "do-fewer" is an active

research area [37], [38]. For instance, Ammann et al. created a theoretical framework for mutant set minimisation, producing sets considerably smaller than other best-practice approaches [39]. However, Gopinath et al. showed that mutant reduction strategies performed either worse or similar to random sampling, suggesting that these methods need further justification. Gopinath et al. also proposed super-mutants that group mutants for execution and only further dividing the group if it is detected [40]. While our approach is similar to this one in that it generates mutants at the level of both methods and statements, we focus on identifying pseudo-tested elements and they aimed to efficiently create the mutant kill matrix. Finally, Petrović et al. revealed how Google reduces the number of mutants shown to developers by focusing on those that are the most productive and would best motivate test improvements [8], [9], [36]. Similarly, using PSEUDOSWEEP to detect pseudo-tested methods and statements can surface insights that enable developers to enhance their test suites.

Deletion Mutation Operators: Untch investigated the SDL operator by itself and found that, compared to other reduction strategies, it best predicted the traditional mutation score [6]. Deng et al. studied the SDL operator further, adding it to muJava [21] and empirically evaluating the traditional mutation scores achieved by statement-deletion-adequate test suites (i.e., when the test suite killed all SDL mutants). Delamaro et al. expanded the set of deletion operators to include variable, constant, and operator deletion. Delamaro et al. also evaluated the SDL operator and one-op mutation for the C programming language [12]. Durelli et al. evaluated whether equivalent mutants generated by deletion mutation operators were easier to identify than those produced by traditional mutation operators [41]. In the context of education, Balfroid et al. studied XMT as an alternative to regular mutation testing, ultimately finding the traditional method was slightly more effective even though similar numbers of students classified both mutation testing approaches as useful [42].

Higher-order Mutation: Higher-order mutants (HOMS) are compositions of first-order mutants aimed to create a single, harder-to-kill mutant [43]. However, without techniques for improving efficiency, the search for meaningful HOMS is often computationally expensive [44]. Although PSEUDOSWEEP uses a metamutant to instrument the program under test, our approach differs from a HOM-based one since it does not involve running multiple mutants simultaneously but rather applying them sequentially based on prior test executions.

State Coverage: State coverage identifies unchecked outputs by identifying all outputs available to the test in memory at the time of checks [45]. Yet, initial implementations proved 70 times slower than the standard JUnit test runner, limiting its practicality [46]. Further work proposed some generalised state coverage algorithms that offer limited feedback [47].

Direct and Indirect Coverage: Indirect coverage highlights program code not directly covered by any test case [48]. Huo and Clause classify code in the program under test as "indirectly covered" if it is not in a method immediately called by the test suite. Instead of focusing on program code that is indirectly tested, our approach automatically detects both methods and statements that are pseudo-tested since, as an example, private methods cannot be called directly by a test.

Checked Coverage: Checked coverage (CC) detects statements influencing the oracle with dynamic backward slices from test assertions. Notably, Schuler and Zeller found it more sensitive than mutation testing [49]. Hossain et al. defined the "coverage gap" to show developers where to focus and developed a static analysis that provides testers with advice [32]. Ultimately, our approach assesses oracle strength by removing covered elements, whereas CC identifies those contributing to the test outcomes on a dynamic backward slice.

VII. Conclusions and Future Work

An extreme mutation testing (XMT) tool individually deletes method bodies in covered program code and observes whether the test suite detects their absence [1]. XMT labels as "pseudo-tested" any method that it removes without influencing test outcomes and calls a method "required" if its deletion causes a change in test status. Focusing on a finer granularity than XMT, this paper presents the first empirical study of pseudo-testedness in program statements. Using 27 open-source Java projects, the experiments use the PSEUDOSWEEP tool to perform statement deletion, revealing that XMT would overlook 48% of the pseudo-tested statements since they appear in required methods. Pseudo-tested statements were also locations of lower mutation scores, suggesting that such statements are points of test suite weakness. The results also show that PIT's default set of mutation operators generated significantly fewer mutants for certain statement types, meaning that neither traditional nor extreme mutation testing will likely highlight issues within these statements. Finally, a manual analysis of 119 pseudo-tested statements across 30 methods surfaced testing concerns that code coverage, extreme mutation testing, and traditional mutation testing did not fully identify.

Given the promise of this paper's results, it is important to further study the full impact of pseudo-testedness at the statement level, thus motivating both tool improvements and new experiments. After enhancing PSEUDOSWEEP to filter unproductive program elements and better handle test flakiness, we plan to run experiments with additional projects to replicate this paper's experiments, measure the approach's efficiency, and perform studies to discern how PSEUDOSWEEP helps software developers. Combining this paper's contributions with those arising from future work will position PSEUDOSWEEP as a compelling way to detect the pseudo-tested statements neglected by extreme mutation testing, supporting the identification of testing inadequacies that developers can address before committing resources to traditional mutation testing.

Acknowledgments

We thank Firhard Roslan for the mined list of GitHub repositories for the Maven projects. Megan Maton is funded by the EPSRC Doctoral Training Partnership with the University of Sheffield, grant EP/W524360/1. Phil McMinn is supported, in part, by the EPSRC grant "Test FLARE" (EP/X024539/1).

REFERENCES

[1] R. Niedermayr, E. Jürgen, and S. Wagner. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 2016.

[2] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 24(3), 2019.

[3] O. L. Vera-Pérez, M. Monperrus, and B. Baudry. Descartes: A PITest engine to detect pseudo-tested methods. In *Proceedings of the International Conference on Automated Software Engineering*, 2018.

[4] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. Suggestions on test suite improvements with automatic infection and propagation analysis. In *arXiv:1909.04770*, 2019.

[5] R.A. DeMillo and Offutt J. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9), 1991.

[6] R.H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Proceedings of the Annual Southeast Regional Conference*, 2009.

[7] L. Deng, J. Offutt, and N Li. Empirical evaluation of the statement deletion mutation operator. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2013.

[8] G. Petrovic and M. Ivankovic. State of mutation testing at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2018.

[9] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. Practical mutation testing at scale: A view from Google. *IEEE Transactions on Software Engineering*, 48(10), 2022.

[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A practical mutation testing tool for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2016.

[11] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of International Symposium on Software Testing and Analysis*, 2014.

[12] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt. Experimental evaluation of SDL and one-op mutation for C. In *Proceedings of International Conference on Software Testing, Verification and Validation*, 2014.

[13] Maven central repository: https://repo.maven.apache.org/maven2.

[14] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2012.

[15] R. Just, G. M. Kapfhammer, and F. Schweiggert. Major: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering*, 2011.

[16] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014.

[17] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, 2005.

[18] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), 1978.

[19] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2011.

[20] M. E. Delamaro, J. Offutt, and P. Ammann. Designing deletion mutation operators. In *Proceedings of International Conference on Software Testing, Verification and Validation*, 2014.

[21] Y-S. Ma and J. Offutt. Description of muJava's method-level mutation operators. Technical report, Electronics and Telecommunications Research Institute, Korea, 2005, updated 2016.

[22] M. Betka and S. Wagner. Towards practical application of mutation testing in industry — Traditional versus extreme mutation testing. *Journal of Software: Evolution and Process*, 34(11), 2022.

[23] M. Maton, G. M. Kapfhammer, and P. McMinn. PseudoSweep: A pseudo-tested code identifier. In *Proceedings of the International Conference on Software Maintenance and Evolution, Tool Track*, 2024.

[24] Replication package: https://github.com/PseudoTested/icsme-2024-replication-package.

[25] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of International Symposium on Software Testing and Analysis*, 1993.

[26] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the International Workshop on Automation of Software Test*, 2011.

[27] O. Parry, M. Hilton, G. M. Kapfhammer, and P. McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology*, 31(1), 2022.

[28] N. Smith, D. van Bruggen, and F. Tomassetti. *JavaParser: Visited*. LeanPub, 2023.

[29] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser. Do automatic test generation tools generate flaky tests? In *Proceedings of the International Conference on Software Engineering*, 2024.

[30] Y-S. Ma, J. Offutt, and Y-R. Kwon. muJava: An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2), 2005.

[31] Y-S. Ma, J. Offutt, and Y-R. Kwon. muJava: A mutation system for Java. In *Proceedings of the International Conference on Software Engineering*, 2006.

[32] S. Hossain, M. B. Dwyer, S. Elbaum, and A. Nguyen-Tuong. Measuring and mitigating gaps in structural testing. In *Proceedings of the International Conference on Software Engineering*, 2023.

[33] R. Gopinath, I. Ahmed, M-A. Alipour, C. Jensen, and A. Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3), 2017.

[34] A. Shi, J. Bell, and D. Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2019.

[35] JaCoCo: Java code coverage library, 2014. https://www.jacoco.org/jacoco/.

[36] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. Please fix this mutant: How do developers resolve mutants surfaced during code review? In *Proceedings of the International Conference on Software Engineering*, 2023.

[37] A. J. Offutt and R. H. Untch. *Mutation Testing for the New Century*, chapter Mutation 2000: Uniting the Orthogonal. Springer, 2001.

[38] A. V Pizzoleto, F. C Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157, 2019.

[39] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of International Conference on Software Testing, Verification and Validation*, 2014.

[40] R. Gopinath, B. Mathis, and A. Zeller. If you can't kill a supermutant, you have a problem. In *International Workshop on Mutation Testing*, 2018.

[41] V. H. S. Durelli, N. M. De Souza, and M. E. Delamaro. Are deletion mutants easier to identify manually? In *Proceedings of International Workshop on Mutation Testing*, 2017.

[42] M. Balfroid, P. Luycx, B. Vanderose, and X. Devroey. An empirical evaluation of regular and extreme mutation testing for teaching software testing. In *Proceedings of International Workshop on Software Testing Education*, 2023.

[43] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10), 2009.

[44] C.-P. Wong, J. Meinicke, L. Chen, J. P. Diniz, C. Kästner, and E. Figueiredo. Efficiently finding higher-order mutants. In *Proceedings of the Joint Meeting on European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, 2020.

[45] K. Koster and D. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, 2007.

[46] K. Koster. A state coverage tool for JUnit. In *Proceedings of the Companion of the International Conference on Software Engineering*, 2008.

[47] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science*, 2012.

[48] C. Huo and J. Clause. Interpreting coverage information using direct and indirect coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2016.

[49] D. Schuler and A. Zeller. Checked coverage: An indicator for oracle quality. *Software Testing, Verification and Reliability*, 23(7), 2013.