ExLi: An Inline-Test Generation Tool for Java

Yu Liu⋆, Aditya Thimmaiah⋆, Owolabi Legunsen†, Milos Gligoric⋆
⋆The University of Texas at Austin; †Cornell University
⋆Austin, †Ithaca, USA
yuki.liu@utexas.edu,auditt@utexas.edu,legunsen@cornell.edu,gligoric@utexas.edu

ABSTRACT

We present ExLi, a tool for automatically generating inline tests, which were recently proposed for statement-level code validation. ExLi is the first tool to support retrofitting inline tests to existing codebases, towards increasing adoption of this type of tests. ExLi first extracts inline tests from unit tests that validate methods that enclose the target statement under test. Then, ExLi uses a coverage-then-mutants based approach to minimize the set of initially generated inline tests, while preserving their fault-detection capability. ExLi works for Java, and we use it to generate inline tests for 645 target statements in 31 open-source projects. ExLi reduces the initially generated 27,415 inline tests to 873. ExLi improves the fault-detection capability of unit test suites from which inline tests are generated: the final set of inline tests kills up to 24.4% more mutants on target statements than developer written and automatically generated unit tests. ExLi is open sourced at https://github.com/EngineeringSoftware/exli and a video demo is available at https://youtu.be/qaEB4qDeds4.

CCS CONCEPTS

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

KEYWORDS

Inline tests, unit tests, automatic test generation, test carving

ACM Reference Format:
Yu Liu⋆, Aditya Thimmaiah⋆, Owolabi Legunsen†, Milos Gligoric⋆. 2024. ExLi: An Inline-Test Generation Tool for Java. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663817

1 INTRODUCTION

Inline tests were recently proposed for statement-level code (i.e., target statements) validation [13]. Inline tests complement traditional levels of test granularity, such as unit and integration tests, and can help find single-statement bugs [10, 20] that are often missed by unit tests [11]. Statements with harder-to-understand or error-prone logic, such as regular expressions [16], or those that are buried in complicated logic [21], can particularly benefit from inline testing. Section 2 provides a detailed example.

Two frameworks—iTest for Java [7] and `pytest-inline` for Python [14]—were proposed to provide APIs for writing and executing inline tests. Those APIs allow developers to specify an inline test's inputs, expected outputs, and oracles immediately after a target statement that is being tested. Then, these frameworks run each inline test independently in a fresh environment. A prior user study showed that inline tests are straightforward to learn [13]. Also, `pytest-inline` has been integrated into pytest, the most popular Python testing framework [6] and has now been downloaded 6,128 times since March 2023 [18]. Despite these advances, developers must still write inline tests manually.

ExLi [12] was proposed to automatically generate inline tests. ExLi can help to reduce developer time for manually writing inline tests, retrofit inline tests to existing code, grow the dataset of available inline tests for research, and increase the chance for inline test adoption in practice.

ExLi generates inline tests by extracting them from the execution of unit tests for methods that enclose target statements. To do so, ExLi follows a four-step process: (1) analyze the code under test to find target statements, (2) instrument a target statement to collect inputs and outputs during unit-test execution, (3) execute unit tests that cover the target statement, and (4) generate inline tests using the collected inputs as test inputs and collected output as expected output in a test oracle. ExLi currently supports four kinds of target statements: regular expressions, string manipulation, bit manipulation, and stream operations, which were identified in prior work [13] as being likely to benefit from inline testing. More kinds of statements can be added in the future.

The extraction-only approach described above can generate an excessive number of inline tests if unit tests execute a target statement many times with varying inputs. To mitigate this excess, ExLi also utilizes a *coverage-then-mutants based* reduction process to reduce redundancy among extracted inline tests. (One inline test is redundant with respect to another if it does not increase the *coverage* [4] and *mutation score* [8] on the target statement.) ExLi tracks the number of covered instructions on the target statement and its context during unit test executions, recording values that cover instructions that were not previously covered. Also, ExLi mutates the target statement and ensures that each generated inline test kills a unique mutant. If no mutant is generated for a target statement, then ExLi's reduction is based only on coverage.

Improvements over previous prototype. This paper extends ExLi from a prototype to a tool with a more user-friendly interface, to facilitate easier adoption. New support that we add include (1) generating inline tests in docker containers to isolate ExLi from host's file system and reduce flakiness, (2) allowing developers to specify the target statement using its line number, and (3) exposing an interface that allows users to supply their own algorithms for target-statement identification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663817

```
1 class DFAssinaturaDigital{
2  void assinarDocumento(...) {
3   final PrivateKeyEntry keyEntry = getPrivateKeyEntry();
4   final String dn = ((X509Certificate) keyEntry.getCertificate()).
        getSubjectX500Principal().getName();
5   this.getLogger().debug("DN: {}", dn);
6   final String cn = new LdapName(dn).getRdns().stream().filter(rdn ->
        StringUtils.equalsIgnoreCase(rdn.getType(), "CN")).map(val ->
        String.valueOf(val.getValue())).findFirst().orElse("");
7   itest().given(dn, "1.2.840=#1612646965676f,CN=NFe,OU=TI,O=NFe,
        L=Florianopolis,ST=SC,C=BR").checkEq(cn, "NFe");
8   this.getLogger().debug("CN: {}", cn);
9   ...
10  }
11 }
```

Figure 1: An example target statement (line 6) and an ExLi-generated inline test (line 7).

Evaluation. We evaluate ExLi on 645 target statements in 31 open-source projects. We generate 873 inline tests in total. The final set of generated inline tests kills up to 24.4% more mutants on target statements than developer written and automatically generated unit tests combined. That is, ExLi generates inline tests that can improve the fault-detection capability of the test suites from which they are extracted. We make ExLi open source and it is available at https://github.com/EngineeringSoftware/exli.

2 EXAMPLE

Figure 1 shows an example target statement (line 6) and an inline test that ExLi generates (line 7). This example is simplified from the open-source project, `wmixvideo/nfe` [24], a Brazilian electronic invoices management system. Method `assinarDocumento` implements code for digitally signing XML documents. Line 6 extracts the common name (CN) from a distinguished name (DN) in an X.509 certificate. If the distinguished name contains a CN component, then line 6 extracts that component using Java streams. Otherwise, line 6 assigns an empty string to the `cn` variable.

This target statement is worth testing because it utilizes complex stream operations and string manipulation, which can be error-prone or hard to understand [13]. However, writing a unit test to check this target statement is challenging: doing so requires setting up a certificate with a specific DN. Also, the local variable `cn` is not directly accessible from outside the method, making it hard to write assertions for unit tests. So, an inline test is useful in this case.

Inline tests have three parts. First, the "Declare" part—`itest()`—marks a statement as an inline test. Second, the "Assign" part—`given(dn, "1.2.840=#1612646965676f, CN=NFe,OU=TI,O=NFe, L=Florianopolis,ST=SC,C=BR")`—assigns values to target statements' right-hand side variables. Third, the "Assert" part—`checkEq( cn, "NFe")`—specifies a test oracle, including an expected output. The inline test on line 7 assigns values to the variable `dn` and checks whether the target statement returns the expected value of `cn`.

3 FRAMEWORK

Figure 2 shows ExLi's procedure for generating inline tests. The code under test (CUT) is a required input; optional inputs are (1) unit tests and (2) file paths and line numbers of target statements. If unit tests are not provided, ExLi will generate them using Randoop [17] and EvoSuite [3]. If target statements are not provided, ExLi will automatically find them based on a default set of previously defined APIs (step ①): regular expressions, string manipulation, bit manipulation, and stream operations. The final outputs are the inline tests after coverage-then-mutants based reduction, namely ExLi-UM. There are two intermediate outputs: ExLi-Base—all unique inline tests that are collected during unit-test execution (before reduction)—and ExLi-Cov—inline tests that remain after reduction based only on code coverage, but not mutation scores.

3.1 Generating Inline Tests

ExLi's inline test generation phase consists of steps ①, ②, ③, ④, ⑤ and ⑦ in Figure 2. In step ①, `TargetStmtFinder` parses the abstract syntax tree (AST) of the CUT and identifies target statements. Users can extend `TargetStmtFinder` by overriding method `isTargetStmt` to define their own rules for what target statements to find. Then, in step ②, `VariablesFinder` identifies the variables used in each target statement, which will be the input or output variables in the generated inline tests. After that, the `Instrumenter` in step ③ adds code *before* each target statement to collect the values of input variables and *after* each target statement to collect the values of output variables. Then, the `Executor` (step ④) runs unit tests on the instrumented code, and the `Collector` stores (in memory) the *unique* sets of values observed during unit testing (step ⑤). Using the collected sets of values, `InlineTestConstructor` (step ⑦) constructs inline tests. If an input or output value is primitive or String typed, then it is used directly in an inline test. Otherwise, the value is serialized using XStream [15] and the location of the serialized object is used. A generated inline tests that is too long (e.g., it is unreadable or it surpasses Java's 65,536-character limit) is not saved. The default maximum length for each inline test is 500 characters, and the maximum number of inline tests generated for each target statement is 300, but users can adjust these parameters.

3.2 Reducing Inline Tests

ExLi's reduction phase consists of steps ⑥ and ⑧ in Figure 2. While executing unit tests, `CovReducer` (step ⑥) processes each collected set of values and instruction-level coverage information. Only sets of values that increase target coverage or context coverage of a corresponding target statement are kept and sent to `InlineTestConstructor`. The intuition is that if an inline test can increase the instruction coverage of a target statement or statements that follow it, that inline test is more likely to be able to find bugs in the target statement. *Target coverage* is the instruction coverage of the target statement alone. *Context coverage* is the instruction coverage of the *context* of the target statement. The context of a target statement is defined as code between the target statement and the end of its enclosing basic block. For example, the context of the target statement in Figure 1 (line 6) is lines 8 to 10.

To collect target coverage and context coverage, `Instrumenter` (step ③) first wraps the target statement in a try-catch block to ensure that the code coverage is collected, even if the target statement or its context throws an exception. Then, `Instrumenter` modifies the source code to collect coverage at three points. See `collectCov` calls in Figure 3: (1) instruction-level coverage just before the target statement (line 6, `cov1`); (2) instruction-level coverage right

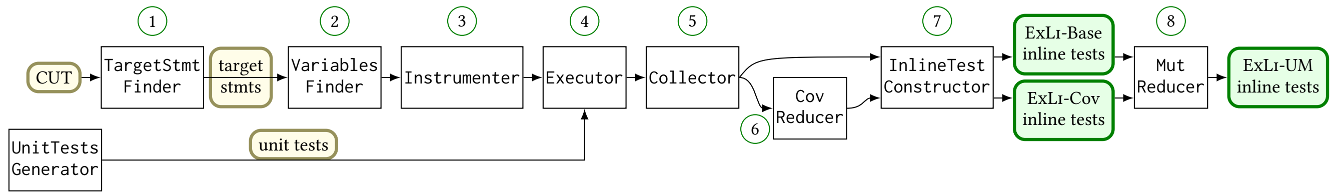


Figure 2: An overview of ExLi and its components.

```
1  void assinarDocumento(...) {
2    final PrivateKeyEntry keyEntry = getPrivateKeyEntry();
3    final String dn = ((X509Certificate) keyEntry.getCertificate()).
         getSubjectX500Principal().getName();
4    this.getLogger().debug("DN: {}", dn);
5    try {
6      collectCov(); // cov1
7      collectInputs(dn);
8      final String cn = new LdapName(dn).getRdns().stream()
9              .filter(rdn -> StringUtils.equalsIgnoreCase(rdn.getType(),
                     "CN"))
10                 .map(val -> String.valueOf(val.getValue()))
11                 .findFirst()
12                 .orElse(""); // target statement
13     collectOutputs(cn);
14     collectCov(); // cov2
15     ... // source code after target statement
16   } finally {
17     collectCov(); }} // cov3
```

Figure 3: Example: how ExLi instruments code in Figure 1.

after the target statement (line 14, cov2); and (3) instruction-level coverage inside the newly added finally block (line 17, cov3).

Step (6) can efficiently reduce the number of inline tests. However, it is possible that it misses some inline tests that could find bugs because it only considers one level of context. For example, CovReducer only considers, as context, instructions after a target statement that is in a loop, but that are within the loop body. However, the loop condition could affect code outside the loop. To address this limitation, MutReducer (step (8)) adds back inline tests collected before reduction if the mutants are not killed by inline tests that remain after CovReducer. Subsequently, MutReducer applies an algorithm to further reduce the number of inline tests, based on mutation scores; it first runs mutation analysis on the CUT and maps killed mutants to each inline test. Here, ExLi supports generating mutants with universalmutator [5] or Major [23] because they are source code level mutators, which can be easily applied to target statements. Then, MutReducer uses one of the four test-suite reduction algorithms [25] implemented by an existing script [22] (the default is Greedy). If a target statement has no mutant, then MutReducer skips it and keeps all inline tests that remained for that statement after applying CovReducer. Finally, ExLi outputs inline tests after coverage-then-mutants based reduction.

4 INSTALLATION AND USAGE

Installation. We provide an ExLi docker image, which can be installed and run using the following commands:

```
~/exli$ docker build -t exli .
~/exli$ docker run -it exli /bin/bash
```

We suggest using Conda [2] to manage packages for ExLi's Python scripts. In the docker container, users can install the Python dependencies by running the following commands:

```
~$ cd exli/python && bash prepare-conda-env.sh
~/exli/python$ conda activate exli
```

Usage. To run ExLi for test generation, a user needs to provide the following parameters: (1) the project name (format: {org}_{repo}), (2) the commit SHA, (3) whether to run Randoop generated tests, (4) the time limit (in seconds) per class for Randoop test generation, (5) whether to run EvoSuite generated tests, (6) the time limit (in seconds) per class for EvoSuite test generation, (7) the seed(s) for Randoop and EvoSuite test generation, and (8) the path to the log file. All parameters other than project name and commit SHA are optional. Here's an example command:

```
~/exli/python$ python -m exli.main run \
  --project_name=Bernardo-MG_velocity-config-tool --sha=26226f5
```

To execute generated inline tests using iTest [7], users provide the following parameters: (1) the project name (format: {org}_{repo}), (2) the commit SHA, (3) the path to the directory with Java files containing inline tests, (4) the path to the directory of parsed inline tests (in JUnit format), (5) the path to the inline tests report, (6) the path to the cached objects, (7) the path to the file that contains the project's dependencies, and (8) the path to the log file. All parameters other than project name and commit SHA are optional. Here's an example command:

```
~/exli/python$ python -m exli.main run_inline_tests \
--project_name=Bernardo-MG_velocity-config-tool --sha=26226f5
```

To reduce inline tests with MutReducer, users can run the following commands, which (1) generate mutants with universalmutator, (2) execute the inline tests on the mutated code, (3) collect a mapping from inline tests to killed mutants, (4) add back inline tests that can kill more mutants than CovReducer-reduced tests, (5) perform inline test reduction using the Greedy algorithm, and (6) add back inline tests whose target statements have no (killed) mutants:

```
~/exli/python$ python -m exli.main generate_mutants \
--project_name=Bernardo-MG_velocity-config-tool --sha=26226f5
~/exli/python$ python -m exli.eval run_tests_with_mutants \
--project_name=Bernardo-MG_velocity-config-tool --sha=26226f5
~/exli/python$ python -m exli.eval get_r2_tests \
--project_name=Bernardo-MG_velocity-config-tool --sha=26226f5\
--mutator=universalmutator --algo=greedy \
--output_path=${HOME}/exli/results/r2/Bernardo-MG_velocity-
     config-tool-26226f5.txt
```

5 EVALUATION

We evaluate ExLi on 31 open-source projects, using the same setup as in previous work [12]. Unlike that work, we exclude 147 target statements that are in automatically generated code (i.e., parser code produced by JavaCC during build time in jkuhnert/ognl [9]).

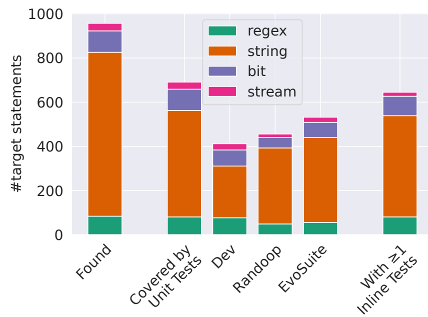


Figure 4: No. of target statements that we find for four kinds of APIs, covered by (all, developer written, Randoop, and EvoSuite) unit tests, and where ExLi generates inline tests.

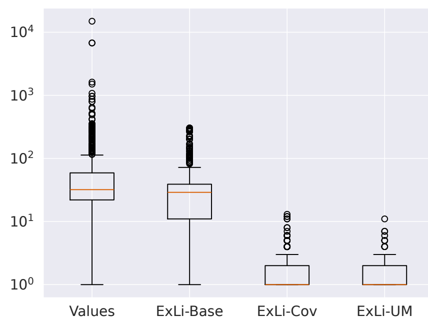


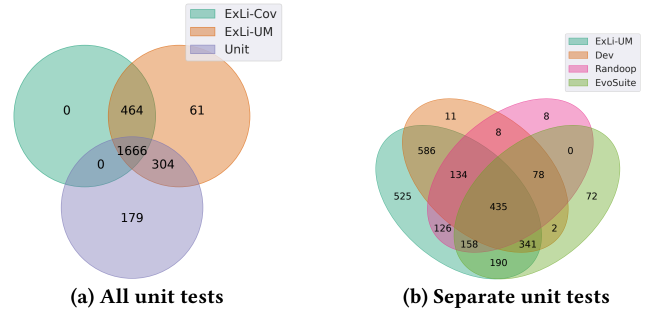
Figure 5: Distribution of inline tests per target statement.

Target statements. Figure 4 shows the distribution of target statements in the 31 projects. ExLi initially identifies 957 target statements: 84 with regular expressions, 742 with string manipulation, 97 with bit manipulation, and 34 with stream operations.

Out of these, 691 target statements are covered by at least one unit test: 412 by at least one developer written unit test, 456 by at least one Randoop-generated unit test, and 532 by at least one EvoSuite-generated unit test. After removing failed inline tests and their corresponding target statements, ExLi generates passing inline tests for 645 target statements: 81 in regular expression, 458 in string manipulation, 88 in bit manipulation, and 18 in stream operations. We use this set in subsequent experiments.

Inline tests. Figure 5 shows the distribution of the number of inline tests per target statement. Additionally, we include the number of unique sets of variable values collected during execution of unit tests (denoted as **Values**), to show the number of inline tests that ExLi would generate without setting the 300 upper limit. The average number of inline tests per target statement for Values, ExLi-Base, ExLi-Cov, and ExLi-UM are 117.5, 42.5, 1.7 and 1.4, respectively. The median values for Values, ExLi-Base, ExLi-Cov and ExLi-UM are 32.0, 29.0, 1.0, and 1.0.

To evaluate the effectiveness of ExLi's reduction, we consider ExLi-Base as the baseline, which generates 27,415 inline tests. ExLi's coverage-based reduction, referred to as ExLi-Cov, reduces the number of inline tests to 1,109, achieving a reduction rate of 96.0%. Subsequently, after performing mutation-based reduction using universalmutator, i.e., ExLi-UM, the number of inline tests is further reduced to 873. This results in a cumulative reduction rate of 96.8%, highlighting the efficiency of ExLi's reduction strategies.



(a) All unit tests **(b) Separate unit tests**

Figure 6: Sets of mutants killed by inline tests and unit tests.

Mutation analysis. Mutation testing is widely used to assess the quality of test suites [1, 19]. We perform mutation analysis using the mutants on the target statements generated by universalmutator.

Figure 6a shows a Venn diagram illustrating the overlap among the sets of mutants killed by all unit tests and inline tests from ExLi-Cov, and ExLi-UM (which is the same as ExLi-Base). All inline tests and unit tests kill 2,674 mutants in total. Figure 6b separates all unit tests into developer written, Randoop generated and EvoSuite generated unit tests and shows their killed mutants. 1,970 mutants are killed by both inline tests and unit tests. The number of mutants killed by inline tests but not by unit tests is 525, and the number of mutants killed by unit tests but not by inline tests is 179. This result shows that inline tests generated by ExLi can improve the fault-detection capability of test suites from which they are extracted.

Performance. Generating inline tests with ExLi-UM takes 1,589.2s on average across projects. This average does not include the per-project times to generate unit tests with Randoop (5,547.7s) and EvoSuite (1,221.2s), and to generate mutants with universalmutator (460.6s); these can be run offline. Other relevant times are: 161.0s to find target statements and instrument code, 1,719.9s to run unit tests, run coverage-based reduction, and generate inline tests, and 968.9s for mutation-based reduction.

6 CONCLUSION

We presented ExLi, a tool for automatic generation of inline tests. The idea behind ExLi is to (1) extract the input values and expected outputs for inline tests while running unit tests for methods that contain the target statement being tested, and (2) reduce redundancy among extracted inline tests while preserving fault-detection capability. We add several new features to make a previous prototype more usable and extensible, and to make the inline test generation process more stable. Our evaluation shows that ExLi generates inline tests for many target statements in several open-source projects and the resulting inline tests improve the fault-detection capability of existing test suites. ExLi is open sourced.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Pengyu Nie, Fred B. Schneider, August Shi, Ayaka Yorihiro, Zhiqiang Zang, and Jiyang Zhang for their comments and feedback. This work is partially supported by an Intel Rising Star Faculty Award, a Google Cyber NYC Institutional Research Award, and the US National Science Foundation under Grant Nos. CCF-2045596, CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2319473.

REFERENCES

- [1] 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 *Automated Software Engineering*. 237–249. https://doi.org/10.1145/3324884.3416667
- [2] Conda 2024. Conda. https://docs.conda.io/projects/conda/en/stable.
- [3] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: Automatic test suite generation for object-oriented software. In *International Symposium on the Foundations of Software Engineering*. 416–419. https://doi.org/10.1145/2025113.2025179
- [4] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis*. 302–313. https://doi.org/10.1145/2483760.2483769
- [5] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering, Demonstrations*. 25–28. https://doi.org/10.1145/3183440.3183485
- [6] Inline Testing Team 2023. pytest-inline on PyPi. https://pypi.org/project/pytest-inline.
- [7] ITest Team. 2023. ITest. https://github.com/EngineeringSoftware/inlinetest/tree/main/java.
- [8] Dennis Jeffrey and Neelam Gupta. 2007. Improving fault detection capability by selectively retaining test cases during test suite reduction. *Transactions on Software Engineering* 33, 2 (2007), 108–123. https://doi.org/10.1109/TSE.2007.18
- [9] jkuhnert Team. 2024. Jkuhnert Ognl. https://github.com/jkuhnert/ognl.
- [10] Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? The ManySStuBs4J dataset. In *International Working Conference on Mining Software Repositories*. 573–577. https://doi.org/10.1145/3379597.3387491
- [11] Jasmine Latendresse, Rabe Abdalkareem, Diego Elias Costa, and Emad Shihab. 2021. How effective is continuous integration in indicating single-statement bugs?. In *International Working Conference on Mining Software Repositories*. 500–504. https://doi.org/10.1109/MSR52588.2021.00062
- [12] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2023. Extracting Inline Tests from Unit Tests. In *International Symposium on Software Testing and Analysis*. 1458–1470. https://doi.org/10.1145/3597926.3598149
- [13] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline tests. In *Automated Software Engineering*. 1–13. https://doi.org/10.1145/3551349.3556952
- [14] Yu Liu, Zachary Thurston, Alan Han, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. pytest-inline: An inline testing tool for Python. In *International Conference on Software Engineering, Demonstrations*. 161–164. https://doi.org/10.1109/ICSE-Companion58688.2023.00046
- [15] LogstashGelf 2022. XStream developer. https://x-stream.github.io/index.html.
- [16] Louis G Michael, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *ASE*. IEEE, 415–426. https://doi.org/10.1109/ASE.2019.00047
- [17] Carlos Pacheco and Michael D Ernst. 2007. Randoop: Feedback-directed random testing for Java. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 815–816. https://doi.org/10.1145/1297846.1297902
- [18] PePy Team. 2024. pytest-inline downloads. https://pepy.tech/project/pytest-inline.
- [19] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical mutation testing at scale: A view from Google. *Transactions on Software Engineering* 48, 10 (2021), 3900–3912. https://doi.org/10.1109/TSE.2021.3107634
- [20] Cedric Richter and Heike Wehrheim. 2022. TSSB-3M: Mining single statement bugs at massive scale. In *International Working Conference on Mining Software Repositories*. 418–422. https://doi.org/10.1145/3524842.3528505
- [21] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In *Automated Software Engineering*. 201–211. https://doi.org/10.1109/ASE.2015.86
- [22] August Shi. 2023. Collection of scripts to conduct test-suite reduction. https://github.com/august782/testsuite-reduction.
- [23] Major Team. 2023. Major mutation framework. https://mutation-testing.org.
- [24] Wmixvideo Team. 2024. Wmixvideo Nfe. https://github.com/wmixvideo/nfe.
- [25] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

Received 2024-01-29; accepted 2024-04-15