EvoSuite at the SBST 2017 Tool Competition

Gordon Fraser*, José Miguel Rojas†, José Campos‡
The University of Sheffield
Sheffield, United Kingdom
{*gordon.fraser, †j.rojas, ‡jose.campos}@sheffield.ac.uk

Andrea Arcuri
Westerdals Oslo ACT, Norway
and University of Luxembourg, Luxembourg
arcand@westerdals.no

***Abstract*—EvoSuite is a search-based tool that automatically generates unit tests for Java code. This paper summarises the results and experiences of EvoSuite's participation at the fifth unit testing competition at SBST 2017, where EvoSuite achieved the highest overall score.**

I. Introduction

The annual unit test generation competition aims to drive and evaluate progress on unit test generation tools. In the 5th instance of the competition at the International Workshop on Search-Based Software Testing (SBST) 2017, two tools, EvoSuite and jTExpert, competed on a set of 69 open-source Java classes. Two other tools, Randoop and T3, and existing open-source test suites were used as baseline. This paper describes the results obtained by the EvoSuite test generation tool [7] in this competition. Details about the procedure of the competition, the technical framework, and the benchmark classes can be found in [21]. In this competition, EvoSuite achieved an overall score of 1457, which was the highest among the competing and baseline tools.

II. About EvoSuite

EvoSuite [7] is a search-based tool [11] that uses a genetic algorithm to automatically generate test suites for Java classes. Given the name of a target class and the full Java classpath (i.e., where to find the compiled bytecode of the class under test and all its dependencies), EvoSuite automatically produces a set of JUnit test cases aimed at maximising code coverage. EvoSuite can be used on the command line, or through plugins for popular development tools such as IntelliJ, Eclipse, or Maven [2].

The underlying genetic algorithm uses test suites as representation (chromosomes). Each test suite consists of a variable number of test cases, each of which is represented as a variable length sequence of Java statements (e.g., calls on the class under test). A population of randomly generated individuals is evolved using suitable search operators (e.g., selection, crossover and mutation), such that iteratively better solutions with respect to the optimisation target are produced. The optimisation target is to maximise code coverage. To achieve this, the fitness function uses standard heuristics such as the branch distance; see [11] for more details. EvoSuite can be configured to optimise for multiple coverage criteria at the same time, and the default configuration combines branch coverage with mutation testing [12] and other basic criteria [18]. Once the search is completed, EvoSuite applies various optimisations to improve the readability of the generated tests. For example, tests are minimised, and a minimised set of effective test assertions is selected using mutation analysis [16]. For more details on the tool and its abilities we refer to [7, 8].

Table I
Classification of the EvoSuite unit test generation tool

Prerequisites	
Static or dynamic	Dynamic testing at the Java class level
Software Type	Java classes
Lifecycle phase	Unit testing for Java programs
Environment	All Java development environments
Knowledge required	JUnit unit testing for Java
Experience required	Basic unit testing knowledge
Input and Output of the tool	
Input	Bytecode of the target class and dependencies
Output	JUnit 4 test cases
Operation	
Interaction	Through the command line, and plugins for IntelliJ, Maven and Eclipse
User guidance	Manual verification of assertions for functional faults
Source of information	http://www.evosuite.org
Maturity	Mature research prototype, under development
Technology behind the tool	Search-based testing / whole test suite generation
Obtaining the tool and information	
License	Lesser GPL V.3
Cost	Open source
Support	None
Does there exist empirical evidence about	
Effectiveness and Scalability	See [11, 13]

The effectiveness of EvoSuite has been evaluated on open source as well as industrial software in terms of code coverage [13, 20], fault finding effectiveness [1, 23], and effects on developer productivity [15, 19].

In the first two and the fourth editions of the unit testing tool competition, EvoSuite ranked first [9, 10, 14], whereas it ranked second in the third one.

III. Competition Setup

The configuration of EvoSuite for the competition is largely based on its default values, since these have been tuned extensively [4]. We used the default set of coverage criteria [18] (e.g., line coverage, branch coverage, branch coverage by direct method invocation, weak mutation testing, output coverage, exception coverage). The use of an archive of solutions [20], which iteratively removes covered goals from the fitness function and stores the corresponding test cases, is now enabled by default in EvoSuite.

A new feature in EvoSuite is the use of Mockito mock classes [3]. After a certain percentage of the search budget has passed, EvoSuite starts considering the use of mock objects instead of real classes. Only branches that cannot be covered without mocks will result in tests with mock objects in the end. We further added frequency based weighting to constants for seeding [22], and included extensions to support Java Enterprise Edition features [5]. Besides these changes, several bug fixes were applied since the last instance of the competition, in particular in relation to non-determinism and flaky tests [6].

Like in previous instances of the competition, we enabled the post-processing step of test minimisation—not for efficiency reasons, but because minimised tests are less likely to break. To reduce the overall time of test generation we included all assertions rather than filtering them with mutation analysis [16], which is a computationally expensive process. The use of all assertions has a negative impact on readability, but this is not evaluated as part of the SBST contest.

Like in the 2016 competition, tools were called with different time budgets. We used the same strategy as for the previous competition [14] to distribute the overall time budget onto the different phases of EvoSuite (e.g., initialisation, search, minimisation, assertion generation, compilation check, removal of flaky tests). That is, 50% of the time was allocated to the search, and the rest was distributed equally to the remaining phases.

IV. Benchmark Results

A. Overall Results

Table II lists the branch coverage and mutation analysis results achieved by EvoSuite on all benchmark classes in the contest. Coverage is generally in the expected range, with clear overall increases for higher time budgets. With the highest time budget of $480s$, the average branch coverage achieved was 66.5% (slightly higher than last year's 65.6%) and the average mutation score was 50.7% (considerably higher than last year's 41.0%).

This year, the contest also included manually written test suites as baseline (only for 63 out of the 69 benchmarks). The results indicate that there is no significant difference in branch coverage between EvoSuite-generated test suites (avg. 50.8%) and manually written ones (avg. 53.8%), with Vargha-Delaney's $A_{12} = 0.45$ and $p = 0.367$. In terms of mutation scores, however, the results show that—unsurprisingly—automatically generated assertions are weaker than manually written ones: the average EvoSuite mutation score for these 63 benchmarks is 36.9%, significantly lower than the average 54.3% obtained by developer-written test suites, with $A_{12} = 0.31$ and $p < 0.001$.

EvoSuite generated 0.4 flaky tests per run on average, much lower than the number of flaky tests produced by the competing and baseline tools (1.3 by T3, 9.9 by jTExpert, and 32.1 by Randoop). We attribute these results—also consistent with previous years' results—to the way EvoSuite handles execution environments during test generation (e.g., controlling the static state of the class under test, mocking interactions with the file system, system calls like `System.in` and `System.currentTimeMillis`, etc.) [6].

B. Challenges

EvoSuite failed to produce any test suites for benchmarks JXPATH-7 and OKHTTP-8, and also struggled often for benchmarks LA4J-3, LA4J-7, BCEL-9 (highlighted in Table II). All executions of EvoSuite for JXPATH-7 failed in the instrumentation phase, where Java's 64k limit on the size of methods was exceeded. A viable fix would be to stop instrumenting before the limit is reached, at the price of limited search guidance; a more effective solution would involve identifying parts of the code that are worth instrumenting. A missing dependency for OKHTTP-8 caused all executions—*for all tools*—to fail. For the other mentioned benchmarks, the failure reasons (in some cases related to sandboxing, mocking and timeouts) will require further investigation.

As expected, both branch coverage and mutation score generally increased with higher time budgets, although in some cases, especially with $budget = 300s$, a decrease was observed for some benchmarks. Having run only three repetitions of the tool per time budget, it is fair to assume the decreases are due to bugs affecting executions by chance. A preliminary investigation on the source of this loss of coverage revealed that memory management (e.g., out-of-memory errors for OKHTTP-2 and RE2J-7), sandboxing (e.g., affecting all executions for LA4J-3 and LA4J-7), and mocking (affecting BCEL-1, BCEL-5 and BCEL-7, for example) are some of the aspects that require attention and debugging in EvoSuite.

Flaky tests continue to be a challenge for test generation. Benchmarks FREEHEP-7 and JXPATH-10 were the ones with the highest number of flaky tests in one run (12 and 14, respectively). In both cases, the minimisation phase timed out and hence EvoSuite reverted the resulting test suite to its previous, unminimised version, which as mentioned before is more likely to break. A *partial minimisation* approach to select and minimise only a reduced subset of tests, or a more efficient minimisation approach based on *delta-debugging* [17], might be worth exploring to alleviate this issue.

As mentioned in the previous section, EvoSuite now uses private API access and functional mocking [3]. In the competition, a total of 20,921 objects were mocked, 1,485 private fields were set and 3,090 private methods were invoked using mocked access, adding up to represent only 1.09% of the total number of lines of code in the EvoSuite-generated test suites

(2,329,361 LOC). Further investigation would be needed to assess the impact of these mocking features on test generation effectiveness for the benchmarks in the competition.

V. Conclusions

This paper reports on the participation of the EvoSuite test generation tool in the 5th SBST Java Unit Testing Tool Contest. With an overall score of 1457, EvoSuite achieved the highest score of all tools in the competition.

To learn more about EvoSuite, visit our Web site:

`http://www.evosuite.org`

Acknowledgments: Many thanks to all the contributors to EvoSuite. This project has been funded by the EPSRC project "GREATEST" (EP/N023978/1), and by the National Research Fund, Luxembourg (FNR/P10/03).

References

[1] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. IEEE, 2017, to appear.

[2] A. Arcuri, J. Campos, and G. Fraser, "Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2016, pp. 401–408.

[3] A. Arcuri, G. Fraser, and R. Just, "Private api access and functional mocking in automated unit test generation," in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2017, to appear.

[4] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Software Engineering (EMSE)*, pp. 1–30, 2013, dOI: 10.1007/s10664-013-9249-9.

[5] ——, "Java enterprise edition support in search-based junit test generation," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 3–17.

[6] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 79–90.

[7] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software." in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.

[8] ——, "EvoSuite: On the challenges of test case generation in the real world (tool paper)," in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[9] ——, "Evosuite at the SBST 2013 tool competition," in *International Workshop on Search-Based Software Testing (SBST)*, 2013, pp. 406–409.

[10] ——, "Evosuite at the second unit testing tool competition." in *Fittest Workshop*, 2013.

[11] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[12] ——, "Achieving scalable mutation-based generation of whole test suites." *Empirical Software Engineering (EMSE)*, 2014.

[13] ——, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[14] ——, "Evosuite at the SBST 2016 tool competition," in *International Workshop on Search-Based Software Testing (SBST)*, 2016, pp. 33–36.

[15] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, p. 23, 2015.

[16] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 278–292, 2012.

[17] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2007, pp. 417–420.

[18] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*. Springer, 2015, pp. 93–108.

[19] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA '15. ACM, 2015, pp. 338–349.

[20] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A Detailed Investigation of the Effectiveness of Whole Test Suite Generation," *Empirical Software Engineering (EMSE)*, 2016, to appear.

[21] U. Rueda and A. Panichella, "Unit testing tool competition - fifth round," in *International Workshop on Search-Based Software Testing (SBST)*, 2017.

[22] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2015.

[23] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

Table II
DETAILED RESULTS OF EVOSUITE ON THE SBST BENCHMARK CLASSES.

Benchmark	Branch Coverage							Mutation Score						
	10s	30s	60s	120s	240s	300s	480s	10s	30s	60s	120s	240s	300s	480s
BCEL-1	14.0%	0.0%	18.0%	36.6%	17.2%	0.0%	0.0%	5.9%	0.0%	14.9%	18.6%	16.8%	0.0%	0.0%
BCEL-10	1.4%	1.4%	41.2%	45.8%	71.3%	76.4%	77.3%	1.3%	1.3%	35.6%	39.6%	62.2%	68.9%	65.3%
BCEL-2	1.1%	1.3%	1.5%	1.9%	2.0%	2.4%	2.1%	0.4%	0.4%	0.7%	1.1%	1.2%	1.5%	1.5%
BCEL-3	57.9%	63.8%	65.7%	73.3%	78.6%	79.8%	52.9%	33.9%	38.2%	29.4%	43.6%	44.4%	49.7%	53.6%
BCEL-4	62.4%	66.2%	74.3%	80.2%	81.0%	82.6%	84.6%	46.3%	45.8%	58.2%	55.9%	59.9%	62.1%	81.9%
BCEL-5	6.9%	6.9%	16.7%	55.1%	53.7%	31.9%	51.9%	4.8%	3.9%	17.4%	50.2%	50.7%	32.4%	48.3%
BCEL-6	55.6%	54.9%	59.9%	66.7%	68.5%	68.5%	68.5%	56.7%	55.3%	66.0%	69.3%	76.7%	76.0%	78.0%
BCEL-7	57.8%	63.7%	37.8%	63.7%	65.2%	21.5%	71.9%	38.1%	43.7%	27.8%	49.2%	65.1%	14.3%	50.8%
BCEL-8	37.5%	37.5%	93.1%	93.1%	97.2%	93.1%	94.4%	21.9%	21.9%	91.7%	93.8%	94.8%	92.7%	94.8%
BCEL-9	0.0%	0.0%	36.9%	0.0%	23.1%	0.0%	0.0%	0.0%	0.0%	33.3%	0.0%	17.5%	0.0%	0.0%
FREEHEP-1	83.6%	83.6%	87.4%	89.1%	92.5%	93.4%	93.4%	21.8%	23.3%	38.8%	37.4%	40.2%	38.2%	37.9%
FREEHEP-10	48.6%	67.6%	82.4%	95.4%	94.0%	95.8%	95.8%	22.0%	22.9%	69.9%	77.4%	72.6%	78.3%	77.1%
FREEHEP-2	0.0%	0.0%	64.2%	95.4%	96.1%	96.8%	96.5%	0.0%	0.0%	4.0%	7.1%	7.4%	7.6%	7.1%
FREEHEP-3	0.0%	0.0%	48.1%	74.8%	86.7%	81.7%	87.1%	0.0%	0.0%	11.1%	14.4%	16.2%	19.2%	21.5%
FREEHEP-4	13.2%	50.8%	49.2%	62.4%	79.4%	70.9%	84.7%	4.2%	18.0%	17.5%	24.7%	31.7%	28.9%	34.4%
FREEHEP-5	32.1%	32.7%	59.1%	68.5%	67.3%	72.4%	75.2%	17.8%	21.5%	46.2%	52.3%	53.3%	53.3%	56.3%
FREEHEP-6	0.0%	0.0%	86.4%	87.7%	88.3%	89.5%	89.5%	0.0%	0.0%	38.0%	46.7%	50.0%	41.3%	46.0%
FREEHEP-7	54.2%	55.6%	85.4%	93.1%	94.4%	99.3%	93.8%	14.2%	14.2%	42.8%	51.2%	61.4%	47.5%	45.9%
FREEHEP-8	25.9%	28.2%	58.8%	82.4%	81.0%	86.1%	91.2%	1.0%	2.4%	10.2%	51.8%	54.3%	43.1%	65.9%
FREEHEP-9	1.4%	1.4%	13.1%	22.5%	49.1%	29.7%	33.8%	0.0%	0.0%	0.0%	0.0%	16.7%	15.8%	16.2%
GSON-1	0.0%	0.0%	46.3%	68.5%	70.4%	70.4%	70.4%	0.0%	0.0%	45.4%	63.0%	65.7%	65.7%	63.9%
GSON-10	43.9%	45.1%	63.0%	62.2%	61.8%	62.2%	62.2%	22.9%	22.1%	54.5%	53.2%	52.4%	53.7%	55.4%
GSON-2	3.3%	9.6%	25.5%	32.4%	36.7%	37.1%	39.6%	2.8%	8.1%	26.4%	34.1%	36.4%	37.2%	42.1%
GSON-3	60.8%	54.6%	76.7%	80.4%	82.9%	85.4%	88.3%	46.7%	47.9%	78.9%	78.2%	82.4%	83.5%	77.8%
GSON-4	3.1%	2.4%	13.1%	18.9%	27.5%	41.7%	46.8%	0.4%	0.6%	8.7%	12.8%	19.5%	17.2%	34.2%
GSON-5	16.0%	21.8%	30.7%	32.7%	38.7%	40.4%	50.7%	17.7%	22.0%	32.8%	33.6%	37.7%	40.3%	44.3%
GSON-6	17.4%	24.4%	63.6%	86.4%	86.4%	91.9%	91.9%	15.3%	19.6%	51.9%	72.0%	61.8%	74.2%	76.6%
GSON-7	42.1%	43.0%	78.1%	82.5%	83.3%	82.5%	81.6%	71.3%	72.0%	90.0%	92.0%	92.7%	91.3%	91.3%
GSON-9	32.8%	40.0%	51.1%	53.9%	46.7%	40.0%	48.3%	17.4%	27.4%	63.7%	64.7%	58.7%	53.7%	60.2%
IMAGE-1	21.3%	27.8%	44.6%	55.4%	53.3%	49.8%	57.0%	7.5%	19.9%	18.4%	21.5%	28.0%	32.1%	36.4%
IMAGE-2	65.0%	70.0%	78.9%	88.3%	87.8%	91.7%	96.1%	21.2%	20.8%	56.6%	62.8%	72.2%	72.6%	72.9%
IMAGE-3	27.8%	29.0%	24.9%	28.5%	26.8%	39.0%	44.4%	8.6%	8.6%	9.2%	9.7%	14.4%	18.1%	21.4%
IMAGE-4	33.6%	35.8%	47.3%	75.9%	79.4%	77.9%	77.9%	16.7%	17.4%	38.0%	63.4%	55.4%	64.1%	71.4%
JXPATH-1	62.4%	65.4%	73.5%	76.8%	77.5%	76.8%	77.4%	41.3%	38.5%	50.3%	68.0%	68.3%	66.7%	67.2%
JXPATH-10	17.6%	23.9%	35.5%	22.3%	52.2%	48.2%	48.0%	12.7%	15.4%	24.2%	19.0%	39.6%	35.4%	42.6%
JXPATH-2	59.7%	67.1%	69.0%	74.4%	70.9%	77.9%	73.6%	40.9%	44.3%	50.8%	65.2%	62.5%	68.2%	62.9%
JXPATH-3	73.4%	83.3%	79.7%	82.8%	83.3%	87.0%	90.1%	28.6%	39.5%	70.1%	86.4%	84.4%	93.2%	89.1%
JXPATH-4	76.8%	78.1%	80.1%	83.0%	85.3%	89.5%	87.6%	68.5%	68.1%	71.4%	75.0%	78.3%	82.6%	83.7%
JXPATH-5	66.0%	73.3%	82.7%	86.7%	88.0%	88.7%	93.3%	31.0%	31.0%	75.4%	89.7%	88.1%	90.5%	95.2%
JXPATH-6	0.0%	0.0%	0.0%	48.6%	84.8%	83.3%	83.3%	0.0%	0.0%	0.0%	51.3%	82.7%	86.7%	78.7%
JXPATH-7	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
JXPATH-8	38.9%	40.2%	52.0%	52.4%	63.6%	60.4%	67.3%	22.1%	21.3%	39.7%	42.8%	53.5%	49.6%	56.6%
JXPATH-9	9.5%	11.9%	45.9%	91.2%	85.4%	88.1%	91.8%	1.1%	3.3%	26.8%	59.8%	55.7%	54.4%	63.1%
LA4J-1	36.2%	68.5%	73.5%	77.2%	77.5%	87.0%	79.9%	23.9%	47.1%	55.6%	55.9%	64.7%	74.5%	55.9%
LA4J-10	31.8%	83.3%	90.9%	92.4%	92.4%	92.4%	92.4%	5.7%	22.6%	49.1%	70.4%	77.4%	69.2%	69.2%
LA4J-2	10.7%	19.4%	19.8%	79.1%	81.4%	76.0%	78.1%	1.0%	2.2%	4.7%	14.8%	10.2%	17.0%	17.8%
LA4J-3	0.0%	15.6%	0.0%	17.1%	38.9%	0.0%	20.3%	0.0%	10.8%	0.0%	10.5%	27.1%	0.0%	7.1%
LA4J-4	0.0%	49.5%	71.7%	68.2%	71.7%	77.8%	80.8%	0.0%	33.1%	48.9%	52.3%	53.0%	43.9%	58.4%
LA4J-5	0.0%	23.5%	32.6%	42.3%	54.4%	52.0%	44.4%	0.0%	14.9%	20.5%	24.1%	34.1%	20.3%	27.2%
LA4J-6	13.3%	61.7%	70.0%	93.3%	95.0%	93.3%	91.7%	4.6%	16.7%	26.9%	78.7%	67.6%	63.0%	67.6%
LA4J-7	0.0%	7.5%	0.0%	21.7%	0.0%	0.0%	21.9%	0.0%	6.1%	0.0%	17.6%	0.0%	0.0%	17.1%
LA4J-8	32.4%	59.8%	74.0%	75.5%	80.9%	77.0%	77.9%	34.2%	58.5%	69.1%	72.7%	82.7%	78.8%	75.5%
LA4J-9	15.0%	39.8%	67.4%	78.5%	77.0%	84.8%	93.1%	0.4%	5.6%	16.6%	24.2%	28.0%	29.4%	22.8%
OKHTTP-1	4.2%	6.4%	5.9%	8.7%	14.4%	17.8%	16.8%	2.2%	3.4%	4.9%	6.5%	10.5%	8.0%	10.5%
OKHTTP-2	33.3%	33.3%	33.3%	50.0%	50.0%	33.3%	50.0%	36.3%	38.2%	32.4%	48.0%	49.0%	32.4%	50.0%
OKHTTP-3	70.7%	75.3%	75.8%	77.3%	74.7%	76.8%	76.3%	36.2%	37.6%	67.6%	69.5%	68.1%	68.5%	68.1%
OKHTTP-4	29.6%	33.9%	54.8%	53.8%	55.9%	56.5%	58.6%	14.1%	17.4%	46.9%	54.5%	55.4%	55.9%	55.9%
OKHTTP-5	0.0%	0.0%	3.6%	11.1%	23.8%	12.7%	33.3%	0.0%	0.0%	5.0%	12.0%	19.0%	11.8%	29.3%
OKHTTP-6	54.8%	72.9%	77.6%	76.2%	83.3%	83.3%	83.8%	9.6%	15.3%	79.7%	75.7%	80.2%	83.6%	81.9%
OKHTTP-7	27.6%	36.5%	24.0%	39.6%	42.7%	26.0%	46.4%	10.1%	12.2%	11.6%	28.6%	34.4%	12.7%	28.0%
OKHTTP-8	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
RE2J-1	30.4%	50.7%	49.2%	51.4%	60.0%	60.5%	65.3%	19.7%	33.2%	32.0%	34.3%	43.8%	42.9%	39.9%
RE2J-2	9.2%	73.8%	77.7%	92.0%	94.0%	95.2%	96.4%	3.9%	32.4%	36.6%	67.3%	69.6%	65.7%	71.6%
RE2J-3	1.8%	1.8%	36.9%	48.8%	48.8%	50.0%	57.7%	3.4%	3.4%	23.0%	41.4%	43.7%	43.7%	48.9%
RE2J-4	81.2%	85.1%	93.1%	94.8%	95.1%	95.5%	96.9%	27.6%	27.1%	91.5%	91.2%	92.0%	89.5%	84.6%
RE2J-5	11.0%	22.7%	57.3%	61.2%	65.9%	58.0%	80.4%	2.9%	3.9%	31.1%	34.6%	36.9%	29.4%	42.7%
RE2J-6	0.0%	0.0%	55.1%	56.7%	83.7%	84.3%	86.8%	0.0%	0.0%	42.0%	51.0%	69.2%	68.9%	66.3%
RE2J-7	15.4%	39.2%	40.3%	53.2%	48.2%	31.4%	51.5%	5.9%	17.0%	15.4%	40.1%	39.2%	27.5%	47.8%
RE2J-8	59.2%	90.8%	89.5%	92.9%	91.5%	92.9%	93.2%	26.7%	44.1%	67.7%	75.6%	81.0%	75.6%	81.2%
Average	27.4%	36.4%	50.9%	60.6%	64.6%	62.3%	66.5%	15.3%	19.8%	36.5%	45.7%	49.6%	46.6%	50.7%