Dependability analysis of Apache Commons Compress

Luca Morelli
l.morelli6@studenti.unisa.it
Università degli Studi di Salerno
Fisciano (SA), Campania, Italy

Daniele Galloppo
d.galloppo@studenti.unisa.it
Università degli Studi di Salerno
Fisciano (SA), Campania, Italy

1 INTRODUCTION

In this report we present the dependability analysis of an open source project: the Apache Commons Compress library. It is a library that implements a large variety of compressors and archivers: the former (un)compress streams that usually store a single entry, the latter deal with archives that contain structured content, like files and directories. The supported formats include: bzip2, gzip, pack200, lzma, xz, Snappy traditional Unix Compress, DEFLATE, DEFLATE64, LZ4, Brotli, Zstandard and ar, cpio, jar, tar, zip, dump, 7z, arj. To conduct the analysis, we forked the Apache repository; all the work we have done can be found here: https://github.com/DG266/commons-compress. The branches of interest are three:

- The "master" branch, which includes everything we have done, except for the automatically generated tests and the web application;
- The "evosuite-randoop" branch, which is a less updated version of the master branch that includes tests generated with two tools, *EvoSuite* and *Randoop*;
- The "docker-web-app" branch, which includes all the code necessary to create and run a very simple Spring web application on a Docker container.

The Docker images for the project can be found here: https://hub.docker.com/r/dg266/commons-compress/tags, the relevant image is the one tagged with "web-app".

Structure of the report. In Section 2 we present the operations related to Docker and the CI/CD aspects of the project. Section 3 presents the code coverage of the library and the tool that we have used to get the coverage report. Section 4 shows the results obtained after running a mutation testing campaign. In Section 5 we present the initial studies about refactoring following SonarCloud reports. Section 6 shows the results of the code refactoring phase, presented in part in the previous section. Section 7 pertains to performance testing. Section 8 describes the operations performed in the context of automatic test case generation. Section 9 shows how we used some specific tools to detect possible vulnerabilities in the project while Section 10 briefly summarizes the considerations and findings of the report.

2 CI/CD & DOCKER

First and foremost, the project was already buildable without any particular issues and various GitHub Actions workflows were already configured in the appropriate YAML files.

After ensuring that the library was buildable, we then proceeded to containerize the project, and to do this, a Dockerfile had to be created. As a first step, we created a Dockerfile that did nothing more than assemble a very simple image and run the JAR of the Apache library with no arguments: we started from a Docker base image that included the Java 17 JDK, then we installed Maven (using the Maven wrapper of the project) and after that we built the JAR file of the project. As a final step, we used the ENTRYPOINT instruction in order to run the JAR file after the container initialization.

However, we encountered a few problems: the Docker image build process failed because of failing tests in the original test suite of the library. These test cases were run exclusively on the Linux operating system, and to solve this issue, we had to make sure that all the instructions in the Dockerfile were executed by a non-root user. As a last step, we switched to a Docker base image that already included a Maven installation, in order to avoid the Maven installation step. We have also created a GitHub Actions workflow that builds and pushes the Docker image on DockerHub; this happens after pushing a commit to our repository.

For what concerns the Spring web application runnable on a Docker container, we used another Dockerfile which is, to a large extent, very similar to the one described above. There's one key difference though: we used a multi-stage build in order to reduce the size of the resulting Docker image. The web-app is very basic: it allows the user to compress a file using the gzip or bzip2 compression method. The web application can be found in the "docker-web-app" branch on the GitHub repository.

3 CODE COVERAGE

The number of lines of code covered by tests has been calculated through the use of JaCoCo (Java Code Coverage), a code coverage report generator for Java projects. The results obtained are presented in Figure 1. Overall, at first glance, it's possible to conclude that this open source project has a good code coverage.

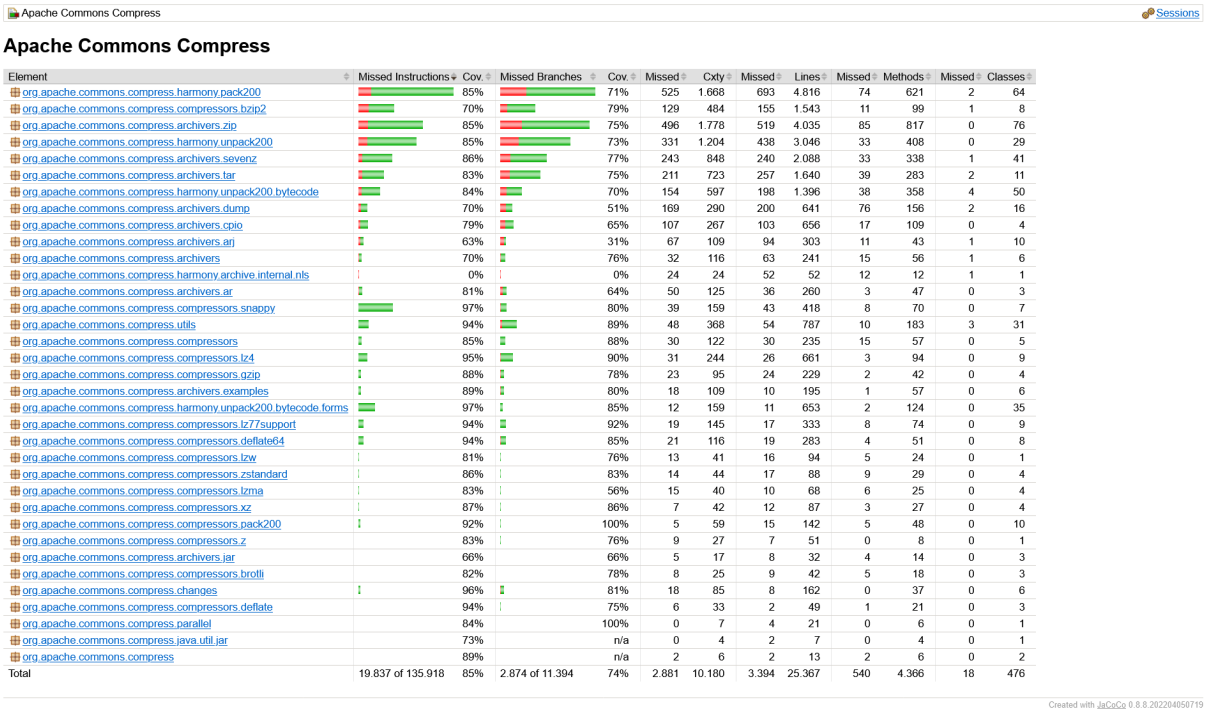
4 MUTATION TESTING

In order to assess the quality of the tests included in the test suite of the Apache Commons Compress library, it has been decided to use PiTest, a mutation testing system.

Although simple in its use, the tool initially reported unreliable results. PiTest was not able to find executable tests, so we had to properly setup the plugin inside the POM of the project.

Having solved this problem, the mutation testing campaign took more than 4 hours to be fully executed, and at the end of this process, it reported satisfactory results. As also presented in Figure 2, the current mutation coverage stands at 71%, an excellent percentage which leads us to the conclusion that the tests of Commons Compress are quite robust.

Fisciano, June 26, 2023, Fisciano (SA)
2023. ACM ISBN 978-1-XXXX-XXXX-X/XX/XX…$999.99
https://doi.org/11.1111/1111111.1111111



Apache Commons Compress — Sessions

Apache Commons Compress

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.compress.harmony.pack200		85%		71%	525	1.668	693	4.816	74	621	2	64
org.apache.commons.compress.compressors.bzip2		70%		79%	129	484	155	1.543	11	99	1	8
org.apache.commons.compress.archivers.zip		85%		75%	496	1.778	519	4.035	85	817	0	76
org.apache.commons.compress.harmony.unpack200		85%		73%	331	1.204	438	3.046	33	408	0	29
org.apache.commons.compress.archivers.sevenz		86%		77%	243	848	240	2.088	33	338	1	41
org.apache.commons.compress.archivers.tar		83%		75%	211	723	257	1.640	39	283	2	11
org.apache.commons.compress.harmony.unpack200.bytecode		84%		70%	154	597	198	1.396	38	358	4	50
org.apache.commons.compress.archivers.dump		70%		51%	169	290	200	641	76	156	2	16
org.apache.commons.compress.archivers.cpio		79%		65%	107	267	103	656	17	109	0	4
org.apache.commons.compress.archivers.arj		63%		31%	67	109	94	303	11	43	1	10
org.apache.commons.compress.archivers		70%		76%	32	116	63	241	15	56	1	6
org.apache.commons.compress.harmony.archive.internal.nls		0%		0%	24	24	52	52	12	12	1	1
org.apache.commons.compress.archivers.ar		81%		64%	50	125	36	260	3	47	0	3
org.apache.commons.compress.compressors.snappy		97%		80%	39	159	43	418	8	70	0	7
org.apache.commons.compress.utils		94%		89%	48	368	54	787	10	183	3	31
org.apache.commons.compress.compressors		85%		88%	30	122	30	235	15	57	0	5
org.apache.commons.compress.compressors.lz4		95%		90%	31	244	26	661	3	94	0	9
org.apache.commons.compress.compressors.gzip		88%		78%	23	95	24	229	2	42	0	4
org.apache.commons.compress.archivers.examples		89%		80%	18	109	10	195	1	57	0	6
org.apache.commons.compress.harmony.unpack200.bytecode.forms		97%		85%	12	159	11	653	2	124	0	35
org.apache.commons.compress.compressors.lz77support		94%		92%	19	145	17	333	8	74	0	9
org.apache.commons.compress.compressors.deflate64		94%		85%	21	116	19	283	4	51	0	8
org.apache.commons.compress.compressors.lzw		81%		76%	13	41	16	94	5	24	0	1
org.apache.commons.compress.compressors.zstandard		86%		83%	14	44	17	88	9	29	0	4
org.apache.commons.compress.compressors.lzma		83%		56%	15	40	10	68	6	25	0	4
org.apache.commons.compress.compressors.xz		87%		86%	7	42	12	87	3	27	0	4
org.apache.commons.compress.compressors.pack200		92%		100%	5	59	15	142	5	48	0	10
org.apache.commons.compress.compressors.z		83%		76%	9	27	7	51	0	8	0	1
org.apache.commons.compress.archivers.jar		66%		66%	5	17	8	32	4	14	0	3
org.apache.commons.compress.compressors.brotli		82%		78%	8	25	9	42	5	18	0	3
org.apache.commons.compress.changes		96%		81%	18	85	8	162	0	37	0	6
org.apache.commons.compress.compressors.deflate		94%		75%	6	33	2	49	1	21	0	3
org.apache.commons.compress.parallel		84%		100%	0	7	4	21	0	6	0	1
org.apache.commons.compress.java.util.jar		73%		n/a	0	4	2	7	0	4	0	1
org.apache.commons.compress		89%		n/a	2	6	2	13	2	6	0	2
Total	19.837 of 135.918	85%	2.874 of 11.394	74%	2.881	10.180	3.394	25.367	540	4.366	18	476

Created with JaCoCo 0.8.8.202204050719

Figure 1: Code coverage of the Apache Commons Compress library.

5 REFACTORING: INITIAL STUDIES

Once the Commons Compress repository was analyzed using SonarCloud CI-based analysis, starting from what was reported by the tool, a preliminary study was carried out: the subdivision of code smells, bugs and security issues into groups, thus being able to define an order of importance for each of them, an order which will then represent the level of urgency in solving these problems.

As already mentioned, the subdivision differs from that carried out by the tool (block, major, critical, minor, info), but rather, the chosen subdivision is: false positives, negligible problems and urgent problems.

For each of them it was possible to extract sub-categories that represent the majority of the problems that compose them. The negligible problems are mainly composed of (i) methods with a wrong scope, (ii) problems related with switch cases, (iii) test cases without assertions and (iv) incorrect use of *system.out* and *system.log* and (v) duplicated literals, while urgent problems that need to be fixed are composed only by (vi) problems correlated with ZipBomb. The reason why a different subdivision was chosen is mainly related to the fact that SonarCloud is not always accurate, so further analysis may be required based on the results obtained. In particular:

(i) accounts for nearly half of the code smells. Although this problem violates information hiding, it is reported mainly for test functions, which is why it would be necessary to analyze case by case, or even define the problem as a false positive, since the modification of the scope of the functions could lead to problems in the testing phase.

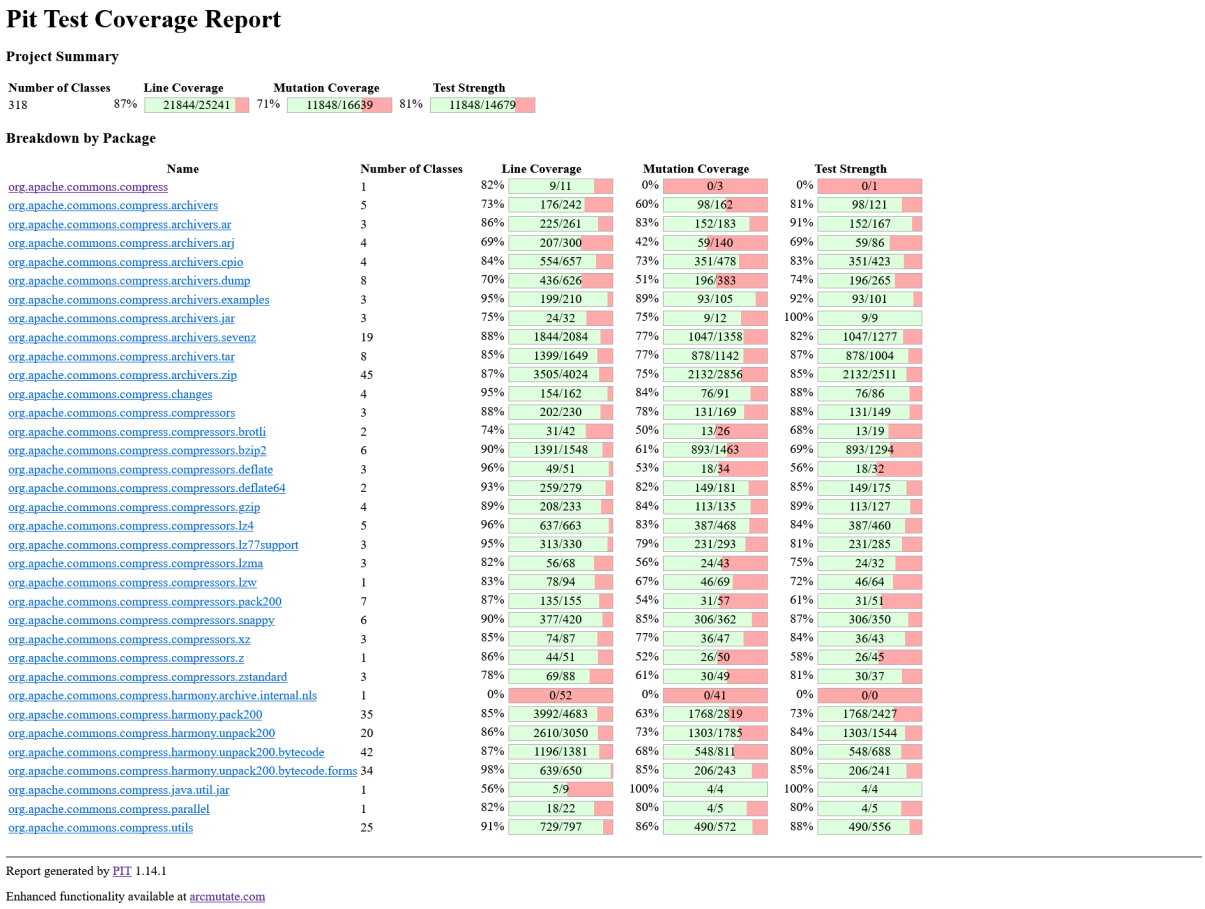
(ii) the use of breaks or default cases are not mandatory but rather recommended. However, it should be specified that the addition of breaks and default conditions could lead to a decrease in the execution time and expended energy.

(iii) a test containing no assertions is an useless test, as it only checks that the code is executed, without checking the result. It would therefore be important to be able to define assertions for these test cases if necessary, alternatively remove them.

(iv) as also reported by SonarCloud, the problem related to this code smell lies in the fact that if a program writes directly to standard output, it would be impossible to log sensitive user data in a secure way, making the record of such data also complex.

(v) with this type of problem, albeit minor, the quality of the code tends to decrease, so actions aimed at solving this error could be performed in order to make the code less complex, smoother and easier to maintain.

(vi) represents the totality of the issues that Commons Compress presents, the problem is in itself complex. Although it represents a type of denial of service attack studied extensively in the literature, the complexity in solving it lies in the definition, using heuristics, of the limit that separates a zip from a zip bomb. In fact, defining a size within which the file would be defined safe could create various problems. Defining a too small value could result in harmless zip folders being classified as zip bombs, while using a too large classification value might cause zip bomb folders to be classified as harmless. To be less doubtful about the classification, the best thing would therefore be to rely on the literature, which defines a zip bomb as a file which, upon decompression, will have a size equal to or greater than a factor equal to 3 times the size of the compressed file.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
318	87%	21844/25241	71%	11848/16639	81%	11848/14679

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.compress	1	82%	9/11	0%	0/3	0%	0/1
org.apache.commons.compress.archivers	5	73%	176/242	60%	98/162	81%	98/121
org.apache.commons.compress.archivers.ar	3	86%	225/261	83%	152/183	91%	152/167
org.apache.commons.compress.archivers.arj	4	69%	207/300	42%	59/140	69%	59/86
org.apache.commons.compress.archivers.cpio	4	84%	554/657	73%	351/478	83%	351/423
org.apache.commons.compress.archivers.dump	8	70%	436/626	51%	196/383	74%	196/265
org.apache.commons.compress.archivers.examples	3	95%	199/210	89%	93/105	92%	93/101
org.apache.commons.compress.archivers.jar	3	75%	24/32	75%	9/12	100%	9/9
org.apache.commons.compress.archivers.sevenz	19	88%	1844/2084	77%	1047/1358	82%	1047/1277
org.apache.commons.compress.archivers.tar	8	85%	1399/1649	77%	878/1142	87%	878/1004
org.apache.commons.compress.archivers.zip	45	87%	3505/4024	75%	2132/2856	85%	2132/2511
org.apache.commons.compress.changes	4	95%	154/162	84%	76/91	88%	76/86
org.apache.commons.compress.compressors	3	88%	202/230	78%	131/169	88%	131/149
org.apache.commons.compress.compressors.brotli	2	74%	31/42	50%	13/26	68%	13/19
org.apache.commons.compress.compressors.bzip2	6	90%	1391/1548	61%	893/1463	69%	893/1294
org.apache.commons.compress.compressors.deflate	3	96%	49/51	53%	18/34	56%	18/32
org.apache.commons.compress.compressors.deflate64	2	93%	259/279	82%	149/181	85%	149/175
org.apache.commons.compress.compressors.gzip	4	89%	208/233	84%	113/135	89%	113/127
org.apache.commons.compress.compressors.lz4	5	96%	637/663	83%	387/468	84%	387/460
org.apache.commons.compress.compressors.lz77support	3	95%	313/330	79%	231/293	81%	231/285
org.apache.commons.compress.compressors.lzma	3	82%	56/68	56%	24/43	75%	24/32
org.apache.commons.compress.compressors.lzw	1	83%	78/94	67%	46/69	72%	46/64
org.apache.commons.compress.compressors.pack200	7	87%	135/155	54%	31/57	61%	31/51
org.apache.commons.compress.compressors.snappy	6	90%	377/420	85%	306/362	87%	306/350
org.apache.commons.compress.compressors.xz	3	85%	74/87	77%	36/47	84%	36/43
org.apache.commons.compress.compressors.z	1	86%	44/51	52%	26/50	58%	26/45
org.apache.commons.compress.compressors.zstandard	3	78%	69/88	61%	30/49	81%	30/37
org.apache.commons.compress.harmony.archive.internal.nls	1	0%	0/52	0%	0/41	0%	0/0
org.apache.commons.compress.harmony.pack200	35	85%	3992/4683	63%	1768/2819	73%	1768/2427
org.apache.commons.compress.harmony.unpack200	20	86%	2610/3050	73%	1303/1785	84%	1303/1544
org.apache.commons.compress.harmony.unpack200.bytecode	42	87%	1196/1381	68%	548/811	80%	548/688
org.apache.commons.compress.harmony.unpack200.bytecode.forms	34	98%	639/650	85%	206/243	85%	206/241
org.apache.commons.compress.java.util.jar	1	56%	5/9	100%	4/4	100%	4/4
org.apache.commons.compress.parallel	1	82%	18/22	80%	4/5	80%	4/5
org.apache.commons.compress.utils	25	91%	729/797	86%	490/572	88%	490/556

Report generated by PIT 1.14.1

Enhanced functionality available at arcmutate.com

Figure 2: PiTest results. We analysed the original test suite which does not include the tests generated with *EvoSuite* and *Randoop*.

6 CODE REFACTORING

As already mentioned in the previous Section, Commons Compress does not have code smells or problems big enough to be classified as a highly refactoring-needed project. Following the previous classifications, only (ii) and (v) were deemed not false positives, adding, to the latter, other problems such as (vii) commented code and (viii) assignments contained in other complex expressions.

(ii) for this problem, a clarification is necessary, since different problems with the switches were reported. The main split was between (a) missing default cases and (b) missing breaks. While for (a) it was possible to carry out refactoring operations, with (b) we ran into problems. In fact, the presence of switch cases without break is, in the context of the Commons Compress project, desired. The structure of these switches is in fact linked to reasons of performance which would decrease if (b) would be fixed (see the concept of loop unrolling and Duff's device).

(v) to solve this problem it was necessary to define global, static and private variables within the classes that had repeated strings, allowing to uniquely define the variable. In this way, although the problem may seem of minor importance despite being reported by SonarCloud as a "critical" level code smell, it could be useful in subsequent stages in the processing of the code, as it is more readable and more maintainable.

(vii) as also reported by SonarCloud, this problem greatly decreases the legibility of the code, so refactoring operations were necessary. However, it should be noted that, in some cases, the commented code blocks have been ignored, as they are not deprecated and unused code but rather useful code for more detailed testing operations.

(viii) is present in a limited number of classes. The resolution of this problem, as (vi), has allowed to have a more readable and understandable code. The variables used for the assignment had, in fact, already been defined, so the extraction of the assignment did not lead to the definition of possibly superfluous variables.

Using SonarQube with the EcoCode plugin, other code smells have been discovered. Of these, the two most reported were (ix) incorrect increment of variables and (x) possible substitutions from *if in sequence* to *switch*.

(ix) the problem reported is related to the increment of the variables following the "i++" format. This format, in fact, creates a temporary variable whereas "++i" does not. Although this does not change the execution times of the algorithm, not creating this

variable would save CPU cycles, which, in a project of this size, could lead to significant results.

(x) an excessive use of else-if in cascade could impact the performances of the software, as the JVM would be forced to compare the conditions. It would therefore be important to carry out, as suggested by SonarQube, a substitution from *if* to *switch*. In this case, however, the problem of false positives arises. In fact, almost all of the reports refer to cases in which, unfortunately, it is not possible to replace the statements. This code smell has therefore been partially resolved.

It is to be specified that (vi) was solved by delegating the task of understanding whether or not the zip is a Zip Bomb to those who unpack the package. An InputStreamStatistics interface is, in fact, implemented by many streams and may be used to provide feedback or detect abnormally high compression ratios that may indicate a ZIP bomb during decompression.

In conclusion, given the above classification, it is possible to define 3 "levels of priority" (1) false positives (2) negligible problems and (3) urgent problems. In (1) the subclasses (i), (iii), (iv) and (vi) are grouped, in (2) the subclasses (ii), (v), (vii) till (x) are grouped while (3) is left empty, as no problem has been classified "urgent" enough to fall into this level, thus varying from the initial classification. The results extracted from SonarCloud are shown in Figure 3.

7 PERFORMANCE TESTING

To assess the current performance of the Commons Compress library, we have decided to use a popular Java benchmarking framework, that is Java Microbenchmark Harness (JMH). It has been used to craft microbenchmarks that can stress the code of the main components of the Apache library, compressors and decompressors, archivers and unarchivers. The microbenchmarks can be found in the "master" branch (although a large part of the development was done in the "microbenchmark" branch, merged in the "master" branch).

In the context of this report, we show the results obtained by running a microbenchmark that stresses the code of a wide variety of compression algorithms implemented in the library. The results are presented in Figure 4. To obtain accurate results, a JAR file containing the microbenchmarks and the necessary dependencies was generated: this file was run on a PC with no programs (except, of course, the operating system) running. You can find more microbenchmarks in the GitHub repository.

8 TEST SUITE GENERATION

In order to increase the code coverage of the Commons Compress library it was necessary to use two specific tools: *EvoSuite* and *Randoop*. The tool that has proven to be most effective was the former, while the latter did not perform as well for a variety of a reasons that we will mention. We have successfully increased code coverage, but not by a substantial amount. The test cases can be found in the "evosuite-randoop" branch on the GitHub repository.

First of all, it should be noted that this library works with Java files and streams, which can complicate the process of automatically generating tests; furthermore, the Apache library also implements the "pack200" algorithm, optimized for compressing JAR archive files. This means that the code that implements the algorithm has to modify the contents of the Java class files in the JAR archive undergoing compression, and it's a delicate matter: it's necessary to handle Java bytecode.

So, the reason why it was not possible to increase the code coverage by a substantial amount, is closely related to the nature of the project. In fact, by running tests that contain file-related operations, it would be possible to rewrite or even delete files of fundamental importance for the functioning of the device on which they are being run.

The first tool we have used to generate tests is *EvoSuite*. Being the code coverage in itself high, the classes that had a coverage lower than 60/70% and that did not have many file-related operations were few. From these, however, a further operation was required. Since some test cases worked with dates, subject to timezone problems, a removal of all those tests that could have been flaky was necessary. In the end, we have generated a total of 460 tests for 11 classes: the code coverage went up by around 1%.

Subsequently, we used *Randoop* to try to further increase the code coverage. First of all, we tried to generate tests for single classes, but this did not yield good results: *Randoop* was not able to get good code coverage values, but we expected this. Afterwards, we tried to run the tool on the JAR file of Commons Compress (as suggested in the official manual), so that it could generate tests for the whole codebase, but this strategy did not work well either: *Randoop* would often crash due to file and stream operations and out-of-memory errors. In the end, we carefully excluded certain classes from the collection that Randoop considers during test generation and we managed to get a working regression test suite. However, there was one last problem to be solved: flaky tests. The tool generated some test cases that used time and date functions to obtain the current time and these tests failed when run on GitHub actions workflows, so we had to remove the classes that relied on these particular Java functions. We have generated a total of 1647 tests for around 100 classes of the library: the code coverage went up by almost 1%.

9 FINDING VULNERABILITIES

The tools that were used to carry out a security analysis of the library are two: *Find Security Bugs*, a plugin for the well-known *SpotBugs*, and *OWASP Dependency-Check*. For both tools we used the available Maven plugins.

The potential vulnerabilities identified by *Find Security Bugs* include several false positives and many of them are not particularly relevant. On the other hand, *OWASP Dependency-Check* detected no publicly disclosed vulnerabilities in the Commons Compress library's dependencies.

To be more specific, and considering only high-priority possible security bugs, *Find Security Bugs* found two potential path traversal vulnerabilities in CLI (Command Line Interface) classes that allow the user to check the contents of a specific archive, and two potential vulnerabilities related to the integrity of the cipher (AES/CBC/NoPadding) used in the encryption of 7z archives (basically, the cipher provides no way to detect that the data has been tampered with by a malicious attacker, the ciphertext is susceptible to alteration). The path traversal vulnerabilities are false positives: the CLI classes need a filename that the user has to enter on the command line and there's no need to filter or sanitize

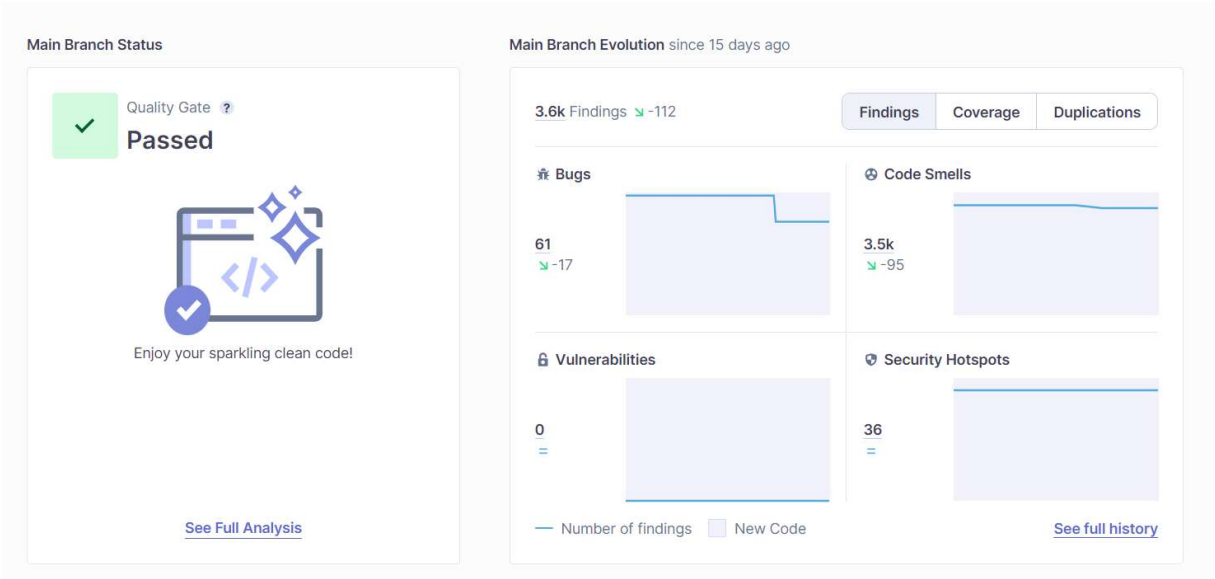


Figure 3: SonarCloud results after applying refactoring.

Benchmark	(bufferSize)	(filePath)	(format)	Mode	Cnt	Score	Error	Units
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	gz	thrpt	5	16796,386	± 139,489	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	bzip2	thrpt	5	1195,685	± 10,614	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	xz	thrpt	5	201,903	± 0,659	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	lzma	thrpt	5	203,190	± 3,937	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	deflate	thrpt	5	15931,611	± 104,330	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	snappy-framed	thrpt	5	11378,017	± 64,839	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	lz4-block	thrpt	5	3327,962	± 30,119	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/bla.tar	lz4-framed	thrpt	5	2102,428	± 10,426	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	gz	thrpt	5	33,877	± 0,381	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	bzip2	thrpt	5	6,479	± 0,062	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	xz	thrpt	5	4,363	± 0,146	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	lzma	thrpt	5	4,427	± 0,027	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	deflate	thrpt	5	33,392	± 0,343	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	snappy-framed	thrpt	5	37,530	± 0,115	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	lz4-block	thrpt	5	18,091	± 0,067	ops/s
GeneralCompressionBenchmark.compressFileBench	8192	./src/test/resources/COMPRESS-592.7z	lz4-framed	thrpt	5	17,648	± 0,109	ops/s

Figure 4: Results of the *compressFileBench* microbenchmark obtained on a computer with an Intel® Core™i5-6400 2.70GHz quad-core processor and 16 GB of RAM. Two files have been used: bla.tar is the smaller one and is 10 KB in size, COMPRESS-592.7z is the larger one and is 1 MB in size. The values under the "Score" column indicate how many times you can compress a given file in one second, using a given compression algorithm.

this filename. As for the other two possible vulnerabilities, the AES/CBC/NoPadding cipher is, in fact, vulnerable, just as indicated by the plugin. However, changing the cipher breaks the implementation of the encryption of 7z archives and consequently this issue should be further investigated. At the moment, in the specific case of this library, we do not think it's an urgent matter.

10 CONCLUSIONS

In conclusion, taking into account the work extensively explained in the previous Sections, Commons Compress has not been classified as a project with a strong need for refactoring or maintenance operations, instead it turned out to be a project with a decent level of dependability.