A Comparison of Closed-Source and Open-Source Code Static Measures

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
e-mail: luigi.lavazza@uninsubria.it

***Abstract*—Most software engineering empirical studies are based on the analysis of open-source code. The reason is that open-source code is readily available, while usually software development organizations do not give access to their code, not even when the purpose is research and the code itself will not be disclosed. As a consequence, the corpus of empirical knowledge is related almost exclusively to open-source software. This poses a quite important question: do the conclusions we draw from the analysis of open-source code apply to close-source code as well? In this paper, a comparison of open-source and closed-source code is performed, to provide some preliminary answers to the question. Specifically, the goal of the paper is to evaluate whether static code measures from open-source code are similar to those obtained from close-source code. To this end, an empirical study was performed, involving closed-source code from two organizations and open-source code from a few different projects. The most popular static code measures were collected using a commercial tool, and compared. The study shows that open-source code measures appear similar to the measures obtained from industrial closed-source code. However, we must note that the study reported here involved just a few industrial projects' measures. Therefore, replications of the work presented here would be very useful.**

Keywords-software code measures; static code measures; open-source code; closed-source code.

I. INTRODUCTION

Software development organizations make their code available to researchers very rarely. This is due to their need for preserving the competitive advantage deriving from code ownership. As a consequence, the great majority of the empirical studies involving source code analyze open-source code, which is freely available. The conclusions reached by these studies are expected to apply to all code, including industrial closed-source code. However, the generalizability of studies based on open-source software relies on the assumption that closed-source software is "similar" to open-source software. Specifically, it is expected that the measures of open-source code are representative of closed-source software as well.

This paper describes an empirical study that aims at verifying if and to what extent code measures of open- and closed-source projects are similar. To this end, we measured a set of industrial closed-source projects and a set of open-source projects and compared the resulting measures.

Based on our results, there are no major differences among the measures collected from industrial and open-source projects. The study reported here has the merit to provide some initial objective evidence that studying open-source projects as representative of closed-source projects is sound.

In this study, the investigation is limited to static code measures for Java projects. Static measures can be defined at various levels of granularity (e.g., method, class, file, subsystem, etc.): here we deal only with method-level measures.

The paper is structured as follows. Section II describes the static code measures investigated in this study. Section III describes the empirical study, whose results are given in Section IV. Section V discusses the results obtained by the study. Section VI discusses the threats to the validity of the study. Section VII accounts for related work. Finally, in Section VIII some conclusions are drawn, and future work is outlined.

II. CODE MEASURES

Since the first high-level programming languages were introduced, several measures were proposed, to represent the possibly relevant characteristics of code [1]. For instance, the size of a software module is usually measured in terms of Lines Of Code (LOC), while McCabe Complexity (also known as Cyclomatic Complexity) [2] was proposed to represent the "complexity" of code, with the idea that high levels of complexity characterize code that is difficult to test and maintain. The object-oriented measures by Chidamber and Kemerer [3] were proposed to recognize poor software design. For instance, modules with high levels of coupling are supposed to be associated with difficult maintenance.

We have considered some of the most popular method-level measures used in the research literature and the software industry: they are listed in Table I.

TABLE I. THE MEASURES COLLECTED VIA SOURCEMETER.

Metric name	Abbreviation
Halstead Calculated Program Length	HCPL
Halstead Volume	HVOL
Maintainability Index (Original version)	MI
McCabe's Cyclomatic Complexity	McCC
Lines of Code	LOC

Halstead proposed several code metrics [4], based on the total number of occurrences of operators N_1, the total number of occurrences of operands N_2, the number of distinct operators η_1 and the number of distinct operands η_2. Halstead Volume (HVOL) is defined as $HVOL = (N_1 + N_2) * log_2(\eta_1 + \eta_2)$; Halstead Calculated Program Length (HCPL) is defined as $HCPL = \eta_1 * log_2(\eta_1) + \eta_2 * log_2(\eta_2)$. McCabe's complexity (McCC) is used to indicate the complexity of a program, being the number of linearly independent paths through a program's source code [2]. The Maintainability Index (MI) [5] is defined as $MI = 171 - 5.2 * ln(HVOL) - 0.23 * (McCC) - 16.2 * ln(LLOC)$, where LLOC is the number of Logical LOC, i.e., the number of non-empty and non-comment code lines.

Interested readers can find additional information concerning the definition and meaning of the selected metrics in the documentation of SourceMeter [6], the tool we used to to collect code measures.

III. The Empirical Study

The empirical study involved closed-source and open-source Java programs. This code was measured, and the collected data were analyzed via well established statistical methods. The dataset is described in Section III-A, while the measurement and analysis methods are described in Section III-B. The results we obtained are reported in Section IV.

A. The Dataset

As already mentioned, obtaining source code from software industries is not easy. Therefore, the closed-source code analyzed within the study is a convenience sample: it is the code that we were able to obtain from industrial developers. The open-source code analyzed within the study is the open-source code used within or together with the analyzed industrial projects. This guarantees a sort of “homogeneity” of code with respect to the required quality.

The projects that supplied the code for the study are listed in Table II, where some descriptive statistics are also given. Because of confidentiality reasons, the names of the industrial projects that supplied the code to be measured are not given: these projects are named Industrial1, Industrial2, Industrial3 (abbreviated Ind1, Ind2 and Ind3 where necessary). Ind1 and Ind2 are client and contract management systems from a large service company, Ind3 is the back-end of a web application. All of the industrial projects aimed to develop software supporting the main business of the owner companies, i.e., none of the considered projects delivered a product to be sold on the market. Also, all projects were developed by external software houses on behalf of the owner companies. Because of confidentiality reasons, the code and the raw measures are not available.

TABLE II. Descriptive statistics of the datasets.

	Number	LOC	LOC per file			
	of files	total	mean	sd	median	range
Industrial1	1507	202299	134	268	91	[1–6851]
Industrial2	280	56419	201	286	93	[3–2336]
Industrial3	1323	250193	189	307	100	[6–3644]
Log4J	1067	126354	118	121	80	[20–1357]
JCaptcha	248	25292	102	99	75	[16–691]
Pdfbox	1215	252158	208	251	125	[21–2966]
JasperReports	3177	533008	168	285	89	[27–4398]
Hibernate	2392	236527	99	127	63	[9–2146]

Table II provides, for each analyzed project, the number of files, the total number of LOC, and the mean, standard deviation, median and range of the LOC per file.

B. The Method

The first phase of the study consisted in measuring the code. We used SourceMeter [6] to obtain the measures.

The second step consisted in selecting the data for the study. We excluded from the study all the methods having unitary McCabe complexity, i.e., the methods that contain no decision points, since those methods would bias the results. In fact, these methods are quite numerous (since they include all the setters and getters) and very small (the excluded methods have mean and median LOC in the [3,6] range).

After removing the methods having unitary McCabe complexity, we got the dataset whose descriptive statistics are given in Table III.

TABLE III. Descriptive statistics of the datasets, after removing methods with unitary McCabe complexity.

	Num.	LOC	LOC per method			
	methods	total	mean	sd	median	range
Industrial1	1342	32654	24	38	15	[3,626]
Industrial2	703	17099	24	25	16	[3,197]
Industrial3	3339	127170	38	61	21	[3,1272]
Log4J	1729	29948	17	17	12	[3,176]
JCaptcha	362	6386	18	15	13	[3,100]
Pdfbox	3738	92679	25	26	16	[3,380]
JasperReports	6815	180104	26	31	17	[3,453]
Hibernate	2746	46505	17	15	12	[3,221]

The third step consisted in comparing the collected measures. To this end, we provide a visual representation of the data via boxplots that describe the distributions, the mean and the median of the measures collected from each project. We also performed statistical analysis:

1) We performed a Kruskal-Wallis test for all the considered metrics, since the conditions for performing ANOVA tests did not hold. As a result, we obtained that, for all metrics, projects are not all equivalent with respect to the considered measure.
2) To explore in detail the differences among projects, we performed Wilcoxon rank sum tests for all project pairs, for all the considered metrics.
3) When a Wilcoxon rank sum test excluded that the measures are equivalent, we evaluated the effect size via Hedge’s g.

In all the performed analysis, we considered the results significant at the usual $\alpha = 0.05$ level.

IV. Results of the Study

This section reports the data collected via the empirical study, grouped according to the type of property being measured.

A. Size measures

Boxplots of LOC measures are given in Figure 1. For the sake of readability, Figure 2 provides the same data, excluding outliers. The mean values are represented as blue diamonds.

The results of the Wilcoxon rank sum tests and Hedges’s g evaluations are given in Table IV. Specifically, a cell includes symbol ‘=’ if the Wilcoxon rank sum test could not exclude that the considered measures are equivalent; otherwise, a cell includes one of the symbols ‘n,’ ‘s,’ ‘m’ for negligible, small

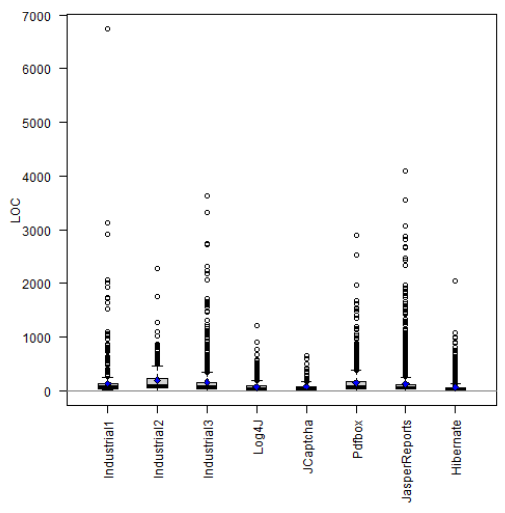

Figure 1. Boxplots of size (measured in LoC) distributions.

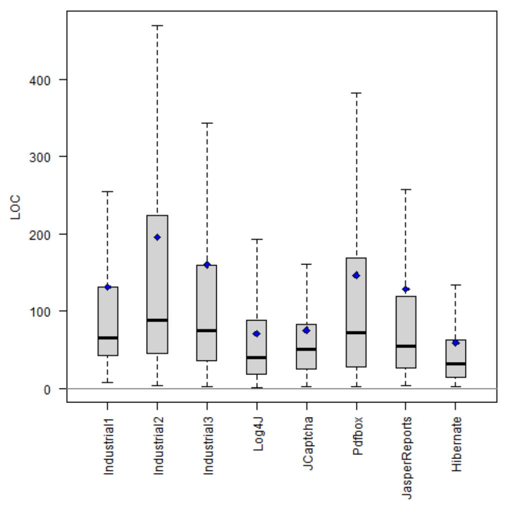

Figure 2. Boxplots of size (measured in LoC) distributions. Outliers omitted.

and medium effect size, respectively (in no case a large effect size was found).

TABLE IV. WILCOXON RANK SUM TEST AND HEDGES'S G RESULTS FOR LOC.

	Ind1	Ind2	Ind3	Log4J	JCaptcha	Pdfbox	JReports	Hibernate
Ind1	–	n	s	s	n	n	n	s
Ind2	n	–	s	s	s	=	n	s
Ind3	s	s	–	s	s	s	s	s
Log4J	s	s	s	–	n	s	s	n
JCaptcha	n	s	s	n	–	s	s	n
Pdfbox	n	=	s	s	s	–	n	s
JReports	n	n	s	s	s	n	–	s
Hibernate	s	s	s	n	n	s	s	–

B. Complexity

Boxplots of McCabe cyclomatic complexity measures are given in Figure 3. For the sake of readability, Figure 4 provides the same data, excluding outliers.

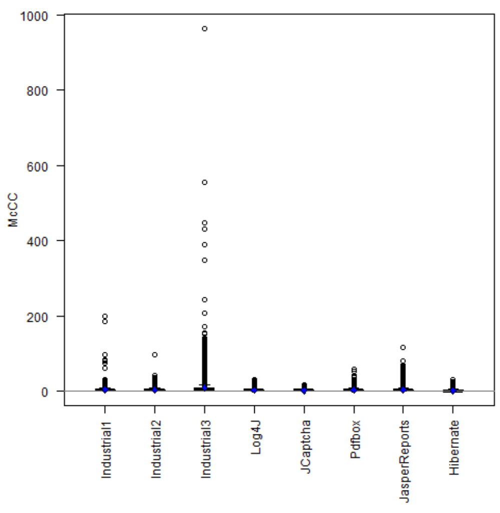

Figure 3. Boxplots of complexity (measured using McCabe cyclomatic complexity) distributions.

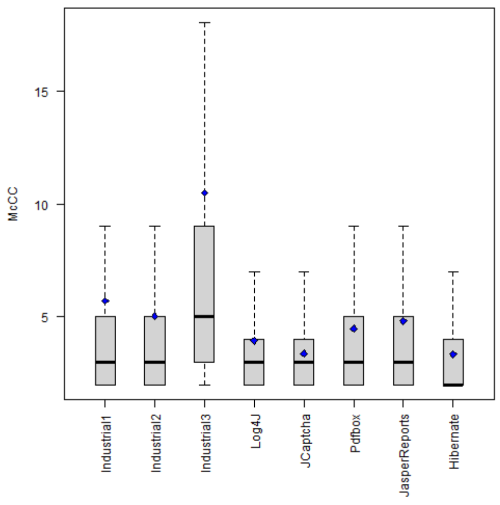

Figure 4. Boxplots of size (measured using McCabe cyclomatic complexity) distributions. Outliers omitted.

The results of the Wilcoxon rank sum tests and Hedges's g evaluations are given in Table V.

TABLE V. WILCOXON RANK SUM TEST AND HEDGES'S G RESULTS FOR MCCABE COMPLEXITY.

	Ind1	Ind2	Ind3	Log4J	JCaptcha	Pdfbox	JReports	Hibernate
Indl1	–	n	n	s	s	n	=	s
Ind2	n	–	s	s	s	=	n	s
Ind3	n	s	–	s	s	s	s	s
Log4J	s	s	s	–	n	n	n	s
JCaptcha	s	s	s	n	–	s	s	n
Pdfbox	n	=	s	n	s	–	n	s
JReports	=	n	s	n	s	n	–	s
Hibernate	s	s	s	s	n	s	s	–

C. Maintainability

Maintainability is measured via the Maintainability Index (MI) [5].

Boxplots of MI measures are given in Figure 5. For the sake of readability, Figure 6 provides the same data, excluding outliers.

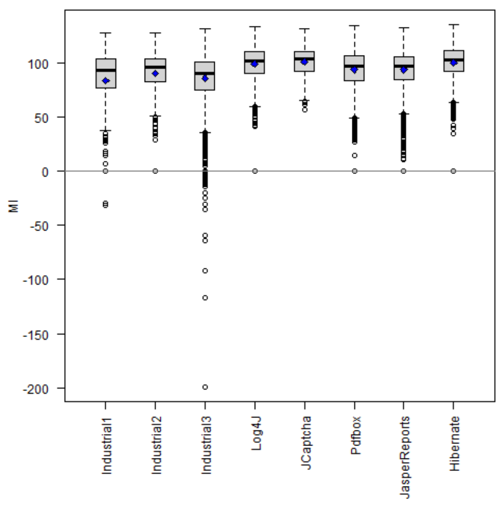

Figure 5. Boxplots of Maintainability Index MI distributions.

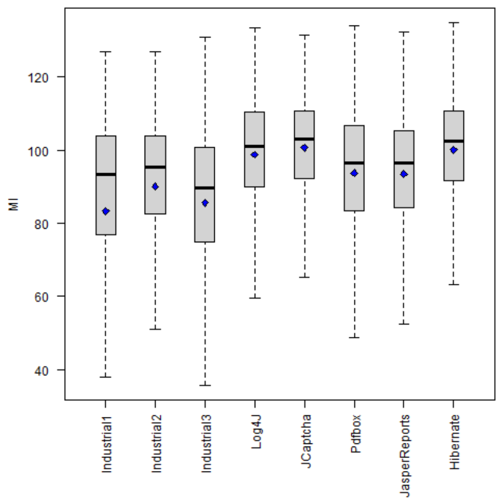

Figure 6. Boxplots of Maintainability Index MI distributions. Outliers omitted.

The results of the Wilcoxon rank sum tests and Hedges's g evaluations are given in Table VI.

TABLE VI. WILCOXON RANK SUM TEST AND HEDGES'S G RESULTS FOR THE MAINTAINABILITY INDEX (MI).

	Ind1	Ind2	Ind3	Log4J	JCaptcha	Pdfbox	JReports	Hibernate
Ind1	–	s	n	m	m	s	s	m
Ind2	s	–	n	s	m	n	n	m
Ind3	n	n	–	m	m	s	s	m
Log4J	m	s	m	–	n	s	s	n
JCaptcha	m	m	m	n	–	s	s	=
Pdfbox	s	n	s	s	s	–	n	s
JasperReports	s	n	s	s	s	n	–	s
Hibernate	m	m	m	n	=	s	s	–

D. Halstead measures

Halstead identified measurable properties of software in analogy with the measurable properties of matter [4]. Among these properties is the volume, measured via the Halstead Volume (HVOL). Boxplots of HVOL measures are given in Figure 7. For the sake of readability, Figure 8 provides the same data, excluding outliers. The results of the Wilcoxon rank sum tests and Hedges's g evaluations are given in Table VII.

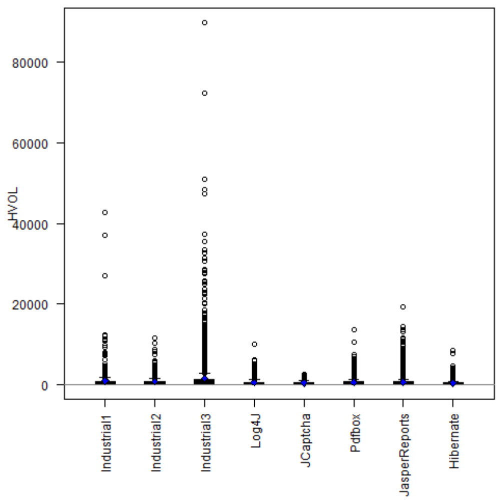

Figure 7. Halstead volume distributions.

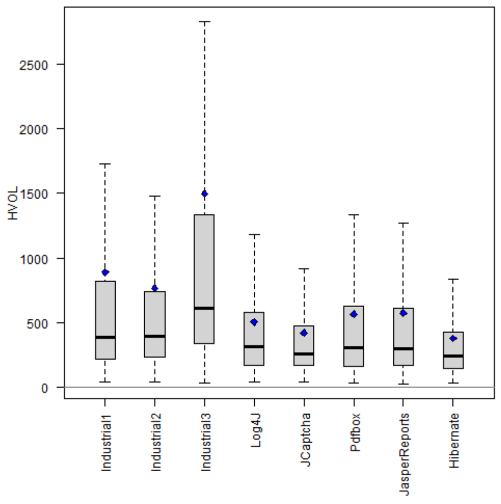

Figure 8. Halstead volume distributions. Outliers omitted.

Boxplots of Halstead Calculated Program Length (HCPL) measures are given in Figure 9. For the sake of readability, Figure 10 provides the same data, excluding outliers. The results of the Wilcoxon rank sum tests and Hedges's g evaluations are given in Table VIII.

V. DISCUSSION

Figures 1 and 2 show that the set of chosen projects are quite homogeneous with respect to size, all projects having

TABLE VII. WILCOXON RANK SUM TEST AND HEDGES'S G RESULTS FOR THE HALSTEAD VOLUME.

	Ind1	Ind2	Ind3	Log4J	JCaptcha	Pdfbox	JReports	Hibernate
Ind1	–	=	n	s	s	s	s	s
Ind2	=	–	s	s	s	s	s	m
Ind3	n	s	–	s	s	s	s	s
Log4J	s	s	s	–	n	n	=	s
JCaptcha	s	s	s	n	–	n	n	n
Pdfbox	s	s	s	n	n	–	n	s
JasperReports	s	s	s	=	n	n	–	s
Hibernate	s	m	s	s	n	s	s	–

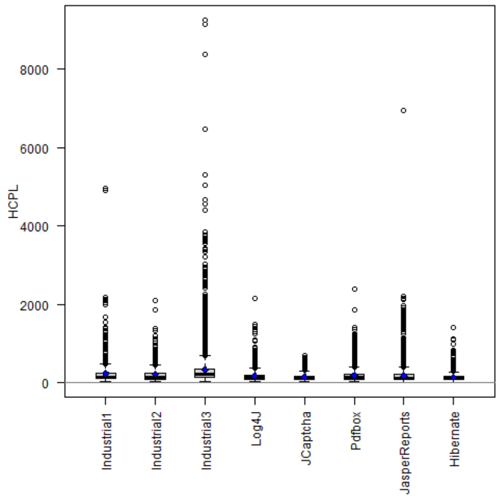

Figure 9. Halstead computed program length distributions.

the great majority of methods no longer than 200 LOC. This homogeneity is confirmed by the effect size evaluations given in Table IV: only negligible and small effect sizes were found.

Similarly, the great majority of methods have McCabe complexity not greater than 5 for all projects, with the only exception of Industrial3. However, also in project Industrial3, only outliers have alarmingly high McCabe complexity. As for LOC, the effect size is at most small, indicating substantial

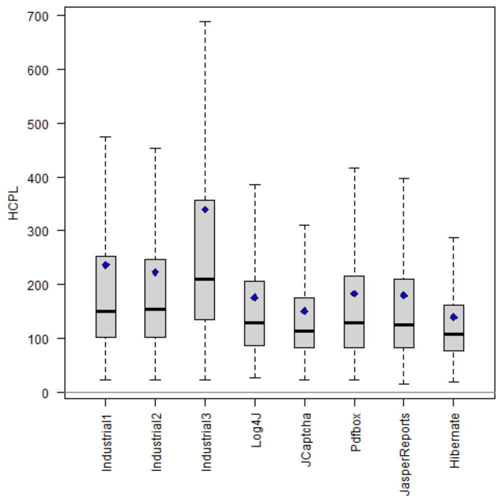

Figure 10. Halstead computed program length distributions. Outliers omitted.

TABLE VIII. WILCOXON RANK SUM TEST AND HEDGES'S G RESULTS FOR THE HALSTEAD COMPUTED PROGRAM LENGTH.

	Ind1	Ind2	Ind3	Log4J	JCaptcha	Pdfbox	JReports	Hibernate
Ind1	–	=	s	s	s	s	s	m
Ind2	=	–	s	s	s	s	s	m
Ind3	s	s	–	s	s	s	s	m
Log4J	s	s	s	–	n	=	n	s
JCaptcha	s	s	s	n	–	n	n	n
Pdfbox	s	s	s	=	n	–	n	s
JasperReports	s	s	s	n	n	n	–	s
Hibernate	m	m	m	s	n	s	s	–

equivalence of the projects' complexity measures.

Concerning the Maintainability Index, Figure 5 shows that Industrial1 and Industrial3 are the only projects that include methods with negative MI; specifically, Industrial3 has several methods with negative MI, some with alarmingly low values. So, even though the situation excluding outliers (Figure 6) seems to indicate a rather homogeneous situation, industrial projects appear to be less maintainable then open-source projects in several cases: according to Table VI, in 8 out of 15 comparisons involving a closed-source and an open-source project, the effect size was medium. Instead, comparisons involving only open-source projects and comparisons involving only closed-source projects revealed at most small effect size.

Finally, we can see that all projects are fairly homogeneous with respect to Halstead volume (Figures 7 and 8 and Table VII). Similar considerations apply for Halstead Computed Program Length (HCPL), with medium effect size differentiating industrial projects only with respect to Hibernate (Table VIII).

In conclusion, we can observe that the analyzed open-source and closed-source code appear sufficiently similar.

VI. THREATS TO VALIDITY

Concerning the application of traditional measures, we used a state-of-the-art tool (SourceMeter), which is widely used and mature, therefore we do not see any threat on this side.

A risk with the type of work presented here is that the code that companies are willing to provide to researchers might differ from the code they will not provide. This is usually due to the desire to "hide" low-quality code. In our case, it is not so: the closed-source code being measured is the complete code being used to build production applications and is representative of the companies' software in general.

Concerning the external validity of the study, as with most Software Engineering empirical studies, we cannot claim that the obtained results are generalizable. Specifically, the limited number of considered projects calls for replications of this study, involving more industrial closed-source code projects.

VII. RELATED WORK

Open-source projects have been compared with closed-source ones multiple times, but usually with respect to external perceivable qualities. In fact, many of the published papers aimed at answering questions like "Should I use this open-source software product or this closed-source one?" These papers considered issues like reliability, speed and effectiveness of defect removal, evolution, security, etc.

Bachmann and Bernstein [7] surveyed five open source projects and one closed source project to evaluate the quality and characteristics of data from bug tracking databases and version control system log files. Among other things, they discovered a poor quality in the link rate between bugs and commits.

The debate on the security of open-source software compared to that of closed-source software have produced several studies. This is due not only to the relevance of the problem, but also to the fact that security issues concerning closed-source software are publicly available, even when the source code is not.

Schryen and Kadura [8] analyzed and compared published vulnerabilities of eight open-source software and nine closed-source software packages. They provided an empirical analysis of vulnerabilities in terms of mean time between vulnerability disclosures, the development of disclosure over time, and the severity of vulnerabilities.

Schryen [9] also investigated empirically the patching behavior of software vendors/communities of widely deployed open-source and closed-source software packages. He found that it is not the particular software development style that determines patching behavior, but rather the policy of the particular software vendor.

Paulson et al. [10] compared open- and closed-source projects to investigate the hypotheses that open-source software grows more quickly, that creativity is more prevalent in open-source software, that open-source projects succeed because of their simplicity, that defects are found and fixed more rapidly in open-source projects.

As opposed to the papers mentioned above, here a fairly systematic comparison of code measures is proposed. Previously, MacCormack et al. compared the structure of an open-source system (Linux) an a closed-source system (Mozilla) [11]. With respect to our work, they evaluated just one code property (modularity) for a single pair of products.

A comparison based on code metrics involving multiple open-source and closed-source projects [12] was performed from a different point of view and using different techniques: the authors modified the Least Absolute Deviations technique where, instead of comparing metrics data to an ideal distribution, metrics from two programs are compared directly to each other via a data binning technique.

VIII. Conclusions

Open-source projects provide the code used in many empirical studies. The applicability of the results of these studies to software projects in general, i.e., including closed-source projects, is questionable, in that we are not sure that open-source code is representative of closed-source code as well.

To address this issue, in this paper, a comparison of open-source and closed-source code is performed. Specifically, static code measures from five open-source projects were compared to those obtained from three close-source projects. The study—which addressed only Java code—shows that some of the most well-known static code measures appear similar in open-source and in industrial closed-source products.

However, we recall that the study reported here involved just a few industrial projects' measures, because getting access to industrial code is not easy. Hence, the presented analysis should be regarded as a preliminary results, which needs replications before it can be considered valid in general.

Acknowledgment

The work reported here was partly supported by Fondo per la Ricerca di Ateneo, Università degli Studi dell'Insubria.

References

[1] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[2] T. J. McCabe, "A complexity measure", *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[4] M. H. Halstead, *Elements of software science*. Elsevier North-Holland, 1977.

[5] K. D. Welker, P. W. Oman, and G. G. Atkinson, "Development and application of an automated source code maintainability index", *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127–159, 1997.

[6] *SourceMeter*, https://www.sourcemeter.com/, [retrieved August, 2024].

[7] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: A historical view on open and closed source projects", in *Proceedings of the Joint int. ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009, pp. 119–128.

[8] G. Schryen, "Security of open source and closed source software: An empirical comparison of published vulnerabilities", *AMCIS 2009 Proceedings*, p. 387, 2009.

[9] G. Schryen, "A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors", in *2009 Fifth International Conference on IT Security Incident Management and IT Forensics*, IEEE, 2009, pp. 153–168.

[10] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products", *IEEE transactions on software engineering*, vol. 30, no. 4, pp. 246–256, 2004.

[11] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code", *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006.

[12] B. Robinson and P. Francis, "Improving industrial adoption of software engineering research: A comparison of open and closed source software", in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–10.