Evaluation of Rust code complexity and maintainability

Luca Ardito Corresp., 1 , **Luca Barbato** 2 , **Riccardo Coppola** 1 , **Michele Valsesia** 1

1 Department of Control and Computer Engineering, Polytechnic Institute of Turin, Torino, Piemonte, Italia

2 Luminem, Torino, Piemonte, Italia

Corresponding Author: Luca Ardito
Email address: luca.ardito@polito.it

Rust is an innovative programming language initially implemented by Mozilla, developed to ensure high performance, reliability, and productivity. The final purpose of this study consists of applying a set of common static software metrics to programs written in Rust to assess the size, structure, complexity, and maintainability of the language. To that extent, we selected nine different implementations of algorithms available in different languages. We computed a set of metrics for Rust, comparing them with the ones obtained from C and a set of object-oriented languages: C++, Python, JavaScript, TypeScript. To parse the software artifacts and compute the metrics, we have leveraged a tool called rust-code-analysis, that we extended with a software module, written in Python, with the aim of uniforming and comparing the results. The Rust code had an average verbosity in terms of the raw size of code. It exposed the most structured source organization in terms of the number of methods. Rust code had a better Cyclomatic Complexity, Halstead Metrics, and Maintainability Indexes than C and C++ but performed worse than the other considered object-oriented languages. Lastly, the Rust code exhibited the lowest Cognitive Complexity of all languages. The collected measures prove that the Rust language has average maintainability compared to a set of popular languages. It is more easily maintainable and less complex than the C and C++ languages, which can be considered syntactically similar. These results, paired with the memory safety and safe concurrency characteristics of the language, can encourage wider adoption of the language of Rust in substitution of the C language in both the open-source and industrial environments.

Evaluation of Rust Code Complexity and Maintainability

Luca Ardito[1], **Luca Barbato**[2], **Riccardo Coppola**[1], **and Michele Valsesia**[1]

[1]**Politecnico di Torino**
[2]**Luminem**

Corresponding author:
Luca Ardito[1]

Email address: luca.ardito@polito.it

ABSTRACT

Rust is an innovative programming language initially implemented by Mozilla, developed to ensure high performance, reliability, and productivity.
The final purpose of this study consists of applying a set of common static software metrics to programs written in Rust to assess the size, structure, complexity, and maintainability of the language.
To that extent, we selected nine different implementations of algorithms available in different languages. We computed a set of metrics for Rust, comparing them with the ones obtained from C and a set of object-oriented languages: C++, Python, JavaScript, TypeScript. To parse the software artifacts and compute the metrics, we have leveraged a tool called *rust-code-analysis*, that we extended with a software module, written in Python, with the aim of uniforming and comparing the results.
The Rust code had an average verbosity in terms of the raw size of code. It exposed the most structured source organization in terms of the number of methods. Rust code had a better Cyclomatic Complexity, Halstead Metrics, and Maintainability Indexes than C and C++ but performed worse than the other considered object-oriented languages. Lastly, the Rust code exhibited the lowest Cognitive Complexity of all languages.
The collected measures prove that the Rust language has average maintainability compared to a set of popular languages. It is more easily maintainable and less complex than the C and C++ languages, which can be considered syntactically similar. These results, paired with the memory safety and safe concurrency characteristics of the language, can encourage wider adoption of the language of Rust in substitution of the C language in both the open-source and industrial environments.

1 INTRODUCTION

Software maintainability is defined as the ease of maintaining software during the delivery of its releases. It is an integrated software measure that encompasses some code characteristics, such as readability, documentation quality, simplicity, and understandability of source code [Aggarwal et al. (2002)]. Also, maintainability is a crucial factor in software products economic success. It is commonly accepted in the literature that the most considerable cost associated with any software product over its lifetime is the maintenance cost [Zhou and Leung (2007)]. Hence, many practices have consolidated in software engineering research and practice to enhance this property, and many metrics have been defined to provide a quantifiable and comparable measurement for it [Nuñez-Varela et al. (2017)].

The academic and industrial practice has also provided multiple examples of tools that can automatically compute software metrics on source codes developed in many different languages [Sarwar et al. (2008)]. Several frameworks have also been described in the literature that leverage combinations of software code metrics to predict or infer the maintainability of a project [Kaur et al. (2014b)].

However, the benefit of the massive availability of metrics and tooling for their computation is contrasted by the constant emergence of novel programming languages in the software development community. In most cases, the metrics have to be readapted to take into account newly defined syntaxes, and existing metric-computing tools cannot work on new languages due to the unavailability of parsers and metric extraction modules. For recently developed languages, the unavailability of appropriate tooling unavailability represents an obstacle for empirical evaluations on the maintainability of the code developed using them.

This work provides a first evaluation of the maintainability of Rust, a newly emerged programming language similar in characteristics to C++, that has been developed with the premises of providing better maintainability, memory safety, and performance [Matsakis and Klock (2014)]. To this purpose, we (i) developed a tool to compute maintainability metrics that support this language; (ii) developed a set of scripts to arrange the computed metrics into a comparable JSON format; (iii) executed a small-scale experiment by computing maintainability metrics for a set of programming languages, including Rust, analyzing and comparing the final results. To the best of our knowledge, no existing study in the literature has provided maintainability computations for the Rust language and the relative comparisons with other languages.

The remainder of the manuscript is structured as follows: Section 2 provides background information about the Rust language and presents a brief review of state-of-the-art tools available in the literature for the computation of maintainability metrics; Section 3 describes the methodology used to conduct our experiment, along with a description of the developed tools and scripts, in addition to the experimental subjects we used for our evaluation; Section 4 presents and discusses the collected metrics; Section 5 describes the threats to the validity of this study; Section 6 concludes the paper and discusses possible future directions of this study.

2 BACKGROUND AND RELATED WORK

This section provides background information about the Rust language characteristics, studies in the literature that analyzes its advantages, and the list of available tools present in literature to measure quality and maintainability metrics.

2.1 The Rust programming language

Rust is an innovative programming language initially developed by Mozilla and is currently maintained and improved by the Rust Foundation[1].

The main goals of the Rust programming language are: memory-efficiency, with the abolition of garbage collection, with the final aim of empowering performance-critical services running on embedded devices, and easy integration with other languages; reliability, with a rich type system and ownership model to guarantee memory-safety and thread-safety; productivity, with integrated package managers and build tools.

Rust is compatible with multiple architectures, and it is quite pervasive in the industrial world. Many companies are currently using Rust in production today for fast, low-resource, cross-platform solutions: for example, software like Firefox, Dropbox, and Cloudflare use Rust.

The Rust language has been analyzed and adopted in many recent studies from academic literature. Uzlu et al. pointed out the appropriateness of using Rust in the Internet of Things domain, mentioning its memory safety and compile-time abstraction as crucial peculiarities for the usage in such domain [Uzlu and Şaykol (2017)]. Balasubramanian et al. show that Rust enables system programmers to implement robust security and reliability mechanisms more efficiently than other conventional languages [Balasubramanian et al. (2017)]. Astrauskas et al. leveraged Rust's type system to create a tool to specify and validate system software written in Rust [Astrauskas et al. (2019)]. Koster mentioned the speed and high-level syntax as the principal reasons for writing in the Rust language the Rust-Bio library, a set of safe bioinformatic algorithms [Köster (2016)]. Levy et al. reported the process of developing an entire kernel in Rust, with a focus on resource efficiency [Levy et al. (2017)]. Such common usages of Rust in such low-level applications encourage thorough analyses of the quality and complexity of a code with Rust.

2.2 Tools for measuring code maintainability metrics

Several tools have been presented in academic works or are commonly used by practitioners to measure maintainability metrics for software written in different languages.

In our previous works, we conducted a Systematic Literature review that led us to identify fourteen different open-source tools that can be used to compute a large set of different maintainability metrics

[1] https://www.rust-lang.org/

Table 1. Languages supported by the metrics tools

Language	CBR Insight	CCFinderX	CKJM	CodeAnalyzers	Halstead Metrics Tool	Metrics Reloaded	Squale
C	x	x	x	x		x	x
C++	x	x		x	x		x
C#	x	x		x			
Cobol	x	x		x			x
Java	x	x	x	x	x	x	
Rust							
Others	x			x			

[Ardito et al. (2020)]. In the review, it is found that the following set of open-source tools is able to cover most of the maintainability metrics defined in the literature, for the most common programming languages: *CBR Insight*, a tool based on the closed-source metrics computation Understand framework, that aims at computing reliability and maintainability metrics [Ludwig and Cline (2019)]; *CCFinderX*, a tool tailored for finding duplicate code fragments Matsushita and Sasano (2017); *CKJM*, a tool to compute the C&K metrics suite and method-related metrics for Java code [Kaur et al. (2014a)]; *CodeAnalyzers*, a tool supporting more than 25 software maintainability metrics, that covers the highest number of programming languages along with CBR Insight [Sarwar et al. (2008)]; *Halstead Metrics Tool*, a tool specifically developed for the computation of the Halstead Suite [Hariprasad et al. (2017)]; *Metrics Reloaded*, able to compute many software metrics for C and Java code either in a plug-in for IntelliJ IDEA or through command line [Saifan et al. (2018)]; *Squale*, a tool to measure high-level quality factors for software and measuring a set of code-level metrics to predict economic aspects of software quality [Ludwig et al. (2017)].

In Table 1, we report the principal programming languages supported by the tools. For the sake of conciseness, we reported as rows in the table, only the languages that were supported by at least two of the tools. With this comparison, we find that none of the considered tools is capable of providing metric computation facilities for the Rust language.

As additional limitations of the identified set of tools, we found out that the tools do not provide complete coverage of the most common metrics for all the tools (e.g., the Halstead Metric suite is computed only by the Halstead Metrics tool), and in some cases, (e.g., CodeAnalyzer), the number of metrics is limited by the type of acquired license. Also, some of the tools (e.g., MetricsReloaded) appear to have been discontinued by the time of the writing of this article.

3 PROCEDURE

This section reports goal, research questions, metrics, and procedures adopted for the study we conducted.

To report the study goal, we follow the Goal Question Metric (GQM) template, as summarized in Table 2. Following the template, the goal of our evaluation can be expressed as

Analyze and evaluate the characteristics of the Rust programming language, focusing on maintainability measurements, measured in the context of open-source algorithms, and interpreting the results from developers and researchers standpoint.

3.1 Research Questions and Metrics

In this subsection, we describe the research questions that guided the definition of the experiment. We identified four different aspects that deserve to be analyzed for code written in Rust programming language.

Table 2. Goal Question Metric template for the study

Object of Study	Rust programming language
Purpose	Evaluation
Focus	Maintainability
Stakeholder	Developers, researchers
Context factors	Open-source algorithms

Table 3. List of metrics used in this study

RQ	Acronym	Name	Description
RQ1	SLOC	Source Lines of Code	It returns the total number of lines in a file
	PLOC	Physical Lines of Code	It returns the total number of instructions and comment lines in a file
	LLOC	Logical Lines of Code	It returns the number of logical lines (statements) in a file
	CLOC	Comment Lines of Code	It returns the number of comment lines in a file
	BLANK	Blank Lines of Code	Number of blank statements in a file
RQ2	NOM	Number of Methods	It returns the number of methods in a source file
	NARGS	Number of Arguments	It counts the number of arguments for each method in a file
	NEXITS	Number of Exit Points	It counts the number of exit points of each method in a file
RQ3	CC	McCabe's Cyclomatic Complexity	It calculates the code complexity examining the control flow of a program; the original McCabe's definition of cyclomatic complexity is the the maximum number of linearly independent circuits in a program control graph [Gill and Kemerer (1991)]
	COGNITIVE	Cognitive Complexity	It is a measure of how difficult a unit of code is to intuitively understand, by examining the cognitive weights of basic software control structures [Jingqiu Shao and Yingxu Wang (2003)]
	Halstead	Halstead suite	A suite of quantitative intermediate measures that are translated to estimations of software tangible properties, e.g. volume, difficulty and effort (see Table 4 for details)
RQ4	MI	Maintainability Index	A composite metric that incorporates a number of traditional source code metrics into a single number that indicates relative maintainability (see Table 5 for details about the considered variants) [Welker (2001)]

Table 4. The Halstead Metrics Suite

Measure	Symbol	Formula
Program length	N	$N = N1 + N2$
Program vocabulary	η	$\eta = \eta 1 + \eta 2$
Volume	V	$V = N * log_2(\eta)$
Difficulty	D	$D = \eta 1/2 * N2/\eta 2$
Program Level	L	$L = 1/D$
Effort	E	$E = D * V$
Estimated Program Length	H	$H = \eta 1 * log_2(\eta 1) + \eta 2 * log_2(\eta 2)$
Time required to program (in seconds)	T	$T = E/18$
Number of delivered bugs	B	$B = E^{2/3}/3000$
Purity Ratio	PR	$PR = H/N$

Table 5. Considered variants of the MI metric

Acronym	Meaning	Formula
MI_O	Original Maintainability Index	$171.0 - 5.2 * ln(V) - 0.23 * CC - 16.2 * ln(SLOC)$
MI_{SEI}	MI by Software Engineering Institute	$171.0 - 5.2 * log_2(V) - 0.23 * CC - 16.2 * log_2(SLOC) + 50.0 * sin(\sqrt{2.4 * (CLOC/SLOC)})$
MI_{VS}	MI implemented in Visual Studio	$\max(0, (171 - 5.2 * ln(V) - 0.23 * CC - 16.2 * ln(SLOC)) * 100/171)$

We have formulated research questions for each of them. In the following, we list the research questions and briefly describe the metrics adopted to answer them. Table 3 reports a summary of all the metrics.

- **RQ1**: What is the verbosity of Rust code with respect to code written in other programming languages?
- **RQ2**: How is Rust code organized with respect to code written in other programming languages?
- **RQ3**: What is the complexity of Rust code with respect to code written in other programming languages?
- **RQ4:** What are the composite maintainability indexes for Rust code with respect to code written in other programming languages?

We are interested in comparing different programming languages through the use of static metrics. A static metric (opposed to dynamic or runtime metrics) is obtained by parsing and extracting information from a source file without depending on any information deduced at runtime.

To answer RQ1, we resorted to measuring the size of code artifacts written in Rust in terms of the number of code lines in a source file. We define four different metrics to differentiate between the nature of the inspected lines of code:

- *SLOC*, i.e., Source lines of code;
- *CLOC*, Comment Lines of Code;
- *PLOC*, Physical Lines of Code, including both the previous ones;
- *LLOC*, Logical Lines of Code, returning the count of the statements in a file.

To answer RQ2, we analyzed the source code structure in terms of source files properties and functions. To that end, we adopted three metrics: *NOM*, Number of Methods; *NARGS*, Number of Arguments;

Table 6. Selected algorithms for the study

Name	Description
binarytrees	Allocate and deallocate binary trees
fannkuchredux	Indexed-access to tiny integer-sequence
fasta	Generate and write random DNA sequences
knucleotide	Hashtable update and k-nucleotide strings
mandelbrot	Generate Mandelbrot set portable bitmap file
nbody	Double-precision N-body simulation
regexredux	Match DNA 8-mers and substitute magic patterns
revcomp	Read DNA sequences - write their reverse-complement
spectralnorm	Eigenvalue using the power method

NEXITS, Number of exits.NARGS and NEXITS are two software metrics defined by Mozilla and have no equivalent in the literature about maintainability metrics.

To answer RQ3, we adopted three metrics: *CC*, McCabe's Cyclomatic Complexity; *COGNITIVE*, Cognitive Complexity; and the *Halstead suite*. The Halstead suite is one of the most popular static code metrics available in the literature and was originally Maurice Halstead to decide a quantitative measure of complexity specifically from a set of operands and operators computed for each software module [Hariprasad et al. (2017)]. Table 4 reports the details about the computation of all operands and operators. The metrics in this category are more high-level than the previous ones and are based on the computation of previously defined metrics as operands.

To answer RQ4, we resorted to measuring the Maintainability Index, a composite metric originally defined by Oman et al. to provide a single index of maintainability for software [Oman and Hagemeister (1992)]. Three different versions of the Maintainability Index are considered: the original version by Oman et al., the version defined by the Software Engineering Institute (SEI), and the one implemented in the Visual Studio IDE. The Maintainability Index is the highest-level metric considered in this study. It includes an intermediate computation of one of the Halstead suite metrics.

3.2 Software Objects

For our study, we needed a set of simple algorithms to analyze the Rust source code properties and compare them with other programming languages.

To that end, we collected nine simple algorithms written each in 5 different languages: C, C++, JavaSript, Python, Rust, and TypeScript. All implementations of the algorithms have been taken from the Energy-Languages repository[2]. The rationale behind the repository selection is its continuous and active maintenance and the fact that these algorithms are adopted by various other projects for tests and benchmarking purposes, especially for evaluations of the speed of programming languages.

We were restricted to a limited number of 5 programming languages for the comparison since the tooling we adopted currently parses only a few languages (additional details are provided in the next section).

Table 6 lists the algorithms used (sorted out alphabetically) and provides a brief description for each of them.

3.3 Instruments and Procedure

This section provides details about the framework we developed to compare the selected metrics and the existing tools we employed for code parsing and metric computation.

A graphic overview of the framework is provided in Figure 1. The framework only represents the logical flow of the data in our software project since the actual flow of operations is reversed, being the *compare.py* script the entry point of the whole computation as described later in this section.

For each piece of source code passed as input, we use the rust-code-analysis tool to compute the static metrics and save them in the .json format. These .json files, containing the results of the metrics computation, are passed to a Python script, called *analyzer.py*, to be formatted in a common notation.

[2]https://github.com/greensoftwarelab/Energy-Languages

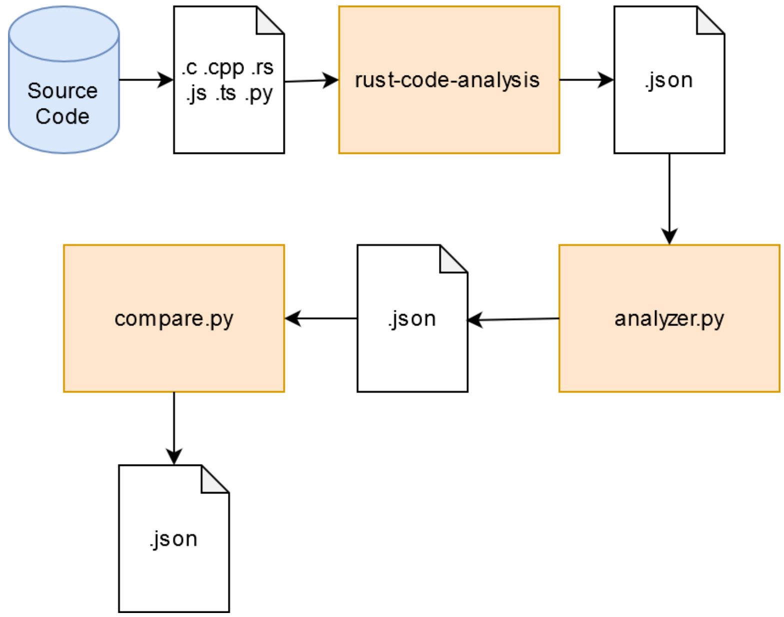

Figure 1. Overview of the evaluation framework

This notation is more focused on academic aspects compared to the ones used by the rust-code-analysis. Then a final script, called *compare.py*, has been developed to perform pair-by-pair comparisons between the .json files provided as output by *analyzer.py*. These comparison files allow us to immediately assess the differences in the metrics computed by the different programming languages on the same software artifacts. We made available the evaluation framework as a repository on GitHub[3].

3.3.1 The Rust Code Analysis tool

All considered metrics have been computed by adopting and extending a Rust language tool called *rust-code-analysis*. We have used the 0.0.18 version of this tool.

This software can receive either single files or entire directories, detect whether they contain any code written in one of its supported languages, and output the resultant static metrics in various formats: textual, JSON, YAML, toml, cbor.

From our point of view, instead, we have decided to adopt and personally extend a project written in Rust because of the advantages guaranteed by this language, such as memory and thread safety, memory efficiency, good performance, and easy integration with other programming languages.

Listing 1. Sample output of the rust-code-analysis tool

```
{
    "name": "/tmp/foo.rs",
    "start_line": 1,
    "end_line": 16,
    "kind": "unit",
    "spaces": [
        {
            "name": "Foo",
            "start_line": 5,
            "end_line": 16,
            "kind": "impl",
            "spaces": [
                {
                    "name": "bar",
                    "start_line": 6,
                    "end_line": 15,
                    "kind": "function",
                    "spaces": [
```

[3] https://github.com/SoftengPoliTo/SoftwareMetrics

```
                        {
                            "name": "<anonymous>",
                            "start_line": 12,
                            "end_line": 12,
                            "kind": "function",
                            "spaces": [],
                            "metrics": {
                                "nargs": 4.0,
                                "nexits": 0.0,
                                "cyclomatic": 1.0,
                                "halstead": {...},
                                "loc": {...},
                                "nom": {...},
                                "mi": {...},
                            }
                        }
                    ],
                    "metrics": {
                        "nargs": 1.0,
                        "nexits": 1.0,
                        "cyclomatic": 1.5,
                        "halstead": {...},
                        "loc": {...},
                        "nom": {...},
                        "mi": {...},
                    }
                }
            ],
            "metrics": {
                "nargs": 0.0,
                "nexits": 1.0,
                "cyclomatic": 1.3333333333333333,
                "halstead": {...},
                "loc": {...},
                "nom": {...},
                "mi": {...},
            }
        }
    ],
    "metrics": {
        "nargs": 0.0,
        "nexits": 1.0,
        "cyclomatic": 1.25,
        "halstead": {...},
        "loc": {...},
        "nom": {...},
        "mi": {...},
    }
}
```

Concerning the original implementation of the rust-code-analysis tool, we have forked the project and performed modifications on it by adding metrics computations (e.g., the COGNITIVE metric) and changes to the possible output format provided by the tool. We made available on GitHub our fork of the rust-code-analysis tool[4].

Listing 1 reports an excerpt of the .json file produced as output by rust-code-analysis.

3.3.2 Analysis

We developed a Python script named *analyzer.py* to analyze the metrics computed from rust-code-analysis. This script can launch different software libraries to compute metrics and adapt their results to a common format.

In this experiment, we used the *analyzer.py* script only with the Rust-code-analysis tool, but in a future extension of this study – or other empirical assessments – the script can be used to launch different tools simultaneously on the same source code.

The *analyzer.py* script performs the following operations:

[4]https://github.com/SoftengPoliTo/rust-code-analysis

- The arguments are parsed to verify their correctness. For instance, *analyzer.py* receives as arguments the list of tools to be executed, the path of the source code to analyze, and the path to the directory where to save the results;
- The selected metric computation tool(s) is (are) launched, to start the computation of the software metrics on the source files passed as arguments to the analyzer script;
- The output of the execution of the tool(s) is converted in Json and formatted in order to have a common standard to compare the measured software metrics;
- The new formatted .json files are saved in the directory previously passed as an argument to *analyzer.py*.

We have modified the output produced by rust-code-analysis through *analyzer.py* for the following reasons:

- The names of the metrics computed by the tool are not coherent with the ones selected from the scientific literature about software maintainability;
- The types of data representing the metrics are floating-point values instead of integers since rust-code-analysis aims at being as versatile as possible;
- The missing aggregation of each source file metrics contained in a directory within a single JSON-object, which is composed of global metrics and the respective metrics for each file. This additional aggregate data allows obtaining a more general prospect on the quality of a project written in a determined programming language.

3.3.3 Comparison

We finally developed a second Python script, *Compare.py*, to perform the comparisons over the .json result files generated by the *Analyzer.py* script. The Compare.py script executes the comparisons between different language configurations, given an analyzed source code artifact and a metric.

The script receives a *Configuration* as a parameter, a pair of versions of the same algorithm, written in two different programming languages.

The script performs the following operations for each received *Configuration*:

- Computes the metrics for the two files of a configuration by calling the analyzer.py script;
- Loads the two JSON files from the Results directory and compares them, producing a JSON file of differences;
- Deletes all local metrics (the ones computed by rust-code-analysis for each subspace) from the JSON file of differences;
- Saves the JSON file of differences, now containing only global file metrics, in a defined destination directory.

The JSON differences file is produced using a JavaScript program called JSON-diff[5].

4 RESULTS AND DISCUSSION

In this section, we report the results gathered by applying the methodology described in the previous section, subdivided according to the research question they answer.

4.1 RQ1 - Code verbosity

The boxplots in Figure 2 and Table 7 report the measures for the metrics that we adopted to answer RQ1. It can be seen that the mean and median values of the SLOC metric (i.e., total lines of code in the source files) are largely higher for the C, C++, and Rust language: the highest mean number of source locs was for C (209 average LOCs per source file), followed by C++ (186) and Rust (144). The mean values are way smaller for Python, TypeScript, and JavaScript (respectively, 98, 107, and 95 lines of code).

[5] https://www.npmjs.com/package/json-diff

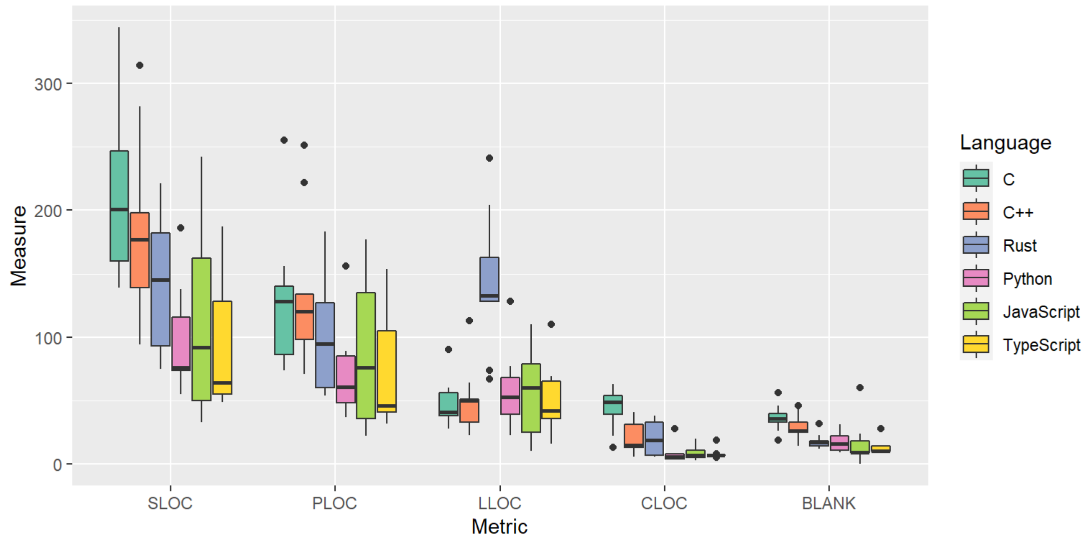

Figure 2. Distributions of the metrics about lines of code for all the considered programming languages

Table 7. Mean (Median) values of the metrics about lines of code for all the considered programming languages

Language	SLOC	PLOC	LLOC	CLOC	BLANK
C	209 (201)	129 (128)	48 (41)	43 (49)	37 (36)
C++	186 (177)	137 (120)	51 (50)	20 (15)	28 (26)
Rust	144 (145)	105 (95)	142 (133)	21 (19)	18 (17)
Python	99 (76)	73 (61)	59 (53)	8 (6)	18 (16)
JavaScript	107 (92)	83 (76)	58 (60)	9 (7)	16 (9)
TypeScript	95 (64)	74 (46)	51 (42)	8 (7)	13 (10)

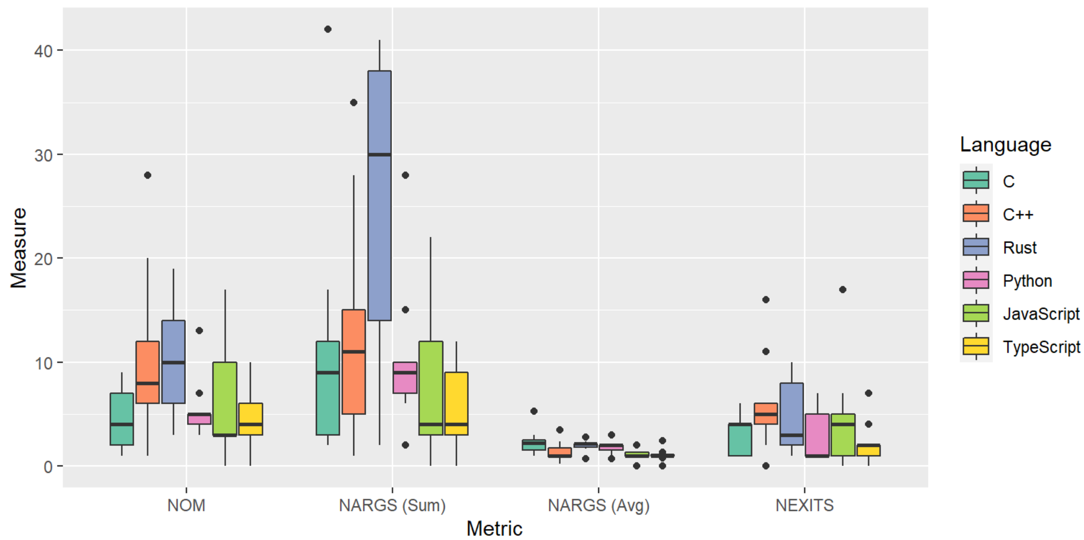

Figure 3. Distributions of the metrics about organization of code for all the considered programming languages

A similar trend is assumed by the PLOC metric (i.e., the total number of instructions and comment lines in the source files). In the examined set, we measured 74 average PLOCs per file for the Rust language. The highest and smallest values were again measured respectively for C and TypeScript, with 129 and 74 average PLOCs per file. The values measured for the CLOC and BLANK metrics showed that a higher number of empty lines of code and comments were measured for C than for all other languages. In the CLOC metric, the Rust language exhibited the second-highest mean of all languages, suggesting a higher predisposition of Rust developers at providing documentation in the developed source code.

An exciting result (especially in contrast with the other ones) is obtained by the LLOC metric (i.e., the number of logical lines of code, or statements, in a file). In this case, the mean number of statements for Rust code is largely higher than the average for all other considered languages (142 mean LLOCs per file, with the second-highest, mean being 59 for the Python language). This result can be interpreted according to the way the LLOC metric is computed by the tools and the type of information that is measured. The metric counts the total number of statements provided in a parsed source file, obtained by searching for the ones that are available for a given language (i.e., in C, *For Statements*, *If Statements*, *Return Statements* are different types of statements, while in Rust *If Let* and While Let are other ones). As an examination of the parsing module of the *rust-code-analysis* tool confirmed, the Rust language offers many more types of statements than the other considered language (24 different types against the 14 provided by C). This higher availability of instruments can translate to a finer decomposition of the lines of source code in statements, and hence to a higher LLOC metric for the same source files.

Answer to RQ1: The examined source files written in rust exhibited an average verbosity (144 mean SLOCs per file and 74 mean PLOCs per file). Such values are lower than C and C++ and higher than the other considered object-oriented languages. Rust exhibited the highest average LLOC value of all considered languages.

4.2 RQ2 - Code organization

The boxplots in Figure 3 and Table 8 report the measures for the metrics that we adopted to answer RQ2. For each source file, we collected two different measures for the NARGS metric: the sum at file level of all the methods arguments and the average at file level of the number of arguments per method (i.e., $NARGS/NOM$).

The Rust language had the highest median value for the NOM metric, with ten median methods per source file. The average NOM value was only lower than the one measured for C++ sources. However, this value was strongly influenced by the presence of one outlier in the set of analyzed sources (namely,

Table 8. Mean (Median) values of the metrics about code organization for all the considered programming languages

Language	NOM	NARGS (Sum)	NARGS (Avg)	NEXITS
C	4.4 (4)	11.6 (9)	2.4 (2)	3.1 (4)
C++	10.6 (8)	13.4 (11)	1.4 (1)	6.0 (5)
Rust	10.3 (10)	25.1 (30)	2.0 (2)	4.7 (3)
Python	5.7 (5)	10.6 (9)	1.8 (2)	2.8 (1)
JavaScript	5.9 (3)	7.4 (4)	1.1 (1)	4.6 (4)
TypeScript	4.7 (4)	5.7 (4)	1.1 (1)	2.1 (2)

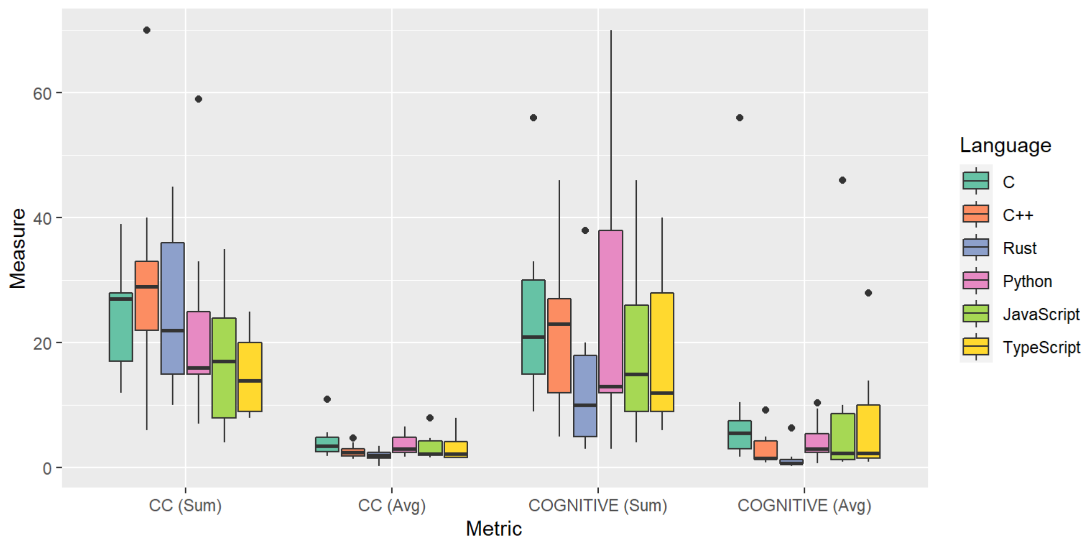

Figure 4. Distributions of complexity metrics for all the considered programming languages

the C++ implementation of *fasta* having a NOM equal to 20). While the NOM values were similar for C++ and Rust, all other languages exhibited much lower distributions, with the lowest median value for JavaScript (3). This high number of Rust methods can be seen as evidence of higher modularity than the other languages considered.

Regarding the number of arguments, it can be noticed that the Rust language exhibited the highest average and median cumulative number of arguments (Sum of Arguments) of all languages. The already discussed high NOM value influences this result.

The lowest average measures for NOM and NARGS_Sum metrics were obtained for the C language. This result can be justified by the lower modularity of the C language. By examining the C source files, we verified that the code presented fewer functions and more frequent usage of nested loops, while the Rust sources were using more often data structures and ad-hoc methods. In general, the results gathered to measure this facet of code maintainability suggests a more structured Rust code organization regarding the C language.

Regarding the NEXITS metric, the values were close for most of the languages, except Python and TypeScript, which respectively contain more methods without exit points and fewer functions. The obtained NEXITS value for Rust shows many exit points distributed among many functions, as demonstrated by the NOM value, making the code much more comfortable to follow.

> **Answer to RQ2**: The examined source files written in Rust exhibited the most structured organization of the considered set of languages (with a mean 10.3 NOM per file, with a mean of 2 arguments for each method).

Table 9. Mean (Median) values of the complexity metrics for all the considered programming languages

Language	CC_{Sum}	CC_{Avg}	$COGNITIVE_{Sum}$	$COGNITIVE_{Avg}$
C	24.4 (27)	4.3 (3.5)	24.3 (21.0)	10.9 (5.5)
C++	31.1 (29)	2.7 (2.4)	22.4 (23.0)	3.2 (1.5)
Rust	25.3 (22)	2.0 (2.0)	13.1 (10.0)	1.5 (0.7)
Python	23.0 (16)	3.6 (3.0)	25.4 (13.0)	4.4 (3.0)
JavaScript	17.6 (17)	3.4 (2.2)	19.9 (15.0)	8.5 (2.3)
TypeScript	15.2 (14)	3.4 (2.2)	17.0 (12.0)	7.2 (2.3)

Table 10. Mean (Median) values of Halstead metrics for all the considered programming languages

Language	Bugs	Difficulty	Effort	Length	Programming Time	Volume
C	1.52 (1.6)	66.7 (55.9)	322,313 (342,335)	726.0 (867.0)	17,906 (19,018)	4,819 (5,669)
C++	1.46 (1.3)	57.8 (56.4)	311,415 (248,153)	728.1 (634.0)	17,300 (13,786)	4,994 (4,274)
Rust	1.1 (1.3)	48.6 (45.9)	199,152 (246,959)	602.2 (550.0)	11,064 (13,719)	4,032 (3610)
Python	0.7 (0.6)	33.7 (30.0)	111,103 (72,110)	393.8 (334.0)	6,172 (4,006)	2,680 (2204)
JavaScript	0.8 (0.9)	43.1 (44.1)	139,590 (140,951)	458.6 (408.0)	7,755 (7,830)	2,963 (2615)
TypeScript	0.8 (0.6)	45.2 (41.9)	132,644 (82,369)	435.7 (302.0)	7,369 (4,576)	2,734 (1730)

4.3 RQ3 - Code complexity

The boxplots in Figure 4 and Table 9 report the measures for the metrics that we adopted to answer RQ3. For the Computational Complexity, we computed the sum of the CC of all *spaces* in a source file (CC_{Sum}), and the averaged value of CC over the number of spaces in a file (CC_{Avg}). A space is defined in *rust-code-analysis* as any structure that incorporates a function. For what concerns COGNITIVE complexity, we computed the sum of the COGNITIVE complexity associated to each function and closure present in a source file, ($COGNITIVE_{Sum}$), in addition to the average value of COGNITIVE, ($COGNITIVE_{Avg}$), always computed over the number of functions and closures. In the table, we report the mean and median values over the set of different source files selected for each language, of the sum and average metrics computed at the file level.

As commonly accepted in the literature and practice, a low cyclomatic complexity generally indicates a method that is easy to understand, test, and maintain. The reported measures showed that the Rust language had a lower median CC_{Sum} (22) than C and C++ and the second-highest average value (25.3). We measured the lowest average and median CC_{Sum} for the TypeScript language. By considering the average of the Cyclomatic Complexity, CC_{Avg}, at the function level, we instead obtain the highest average and mean values for the Rust language. It is worth mentioning that the average CC values for all the languages were rather low, hinting at an inherent simplicity of the software functionality under examination. So an analysis based on different codebases may result in more pronounced differences between the programming languages.

Cognitive complexity is a software metric that assesses the complexity of code starting from human judgment and is a measure for source code comprehension by the developers and maintainers [Barón et al. (2020)]. Moreover, empirical results have also proved the correlation between cognitive complexity and defects [Alqadi and Maletic (2020)]. For both the average cognitive complexity and the sum of cognitive complexity at the file level, we measured that Rust provided the lowest mean and median values. Specifically, Rust guaranteed a Cognitive complexity of 0.7 per method, which is less than half the second-lowest value for C++ (1.5). The highest average Cognitive Complexity per class was measured for C code (5.5). This very low value of the cognitive per method for Rust is related to the highest number of methods for Rust code (described in the analysis of RQ2 results). By considering the sum of the COGNITIVE metric at the file level, Rust had a mean $COGNITIVE_{Sum}$ of 13.1 over the 9 analyzed source files. The highest mean value for this metric was measured for Python (25.4), and the highest median for C++ (23). Such lower values for the Rust language can suggest a more accessible, less costly, and less prone to bug injection maintenance for source code written in Rust.

The boxplots in Figure 4 and Table 9 report the distributions, mean, and median of the Halstead metrics computed for the six different programming languages.

The Halstead Difficulty (D) is an estimation of the difficulty of writing a program that is statically

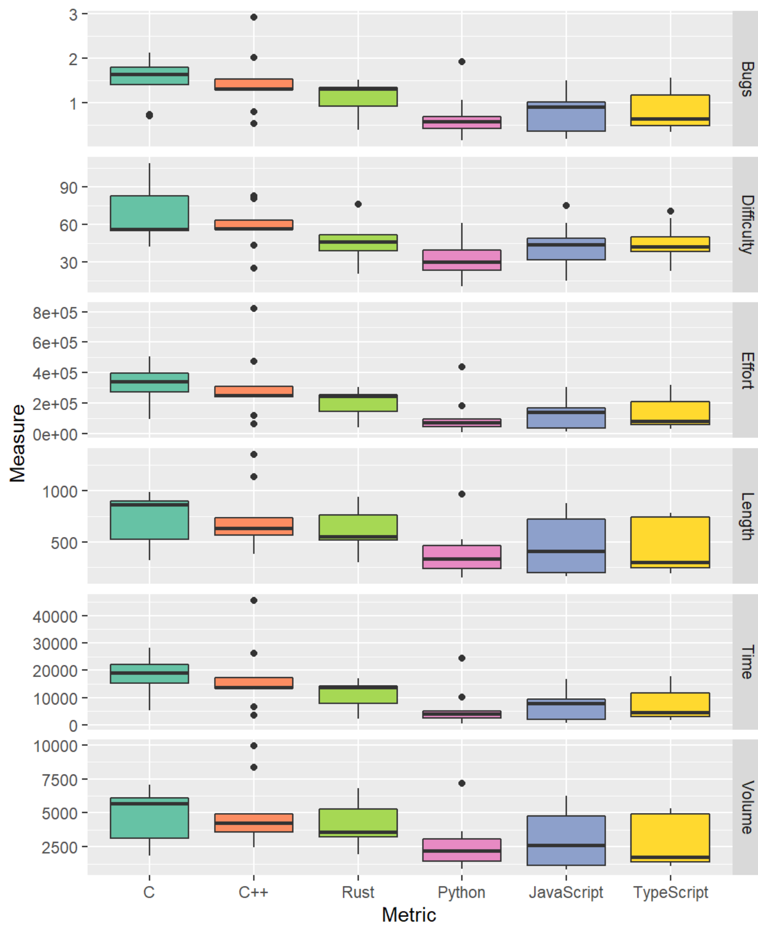

Figure 5. Distributions of Halstead metrics for all the considered programming languages

Table 11. Mean (Median) values of Maintainability Indexes for all the considered programming languages

Language	Original	SEI	Visual Studio
C	35.9 (36.7)	10.5 (5.0)	21.0 (21.5)
C++	36.5 (36.3)	3.6 (9.9)	21.3 (21.2)
Rust	43.0 (43.3)	15.8 (22.6)	25.1 (25.3)
Python	52.5 (55.5)	23.3 (25.7)	30.7 (32.5)
JavaScript	54.2 (51.7)	27.7 (25.3)	31.7 (30.3)
TypeScript	55.9 (61.6)	29.4 (39.2)	32.7 (36.0)

analyzed. The program difficulty is the inverse of the program level metric. Hence, as the volume of the implementation of an algorithm increases, the difficulty increases as well. The usage of redundancy hence influences the Difficulty. It is correlated to the number of operators and operands used in the algorithm implementation. Our results suggest that the Rust programming language has an average difficulty (median of 45.9) on the set of considered languages. The most difficult code to interpret, according to Halstead metrics, was C (median of 55.9), while the easiest to interpret was Python (median of 30.0). A similar hierarchy between the different languages is obtained for the Halstead Effort (E), which estimates the mental activity needed to translate the existing algorithm into code written in a specific language. The Effort is linearly proportional to both Difficulty and Volume. The unit of measure of the metric is the number of elementary mental discriminations [Halstead et al. (1977)].

The Halstead Length (L) metric is given by the total number of operator occurrences and the total number of operand occurrences. The Halstead Volume (V) metric is the information content of the program, linearly dependent on its vocabulary. For Rust code, we measured the third-highest mean and median Halstead Length (602.2 mean, 550.0 median) and Halstead Volume (4,032 mean, 3,610 median), again below those measured for C and C++. The results measured for all considered source files were in line with existing programming guidelines (Halstead Volume lower than 8000). The reported results about Length and Volume were, to some extent, expectable since these metrics are largely correlated to the number of lines of code present in a source file [Tashtoush et al. (2014)].

The Halstead Time metric (T) is computed as the Halstead Effort divided by 18. It estimates the time in seconds that it should take a programmer to implement the code. We measured a mean and median T of 11,064 and 13,719 seconds, respectively, for the Rust programming language. These values are significantly distant from those measured for Python and TypeScript (the lowest) and from those measured for C and C++ (the highest).

Finally, the Halstead Bugs Metric estimates the number of bugs that are likely to be found in the software program. It is given by a division of the Volume metric by 3000. We estimated a mean value of 1.1 (median 1.3) bugs per file with the Rust programming language on the considered set of source artifacts.

433 **Answer to RQ3**: The Rust software artifacts that we examinated exhibited an average Cyclomatic Complexity (mean 2.0 per function) and the lowest Cognitive Complexity (mean 1.5 per function). Rust was the third-highest performing language, after C and C++, for the Halstead metric values.

4.4 RQ4 - Code maintainability

The boxplots in Figure 4 and Table 9 report the distributions, mean, and median of the Maintainability Indexes computed for the six different programming languages.

The Maintainability Index is a composite metric aiming to give an estimate of software maintainability over time. The Metric has correlations with the Halstead Volume (V), the Cyclomatic Complexity (CC), and the number of lines of code of the source under examination.

By using all the formulas for the Maintainability Index, we computed for the source files written in Rust an average MI that placed the fourth among all considered programming languages. Minor differences in placing other languages occurred, e.g., the median MI for C is higher than for C++ with the original formula for the Maintainability Index and lower with the SEI formula. With all the formulas to compute MI, the highest maintainability was achieved by the TypeScript language, followed by Python

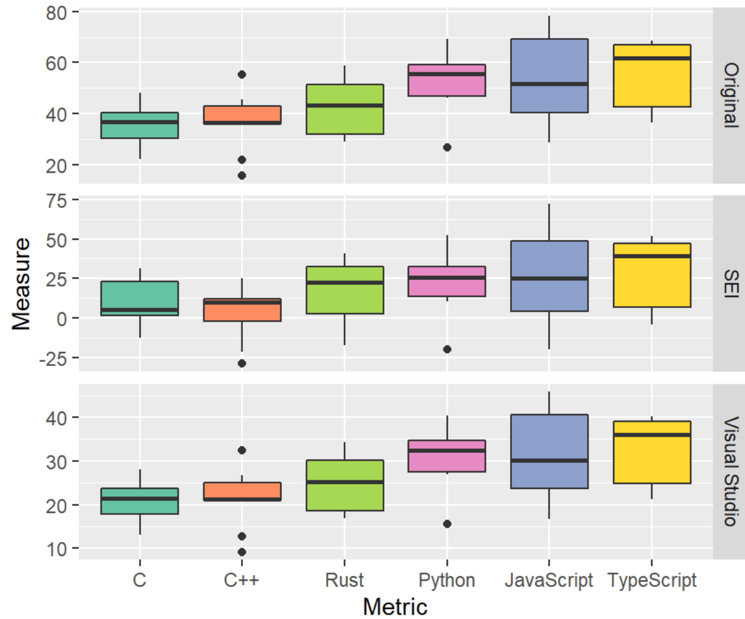

Figure 6. Distributions of Maintainability Indexes for all the considered programming languages

and JavaScript. These results were expectable in light of the previous measures, given the said strong dependency of the MI on the raw size of source code.

It is interesting to underline that, in accordance with the original guidelines for the MI computation, all the values measured for the software artifacts under study would suggest hard to maintain code, being the threshold for easily maintainable code set to 80. On the other hand, according to the documentation of the Visual Studio MI metric, all source artifacts under test can be considered as easy to maintain ($MI_{VS}20$).

> **Answer to RQ4**: Rust exhibited an average Maintainability Index, regardless of the specific formula used (median values of 43.3 for MI_O, 22.6 for MI_{SEI}, 25.3 for MI_{VS}). Highest Maintainability index were obtained for Python, JavaScript and TypeScript.

5 THREATS TO VALIDITY

Threats to Internal Validity. The study results may be influenced by the specific selection of the tool with which the software metrics were computed, namely the *rust-code-analysis* tool. The values measured for the individual metrics (and, by consequence, the reasoning based upon them) can be heavily influenced by the exact formula used for the metric computation. In Halstead metrics, the formulas depend on coefficients defined explicitly in the literature for every software language. Since no previous result in the literature has provided Halstead coefficients specific to Rust, we used the C coefficients for the computation of Rust Halstead metrics. This choice may significantly influence the values obtained for the collected metrics. Future extensions of this work may include studies to infer the optimal Halstead parameters for Rust source code.

Threats to External Validity. The results that we present in this research have been measured on a limited number of source artifacts (namely, nine different algorithms per programming language). Therefore, we acknowledge that the results cannot be generalized to all software written with one of the analyzed programming languages. Another bias can be introduced in the results by the characteristics of the considered code artifacts. All considered source files were small programs collected from a single software repository. Future extensions of the current work should include the computation of the selected metrics on more extensive and more diverse sets of software artifacts to increase the presented results generalizability.

Threats to Conclusion Validity. The conclusions detailed in this work are only based on the analysis of quantitative metrics and do not consider other possible characteristics of the analyzed source artifacts (e.g., the developers' coding style who produced the code). Like the generalizability of the results, this bias can be reduced in future extensions of the study using a broader and more heterogeneous set of source artifacts [Sjøberg et al. (2012)].

In this work, we make assumptions on maintainability, complexity and understandability of source code based on quantitative static metrics. It is not ensured that our assumptions are reflected by maintenance and code understanding effort in real-world development scenarios. It is worth mentioning that there is no unanimous opinion about the ability of more complex metrics (like MI) to capture the maintainability of software programs more than simpler metrics like lines of code and Cyclomatic Complexity.

Researcher bias is a final theoretical threat to the validity of this study since it involved a comparison in terms of different metrics of different programming languages. However, the authors have no reason to favor any particular approach, neither inclined to demonstrate any specific result.

6 CONCLUSION AND FUTURE WORK

In this paper, we have evaluated the complexity and maintainability of Rust code by using static metrics and presented a comparison of the gathered results.

All the evidence collected in this paper suggests that the Rust language can produce more maintainable code than C and C++, the languages to which it is more similar in terms of code structure and syntax. On the other hand, the Rust language provided lower maintainability than measured for more sophisticated and high-level object-oriented languages. Worth underlying that the source artifacts written in the Rust language exhibit the lowest cognitive complexity, meaning that the language can guarantee the highest understandability of source code compared to all others. Understandability is a fundamental feature of code during its evolution since it may significantly impact the required effort for maintaining and fixing it.

As a prosecution of this work, we plan to perform further developments on the *rust-code-analysis* tool such that it can provide more metric computation features and parsers for more programming languages (e.g., Java) to which comparisons can be performed. We also plan to extend our analysis to real projects composed of a significantly higher amount of code lines that embed different programming paradigms, such as the functional and concurrent ones. To this extent, we plan to mine software projects from open source libraries, e.g., GitHub.

REFERENCES

Aggarwal, K. K., Singh, Y., and Chhabra, J. K. (2002). An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*, pages 235–241. IEEE.

Alqadi, B. S. and Maletic, J. I. (2020). Slice-based cognitive complexity metrics for defect prediction. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 411–422. IEEE.

Ardito, L., Coppola, R., Barbato, L., and Verga, D. (2020). A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming*, 2020.

Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. (2019). Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30.

Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L. (2017). System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161.

Barón, M. M. n., Wyrich, M., and Wagner, S. (2020). An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA. Association for Computing Machinery.

Gill, G. K. and Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284.

Halstead, M. H. et al. (1977). *Elements of software science*, volume 7. Elsevier New York.

Hariprasad, T., Vidhyagaran, G., Seenu, K., and Thirumalai, C. (2017). Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1109–1113. IEEE.

Jingqiu Shao and Yingxu Wang (2003). A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering*, 28(2):69–74.

Kaur, A., Kaur, K., and Pathak, K. (2014a). A proposed new model for maintainability index of open source software. In *Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization*, pages 1–6. IEEE.

Kaur, A., Kaur, K., and Pathak, K. (2014b). Software maintainability prediction by data mining of software code metrics. In *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, pages 1–6. IEEE.

Köster, J. (2016). Rust-bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446.

Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., and Levis, P. (2017). The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7.

Ludwig, J. and Cline, D. (2019). Cbr insight: measure and visualize source code quality. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 57–58. IEEE.

Ludwig, J., Xu, S., and Webber, F. (2017). Compiling static software metrics for reliability and maintainability from github repositories. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 5–9. IEEE.

Matsakis, N. D. and Klock, F. S. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104.

Matsushita, T. and Sasano, I. (2017). Detecting code clones with gaps by function applications. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 12–22.

Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., and Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164 – 197.

Oman, P. and Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society.

Saifan, A. A., Alsghaier, H., and Alkhateeb, K. (2018). Evaluating the understandability of android applications. *International Journal of Software Innovation (IJSI)*, 6(1):44–57.

Sarwar, M. I., Tanveer, W., Sarwar, I., and Mahmood, W. (2008). A comparative study of mi tools: Defining the roadmap to mi tools standardization. In *2008 IEEE International Multitopic Conference*, pages 379–385. IEEE.

Sjøberg, D. I., Anda, B., and Mockus, A. (2012). Questioning software maintenance metrics: a comparative case study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 107–110. IEEE.

Tashtoush, Y., Al-Maolegi, M., and Arkok, B. (2014). The correlation among software complexity metrics with case study. *arXiv preprint arXiv:1408.4523*.

Uzlu, T. and Şaykol, E. (2017). On utilizing rust programming language for internet of things. In *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 93–96. IEEE.

Welker, K. D. (2001). The software maintainability index revisited. *CrossTalk*, 14:18–21.

Zhou, Y. and Leung, H. (2007). Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of systems and software*, 80(8):1349–1361.