universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Software Code Metric Libraries and their Differences

verfasst von | submitted by

Matteo Bernard Battaglin BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl

Contents

1 Acknowledgments

I am profoundly grateful to Dr. Edgar Weippl, Dr. Sebastian Schrittwieser, and Patrick Kochberger for their exceptional mentorship, unwavering support, and invaluable guidance throughout this challenging journey. Their expertise, encouragement, and insights were pivotal in navigating the complexities of this thesis. I also extend my heartfelt appreciation to my father, Bernard Battaglin, whose meticulous proofreading and native English proficiency greatly enhanced the clarity and quality of this work.

Furthermore, I wish to express my deepest gratitude to my mother, Doris Battaglin, and my sister, Elena Battaglin, for their unwavering belief in me and constant encouragement during this endeavor. Their emotional support has been a constant source of strength and motivation.

This thesis would not have been possible without the collective support and encouragement of these individuals, each of whom played a significant role in its successful completion.

2 German Abstract

Softwarecode-Metriken sind quantifizierbare Maße, die zur Bewertung verschiedener Aspekte von Softwarecode verwendet werden, darunter Größe, Qualität, Komplexität, Wartbarkeit und die Schätzung der Entwicklungskosten.

Das Hauptziel dieser Arbeit ist es, ein tieferes Verständnis von Softwarecode-Metriken zu erlangen, indem zwei zentrale Forschungsfragen behandelt werden: Erstens, wird untersucht, wie unterschiedliche Open-Source-Metrik-Bibliotheken bei der Analyse des selben Eingabecodes variierende Ergebnisse liefern. Zweitens, sollen Metriken wie die zyklomatische Komplexität (CC) untersucht werden, welche als Anzahl der unabhängigen Pfade durch ein Programm definiert ist und theoretisch zwischen Programmiersprachen mit ähnlicher Basissyntax vergleichbar sein sollte. In diesem Zusammenhang wird erforscht, wie die Implementierung desselben Algorithmus in verschiedenen Programmiersprachen (z. B. Python vs. C) die Ergebnisse verschiender Tools beeinflusst.

Zur Erreichung dieser Ziele wurden zwei Software-Tools entwickelt. Das Analysis Tool wurde konzipiert, um den Vergleich der Ergebnisse verschiedener Open-Source-Metrik-Tools zu erleichtern. Darüber hinaus wurde ein Transcriber implementiert, der einfache Python-Codedateien in C-Code umwandelt, um die Metriken desselben Algorithmus in beiden Zielprogrammiersprachen vergleichen zu können. Zusätzlich wurden mehrere Testfälle erstellt.

Die Untersuchung ergab beträchtliche Unterschiede in den Ergebnissen verschiedener Metrik-Tools für identische Eingaben. Außerdem zeigte sich, dass der Vergleich der zyklomatischen Komplexität für denselben Algorithmus in unterschiedlichen Sprachen zu signifikant abweichenden Ergebnissen führte, obwohl diese theoretisch ähnlich sein sollten. Dies weist darauf hin, dass beim Vergleich von Ergebnissen, die von unterschiedlichen Bibliotheken berechnet wurden, Vorsicht geboten ist. Die Arbeit verdeutlicht die Notwendigkeit zukünftiger Bemühungen zur Standardisierung von Softwarecode-Metriken.

3 Abstract

Software code metrics are quantifiable measures used to assess various aspects of software code, such as size, quality, complexity, maintainability, and estimating development costs.

The main aim of this thesis is to gain a deeper understanding of software code metrics while focusing on two main areas. First, a better understanding of the variations in results from various open-source code metric libraries for the same input code. Second, to examine metrics such as Cyclomatic Complexity (CC), which is defined as the amount of independent paths through a program and should, in theory, be fairly comparable across programming languages with similar basic syntax. We thus investigated how the implementation of the same algorithm in two different programming languages (e.g., Python vs. C) affects the library results.

Two software tools were developed to accomplish these goals. First, the Analysis Tool was created, which simplifies comparing the results of various open-source metric tools. Second, a Transcriber was implemented that can transcribe simple Python code files to C code with the goal of allowing the metrics of the same algorithm to be easily generated and compared in the two target programming languages. Additionally, multiple test cases were written to verify the methodology.

Considerable differences in results were observed between different metric tools for the same inputs in some cases. Furthermore, it was demonstrated that comparing the cyclomatic complexity metric results for the same algorithm implemented in two languages leads to vastly different results for most tools due to language-specific bugs, although, in theory, they should be the same. This suggests that caution is necessary when comparing metric results calculated by different libraries. This thesis shows that more work is needed in the future to standardize software code metrics.

4 Introduction

Computer programming has long been regarded as a form of artistic expression akin to poetry or writing, blending creativity with technical precision [1]. Like poets or writers, programmers use language and structure to convey ideas and tackle issues [2]. The best programmers craft concise, efficient, elegant, and easy-to-use software solutions to solve complex problems, while striving for simplicity, readability, and maintainability in their solutions [1].

Especially in the early days of software development, programmers commonly wrote source code ad-hoc, mostly using their own intuition rather than following established guidelines [3]. This resulted in code rife with personal choices and artistic expression. As software systems became more complex and more people were required to collaborate on projects, it became apparent that this ad-hoc approach to software development was insufficient and a more scientific approach was necessary [3], [4]. This gave rise to the field of software engineering, which aims to introduce more scientific methods, structure, and discipline into the software development process [3]. The total number of lines in a source file—commonly referred to as Lines of Code (LOC) — and related measures such as the Source Lines of Code (SLOC), which excludes comment and import lines, have been fundamental measures of program size since the early days of software development [4].

In the late 1970s, two pioneers in the field of software metrics, Maurice H. Halstead and Thomas J. McCabe, emphasized the necessity of defining new scientific metrics to measure and numerically quantify to what degree a software system possesses a specific property [5]. During this time, software managers increasingly realized that better methods to estimate development costs and measure the quality of the developed software were desperately needed [6].

In 1976, McCabe suggested using the total number of linearly independent paths through a program (coined "*Cyclomatic Complexity (CC)*") to quantify a program's complexity [5]. McCabe further proposed a testing strategy to ensure software quality, requiring the minimum number of test cases for each program to be at least equal to the Cyclomatic Complexity of the code.

One year later, Halstead published his book "Elements of Software Science" that further emphasized the need to view software development as a "science" rather than an "art form" [7]. Halstead suggested viewing programs as a string of operands (variables, constants) and operators (that alter operands) and proposed multiple new metrics derived from the total and unique number of occurrences of both operands and operators. The book proposed new metrics to estimate the effort needed to write a program as well as specific metrics to quantify a program's vocabulary, length, volume, and other properties, which are detailed in chapter 6.1.2. Various scholars have since proposed their own software code metrics with the goal of obtaining objective, reproducible, and scientific measurements for a variety of properties.

In 1981, Dr. Barry Boehm proposed his Constructive Cost Model (COCOMO) to better estimate development costs of software projects [8]. The model's parameters were determined by fitting a regression formula to data from 63 historical code bases at TRW Aerospace, where Boehm served as a director. The COCOMO model is explained in greater detail in chapter 6.1.4 but is mostly influenced by the number of lines in a program. In the early 90s, Paul Oman and Jack Hagemeister hoped to develop a metric to quantify a program's overall "maintainability." In 1992, Oman and Hagemeister conducted an extensive review of 35 published works pertaining to software maintainability, culminating in the identification of 92 attributes

potentially influencing a program's overall maintainability [9]. Subsequently, they curated existing metrics or devised new ones to quantitatively assess each of these attributes. Expanding upon their earlier investigations, Oman and Hagemeister collaborated with the company Hewlett Packard (HP) in 1994 to conduct significant experiments aimed at determining the minimal set of metrics required for reliably predicting maintainability [10]. To summarize, the experiments concluded that Halstead's effort & volume metrics, along with the cyclomatic complexity, the lines of code, and the number of comment lines were the best predictors of maintainability [10]. This research is studied more closely in chapter 6.1.5.

According to a systematic research review completed in 2017 concerning the state of source code metrics that analyzed 226 primary studies, the most studied metrics were Chidamber and Kemerer, lines of code, and the cyclomatic complexity. Fault prediction and complexity were identified as recurrent quality topics of interest in the metric research community [11].

As of today, there are roughly 12 million professional software developers working in the software industry worldwide, with the average developer writing 10,000 lines of software code each year [12]. This results in 120,000,000,000 lines of code being created or modified per year. Companies report spending 50-60% of their IT budgets on maintaining and fixing existing source code [9]. With more and more time and resources spent on software development and maintenance, there has been a growing incentive to scientifically analyze and measure the size, quality, complexity, and maintainability of software [13]. Source code metrics are seen as a crucial component of the modern software measurement process.

Currently, a diverse array of software metric tools, both open-source and commercial, are extensively utilized in academia and industry [11]. The predominant focus of source code metric research revolves around the Java and C programming languages, with only minimal research focusing on the widely popular Python language. Some of the tools available today are specialized in calculating a single metric for a single input language, while others claim to support a variety of different metrics and input programming languages.

The broad use of multiple different metric tools, presumably implemented in different ways, raises new questions.

4.1 Research Question

This thesis focuses on answering two main research questions. First, this thesis aims to uncover differences in how popular open-source metric tools calculate their results given the same input and are queried for supposedly the same metric. Many of the most cited published experiments on source code metrics only specify the metric but do not specify the tools used to calculate said metric [9] [11]. The goal of this part of the research is to aid in determining how sensible it is to compare metric results across different tools.

The second main research question focuses on determining whether the studied tools apply the same rules and standards when calculating a metric given different input languages. Some metrics, such as Cyclomatic Complexity (the number of independent paths through a program), should in theory be fairly comparable between two imperative programming languages with generally similar syntax. A tool was created to automatically transcribe simple Python files to C code, with the goal of easily being able to compare the results for both languages.

5 Methodology

This section provides an overview of the methodology used to complete this thesis. Major elements of the methodology include:

- **A literature search** - Researched commonly cited source code metrics and gathered information about open-source software metric tools that can be used to calculate them.
- **Development of the *Analysis Tool*** - Developed software to automate testing and collection of results (see Section 5.1).
- **Implementation of the *Transcriber*** - Developed software to transcribe Python code into C (see Section 5.2).
- **Finding Python Test Cases** - Cooperated with the Security and Privacy Research Group to obtain real-world Python test cases. Gathered additional Python source code from common Python open-source libraries.
- **Running and Analyzing Python Tests** - The *Analysis Tool* was used to run tests on the selected source code.
- **Transcribing Python Code** - The *Transcriber* was used to transcribe selected Python source code into C.
- **Running and Analyzing Transcribed C Tests** - The *Analysis Tool* was used to run tests on the generated C code.

5.1 Analysis Tool

The goal of the *Analysis Tool* is to provide a convenient way to calculate and compare the results of all studied metric tools. The *Analysis Tool* is written in Python (version 3.8.10) and uses multi-threading and parallelization to effectively execute and collect metrics results from 13 source code metric programs. The *Analysis Tool* includes an option to use the *Transcriber* described in Section 5.2 to transcribe Python files to C and analyze both sets of source code. Additionally, the tool is designed with modular parsing functionality to make adding new metric tools in the future as easy as possible.

The *Analysis Tool* can operate in two different analysis and output modes: `normal` and `Line by Line`, which are detailed in the next two subsections.

5.1.1 Normal Mode

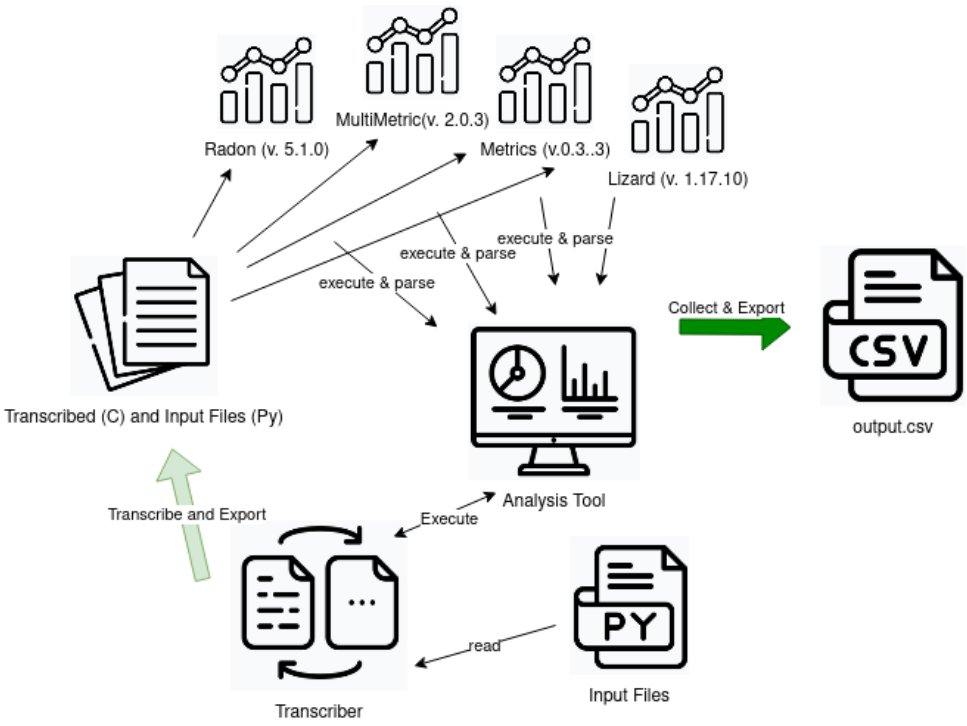

Figure 1: Normal Mode Overview

The `normal` mode of the *Analysis Tool* provides functionality to collect, filter, and output the metric results of the metric tools for one or more input files. The process consists of three steps. First, the user selects the input files and metrics to calculate and output. Next, all tools that support the selected metrics are automatically selected and executed in parallel, and their output is parsed and collected. In the final step, the tool outputs are formatted and saved in a `csv` file. This allows the results to be easily filtered and compared.

The *Analysis Tool* generally supports all input file types; however, the different metric tools may only support a subset of input types. When analyzing Python files, the tool includes an option to first transcribe the inputted files using the *Transcriber*. In this case, both the original and transcribed input files are processed. Please note Figure 1 only shows some of the supported metric tools due to space constraints.

5.1.2 Line by Line Mode

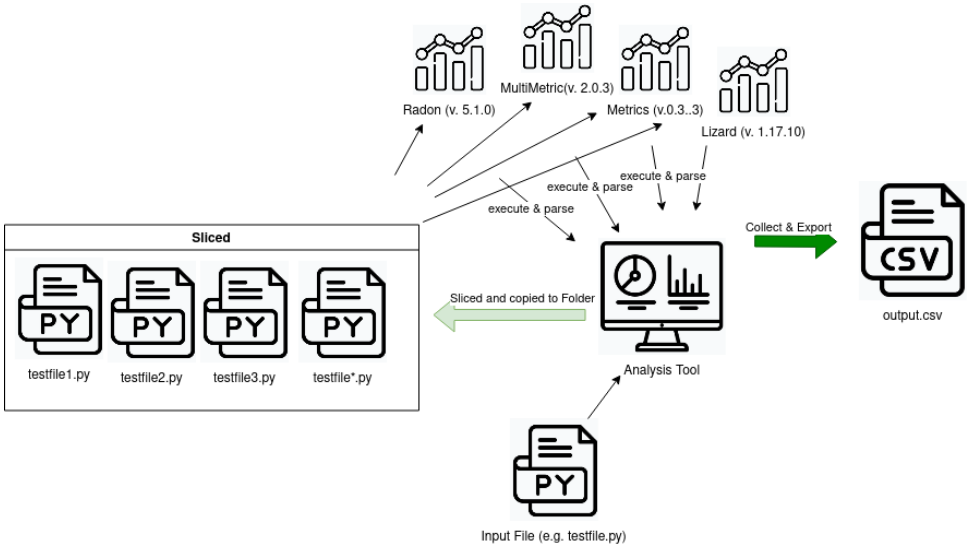

Figure 2: Line by Line Overview

To better understand how a given metric number gets calculated, the *Analysis Tool* has a mode that runs the analysis in a line-by-line fashion. This aids in understanding the calculation process by the various libraries. This `Line by Line` mode splits and exports a single `py` input file into multiple files, each containing one additional line of the original input file, to the `Sliced` folder. Thus, the first export file contains only the first line of input, while the last export file contains the entire original input file. The analysis tool then runs the various metric tools on each of the exported slice files and adds the results line by line to a `csv` file. The exported spreadsheet is useful to better understand how the individual lines of code affect the overall metric score of the various libraries. Currently, only `py` inputs are supported.

The `Line by Line` mode proved itself to be invaluable in uncovering the limitations and errors in the metric tools studied in this thesis.

5.2 C to Python Transcriber

One of the primary objectives of this thesis is to assess how the choice of programming language impacts the results of the software metric tools that claim to support multiple languages. Our aim is to ascertain whether these tools apply consistent general metric calculation principles across different input languages. Furthermore, we sought to gain insights into the validity of comparing metrics such as Cyclomatic Complexity (CC) between languages, specifically Python and C.

To achieve these objectives, it was necessary to develop a method to efficiently produce algorithm implementations in both Python and C. Therefore, we created a tool known as the *Transcriber*, with the aim of automatically transcribing Python input source code to C output code. Acknowledging the complexity involved in developing a fully adaptable transcriber (capable of transcribing any valid Python input), we decided to focus on a simplified version capable only of converting simple Python Domain Generating Algorithms (DGAs). DGAs can be summarized as algorithms that generate strings in a predictable manner, widely used in software security, and explained in greater detail in Section 5.2.2. We chose DGAs as input due

to their widespread real-world use and because DGAs are generally simple Python programs with a common structure that are fairly straightforward to transcribe. The Security and Privacy Work Group of the University of Vienna is currently researching Domain Generating Algorithms as part of their other projects and agreed to provide multiple real-world DGAs.

The newly created *Transcriber* is based on parsing the Abstract Syntax Tree of the input Python file. The general transcription process and the Abstract Syntax Tree are detailed in the following sections.

5.2.1 Abstract Syntax Tree (AST)

The Abstract Syntax Tree, commonly referred to by its abbreviation AST, is an easily readable tree representation of essential characteristics of the structure of a given source code [14]. An AST captures the essential structure of a program in tree form while omitting unnecessary syntactical details. Each node in the AST tree represents a construct from the source code. In compilers, ASTs are usually generated in the syntax analysis phase. Furthermore, they are frequently used for static code analysis [15].

Python's AST provides a variety of building blocks that can represent any Python file.

Example This subsection of the thesis provides a brief AST example to facilitate understanding in the following sections. The input Python file is a simple function that takes a string variable called "name" as an argument and returns the string "Hello " concatenated with the input name. For example, calling Hello("Joe") would return "Hello Joe". Below is the example source code and its corresponding AST:

```
def Hello(name:str) -> str:
   return("Hello "+name)
```

The Abstract Syntax Tree representation of the `Hello` function definition source code snippet can be viewed in Figure 3 below. It should be noted that the `body` attribute, which starts on the first line on the right, is at the same hierarchical level as the `args` attribute inside the `FunctionDef`.

Figure 3: Abstract Syntax Tree of `Hello` Source Code Example

The first line of Figure 3 defines a module, which represents the entire Python file and includes the rest of the source code representation in its body list.

In the example, the list only contains a single element at index 0, which is a "FunctionDef" object. This object contains 5 keys, of which only 4 are relevant for this example:

- **args**: This object contains the function arguments in a list. In the given example, there is only one "arg" object at index 0, which has two keys:
  - **annotation**: Specifies the type of the argument using type hinting. It can be a Constant, List, Tuple, Set, Dict, or Name (Variable) as in this example. The id of the Name (Variable) specifies the variable type inside an annotation (e.g., str).
  - **arg**: Specifies the name of the argument in the Python code.

- **body**: Lists all the objects in the body of the function. In the example, there is only a return statement. The return value is a binary operator (BinOp) with the operation "add" of a Constant (e.g., "Hello ") and a Name (e.g., the name variable).
- **name**: Represents the name of the function that is defined (e.g., "Hello").
- **returns**: Because the source code uses type hinting, the AST can also specify the type that the function returns. In this case, it is a Name (Variable) of type "str".

The example shows that the AST can be understood and parsed fairly easy and allows us to perform static code analysis and also process the code in a structured manner. One should note that is necessary to use type hints to insure that the types of the variables are known in case one wants to convert from python (untyped) code into a strongly typed language like C. Furthermore it is important to note that the types are only mentioned in the AST at the location of the type hints. They must be saved by the program that's parsing the AST to know what type an object has when used later in the code.

5.2.2 Domain Generating Algorithms

Domain Generating Algorithms (DGAs) [16], [17] are modern tools used by hackers, malware distributors, botnets, and potentially governments to thwart efforts to block specific programs from accessing their command and control servers. Many applications depend on servers to instruct controlled programs. For instance, an infected computer within a botnet connects to a designated domain to receive commands. Similarly, ransomware viruses rely on connecting to a control server operated by the attacker to verify ransom payments. Any program using a fixed domain for its command and control system becomes vulnerable if authorities seize that domain, disrupting communication and rendering the program unusable.

DGAs operate under the premise that both the controlled program and the control server implement the same DGA algorithm and seed. This algorithm generates a new domain each time it runs, with the result being predictable if the algorithm and seed are known. For example, a virus can be programmed to generate a new domain every 24 hours. The hacker, armed with the DGA algorithm and seed, can predict and register the same domain for their control server [17]. This practice enables control servers to change at regular intervals, complicating law enforcement efforts, especially when domains are switched frequently (referred to as "Fast Fluxing" Arntz). Moreover, if a control server is disabled, the virus only loses functionality until the DGA generates a new, uncompromised domain. This flexibility contrasts sharply with older programs that permanently ceased functioning if their control server was compromised.

The first publicly known DGA was used in the "Kraken" family of malware in 2008 [16].

DGA Abstract Class To make the various DGA implementations more readable and tidy, the Security and Privacy (SEC) group of the University of Vienna created a simple abstract class that all the DGAs implement.

The class consists of 3 functions:

- **__init__**: sets three variables that are used by all the DGA algorithms. The number of generated domains, the last domain created and a list of all previously generated domains.

```python
from abc import ABC, abstractmethod
class DGA(ABC):

    def __init__(self) -> None:
        self.generateddomains = 0
        self.lastdomain = ""
        self.domainhistory = list()

    def get_nextdomain(self) -> str:
        self.domainhistory.append(self.generate_domain())
        self.generateddomains += 1
        return self.domainhistory[-1]

    @abstractmethod
    def generate_domain(self) -> str:
        """shall be implemented by the inheriting class."""
        pass
```

Figure 4: Source Code of `dga_abstract_class.py`

- **get_nextdomain**: this function is used to make sure newly generated domains are added to the history correctly and the domain counter is increased.
- **generate_domain**: is an abstract method that is implemented by each of the DGAs and contain the specific implementation of a DGA algorithm.

DGA Example: Ramdo Below is the famous DGA Ramdo implemented in Python, which is used as example to better explain how the transcriber functions. First off it can be seen that it implements the abstract DGA class mentioned above.

5.2.3 Overview of Transcriber

Figure 6 below provides an overview of the Transcriber. It utilizes the Python AST library (version 3.11.5) to generate the Abstract Syntax Tree (AST), which is essential for parsing and transcribing the input and abstract class. Helper functions, such as linked lists in C, are implemented and stored in the "Export" folder, sourced from the "Includes" folder when the functionality is unavailable in the C language. This process is elaborated upon in subsequent chapters.

Following transcription, the output is moved to an Export folder named after the input file (e.g., DGARamdo in this example). In the final step, gcc (version 9.4.0) is employed to compile the final result and conduct error checks. Optionally, the file can also be automatically executed to facilitate comparison of the output.

5.2.4 Requirements for Transcription

The python to C transcriber was implemented using python. The transcriber is based on the python AST package which is used to generates the AST of the given input python file. The resulting AST can then be traversed easily using the *ast.walk* function provided in the package.

The transcriber is built specifically to convert DGAs from python to C. For this reason, there are some rules that have to be followed in order for the process to work correctly:

```python
from pr2.dga_classes.dga_abstract_class import DGA

class DGARamdo(DGA):
        """Implementation of the "Ramdo" DGA.

            This class extends the DGA abstract class.
        """
    def __init__(self, seed: int = 0xD5FFF):
        """Initialize the "DGARamdo" instance."""
        super().__init__()
        self.seed = seed
        self.nr = 0

    def generate_domain(self) -> str:
        """Generate a domain using the Ramdo DGA"""
        s = (2 * self.seed * (self.nr + 1))
        r = s ^ (26 * self.seed * self.nr)
        domain = ""
        for i in range(16):
            r = r & 0xFFFFFFFF
            domain += chr(r % 26 + ord('a'))
            r += (r ^ (s * i ** 2 * 26))
        domain += ".org"
        self.lastdomain = domain
        self.nr += 1
        return(self.lastdomain)
```

Figure 5: Source Code of `DGARamdo.py`

- **seed variable in __init__**: every DGA implementation must define a seed variable and its type.
- **class variables**: All class variables used anywhere in the code have to be defined in the __init__ class. This limitation is necessary because the transcriber creates a C struct representing the class. Even though python allows one to add class variables anywhere in the code, It would be unnecessarily complicated and out of the scope of this thesis to allow class variables to be added to a struct later in C code.
- **types cannot change**: Even though python would variable types to change during runtime, this is complicated to implement in C. To simplify this problem it was decided that only constant variable types are supported for this thesis. Every time a variable is assigned a new value the transcriber checks whether the type stored for this variable name and the assigned type are equal. Thus a variable's type cannot change during run time or a exception is thrown by the transcriber. (e.g. when variable *x* is defined as *int* but later on a *double* is assigned to the variable this results in an exception being thrown during the transcribing process)
- **parameters have to be type hinted**: every parameter used in a function except "self" have to be type hinted. If this where not the case it would not be possible to reliably determine the types via static code analysis.

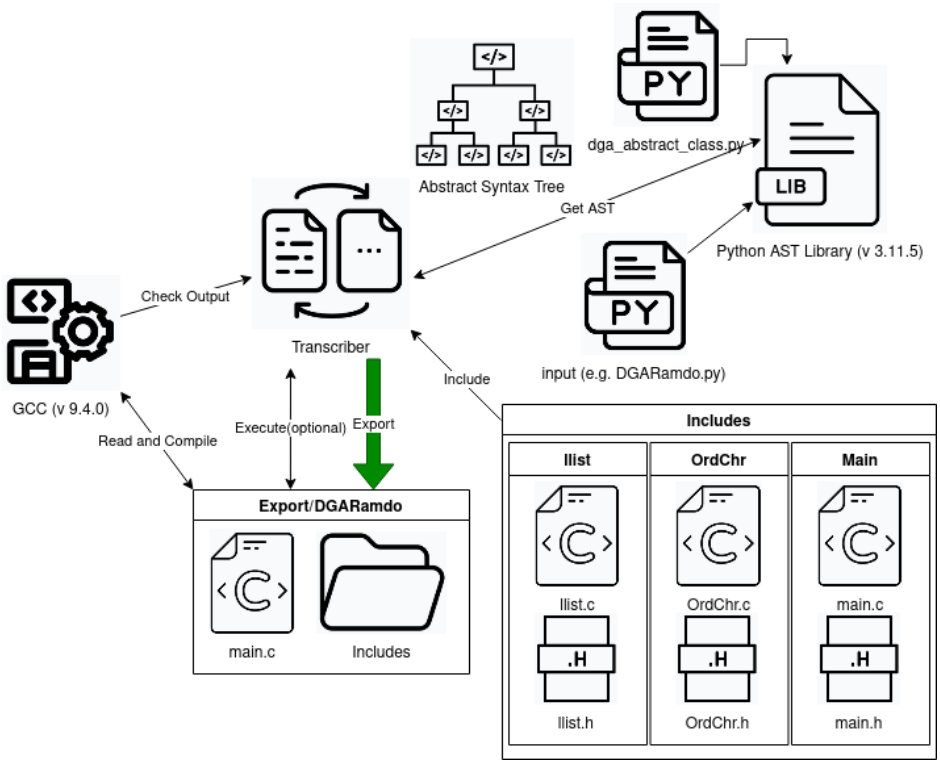

Figure 6: Overview of Transcriber

- **generate_domain** function: Every DGA algorithm implementation must include an implementation of the *generate_domain* abstract method defined in the *dga_abstract_class.py*. This function contains the main implementation of the specific DGA algorithm.

5.2.5 Overview of Transcription Process

This section starts by providing an overview of the transcription process of the *Transcriber*. Figure 7 provides an overview of the eleven steps of the transcription process.

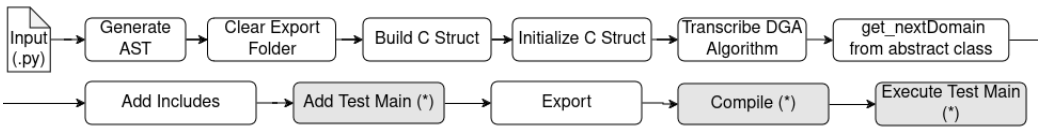

Figure 7: Overview of Transcription Process. The gray steps marked with (*) are optional.

In the first step, the AST of the Python input is generated. Next, the `Export` folder is cleared and the C `struct` representing the Python class is created. The following step is initializing this `struct` . After the initialization the main part of the DGA algorithm is transcribed. The `get_nextDomain()` implementation is transcribed from `dga_abstract_class.py` and appended. Next, the necessary includes are added. The final, *optional(*)*, step before exporting is adding the test main function. The completely transcribed file and all its dependencies (includes) are then exported. The last two *optional* steps are compiling the result and checking for errors and executing the test main function and comparing outputs.

A more detailed description of each step is given in the following subsections.

Generating AST In the first step the DGA file is input as Python code. The file has to conform to the format specified in section 5.2.2 for the transcription process to run smoothly. The transcriber then uses the *python.ast* library to generate an Abstract Syntax Tree (AST), which is explained in greater detail in section 5.2.1. The AST is generated for the input file and the `dga_abstract_class.py` file specified in section 5.2.2. It is used in the following steps to parse and transcribe the Python code.

Clear Export Folder To ensure a fresh execution and avoid errors caused by stale data, any remaining files in the export folder are deleted in this step prior to continuing with the actual transcription. The *IncludeList*, in the Transcriber that contains all the necessary include files for the C code, is also cleared. In addition, the Global- *VariableList*, that includes all the Variables used and their types, as well as the *StructVariableList*, that includea all Variables used in the C structure and their types, is also reset for the new transcription, insuring a fresh starting point.

Building the C Structure The goal of this step is to create a C structure that can represent and hold all the data used by the *DGA* python class. As mentioned above this only works correctly if all the class variables used are defined in the __init__ functions and the other requirements mentioned in 5.2.4 are fulfilled.

The transcriber first parses the *__init__* function of the *dga_abstract_class.py* class shown in section 5.2.2 above. The type of each of the variables used is determined and a corresponding C type or implementation is added to the C structure. In this case these variables are *self.generateddomains* of type *int*, **self.lastdomain** of type *str* and *self.domainhistory* of type *list*. These variables and their types are also saved in the Transcriber's *StructVariableList* so every previously defined variable's type is known later while transcribing.

Next, the transcriber parses the input file's *__init__* function for any class variables which are needed and appends the necessary variables in C code to the export. For the Ramdo code from section 5.2.2 above, the variables would be *self.seed* and *self.nr* of type *int*. By the end of this step the transcriber has produced the code for creating the structure in C that represents the python class.

Initialize C Structure The goal of this step is to generate a function named *init* that initializes the previously created C structure with the correct starting values. The values are parsed out of the input file's and the abstract classes' *__init__* functions. The *init* function generated in this step is appended to the export file after the structure definition to ensure that the structure is in the expected state when accessed later.

Transcribe DGA Algorithm In this step all other functions of the **input file** are transcribed. This includes the main algorithm implementation in the *generate_domain* function as well as any helper functions used. The functions are transcribed line by line. Whenever a class variable (e.g. *self.**) is used in the python code, this code is transcribed to use the *self* structure in the C code. Because all the variables are type hinted it is easy to determine the result type for a given operation, making sure all the C types are correct. If a function/operator is used with a return type that is not type hinted (e.g. *multiplication operator (*)*) the transcriber executes

the given function with some test input of the same type as the function input and check the return value type. For example when parsing *type int * type double* the transcriber executes *2 * 2.2* and typechecks the result to "see" that a double is returned in this example, this way the transcriber can often predict the return type without hard coding the return value types for all operations/functions.

Transcribe get_nextdomain from abstract class In this step the *get_nextdomain* function from the *dga_abstract_class.py* file is transcribed and added to the end of the export file.

Add Required C Implementations (includes) Whenever a python function/type is used that does not exist in C a corresponding C implementation must be provided. If for example the *list* type does not exist in C, a C list implementation was created that is used whenever a list is used in a python source file. At run time, the transcriber contains a *IncludeList* list variable that contains all used imports. Thus if a list is used for the first time in the python code, the corresponding C list implementation is added to the *IncludeList*. If a list is used again further down the source code file the transcriber checks the *IncludeList* and sees that the necessary input is already included. This prevents importing the required code multiple times. These required C implementations are stored in the *includes* folder of this project. The transcriber is architected so that new implementations can be easily added as future work to provide more functionality when transcribing. In this step the transcriber also adds all includes previously added to the *IncludeList* to the **beginning** of the export file to insure that all necessary includes are available when the program is executed normally: starting at the top of the file.

Add Test Main (*) This **optional(*)** step adds a test *main()* function to the exported C code that generates 3 new domains and print them to output. This step is further described in section 5.2.5.

Export A new folder is created for the transcribed output. This folder is named after the input DGA filename. So if the input file was named *DGARamdo.py* the export folder is named *DGARamdo* and multiple sub-folders are created for all imports (e.g. *llist*) within it. Finally the transcribed source code and all the import C implementations are copied to the corresponding folders. This is the last required step.

Compile (*) **Optionally** the exported code can be automatically compiled by the Transcriber using *gcc*. The Transcriber checks the *gcc* output to make sure no errors where thrown and the compilation was successful. This compiled file can then optionally be executed automatically in the next and final step.

Execute Test Main (*) In the final step the transcriber generates 3 domains using the input python file. It then executes the exported test *main()* C code and also generates 3 new domains. Checking if the 3 generated domains are the same for both programming languages is a useful indicator to see if transcription worked as expected.

6 Software Code Metrics

Source code metrics are an essential aid in modern software development. A software metric is a function used to measure to what degree a software system possesses a certain property [11]. The goal of using software metrics is to obtain objective, reproducible and quantifiable measurements of a property [10]. The result of applying said metric results in a numeric value.

The resulting information can be useful in a verity of applications including for quality assurance, performance, debugging, management, and estimating costs [18]. Source code metrics are thus an essential aid in modern software development. There are other metrics used in the modern software development. Indirect software measures such as project milestone status and reported system failures will not me covered [19]. This thesis solely focuses on software code metrics, thus the measurement of software attributes [20].

There are lot of different metrics being used today [10]. Some are easy to compute others require lots of data. This thesis concentrates on some of the most studied and well received metrics still in use today.

6.1 Studied Metrics

This section explores six widely studied and influential source code metrics [11]. The metrics covered include various aspects such as lines of code metrics, the ABC metric, Halstead metrics, Cyclomatic Complexity, and the Maintainability Index. Additionally, the Constructive Cost Model (COCOMO), which integrates some of these metrics and is commonly employed for predicting development costs, are discussed [6].

6.1.1 Lines of Code Related Metrics

In the early days of software development, the size of programs was commonly measured by counting the punch cards they occupied [4]. As punch cards fell out of use, metrics based on lines of code gained popularity [11]. These metrics, as their name implies, quantify the size of a program by counting the number of lines in its source code files [18]. Over the years, various lines of code metrics have been developed, each with its own rules regarding blank lines, comment lines, import lines, non-executable lines, and single braces [4].

- **Lines of Code (LOC)** - The total number of lines in a source code file is commonly referred to as Lines of Code (LOC) [21].
- **Source Lines Of Code (SLOC)** - The Source Lines of Code (SLOC) metric, sometimes also referred to as physical-SLOC is commonly defined as the number of lines in a source code file excluding empty lines, comments, standalone braces or parenthesis [22]. This metric aims to more accurately represents the amount of work performed [21].
- **Logical Lines of Code (LLOC)** - Logical Lines of Code, also referred to as LLOC, number of executable statements(XQT) or logical-SLOC is the number of statements in a piece of source code. These statements are usually terminated with a semi-colon (in C for Example) [21].
- **Non-Comment Lines of Code (NCLOC)** - Is the number of lines in a source code file excluding empty lines, comments, opening and closing paresis and include/import statements [21].

Lines of Code-related metrics have routinely been used as a tool to measure programmer productivity as well as program quality (errors per LOC). The LOC and SLOC metrics are commonly used as a basis to estimate the development cost of software projects [23] and affect a program's maintainability [21]. It can also be helpful to use the LOC and SLOC metrics to quantify the number of lines that were removed in a development step without impacting functionality.

6.1.2 Halstead Software Metrics

Maruice Howard Halstead introduced several software metrics in his 1977 book "Elements of Software Science" to aid establishing an empirical science of software development. Halstead observed that software metrics should capture the implementation or representation of algorithms across various programming languages, while remaining unaffected by the execution environment [7]. His objective was to define quantifiable characteristics of software and the relationships between them, which could be calculated directly from the source code. Halstead proposes measures for program length, volume and level, potential volume, language level, clar- ity, implementation time, and error rates [24].

Requirements The data required for the calculation of Halstead's metrics is easily obtainable for any given snippet of source code. Halstead states that software must be viewed as a string of operators and operands. Operands are defined as variables or constants, while operators are entities that can alter the value of an operand [24]. These straightforward definitions enable quantitative measures for many useful properties of programs.

The metrics only require:

- η_1 = number of distinct operators
- η_2 = number of distinct operands
- N_1 = total number of occurrences of operators
- N_2 = total number of occurrences of operands

Given these numbers multiple metrics can be calculated:

Length Halstead defined the Program length (N) of a well-structured program as the sum of occurrences of operators and operands [24].

$$N = N1 + N2 \tag{1}$$

Calculated Length Halstead Calculated Length is a metric that takes into account the unique operators and operands in the program, as well as their occurrences. Can be calculated using:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{2}$$

This formula measures the potential number of unique expressions that can be formed from the operators and operands in the program, providing a more refined measure of program complexity than Halstead Length alone [24].

Vocabulary Halstead defined the vocabulary (η) as the number of unique functions and unique operands. The availability of a particular operator in a given language does not produce a contribution to (η) unless that operator is actually used in the program[24]. While one would intuitively think the used programming language sets a upper limit to the number of unique operators in a program this is not the case, given the language allows the definition of new functions.

$$\eta = \eta_1 + \eta_2 \tag{3}$$

Program Volume The program volume (V) represents the size, in bits, of space necessary for storing the program if the vocabulary is encoded using uniform binary encoding [25]. The number of different entities in any program is given by its vocabulary. Thus it follows [24], that the number of bits required to encode each entity is given by:

$$V = N \log_2 \eta \tag{4}$$

Potential Volume While the Volume is defined above, it can be seen that if a program where to be translated into a different, more powerful, language the resulting version would have a smaller volume [24]. Consequently, the volume of a given implementation of a algorithm must inversely reflect the level at which it was written(of the language). Halstead argues that if we continued to translate an algorithm into an even higher language the volume would continue to decrease until it reaches a limit. Because even the most powerful language would still require operators and operands. For any given problem the most powerful language would provide a function ready to be invoked. Expressed in this (hypothetical) language, any algorithm would require exactly only two operators,first the required procedure and second a grouping operator [24]. The number of operands required would be algorithm dependent, but equals the sum of conceptually unique input and output operands.

The potential minimum volume is thus defined:

$$V* = (2 + \eta_2*) * \log_2(2 + \eta_2*) \tag{5}$$

The potential minimum volume of a algorithm is thus independent of programming language!

Implementation Level As mentioned above the volume is directly affected by the language chosen to write a program. Even the same programs written differently in the same language can have different volumes. Halstead defines program level as:

$$L = V^*/V \tag{6}$$

Program Difficulty The difficulty or error-proneness (D) of a program is directly related to the number of unique operators used in the code [25]. As the size or volume of a program grows, its overall level decreases while the difficulty rises [26]. Consequently, coding habits like the redundant use of operands or neglecting higher-level control structures typically lead to an increase in both program volume and difficulty [27].

$$D = V/V^* \tag{7}$$

Programming Effort This metric evaluates the cognitive effort required to convert a given algorithm into its implementation within a specific programming language. [27].

$$E = D * V \tag{8}$$

6.1.3 Cyclomatic Complexity

In 1976 Thomas J. McCabe proposed the *Cyclomatic Complexity (CC)* metric to measure the complexity of source code [20]. The CC measures the amount of decision logic in specific software and is based on the software's control flow graph [19].

Control flow graphs Control flow graphs are used to describe the logic structure of software modules. Each module represents a function in a typical programming language. With a single entry and exit point and can be called and return a value [20]. A control flow graph is composed of nodes, which represent computational expressions, and edges, which signify the transfer of control between nodes (a directed edge connects two nodes when the second command can be executed immediately after the first) [20]. Each potential execution path within a software module corresponds to a specific path from the entry to the exit node of the module's control flow graph.

McCabe Cyclomatic Complexity The cyclomatic complexity ($v(G)$)of a section of source code is defined as the number of linearly independent paths through it [20]. If there are no control flow statements (if/case ect) the complexity is 1, because there is only one specific path the program can run. If a program would contain a single if statement the complexity would be 2, because there would be one path for each possibility of the if statement (True/False). McCabe proposes to create a minimum of one test case for each individual path through a program[5].

Mathematically it is defined as:

$$CYC = E - N + 2 \tag{9}$$

In this context, E represents the number of edges and N represents the number of nodes within the control flow graph. [5].

```
1  void compare(a,b)
2  {
3      if (a && b)
4          x=1;
5      else
6          x=2;
7  }
```

```
1  void compared(a,b)
2  {
3      if (a)
4          if(b)
5              x=1;
6          else
7              x=2;
8      else
9          x=2;
10 }
```

Figure 8: Example of un-intuitive CC

Example One first glance one would assume there is one decision in this example but one has to look closer. The code is interpreted like this [28]:

Thus the correct CYC would be 3 in this Example. Because the CYC can sometime be un-intuitive at first glace, McCabe proposed developers should document the flow graph of a program [5].

Myers Extended Cyclomatic Complexity In 1977 Glenford J Myers, of the IBM research center new york, released a paper called "An extension to the cyclomatic measure of program complexity" [29]. In this paper Myers discussed some weaknesses and anomalies of the orginal proposal by McCabe and suggests an improved version of the algorithm that should eliminate these problems.

Myers Anomaly Example Myers gives an example to explain the anomaly he discovered [29]:

```
A  IF (X = 0)  THEN ...
               ELSE ...

B  IF (X = 0) & (Y > 1) THEN ...
                        ELSE ...

C  IF (X = 0) THEN
               IF (Y > 1) THEN ...
                          ELSE ...
               ELSE ...
```

Figure 9: Caption

The premise is that statement C has more control complexity than statement B while statement B has more complexity than statement A [29]. Thus the complexity ordering is $A < B < C$. If one would now calculate the CYC (V(G)) by diagramming statements one would get $V(A) = 2$, $V(B) = 2$ and $V(C) = 3$. Since $2 < 2 < 3$ is false, there is a problem. If one would compute V(G) by counting individual conditions (plus one) one would get $V(A) = 2$, $V(B) = 3$, $V(C) = 3$. Again, $2 < 3 < 3$ is false [29].

Myers concludes that "apparently both the number of decision statements and the number of conditions in predicates contribute to complexity, so both must be considered" [29].

Myers proposed solution is to calculate V(G) as complexity interval and proposed the number of decision statements + 1 as the lower bound, while the upper bound is the number of individual conditions + 1. N-way CASE statements are counted as N-1 decision and conditions.

Computing V(G) for the above Example, we get $V(A) = 2 : 2$, $V(B) = 2 : 3$, and $V(C) = 3 : 3$, which satisfies the intuitive relationship.

6.1.4 COCOMO

While this paper is primarily focused on source code metrics, a brief description of the the Constructive Cost Model (COCOMO), which estimates software project costs in man/months, is included as two of the metric tools provide support to calculate this value, in addition to other source code metrics. In the late 1970s and early 1980s, as software engineering was emerging, managers sought methods to estimate software development costs and explore project organization options. This led to the development of commercial and proprietary cost/schedule estimation models [6].

First proposed by Barry Boehm in 1981, the Constructive Cost Model (COCOMO), become (and arguably remains) the worlds most widely used cost estimation model [30]. The original COCOMO (now sometimes referred as COCOMO81) is a regression model based on the study of 63 projects. The current version COCOMO II, is based on a study of 161 projects [31], which makes it one of the best-documented models [32].

Unlike other metrics examined during this thesis the COCOMO model relies on additional information, aside from the source code, to calculate its results. Specifically when applying the COCOMO method, software projects are classified as "Organic" , "Semidettached" or "Embedded" based on their size and complexity. There are multiple versions and levels of COCOMO evaluation. The basic version uses the formula $E = a(KLOC)^b$, where $E = numberofman/months$, $KLOC = linesofcode(inthousands)$ and a and b are factors determined by the projects classification.

In the intermediate COCOMO evaluation a collection of cost estimation factors related to environment in which the code is run and the team developing the project are specified [8]. These factors are combined to form an Effort Multiplier, which is multiplied with the result of the basic method.

6.1.5 Maintainability Index

Paul Oman and Jack Hagemeister wanted to come up with a metric for general mentainability [10]. The definition of "maintainabilty" is taken from the IEEE definition:

> "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment." (IEEE, 1990)

In 1991, Paul Oman and Jack Hagemeister examined definitions from 35 published works on the topic and identified 92 maintainability attributes [10]. The idea was to organize these 92 attributes into a hierarchical tree structure, which could be used to evaluate the relative maintainability of software systems [9].

New metrics were created for attributes that did not already have measurement criteria. The authors proposed possible metrics that could be used to measure each individual attribute [33]. In 1994, Oman and Hagemeister, in cooperation with HP, attempted to identify the smallest set of metrics needed to predict maintainability effectively [10]. For this purpose, HP provided eight sets of source code, each ranging from 1,000 to 10,000 lines, along with subjective engineering evaluations of code quality. Software complexity and quality metrics were computed for all eight modules. From the observed correlations and a principal component analysis of the covariances, they developed three distinct maintainability models:

Single Metric Model The simplest model only uses Halstead's Effort calculated individually per module, called (aE) :

$$MI = 125 - 10 * \log(aE) \tag{10}$$

Four Metric Model This model is based on the average Halstead's Effort per module (aE),the average extended cyclomatic complexity per module (aC),the average Lines of Code per module ($aLOC$), and average Comment Lines (CMT) [10].

$$MI = 171 - 3,42 * \ln(aE) - 0,23 * aC - 16,2 * \ln(aLOC) + 0,99 * aCMT \tag{11}$$

Five Metric Model Uses the average Halstead's Effort per module (aE),the average extended cyclomatic complexity per module (aC),the average Lines of Code per module ($aLOC$), and average Comment Lines ($aCMT$) and a 20 point subjective evaluation of external documentation(DOC).

$$MI = 138 - 2.76 * ln(aE) - 0.33 * aC - 12.2 * ln(aLOC) + 0.88 * aCMT + 1.04 * DOC \quad (12)$$

Model Improvements While the inital models did appear to be good predictors of maintainability the MI was fine-tuned over time as more data , from varying industrial sources, and weaknesses became available. According to Welker, Oman, and Atkinson the MI has three problems [33]. First, the five-metric model, by including the DOC metric, became more subjective. Second, both the four and five-metric models appeared overly sensitive to comments. And thirdly there was debate if the Halstead Volume metric should be used instead of Healstead Effort. To combat these shortcomings improved models where suggested [10] [33].

Improved Three-Metric Model Is based on average Halstead's Volume per module(aV), average extended cyclomatic complexity (aC), average Lines of Code (aLOC):

$$MI = 171 - 5,2 * ln(aV) - 0,23 * aC - 16,2 * ln(aLOC) \quad (13)$$

Improved Four-Metric Model Is based on average Halstead's Volume per module(aV), average extended cyclomatic complexity (aC), average Lines of Code ($aLOC$) and the average per cent of lines of comments per module (aCM):

$$MI = 171 - 5,2 * ln(aV) - 0,23 * aC - (5)16,2 * ln(aLOC) + 50 * sin(\sqrt{2,4 * aCM}) \quad (14)$$

6.1.6 ABC Metric

The ABC metric was introduced by Jerry Fitzpatrick in 1997 as an alternative to the Lines of Code (LOC) metric. Fitzpatrick argued that the LOC measure lacks a solid theoretical basis, as there is no clear correlation between the number of lines of code and a program's functionality. He also criticized LOC for being an unreliable metric, offering the following example to illustrate its shortcomings [34].:

```
1 bool SearchList(char *cmd, char **list, unsigned n) {
2 for(unsigned i=0; i < n; i++)
3     if (stricmp(cmd, list[i]) == 0) return true;
4     return false;
5 }
```

Figure 10: SearchList Variation 1

Although these snippets are equal functionally, their LOC value differs by a factor of 3! This demonstrates that the LOC metric is sensitive to programming style [34]. Furthermore when comparing programs in different languages LOC has very little value because a higher level language could provide much more functionality with less lines of code.

```
bool SearchList(
char *cmd, // command string
char **list, // array of strings
unsigned n) // max. no. of elements
{
unsigned i;
for(unsigned i=0; i < n; i++) // for each element
    {
    if (stricmp(cmd, list[i]) == 0) // matched?
        {
        return true; // found match
        }
    }
return false; // no match
}
```

Figure 11: SearchList Variation 2

ABC Calculations The idea of Jerry Fritzpatrick was to base the ABC metric on something with a theoretical foundation for measuring "useful work" [34]. Fitzpatrick recognized that imperative programming languages rely on variables to carry out meaningful tasks. He noted that these languages are built upon just three fundamental operations [34]:

- Assignment - Assigning data to a variable (usually data storage)
- Branch - an explicit out of scope forward program branch (function calls)
- Condition - A logical or Boolean test such as if, else ect.

ABC values are expressed as an ordered triplet of integers, with the counts always listed in the order A, B, and C, enclosed in angle brackets. For example, a program might have ABC values of <9, 4, 7>, where 9 represents the number of assignments, 4 is the number of branches, and 7 is the number of condition tests [34].

Fitzpatrick also suggested using a scalar size measure to simplify code comparison. Since each value in an ABC vector represents a distinct entity, summing the values directly to obtain a single measure is not appropriate. Instead, a single ABC value can be derived by calculating the magnitude of the vector, as the ABC components are orthogonal and have comparable scales [34].:

$$|ABC| = sqrt((A * A) + (B * B) + (C * C)) \tag{15}$$

Fitzpatrick asserts that an ABC magnitude value should always be accompanied by its corresponding ABC vector. Additionally, due to semantic variations among programming languages, the rules for counting ABC values must be interpreted within the context of the specific language used [34].

C Calculation Rules This thesis only covers the proposed calculation for the C programming language but Fitzpatrick also proposes calculations for Java and C++.

The rules for counting are below [34]:

1. Assignment count raised for each instance of an assignment operator in the source code, excluding constant declarations:

```
+= = *= /= %= &= <<= >>= |= ^=
```

2. Assignment count should be increased for each increment or decrement operation:

```
++ --
```

3. Branch count raisFor each goto statement that targets a deeper level of nesting than the level of the goto itself, one is added to the Branch count.ed for each function call
4. For each goto statement that targets a deeper level of nesting than the level of the goto itself, one is added to the Branch count.
5. Condition count raised for each conditional operator:

```
< > <= >= == !=
```

6. Condition counter increase for

```
else case default ?
```

7. Cndition count increased for each unary conditional expression

```
: (for example "a ? b : c")
```

6.2 Areas of Applicability

Software code metrics can be useful in a variety of applications, including quality assurance, performance evaluation, debugging, project management, and cost estimation [18]. The following sections detail how these metrics are used. Additionally, research comparing various metrics is analyzed, and commonly used software metric tools are presented.

6.2.1 Testing and Quality Assurance

McCabe proposed cyclomatic complexity to be a cornerstone of modern software testing [5]. He recommended that a minimum of one test case should be written for each individual path in the control flow graph of a program. Documenting the flow graph of a program and calculating the cyclomatic complexity should help developers discover paths that might otherwise be overlooked. McCabe makes it clear that additional testing is required to cover other conditions in the program.

6.2.2 Software Cost and Time prediction

Leung stresses that accurate software cost estimates are critical to both the companies developing software and their customers [23]. These estimates are used for generating proposals, contract negotiations, scheduling, monitoring, and control. Inaccurate software cost estimation can lead to various problems. Underestimation may cause management to approve proposals that exceed the budget or planned timeframe, resulting in underdeveloped functions and poor quality. Conversely, overestimation may lead to an excessive commitment of resources or failure to win contracts due to inflated offers. Ultimately, customers expect actual development costs to align with the estimated costs.

Most cost estimation models aim to generate an effort estimate (usually in person-months/hours), which is then used to calculate project duration and cost. These models typically rely on program size metrics. However, since the exact program size can only be determined after the project is completed, it must be estimated. Therefore, the accuracy of size estimation has a direct impact on the accuracy of cost estimation. [23].

6.2.3 Related Work

Source code metrics have been well studied in research [4]. Below, four noteworthy papers are looked at in detail. The first and second papers, authored by Curtis, Sheppard, Milliman, and Basili and Phillips, compare different complexity metrics in real-world experiments. The third paper by Lincke, Lundberg, and Löwe evaluates various software metric tools. Finally, the fourth paper is a systematic review that analyzes a total of 226 studies on source code metrics.

Curtis, Sheppard, and Milliman Curtis, Sheppard, and Milliman studied the application of software complexity metrics for predicting programmer performance. [37]. An experiment was conducted with 54 experienced FORTRAN programmers, each tasked with locating a single bug in three provided source code files. Performance was assessed based on the time required to locate and fix each bug.

To account for individual performance differences, a within-subject 3-factorial design was employed. Three different types of control flow were defined for each of the three programs. Each program was presented in three lengths and included three different semantic bugs, resulting in a total of 81 experimental conditions. Participants were divided into two groups of 27 subjects each. Group 1 addressed all 81 conditions, whereas Group 2 faced the same conditions but in a different task order.

For each task, participants received the FORTRAN source code containing the bug, the correct output, and the incorrect output produced by the faulty program. Additionally, explanations for any functions used but not included in the source code were provided.

The following complexity metrics where used:

- Halstead's Effort (E)
- McCabe Cyclomatic Complexity (v(G))
- Number of Statements (Length)

Curtis, Sheppard, and Milliman looked at the inter-correlations among the metrics at both the subroutine level and the total program. It was observed that all

three metric values where similar on the subroutine level[37]. At the program level big differences where observed however. While the correlation between v(G) and Length remained large, the correlation of the two with Halstead's E was considerably smaller.

Table 1 shows the correlations between performance and the three metric values.

	Correlations	
Measures	Subroutine	Program
E	.66	.75
v(G)	.63	.65
Length	.67	.52

Table 1: Correlation among Complexity Metrics [37]

It was observed that the three metrics predicted performance similarly well at the subroutine level. However, significant differences were noted at the programming level. Curtis, Sheppard, and Milliman concluded that as the size of a program increases, Halstead's E appears to be a better measure of complexity compared to LOC and v(G)v(G). Furthermore, they concluded that both Halstead's E and McCabe cyclomatic complexity are directly related to the difficulty programmers experience in locating errors.

Basili and Phillips Basili and Phillips tried to evaluate and compare different software metrics in a cooperation with NASA and the Goddard Space Flight Center [35]. The Source codes analyzed consisted of 50-110,000 lines of FORTRAN code used in the ground support software of satellites. The developers provided data on the effort of coding said source code as well as the number of errors they ran into. This data was then compared with multiple metrics:

- Halstead's Effort (E)
- McCabe Cyclomatic Complexity
- Lines of Source Code
- Number Executable Statements (XQT)

Basili and Phillips found that the executable statements metric correlated the most with the actual effort reported. Followed by LOC, Cyclomatic Complexity and Halstead E in that order.

Lincke, Lundberg, and Löwe Lincke, Lundberg, and Löwe compared different software metrics tools in their article 'Comparing software metrics tools' [36]. They showed that code metrics are implemented differently in various software metrics tools, resulting in divergent metric values. This means that even if the software metrics provide a specific metric by name, it does not guarantee that this metric was implemented exactly according to the original algorithm. This is important to realize because it means one cannot simply compare the values calculated by different software metric tools.

Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, A 2017 systematic mapping study [11] was conducted that selected and analyzed a total of 226 studies about source code metrics. The aim was to provide an overview of the current state of the art on the use of source code metrics. The study found that the Lines of Code Metric, Cyclomatic Complexity, and the Chidamber and Kemerer metrics were the most extensively researched software code metrics, with fault prediction and complexity recurring as key areas of interest within the research community [11].

It is noteworthy that the Chidamber and Kemerer metrics enjoy widespread popularity in Java projects. However, during the research conducted for this thesis, no open-source tools supporting these metrics for Python input could be identified, which is why this metric is not cover in depth in this thesis.

6.3 Studied Metric Tools

This thesis investigated a total of 13 metric tools. The libraries where chosen because they are open-source, popular and easy to install and use. Furthermore, libraries supporting both Python and C where preferred. Some other common non open-source metric tools or metric tools within IDE's are discussed briefly in Section 6.4. Most of the thesis focuses on open source tools to assure possible replication at no cost.

6.3.1 CCCC (v3.1.4)

CCCC is an abbreviation for *C and C++ Code Counter* and was first released in 1997. It is the oldest metrics tool studied in this thesis. CCCC started as Tim Littlefair's master project and is licensed under GPL2 [38]. The original version of CCCC supported C, C++, and Ada95 language files as input. Current versions of CCCC support Java instead of Ada95 [39]. CCCC offers support for various metrics including Lines of Code (LOC), Cyclomatic Complexity (CC), Comment Lines & Ratio, alongside several other lesser-known metrics. These additional metrics are not elaborated upon due to their limited usage and lack of support in the other tools studied. CCCC has detailed documentation and provides definitions for each of its supported metrics [39]. CCCC has been used to calculate metrics in multiple scientific papers since its release [11].

Usage & Output After installing, *CCCC* was run via the command line using the following syntax [40]:

```
cccc <filename>
```

Unique among the tools studied in this thesis, CCCC does not output its results in the console. Instead, it creates a (hidden) folder named CCCC after execution that contains the output in *XML*, *HTML*, and *DB* formats. The metrics results are listed both per method (in the anonymous.xml/.html files) and overall (in cccc.xml/.html).

6.3.2 SLOCCount

As the name suggests, David A. Wheeler's SLOCCount is specialized solely on the Source Lines of Code (SLOC) metric. The current version of the tool supports twenty-seven input languages, including Python and C [41]. SLOCCount can process large amounts of data and made news in 2001 when Wheeler claimed to have calculated

the SLOC of Red Hat Linux (v7.1) at 30 million [42]. Since then, SLOCCount has been well-regarded and used in multiple papers. In 2001 [43] and 2005 [44], SLOCCount was used in an attempt to calculate the SLOC of the entire Debian code base. Similar experiments have been undertaken for multiple Linux distributions since then [45]. SLOCCount is referred to as "the original SLOC counter" in the documentation [46] of the SCC tool detailed in Section 6.3.12.

Usage & Output After installation SLOCCount was run via the command line using the following syntax [40]:

```
sloccount <filename / folder>
```

To suppress the calculation of COCOMO we used:

```
sloccount --details <filename / folder>
```

6.3.3 CLOC (v1.82-1)

The *Count Lines of Code* (CLOC) library was started by AlDanial and first released in 2006 under a GPL2 license [47]. Since then, over 100 people have contributed to the project and more than seventeen thousand people have starred it on GitHub. The CLOC library focuses on wide language support, portability, and execution speed at the cost of supporting fewer metrics. According to the project's website, the tool currently supports 250 programming languages, including Python and C [48]. The CLOC library measures the Source Lines of Code (SLOC) metric and also outputs the number of blank lines and comments. The library is optimized for execution speed and portability, running on a variety of operating systems such as multiple flavors of Linux, FreeBSD, NetBSD, OpenBSD, macOS, AIX, HP-UX, Solaris, IRIX, z/OS, and Windows [47]. It is worth noting that CLOC incorporates code from David Wheeler's SLOCCount [47].

Usage The CLOC library supports four different comment ratio variants depending on which calculation mode is selected via flags:

- The formula $\frac{Comments}{LOC}$ can be used with:
  `"cloc <filename> --by-percent c"`
- For $\frac{Comments}{LOC+Comments}$ one can use:
  `"cloc <filename> --by-percent cm"`
- The formula $\frac{Comments}{LOC+Blanks}$ can be used with:
  `"cloc <filename> --by-percent cb"`
- For $\frac{Comments}{LOC+Comments}$ the following command is used :
  `"cloc <filename> --by-percent cmb"`

6.3.4 Lizard (v. 1.17.10)

Lizard is a static code analysis program that has been developed and maintained by Terry Yin since 2012. According to its GitHub page, Lizard is a "extensible Cyclomatic Complexity Analyzer for many programming languages" [49]. Lizard supports twenty input languages including C/C++/C# , Java, JavaScript and Python. Lizard supports calculating four metrics: Cyclomatic Complexity, Non-Comment

Lines of Code (NCLOC), token count of functions and the parameter count of functions. In this thesis we look only at the CC and the NCLOC metrics because they are also supported by the other studied metric tools and can therefore be compared. LIZARD's Github project currently has 45 contributors, is starred by over 1700 people and LIZARD is used by over 640 other Github projects [49].

Usage After successfully installing the LIZARD library it was accessed via the command line using the following command [40]:

```
lizard --xml <filename>
```

6.3.5 PyCQA's McCabe (v0.7.0)

This complexity checker tool is being developed by the Python Code Quality Authority (PyCQA), an organization for Python code quality tools since 2013. PyCQA's McCabe is published under the Expat license and only supports calculating the Cyclomatic Complexity metric for Python input [50]. Pylint, a popular Python static code analysis tool that checks for errors and enforces coding standards, uses PyCQA's McCabe to provide cyclomatic complexity analysis [51]. The project currently has twenty-one contributors on GitHub and is starred by 620 people.

Usage & Output After successfully installing PyCQA's McCabe, the tool was run via the command line using the following syntax [40]:

```
python3 -m mccabe <filename/folder>
```

The Cyclomatic Complexity is calculated for each function/method and the results are output to the console as text.

6.3.6 Ohcount (v.4.0.0-1)

In 2008, the open-source indexing company Ohloh (now Open Hub) started developing *Ohcount*. The tool's name comes from "Ohloh's source code line counter". As the name suggests, this tool specializes in lines of code-related metrics [52]. Ohcount can be used to calculate the LOC, SLOC, number of blank lines, and comment ratio metrics in over seventy programming languages including Python and C [53]. Ohcount's GitHub repository highlights the support of multiple different languages within a single file (e.g., JavaScript and CSS within an HTML file)[52]. To date, Ohcount has 105 contributors[54] and has been used to count over 6 billion lines of code [55].

Usage After successful installation *Ohcount* the tool was run via the command line using the following syntax [40]:

```
ohcount <filenames>
```

6.3.7 Radon (v. 5.1.0)

Radon is a metrics tool implemented in Python and created by Michele Lacchia, first released on GitHub on September 20, 2012 [56]. Radon supports only Python input but claims to calculate multiple lines of code-related metrics, Halstead Metrics, Cyclomatic Complexity, and the Maintainability Index. The supported lines of

code-related metrics are: LOC, SLOC, LLOC, number of blanks, number of single-line comments, and the number of multi-line comments [56]. Radon is actively being developed and currently has over 1600 stars and 58 contributors on GitHub. It is used as a dependency in over 4600 other GitHub projects [56]. Radon is well-documented and is used to calculate Python code metrics by multiple for-profit companies including codacy.com, codefactor.io, coala.io, and codeclimate.com.

Usage & Output After successful installation Radon can either be run as a command line program or imported into the python program and directly called. The Radon library chose to split its metrics into the four sub-commands "hal", "raw", "cc" and "mi" that each are explained below.

hal All of the Radon library's Halstead metrics are combined into the *hal* sub-command. Radon calculates the Halstead metrics **per file**. The following command was used throughout this thesis:

```
radon hal <filename/folder> -j
```

The *-j* flag returns JSON formatted output. For the *Func_Baseline.py* file this would result in the following output:

```
{
"Method_Baseline.py": {
    "total": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.0, 0.0],
    "functions": [
        ["main", [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.0, 0.0]]
        ]
    }
}
```

raw

The *raw* sub-command calculates the supported Lines of Code metrics **per file**:

```
radon raw <filename> -j
```

Which results in:

```
 {
 "Method_Baseline.py": {
    "loc": 2,
    "lloc": 2,
    "sloc": 2,
    "comments": 0,
    "multi": 0,
    "blank": 0,
    "single_comments": 0
    }
}
```

cc The *cc* sub-command calculates the cyclomatic complexity **per method and class**. The following command was used to call the library via the command line:

```
radon cc <filename/folder> -j
```

The *-j* flag returns the output in JSON format. For easier manual reading one can output everything in the command line neatly by using the following command:

```
radon cc <filename/folder> -s
```

The results of running the above command on the *DGARamdo.py* file can be seen below:

```
DGARamdo.py
    C 4:0 DGARamdo - A (3)
    M 12:4 DGARamdo.generate_domain - A (2)
    M 7:4 DGARamdo.__init__ - A (1)
```

Radon calculates the cyclomatic complexity per method and class. Each line of the output starts with either "C" or "M", depending on if its a class or method. At the end of each line we can see the calculated cyclomatic complexity in the braces.

mi The *mi* sub-command is used to calculate the Maintainability Index:

```
radon mi <filename/folder> -j
```

6.3.8 Metrics (v. 0.3.3)

Metrics is a Python library written by Mark Fink, first released on GitHub in 2013 [40]. The Metrics library officially supports Python, C, C++, Go, and JavaScript source code [40], but was specifically designed to allow for easy addition of additional languages. Many more programming languages have been added by other contributors since the first release of the library, resulting in over 500 programming languages unofficially being supported currently [57]. The library calculates the following four source code metrics **per input file**:

- Cyclomatic Complexity - Cyclomatic complexity is calculated once for the entire input file, not per method.
- Source Lines of Code (SLOC) - According to the documentation the Metrics library "counts the lines but excludes empty lines and comments" [40].
- Lines of Comments - Total comment lines in input file.
- Comment Ratio - The total number of comments lines divided by the SLOC metric.

Usage & Output The *Metrics* library is accessible via the command line after installation. [40]. The following command was used throughout this thesis:

```
metrics <filename> --q --format json
```

The combination of *–q* (quiet) and *–format json* flags are used to get the output returned in JSON format without additional text. Below is an example output for the Func_Baseline.py file:

```
{
    "files": {
        "Method_Baseline.py": {
            "block_positions": [
                {
                    "end": 2,
                    "name": "main",
                    "start": 1,
                    "type": "Function"
```

```
                }
            ],
            "comments": 0,
            "language": "Python",
            "mccabe": 0,
            "ratio_comment_to_code": 0.0,
            "sloc": 2
        }
    }
}
```


6.3.9 Tokei (v12.1.2)

Tokei is a lines-of-code-related metrics tool developed since 2015 that supports the calculation of LOC (lines of code), SLOC (source lines of code), and the total number of comment and blank lines. As of the time of writing, Tokei claimed to support 226 programming languages including Python and C. This project is very popular, with more than 9600 stars on GitHub and over 230 individual contributors [58]. Tokei has been downloaded over 800,000 times from GitHub and Crates.io alone [59][58]. Tokei purports to offer high speed without compromising accuracy and uses the catchphrase "Count your code, quickly". Tokei is implemented as a small state machine rather than with regular expressions found in other code counters [60]. Tokei specifically asserts its capability to accurately tally multi-line comments, nested comments, and to exclude comments embedded within strings [58]. Tokei is distributed under both the MIT and the Apache licenses.

Usage & Output After installation *Tokei* was run via the command line for this thesis using the following syntax [40]:

```
tokei <filename / folder>
```

By default *Tokei* prints its output to the console as text. However, this tool can also output its results in cbor, json and yaml formats by setting the `--output` flag [58]. To get the output formatted as json we used:

```
tokei --output json <filename / name>
```

It should further be noted that Tokei doesn't count hidden files or .gitignore files by default. If these files should be counted, the following command has to be used [58]:

```
tokei --no-ignore <filename / name>
```

6.3.10 GoCLOC (v0.5.2)

As the name suggests, GoCLOC is a metrics tool implemented in Go and based on the CLOC tool detailed in section 6.3.3. GoCLOC was first released in 2016 and is inspired by Tokei (section 6.3.9). GoCLOC claims to be faster than both Tokei and the original CLOC tool. The GoCLOC project, which is released under the MIT license, has twenty-one contributors and is currently starred 739 times on GitHub. GoCLOC claims to support calculating the SLOC and number of comment & blank lines for over 175 input languages [61]. Both the Python and C programming languages are supported.

Usage & Output In the documentation, instructions are provided for a variety of ways to use the GoCLOC tool. GoCLOC can be used in combination with the Jenkins continuous integration server, GitHub Actions, and Docker. Additionally, GoCLOC can be executed as a traditional program that outputs its results to the console. By default, the result is formatted as text; additionally, output can be formatted in the same XML schema that CLOC uses and as JSON. The following command was used for this thesis to format the result as JSON:

```
./gocloc <filename/folder> --output-type json
```

6.3.11 Multimetric (v. 2.0.3)

Multimetric is a metric library first released on Dec 21, 2019 [62]. It is created and maintained by Konrad Weihmann and is mostly written in Python. The library claims full support for 24 programming languages including: Java, Python, C, Go [62]. The library can calculate a variety of different software code metrics **per input file**:

- Cyclomatic Complexity - calculated once overall for entire input file, not per method.
- Lines of Code (LOC) - overall LOC of entire file
- Comment ratio - calculated for entire file.
- Halstead Metrics - calculates halstead metrics for entire file.
- Fan Out - calculates Fanout for entire file.

Usage & Output After successfully installing the *multimetric* library, it was used in this thesis via the command line [40].

```
multimetric <filename>
```

The multimetric library returns *.json* formatted output by default and can optionally takes multiple files as input. The library calculates the metrics for each of the input files and exports a json file with the following information for each input file (output for *Func_Baseline.py* shown) :


```
{
  "files": {
    "Method_Baseline.py": {
      "comment_ratio": 0.0,
      "cyclomatic_complexity": 2,
      "fanout_external": 0,
      "fanout_internal": 0,
      "halstead_bugprop": 0.00655049481813441,
      "halstead_difficulty": 2.5,
      "halstead_effort": 49.12871113600807,
      "halstead_timerequired": 2.7293728408893374,
      "halstead_volume": 19.651484454403228,
      "lang": [
        "Python"
      ],
      "loc": 1,
```

```
      "operands_sum": 2,
      "operands_uniq": 2,
      "operators_sum": 5,
      "operators_uniq": 5
    }
  }
```

If multiple files are input simultaneously the multimetric library calculates the *max*, *mean*, *median*, *min* and the standard deviation (sd) across all input files. This feature was not used or tested in this thesis.

6.3.12 SCC (v3.1.0)

The Succinct Code Counter (SCC) was created by Ben Boyter and first released on GitHub in 2018. SCC was implemented in Go with the stated goal of being the fastest code counter possible, while also performing COCOMO calculations and estimating code complexity [46]. The SCC project, which is dual-licensed under MIT and UNLICENSE, currently has 82 contributors on GitHub and has been starred over 5700 times. The version used in this thesis claims to support 239 input languages including Python and C [46]. SCC is described as 'A tool similar to CLOC, SLOCCount, and Tokei' in its documentation. Currently, SCC supports calculating the LOC, SLOC, number of comment & blank lines, as well as cyclomatic complexity metrics.

Usage & Output After installation *SCC* was run for this thesis via the command line using the following syntax [40]:

```
scc <filename/folder>
```

SCC prints its output to console as text by default, but supports over ten other output formats including json, csv and html. This thesis used the following command to get the output in json format:

```
scc <filename/folder> --format json
```

Enabling the `--trace` flag results in SCC indicating whether a line was classified as code, comment, or blank for all lines encountered during the counting process. No additional information about SCC's cyclomatic complexity measure is given however.

6.4 Other Popular Metric Tools

Software code metrics have been incorporated into multiple different software tools. Below some of the most common non open-source solutions are quickly mentioned due to their widespread use.

6.4.1 Eclipse Metrics Plug-in

Created by Sauer [63], this tool is an open-source metrics calculation plugin for the Eclipse IDE. Once integrated into Eclipse, it facilitates the calculation of various code metrics, including Lines of Code (LOC), McCabe cyclomatic complexity, and Weighted Methods per Class (WMC)—which is the sum of the McCabe cyclomatic complexity values for all methods within a class. Additionally, the plugin

offers support for other metrics beyond the scope of this thesis, providing a comprehensive suite of tools for analyzing and measuring code complexity and quality within the Eclipse development environment.

6.4.2 Visual Studio

Visual Studio has multiple different code metrics build into their IDE [64]. This allows Developers to easily generate code metrics data for their code. The code metric data can be generated for single projects or even entire solutions. Visual Studio supports the calculation of:

- Maintainability Index
- McCabe Cyclomatic Complexity
- Lines of Source Code
- Other more advanced Metrics

6.4.3 IntelliJ CodeMetrics (Add-on)

The CodeMetrics [65] add-on for the IntelliJ IDE is a very useful tool to keep track of source code complexity while programming. This add-on uses close approximation of McCabe cyclomatic complexity and is very useful to show the complexity of classes while coding them.

6.4.4 SAP Code Inspector

SAP Code Inspector allows users to automatically check programs for name conventions and also provides the following metrics:

- Halstead's Effort and Volume
- McCabe Cyclomatic Complexity
- Lines of Code

6.5 Criticism of Source Code Metrics

There has been some criticism of code metrics. Shepperd wrote a paper criticizing McCabe's cyclomatic complexity [66]. Shepperd argued that it was grounded in a flawed theoretical understanding and an inaccurate model of software development. It concluded that cyclomatic complexity is questionable on both theoretical and empirical grounds. The fundamental objection brought forth by this paper is the inconsistent behavior when measuring modularized software. v(G)v(G) is sensitive to the number of subroutines/functions within a piece of source code because McCabe suggests that these subroutines should be treated as unconnected components within the control graph. This results in a higher complexity value as a program is divided into smaller, simpler, and more maintainable modules. Furthermore, Shepperd questions if it is acceptable to completely ignore non-linear code (i.e., code that does not contain any decisions) in a model with which general software complexity is measured.

Coulter criticized Halstead's Software Science [67]. He argued that Halstead's EE predicts the number of elementary mental discriminations needed to be made

by an individual to generate a given program. Halstead assumed that humans performed binary searches on operands and operators when writing programs. Coulter cites studies that do not support the human brain's binary search assumption, thus questioning the foundations of Halstead's work.

7 General Overview of Software Code Metric Tools

This section provides a general overview of the 13 open-source code metric tools studied in this thesis. This section solely summarizes and documents information asserted in the programs respective documentation, while Section 8 and Section 9 scrutinize and compare the actual output of the tools for Python and C input.

This section begins by providing general information about each tool, including the specific version used for this thesis, along with details about their licensing terms, accepted input languages, and an overview of the domains/areas for which metrics are provided. Moreover, the definitions of the metrics that are supported are evaluated and compared, if they are provided in the documentation. Additional information about each tool is provided in Section 6.3, where the tools are first presented. Subsection 7.1 offers a more detailed overview of the individual Lines of Code-related metrics supported by the tools. Subsection 7.2 delves into the Cyclomatic Complexity-related metrics supported by the tools, while Subsection 7.3 provides a deeper insight into the specific Halstead metrics and Subsection 7.4 focuses on Maintainability Index metrics. The final Subsection 7.5, briefly mentions the tools that support the COCOMO models.

The first column in Table 2 identifies the tools used in this thesis along with their specific versions. The "License" column outlines the terms under which the programs were published. Within the "Languages" column, a checkmark (✓) in the ".py" sub-column signifies support for Python input, while ".c" indicates support for the C language. The "total" sub-column tallies the total number of languages supported by each tool. A checkmark in a tool's "LOC" sub-column indicates support for at least one lines of code related metric. The "CC" sub-column denotes support for Cyclomatic Complexity metrics, "Hal" for Halstead metrics, "MI" for Maintainability Index metrics, and "COCOCO" for the COCOMO model calculation.

Library	**License**	**Languages**			**Metrics**				
		.py	*.c*	*total*	LOC	CC	Hal	MI	COCOMO
CCCC (v3.1.4)	GPL2		✓	3	✓	✓			
CLOC (v1.82-1)	GPL2	✓	✓	250+	✓				
Lizard (v.1.17.10)	MIT	✓	✓	20	✓	✓			
PyCGA's CC (v.0.7.0)	MIT	✓		1		✓			
Metrics(v0.3.3)	MIT	✓	✓	500+	✓	✓			
Multimetric(v2.0.3)	Zlib	✓	✓	24+	✓	✓	✓	✓	
Ohcount(v4.0.0-1)	GPL2	✓	✓	70+	✓				
Radon(v5.1.0)	MIT	✓		1	✓	✓	✓	✓	
SCC (v3.1.0)	MIT	✓	✓	239+	✓	✓			✓
SLOCCount (v2.26-5.2)	GPL	✓	✓	27	✓				✓
GoCLOC (v0.5.2)	MIT	✓	✓	176+	✓				
Tokei(v12.1.2)	MIT	✓	✓	226+	✓				

Table 2: General Overview of the Studied Metric Tools

Although CCCC, RADON, and PYCGA's CC only support one of the studied languages, they were included in this thesis as comparison values for the other tools.

Lines of Code-related metrics are the most widely supported with 11 out of 12 of the studied tools supporting metrics from this group. Subsection 7.1 details the specific Lines of Code-related metrics supported by these tools. The calculation of Cyclomatic Complexity-related metrics is supported by 7 out of 12 of the tools. Subsection 7.2 provides a more detailed overview of the Cyclomatic Complexity-related metrics supported by the studied tools. Both the Halstead and Maintainability Index metrics are only supported by MULTIMETRIC and RADON, while functionality to calculate the COCOMO model is only provided by SCC and SLOCCOUNT.

Multi-File Input Additionally, different behaviours can be observed between the tools when calculating metrics for all files in a folder (and sub folders). In this common real world use case, the folder would commonly contain a program that consists of multiple source files which may be implemented in different programming languages.

- METRICS only parses the root folder and ignores folders and files in sub folders. The command `find * -type f | grep '\.py$' | xargs metrics` can be used as a work around.
- MULTIMETRIC throws an error if a folder is encountered. Additionally, unsupported file endings result in an error.
- CLOC, OHCOUNT and GOCLOC ignore empty files by default (0 byte files).
- RADON ignores hidden files by default.
- A bug occurs when the same function/method name is used in multiple input files when calculating CCCC's metrics that are grouped by function/method.

7.1 Lines of Code Related Metrics Overview

This section lists the different lines of code related metrics that are supported by the studied tools. Additionally, the definitions of these metrics are compared if provided in the tools documentation. The findings are summarized in Table 3. The first column in the table names the tools. The "(.c)" after CCCC indicates that Python input is not supported, while the "(.py)" after the PYCGA's CC and RADON tools are indications that neither tool supports C code. Within the "Main Metrics" column, a checkmark (✓) in the "LOC" sub-column signifies support for the Lines of Code (LOC) metric, while the "SLOC" sub-column indicates Source Lines of Code support. The "Other" sub-column lists additional variations of the Lines of Code Metric that are supported by the tools. A checkmark in a tool's "Blank" column indicates the tool includes the number of blank lines in its output. The "Lines" sub-column of the "Comment" column denotes the tool counts and outputs the number of comments, while the "Ratio" sub-column signifies support for some variation of the Comment Ratio metric. The two sub-columns in the "Multi-File Grouping" column identify if the results can be grouped by either "File" or "Language", when analyzing multiple files at once.

The results in Table 3 show that 5 out of the 13 metric tools support calculating the Lines of Code (LOC) metric, while 9 tools claim to support the Source Lines of Code (SLOC) metric. LIZARD is the only tool that offers the Non Comment Lines of Code (NCLOC) metric and RADON solely supports the Logical Lines of Code (LLOC) metric. The number of blank lines is returned by 6 of the 13 tools. 8 tools support calculating the number of comment lines, while 6 implement a variation of the Comment Ratio measure.

Library	Main Metrics			Blank	Comment		Multi-File Grouping	
	LOC	*SLOC*	*Other*	*Lines*	Lines	Ratio	File	Language
CCCC (.c)		✓			✓	✓	✓	
CLOC		✓		✓	✓	✓		✓
Lizard			NCLOC				✓	
PyCGA's CC (.py)								
Metrics		✓			✓	✓	✓	✓
Multimetric	✓					✓	✓	
Ohcount	✓	✓		✓	✓	✓	✓	✓
Radon (.py)	✓	✓	LLOC	✓	✓	✓	✓	
SCC	✓	✓		✓	✓		✓	✓
SLOCCount		✓						✓
GoCLOC		✓		✓	✓		✓	✓
Tokei	✓	✓		✓	✓		✓	✓

Table 3: Overview of Lines of Code Related Functionality of the Tools

7.1.1 Definitions

The objective of this section is to contrast the metric definitions outlined in the documentation of the tools, where available. This is done to ascertain whether the tools utilize consistent definitions for ostensibly identical metrics, a crucial factor in facilitating precise comparisons between the results produced by the tools.

Lines Of Code (LOC)

- RADON claims to calculate "the total number of lines of code" defined as the sum of the SLOC metric, blank lines, and all multi- and single-line comment lines [68].
- MULTIMETRIC claims to calculate the "Lines of code" and further specifies the expected range of "1..(inf)" [62].
- Neither OHCOUNT [52] nor SCC [46] provide additional information aside from "Lines of code".

Source Lines of Code (SLOC)

- CLOC's SLOC is calculated by subtracting the number of blank and comment lines from the total lines in the file [47].
- METRICS "The SLOC metric counts the lines but excludes empty lines and comments" [40] .
- OHCOUNT No definition except "Count lines of code per file" is provided [52].
- RADON The basic definition provided is:"The number of source lines of code". Furthermore Radon's definition of LOC can be expressed as: $SLOC = LOC - (MultiComments + Singlecomments + Blank)$ [56].
- SCC No definition except "For counting the lines of code" is provided [46].
- SLOCCOUNT "SLOCCount counts physical SLOC, also called "non-blank, non-comment lines" [69].

- GoCLOC No specific definition is given except "count lines of code" [61].
- CCCC counts non-blank, non-comment lines of source code. Preprocessor lines are considered as blank. Class and function declarations are counted, while declarations of global data are excluded. There might be some double counting of lines within class definitions, as the algorithm aggregates the total lines in a module by summing the lines within the module itself and the lines associated with its member functions. This means that declarations and definitions of member functions contained within the body of a class definition may contribute to both counts. [39]

Comment Ratio

CLOC, METRICS, MULTIMETRIC, OHCOUNT, and RADON support the comment ratio metrics for Python input files. It is important to note that the libraries use different definitions and formulas to calculate the comment ratio metric. Firstly, RADON is the only studied tool that breaks its comment lines metrics down into three sub-metrics and returns the number of hashtag denoted comments (#C), as well as the number of single line and multi-line comments (MC) that are denoted using quotes (e.g. doc strings). Table 4 provides an overview of the different formulas the tools provide to calculate the comment ratio metric.

Tool	**Comment-Ratio Metric Formulas**						
	$\frac{C}{LOC}$	$\frac{C}{SLOC}$	$\frac{C}{C+SLOC}$	$\frac{C}{Blank+SLOC}$	$\frac{\#C}{LOC}$	$\frac{\#C}{SLOC}$	$\frac{\#C+MC}{LOC}$
CLOC	✓	✓	✓	✓			
METRICS		✓					
MULTIMETRIC			✓				
OHCOUNT			✓				
RADON					✓	✓	✓

Table 4: Overview of Supported Comment-Ratio Formulas

Most noteworthy, it can be seen that RADON uses unconventional formulas to calculate its comment ratio metrics, which effectively ignore all single line non-`#`-comments.

7.2 Cyclomatic Complexity Related Metrics

Table 5 provides an overview of the tools claimed Cyclomatic Complexity functionality and output of the tools when analyzing a single or multiple files at once. In the case of single-file inputs, the table displays whether the tools output the total CC per file, the average CC overall, or the CC per method. Similarly, check marks in the "Multi File" sub-columns indicate whether the tools provide the total and average CC for all files collectively, on a per-file basis, or even per method when multiple files are input.

The main takeaways from Table 5 are that LIZARD is the single studied tool that supports calculating Myers Extended Cyclomatic Complexity and that the tools differ drastically in which Cyclomatic Complexity grouping and output options they provide. CCCC , LIZARD, PYCGA's CC and RADON allow calculating the Cyclomatic Complexity metric individually for each function/method, while METRICS and MULTIMETRIC only allow calculating the overall CC of a file.

Library	Myers' CC	Single File Input			Multi File Input				
		Per File		*Method*	*Overall*		*Per File*		*Method*
		CC	*avg CC*	*CC*	*CC*	*avg CC*	*CC*	*avg CC*	*CC*
CCCC (.c)		✓	✓	✓	✓	✓			✓
LIZARD	✓	✓	✓	✓	✓	✓	✓	✓	✓
PYCGA(.py)				✓					
RADON (.py)			✓	✓			✓		✓
METRICS		✓			✓		✓		
MULTIMETRIC		✓			✓		✓		
SCC		✓	✓		✓		✓		

Table 5: Overview of Cyclometic Complexity Related Functionality of the Tools

7.3 Halstead Metrics

RADON and MULTIMETRIC are the sole two programs among the 13 studied metric tools that offer support for Halstead metrics. Table 6 provides an overview of the Halstead metrics both tools claim to provide.

Description	Radon	Multimetric
η_1 = number of distinct operators	✓	✓
η_2 = number of distinct operands	✓	✓
N_1 = total number of occurrences of operators	✓	✓
N_2 = total number of occurrences of operands	✓	✓
Program vocabulary ($\eta = \eta_1 + \eta_2$)	✓	
Program length ($N = N_1 + N_2$)	✓	
Calculated program length ($\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$)	✓	
Volume ($V = N \log_2 \eta$)	✓	✓
Difficulty ($D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$)	✓	✓
Effort ($E = D * V$)	✓	✓
Time required to program ($T = \frac{E}{18sec}$)	✓	✓
Number of delivered bugs ($B = \frac{V}{3000}$)	✓	✓

Table 6: Halstead Operator and Operand Counting Differences

Notably MULTIMETRIC does not output the program vocabulary (η) or program length (N), even though multiple other metrics that depend on these values are provided in the output. Additionally, MULTIMETRIC provides two methods and definitions to calculate the numver of delivered bugs which are covered in Subsection 7.3.1.

7.3.1 Definitions

The documentation of RADON and MULTIMETRIC provide the same definition for all of their supported Halstead metrics except for the number of delivered bugs (B). RADON provides the definition: $B = \frac{V}{3000}$, while MULTIMETRIC provides two different methods to calculate the number of delivered bugs ($B = \frac{V}{3000}$ and $B = \frac{(E*\frac{2}{3})}{3000}$).

7.4 Maintainability Index

As discussed in Section 6.1.5 various iterations of the Maintainability Index (MI) metric have been developed. These iterations share a common reliance on a combination of Halstead metrics and Cyclomatic Complexity and Lines of Code related metrics for MI calculation. Consequently, RADON and MULTIMETRIC, which where the only studied tools that incorporate the Halstead metric, are the only tools that offer features to compute the Maintainability Index.

7.4.1 Definitions

Unfortunately the tools share no common definition for calculating the MI. MULTIMETRIC claims to support three different variations of the Maintainability Index [62]:

- The `classic` mode uses the original formula proposed by Halstead:

  $$MI = 171 - 5.2 * ln(aveVol) - 0.23 * aveV(g') - 16.2 * ln(aveLoC)$$

  $aveVol$ denotes the average Halstead Volume, $aveV(g')$ is average extended cyclomatic complexity per module and $aveLoC$ is the average Lines Of Code
- The `SEI` mode calculates the maintainability index using:

  $$\begin{aligned} MI =& 171 - 5.2 \cdot \ln(aveV) - 0.23 \cdot aveV(g') \\ & - 16.2 \cdot \ln(aveLOC) - 50 \cdot \sin(\sqrt{2.4 \cdot perCM}) \end{aligned}$$

  , where $aveV$ the average Halstead Volume V per module, $aveV(g')$ is the average extended cyclomatic complexity per module and $aveLoC$ is the average Lines Of Code and $perCM$ is the average percent of lines of comments per module.
- The `microsoft` mode uses the formula:

  $$MI = \max[0, (171 - 5.2 * ln(V) - 0.23 * CC - 16.2 * ln(LoC)) * 100/171]$$

  Where V is the Halstead volume, CC is the cyclomatic complexity and LoC is the lines of code.

While RADON' Maintainability Index metric is a derivative, computed from both SEI and Visual Studio's Maintainability Index:

$$MI = \max\left[0, 100\frac{171 - 5.2\ln V - 0.23G - 16.2\ln L + 50\sin(\sqrt{2.4C}))}{171}\right]$$

Where V is the Halstead Volume, G is the total Cyclomatic Complexity, L is the number of Source Lines of Code (SLOC) and C is the percentage of comment lines (important: converted to radians) [68]. Note that the SEI formula used by RADON uses the comment ratio value rather than converting the comment percentage to radians. This difference also contributes to dissimilar results [70].

By examining the formulas used by the tools it is clear that results are not directly comparable, because there is no way to set both tools to use a common formula.

7.5 COCOMO

SLOCCount and SCC were the only two studied metric tools that provided functionality to calculate COCOMO. As described in Section 6.1.4, additional information beyond the source code is required for the calculation. Both tools allow the project classification to be specified via their options. When the same classification settings are used, the COCOMO result is solely dependent on the SLOC metric.

8 Python Specific Results

This section compares the results of the different metric libraries when analyzing the same Python input code. By doing this we achieved multiple goals. First, we increase our understand of the variance in results returned by the different libraries for the same metric. Second, the results also determined how sensible it is for individual metric results to be compared across different metric tools. Finally, some shortcomings and bugs of the individual libraries and their impact on the results are described.

Table 3 provides an overview of the metrics that the libraries claim to support. Due to lack of Python language support the CCCC metric tool is not covered in this section. The *Analysis Tool* mentioned in Section 5.1 was used to automatically calculate, bundle and export the results of the various metric tools for different Python input files. The *Line by Line* functionality detailed in section 5.1.2 was used to better determine how an overall metric score was achieved and which lines of code where most influential. In addition, test cases where written to better understand the differences in the results returned by the libraries. The detailed approach as well as the results are studied more thoroughly in the respective individual subsections. Each experiment can be reproduced easily by following the instructions provided in each subsection.

The following abbreviations where used for the tools names to conserve space when necessary: SLOCCount (SloC.), OhCount (OhC.), Radon (Rad.), Metrics (Met.) and GoCLOC (GoC.).

Invalid Syntax Check

Both Radon and PyCGA's CC perform Python syntax checking on the input files during CC metric computation. For this reason, the tools do not report results if they detect syntax errors. Because the *Analysis Tool*'s line-by-line mode only sends a portion of the file for analysis in each iteration, syntax errors may appear when subsequent lines are necessary to complete a syntactical structure. These errors automatically resolve themselves when the subsequent lines are submitted and the syntactical structure is complete.

8.1 Lines of Code (LOC) Metric Results

Multimetric, OhCount, Radon, SCC and Tokei are the five out of 13 studied tools that claims to support calculating the traditional Lines of Code (LOC) metric for Python code.

8.1.1 Results

OhCount, Radon, SCC and Tokei always returned the same LOC result for a variety of different input files, while Multimetric returns the number of input files instead

of the lines of code for its *loc* attribute in the returned JSON, which must be considered an error. This pattern can be observed in the test results shown in table 7 and 8.

Three DGA Files and DGA Abstract Class In this test case we compare the LOC metric results of the MULTIMETRIC, OHCOUNT, RADON, SCC and TOKEI tools for the three Python DGA files `DGARamdo.py` , `DGABanjori.py` , `DGAFobberV1.py` and their common abstract class `dga_abstract_class.py` . The results in Table 7 show that all tools except MULTIMETRIC return the same results for all four input files.

	Lines Of Code (LOC)				
File	Multimetric	OhCount	Radon	SCC	Tokei
`dga_abstract_class.py`	1	21	21	21	21
`DGARamdo.py`	1	26	26	26	26
`DGABanjori.py`	1	28	28	28	28
`DGAFobberV1.py`	1	29	29	29	29

Table 7: Comparison of LOC Results for Python DGA Files

Biggest Python File of 5 Popular Python Libraries Python DGAs are generally fairly short and simple programs without complex syntactic sugar or complex data structures. Therefore the biggest python file of five popular open source python libraries was used as more complex real world test data. The files used where:

- `app.py` - the file containing the main implementation of the `Flask` python web server library(v3.0).
- `test_feedexport.py` - a file of test cases that can be found in the `Scrapy` webscraping library(v2.11.0).
- `figure.py` - part of the `Mathplotlib` (v3.8.2)
- `compiler.py` - the biggest file that contains a lot of logic of `Sqlalchemy` library (v2.0.23).
- `generic.py` - biggest file of `Pandas` python library (v2.1.4)

The results of the LOC metrics for these much bigger and more complex files can be seen in Table 8.

8.1.2 Analysis of Results

The test in this section demonstrate that all metric programs, with exception of MULTIMETRIC returned the same results. The MULTIMETRIC tool returns the number of files analyzed as the Lines of Code metric, which must be considered an error.

File	Lines Of Code (LOC)				
	MULTIMETRIC	OHCOUNT	RADON	SCC	TOKEI
(FLASK) `app.py`	1	1481	1481	1481	1481
(SCRAPY) `test_feedexport.py`	1	2966	2966	2966	2966
(MATPLOTLIB) `figure.py`	1	3626	3626	3626	3626
(SQLALCHEMY) `compiler.py`	1	7690	7690	7690	7690
(PANDAS) `generic.py`	1	13976	13976	13976	13976

Table 8: LOC Results of the Biggest Python File of Five Popular Libraries

8.2 Source Lines of Code (SLOC) Metric

CLOC, METRICS, OHCOUNT, RADON, SCC, SLOCCOUNT, GOCLOC and TOKEI are the eight tools that claim to support calculating the Source Lines of Code (SLOC) metric for Python code. It's important to highlight that TOKEI offers two distinct configurations for managing multi-line comments in Python language input. By default, TOKEI considers Python multi-line comments indicated with triple-quotes (docstring comment notation) as code. However, setting the `treat_doc_strings_as_comments` flag to `True` in `.tokeirc` interprets these occurrences as comments, as per the documentation[60]. Thus, each table in this Section lists both outputs in the *default* and *.ini* columns of the TOKEI tool.

8.2.1 Results

This section compares the metric tools SLOC results, using real world Python files as input. Table 54 and 9 show significant differences for the SLOC metric values calculated by the different metric tools.

Three DGA Files and DGA Abstract Class Table 9 lists the SLOC results of the tools when analyzing the three Python DGA files and their common abstract class.

	Source Lines of Code Comparison							
	SloC.	CLOC	OhC.	Rad.	Met.	Tokei	GoC.	SCC
						default | ini		
`dga_abstract_class.py`	13	13	13	13	13	16 | 16	13	16
`DGARamdo.py`	18	18	18	18	18	24 | 20	18	25
`DGABanjori.py`	16	16	16	16	16	22 | 17	16	21
`DGAFobberV1.py`	19	19	19	19	19	26 | 23	19	28

Table 9: Comparison of SLOC Result for Python DGA Files

Five Popular Python Libraries Table 10 provides an overview of the magnitude of differences of the tools SLOC result when analyzing all python files of the five popular libraries previously mentioned. The percentages in the tools columns denote the difference from the median value of all tools results for that file. The median was chosen over the average because it is better suitable to uncover outlier values. The raw output of the tools can be seen in the appendix in Section 11.2.1.

Tool & Mean SLOC	SloC.	CLOC	OhC.	Rad.	Met.	Tokei default | ini		GoC.	SCC
Flask [9,977]	-1%	≈	=	≈	≈	+34%	≈	≈	+50%
Scrapy [46,872]	+1%	≈		+3%	≈	+11%	≈	≈	+18%
Matplotlib [137,697]	+1%	-2%	-2%	=	=	+39%	-1%	-2%	+50%
SQLAlchemy [400,724]	≈	≈	=	+3%	≈	+18%	≈	≈	+31%
Pandas [389,043]	+2%	=	≈	+4%	≈	+23%	≈	≈	+31%

Table 10: Variance from Median overall SLOC results for the Flask , Scrapy , Matplotlib, SQLAlchemy and Pandas Libraries.

The results summarized in table 10 show that all tools, except Tokei in default mode and SCC, delivered results within 4% of the median value for all tests. If the default settings for Tokei are used the results are also between 11% and 39% higher than the median results. SCC's results were between 18% and 50% percent higher.

8.2.2 Analysis of Results

In order to determine the reasons for the differences in results described in Section 8.2.1, the Line-by-Line mode of the Analysis tool was run on several Python files. The results of the Line-by-Line analysis were used to develop a total of 13 test cases to isolate the causes of the observed differences in SLOC results of the studied tools. Table 11 provides an overview of scenarios that lead to differences between the studied tools' SLOC metric for Python input. A checkmark in a specific tool's column of a specific row means the condition stated in the first column of that row influences the overall SLOC result. The cases marked with [1] only occur when using Tokei's default settings.

	SLOC Counting Errors							
	SlocC.	CLOC	OhC.	Rad.	Met.	Tokei	GoC.	SCC
All lines following quote inside non-`#` comment								✓
In `"""`-comments:								
Non blank lines						✓[1]		
Blank lines						✓[1]		
Single line `"""` comments						✓		
In `'''`-comments:								
Non blank lines	✓					✓[1]	✓	
Blank lines						✓[1]		
Single line `'''` comments						✓		
In function parameters:								
multi line `str` (`"`)		✓	✓	✓	✓	✓	✓	✓
multi line `str` (`'`)	✓	✓	✓	✓	✓	✓	✓	✓
multi line `str` (`"""`)				✓		✓		
multi line `str` (`'''`)	✓			✓		✓	✓	
`#`-comments				✓				

Table 11: Overview of the Tools SLOC Counting Differences

Multiple differences in the SLOC metric results where observed between the tools, depending on which quotes are used to denote multi-line string parameters (see 8.2.2). The most important difference observed are:

- The SCC metric tool contains a major bug that occurs when quotes (`"` / `'`) are encountered within a triple-quote (`"""` / `'''`) denoted comment. In this specific case the library continues to count all following lines as code - even if the lines are blank or comments (see 8.2.2).
- By default TOKEI counts all lines (including blanks) inside a tripple-quote denoted comments as code (see test 8.2.2). If TOKEI's Python specific flag `treat_doc_strings_as_comments` is set, the SLOC metric is unaffected by all multi-line comments. However, **single line** tripple-quote denoted comments are always counted by TOKEI, independent of the setting used.
- SLOCCOUNT and GOCLOC count non blank lines inside multi-line tripple-**single**-quotes (`'''`) denoted comments as code for their SLOC metric. However their SLOC result is unaffected if the same comment is denoted using tripple-**double**-quotes (`"""`).
- SLOCCOUNT only raises its SLOC metric by one for each multi-line (`"`)-string parameter, independent of the number of lines the string spans. All the other

tools raise the SLOC by one per line of the multi-line string (blank lines and `#`-comments are ignored).

- When triple double quotes (`"""`) are used to denote the multi line string parameter only Radon and Tokei raise there SLOC for each line, while the other tools only add one (independent of number of lines).
- Only Radons SLOC metric is affected by hashtag comments (`#`) between the parameters of a function. The other tools (correctly) ignored these comments and only count the lines containing parameters.
- Denoting the multi-line string parameter using triple single quotes (`'''`) resulted in SLOCCount, Radon, Tokei and GoCLOC counting all lines, while CLOC, OhCount, Metrics and SCC only raised their SLOC by one.

The following paragraphs present Python test cases to illustrate discrepancies in the Source Lines of Code (SLOC) calculations performed by the studied tools.

Single Line Comments and Blank Lines This first test case showcases the tools handling of single line comments, blank lines and a variable decleration code line. This test's source code and the SLOC results of the tools for this test case can be seen in Table 12.

		SLOC Line by Line of SingleLineComments.py							
	Line of Code	SloC.	CLOC	OhC.	Rad.	Met.	Tokei def|ini	GoC.	SCC
1	`def ComTest():`	1	1	1	1	1	1|1	1	1
2	`  # comment`	1	1	1	1	1	1|1	1	1
3	`  """ DQ """`	1	1	1	1	1	**2**|**2**	1	1
4	`  ''' SQ '''`	**2**	1	1	1	1	**3**|**3**	**2**	1
5		2	1	1	1	1	3|3	2	1
6	`  x = 1`	3	2	2	2	2	4|4	3	2

Table 12: Line by Line SLOC Comparison for `SingleLineComments.py`

The line by line SLOC results in table 12 show that the SLOC metric of all tested tools was unaffected by the `#`-comment. Tokei is the sole tool affected by single-line triple-double-quote comments (`"""`) in the calculation of SLOC. Notably, this influence persists irrespective of the tool's configuration settings (see 6.3.9). SLOCCount, GoCLOC and Tokei (irrespective of configuration) increased their SLOC metric for the single-line triple-single-quote comments (`'''`). None of the tools SLOC metric was affected by blank lines and all tools increased their SLOC metric for a simple variable assignment.

Multi Line Comments (Double Quote Notation) This test case focuses on differences when encountering tripple-double-quote (`"""`) denoted multi line comments for Python input. The source code and results of the test can be seen in Table 13. In addition, Radon returns an error if there are syntax errors in the input code (see lines 2-5). These error values do not affect the overall test results once the syntax errors are resolved. CLOC and OhCount may return incorrect results if syntax errors are present (in parenthesis), but their results readjust once the syntax is correct (see line 6).

Line of Code	SLOC Line by Line of `DQMultiLineComment.py` SloC.	CLOC	OhC.	Rad.	Met.	Tokei def|ini	GoC.	SCC
1 `def DQTest():`	1	1	1	1	1	1|1	1	1
2 `"""`	1	(2)	(2)	err	1	**2**|1	1	1
3 `double quote`	1	(3)	(3)	err	1	**3**|1	1	1
4 `comment`	1	(4)	(4)	err	1	**4**|1	1	1
5	1	(4)	(4)	err	1	**5**|1	1	1
6 `"""`	1	1	1	1	1	**6**|1	1	1
7 `print("DQT")`	2	2	2	2	2	7|2	2	2

Table 13: Line by Line comparison of SLOC metrics for `DQMultiLineComment.py`

Multi Line Comments (Single Quote Notation) This test is similar to the previous test case (8.2.2). The only difference is that tripple-**single**-quotes (`'''`) are used instead of tripple-**double**-quotes to denote the multi line comment. The results and source code of the experiment can be seen in Table 17.

Line of Code	SLOC Line by Line of `SQMultiLineComment.py` SloC.	CLOC	OhC.	Rad.	Met.	Tokei def|ini	GoC.	SCC
1 `def SQTest():`	1	1	1	1	1	1|1	1	1
2 `'''`	**2**	(2)	(2)	err	1	**2**|1	**2**	1
3 `single quote`	**3**	(3)	(3)	err	1	**3**|1	**3**	1
4 `comment`	**4**	(4)	(4)	err	1	**4**|1	**4**	1
5	4	(4)	(4)	err	1	**5**|1	4	1
6 `'''`	**5**	1	1	1	1	**6**|1	**5**	1
7 `print("DQT")`	6	2	2	2	2	7|2	6	2

Table 14: Line by Line SLOC Results for `SQMultiLineComment.py`

The key take away from this experiment is that SLOCCOUNT, GOCLOC and TOKEI (using default settings) count text inside of the tripple-**single**-quote multi-line comments as code. However, if TOKEI's `treat_doc_strings_as_comments` flag is set this tool ignores these multi line comments.

Multi Line String Parameter Variations The following four test cases uncover differences in the tools handling of string parameters that span multiple lines. All four test cases are identical, except for using different quotes to denote the string parameter.

Table 15: Multi Line String Parameter Variations Source Codes

```
def main():
 print(
  "multi"
  "line"
  "string"
 )
```

```
def main():
 print(
  'multi'
  'line'
  'string'
 )
```

```
def main():
 print(
  """multi"""
  """line"""
  """string"""
 )
```

```
def main():
 print(
  '''multi'''
  '''line'''
  '''string'''
 )
```

The SLOC results of the studied tools for these four files are listed in Table 16.

The results in Table 16 show the final output of the different tools for these four test files. Note that CLOC, OHCOUNT, METRICS and SCC all produced identical

	SLOC Tests of Multi-line String Parameter							
	SloC.	CLOC	OhC.	Rad.	Met.	Tokei def|ini	GoC.	SCC
`ParamsDQ.py`	**3**	6	6	6	6	6|6	6	6
`ParamsSQ.py`	6	6	6	6	6	6|6	6	6
`ParamsTDQ.py`	**3**	**3**	**3**	6	**3**	6|6	**3**	**3**
`ParamsTSQ.py`	6	**3**	**3**	6	**3**	6|6	6	**3**

Table 16: SLOC Results of `ParamsDQ.py`, `ParamsSQ.py`, `ParamsTDQ.py` and `ParamsTSQ.py`

(correct) results. The other tools yielded values that where either half or twice of the correct value depending on how parameter strings are delimited in the source code.

Quote Inside of Quote Denoted Comment (SCC Bug) This test case showcases SCC's bug when a quote is encountered inside a quote denoted comment.

		SLOC Line by Line of `QuoteBug.py`							
	Line of Code	SloC.	CLOC	OhC.	Rad.	Met.	Tokei def|ini	GoC.	SCC
1	`def QuoteBug():`	1	1	1	1	1	1|1	1	1
2	`  """ >"< """`	1	1	1	1	1	2|2	1	**2(1)**
3		1	1	1	1	1	2|2	1	**3**
4	`  # comment`	1	1	1	1	1	2|2	1	**4**
5	`  x = 1`	2	2	2	2	2	3|3	2	**5**

Table 17: Line by Line SLOC Results of `QuoteBug.py`.

Table 17 showcases how SCC counts all lines after the quote is encountered inside a quote denoted comment are incorrectly classified as code. The (1) highlights that SCC did not count the single line `"""` denoted comment in the first Test Case 12.

Empty String Declarations (SCC Bug) This test case is designed to show case a bug that occurs in SCC if a empty string variable is declared using quotes. Importantly the same result can be observed independent of which quotation style is used (single and tripple double/single quote notation).

		SLOC Line by Line of `EmptyStringBug.py`							
	Line of Code	SloC.	CLOC	OhC.	Rad.	Met.	Tokei def|ini	GoC.	SCC
1	`def StringBug():`	1	1	1	1	1	1|1	1	1
2	`  string = ""`	2	2	2	2	2	2|2	2	2
3		2	2	2	2	2	2|2	2	**3**
4	`  # comment`	2	2	2	2	2	2|2	2	**4**
5	`  x = 1`	3	3	3	3	3	3|3	3	5

Table 18: Line by Line SLOC Results of `EmptyStringBug.py`.

The key take away from Table 18 is that SCC counts all lines after line 2 as code independent of the lines content.

8.3 Cyclomatic Complexity Related Metrics

This section documents the differences between the Cyclomatic Complexity results of Lizard, Metrics, Multimetric, SCC, PyCQA's CC, and Radon tools for Python input. As previously stated in Section 7.2, it is important to understand that the tools output their CC metric with different granularity and groupings. Lizard, PyCQA's CC, and Radon primarily calculate the CC metric independently for each function/method in a file, while Metrics, Multimetric, and SCC predominantly calculate one overall CC value per file. The overall value is the sum of all the function/method values combined. It was decided to sum the individual outputs of PyCQA's CC and Radon to make a comparison possible.

8.3.1 Results

Domain Generating Algorithms First, the results of the tools are compared for the three Python DGA implemenentations `DGARamdo.py`, `DGABanjori.py`, `DGAFobberV1.py` and their abstract class `dga_abstract_class.py`.

File & [Total # Lines *] (*) using `"wc -l"`	Overall Cyclomatic Complexity					
	Liz.	**Met.**	**MMe.**	**SCC**	**PyCGA***	**Radon***
`dga_abstract_class.py` [21]	3	0	1	0	3	3
`DGARamdo.py` [26]	3	3	4	0	6	3
`DGABanjori.py` [28]	5	4	4	0	6	5
`DGAFobberV1.py` [29]	4	3	3	0	7	4

Table 19: Cyclomatic Complexity Results for Python DGA Files

The results in Table 19 show that the Cyclomatic Complexity metric results vary between the studied tools for all four of the DGA related Python files.

File & [Total # Lines *] (*) using `"wc -l"`	Cyclomatic Complexity (CC)					
	Liz.	**Met.**	**MMe.**	**SCC**	**PyCGA***	**Radon***
`app.py` [1481]	146	98	71	14	137	144
`test_feedexport.py` [2966]	326	129	0	10	264	300
`figure.py` [3576]	438	304	268	58	350	388
`compiler.py` [7690]	1300	1043	738	127	938	1310
`generic.py` [13976]	1101	810	689	311	792	1089

Table 20: The table shows the different cyclomatic complexity results per library for the five analyzed files plus their standard deviation.

5 Biggest Files of Popular Python Libraries

8.3.2 Analysis of Results

Table 21 gives an overview of the differences between the cyclomatic complexity results calculated by the various studied tools for Python input code. It can be seen in Table 21 that Metric, Multimetrics and SCC solely calculate one overall cyclomatic complexity metric per file while Lizard, PyCGA and Radon calculate the CC

Case / Affected By	Cyclomatic Complexity Counting Differences					
	Lizard	Metric	MultiMetric	SCC	PyCGA	Radon
Starting value	0	0	2	0	1	1
Count code outside function		✓	✓	✓	✓	
Func/Method adds +1	✓				n.a.	n.a.
`else` (as part of `if`)		✓	✓	✓		
`and` / `or` in `if`/`while`	✓		✓	✓		✓
`if` and `else` in one line	+1	+2	+2	+2	(!)	+1
`assert()`		+1	-1			+1
`return`			-1			
`try:`				✓	✓	
`except:`	✓				✓	✓
`else:` (as part of try)		✓	✓	✓		✓
`finally:`	✓			✓		
`with open as file:`				✓		
Counts strings that equal a keyword (e.g. `x="else"`)			✓			

Table 21: Overview of Cyclomatic Complexity Metric Differences.

per function/method (and class for Radon). Lizard and Radon calculate the average cc measure per file although Lizard can optionally (`"-xml"` flag) provide the overall CC of a file(sum of all the individual func/methods cc results).

Lizard, Metric and SCC start couting at zero while Multimetric starts at two and PyCGA and Radon start at one (see 8.3.2). It should be noted that the Lizard and Radon tools both ignore source code that is not part of a function, method or class when calculating their CC metrics (see 8.3.2). Lizard, PyCGA and Radon all start counting at one for each method. Because of this Lizards overall sum is affected by the number of methods in the file. The same would be true for PyCGA and Radon if one would just sum their cc results per method to calculate an "overall" CC per file similar to the other tools. All six tools count `if`, `elif`, `for` and `while` statements, however only Metric, Multimetric and SCC count the `else` inside a `if-elif-else` statement(see 8.3.2). Lizard, Multimetric, SCC and Radon additionally raise their CC for each `and`/`or` inside the condition of a `if` or `while`(see 8.3.2). Lizard, SCC and PyCGA ignore `assert()` statements while Metric and Radon raise there CC and Multimetric subtracts one for each occurrence. Multimetric is the only studied tool that subtracts one for each `return`-statement(see 8.3.2). The tests in section 8.3.2 show that none of the six studied tools applied the same rules for `try-except-else-finally` code blocks. SCC and PyCGA's results are affected by `try` while the other tools results are not. Lizard, PyCGA and Radon raise their CC for each `except` while Metric, Multimetric and SCC do not. An `else` as part of a `try`-statement is counted by Metric, Multirmetric, SCC and Radon(eventhough a `else` inside `if` is not).Lizard and PyCGA ignored the `else` in both cases.

Starting Values Tests Test cases were written to help determine the starting values of the Cyclomatic Complexity (CC) metric for the different libraries. The baseline CC value is essential for further tests and to compare the output of the various libraries. Each test case was saved and can easily be rerun.

Starting Value Outside of Function This test file is only one line containing `print("t1")` without a function definition. Because LIZARD, PYCQA, and Radon returned the CC per method, this is an interesting edge case test. LIZARD, Metrics, and SCC returned a CC of zero. Multimetric returned two, while PYCQA and Radon returned an error.

Starting Value in Function This test contains a `print("t2")` statement inside of a function definition.

Line of Code	Cyclomatic Complexity (CC)					
	Lizard	Metric	Multimetric	SCC	PyCGA	Radon
1 `def baseline()`	0	0	2	0	err	err
2 `  print("ok")`	**1**	**0**	**2**	**0**	**1**	**1**

Table 22: Line by Line CC results of `Func_Baseline.py`

Table 23 shows that the results of LIZARD and Radon both increased by one while the results of the other three libraries where unchanged compared to the first test.

If-ElIf-Else, For, While This test tries to check the libraries basic counting abilities for `if-elif-else`, `while` and `for` statements inside of a function definition.

Line of Code	Cyclomatic Complexity (CC)					
	Liz.	Met.	MMe.	SCC	PyCGA	Radon
1 `def cc_3(self,x:int):`	0	0	2	0	err	err
2 `  r = "baseline"`	1	0	2	0	1	1
3 `  if x == 1:r = "one"`	2	1	3	1	2	2
4 `  elif x == 2:r = "two"`	3	2	4	2	3	3
5 `  else:r = "other"`	**3**	3	5	3	**3**	**3**
6 `  while(x < 3):x=x+1`	4	4	6	4	4	4
7 `  for i in range(3):`	5	5	7	5	sytx(+)	sytx(+)
8 `  print(i)`	5	5	7	5	5	5
vs `Func_Baseline.py`	+4	+5	+5	+5	+4	+4

Table 23: Line by Line CC results of `CC_MultiTest.py`

Table 23 shows that the results of LIZARD and Radon both increased by one while the results of the other three libraries where unchanged compared to the first test.

Func_3_If_outside.py Table 24 shows that LIZARD and RADON both ignored code that isn't part of a method. METRIC, MULTIMETRIC , SCC and PYCGA all counted the three `if`-statements outside of the function.

Assert and Return Errors

Line 5 in Table 8.3.2 shows that MULTIMETRIC decreases its CC metric with each

Cyclomatic Complexity (CC)

Line of Code	Lizard	Metrics	MMe.	SCC	PyCGA	Radon
1 `def threeIf()`	0	0	2	0	err	err
2 `  print("test")`	1	0	2	0	1	1
3 `u = 1`	1	0	2	0	1	1
4 `if u:`	**1**	1	3	1	sytx	sytx (=)
5 `  if u:`	**1**	2	4	2	sytx	sytx (=)
6 `  if u:print(u)`	**1**	3	5	3	4	**1**

Table 24: Line by Line CC results of `Func_3_If_outside.py`

`assert` statement, while RADON increases its result. Lines 8 and 9 show that only MULTIMETRIC reduces its CC result for each Python `return` statement.

Cyclomatic Complexity (CC)

Line of Code	Lizard	Metrics	MMe.	SCC	PyCGA	Radon
1 `def if_rtn(x:int):`	0	0	2	0	sytx	syrx
2 `  toTest = 4`	2	1	2	0	1	1
3 `  for i in range(3):i=i+2`	2	1	**3**	**1**	**2**	**2**
4 `  while x < 3: x=x+1`	**3**	**2**	**4**	**2**	**3**	**3**
5 `  assert(toTest == 4)`	3	**3**	**3**	2	3	**4**
6 `  if(x>3) or (x<0) and 1:`	**6**	4	**6**	**5**	synt	synt
7 `  x = 0`	6	4	6	5	4	**7**
8 `  return`	6	4	**5**	5	4	7
9 `  return`	6	4	**4**	5	4	7

Table 25: Line by Line CC results of `For_While_and_Or_Return.py`

Func_Try_Except.py This test aims to check how libraries count `try` and `except` statements within a method.

Cyclomatic Complexity (CC) Func_Try_Except.py

Line of Code	Lizard	Radon	MMe.	Metric	SCC	PyCGA
1 `def main():`	0	error	2	0	0	error
2 `  x = 1`	1	1	2	0	0	1
3 `  try:`	1	error	2	0	**1**	error
4 `  x = 2`	1	error	2	0	1	error
5 `  except:`	**2**	error	2	0	1	error
6 `  x = 3`	2	**2**	2	0	1	**3**
vs `Func_Baseline.py`	**+1**	**+1**	**=**	**=**	**+1**	**+2**

Table 26: Line by Line CC results of `Func_Try_Except.py`

The results in Table 26 show that LIZARDS, Radon and SCC all raised there CC metric results by one compared to the `Func_Baseline.py` result. Looking at the line by line analysis it seems that `except` is counted while the `try` statement is not. For the SCC library it is the opposite, the `try` is counted while the `except` is not. It is not clear which of the two statements was counted by the Radon library, but only one contributed to the over result. The results of Multimetric and Metric where both not affected.

Func_Try_Finally.py This test aims to check how libraries count `try` and `finally` statements within a method. In the results shown in Table 27 the LIZARD library re-

Lines of Code	Cyclomatic Complexity (CC) Func_Try_Finally.py					
	Lizard	Radon	Multimetric	Metric	SCC	PyCGA
1 `def main():`	0	error	2	0	0	error
2 `x = 1`	1	1	2	0	0	1
3 `try:`	1	error	2	0	**1**	error
4 `x = 2`	1	error	2	0	1	error
5 `finally:`	**2**	error	2	0	**2**	error
6 `x = 3`	2	1	2	0	2	**2**
vs `Func_Baseline.py`	**+1**	**=**	**=**	**=**	**+2**	**+1**

Table 27: Line by Line CC results of Func_Try_Finally.py

turned a cyclomatic complexity of two. This is one more than it did for `Func_Baseline.py`. It can be seen in the line by line analysis that `finally` was counted by LIZARD while `try` was not. Radon returned the same CC as it did for `Func\_Baseline.py`, but the result was lower by one than the result of the last test8.3.2. It can therefore be concluded that Radon counted `except` and not the `try`in the previous test in section 8.3.2. Neither Multimetric or Metric's CC results differed from their `Func_Baseline.py` results. The result of the SCC library was two higher compared to its baseline. It can be seen in the line by line analysis that both the`try` and `finally` where added to the overall cyclomatic complexity..

Func_Try_Except_Else_Finally.py This test aims to check how libraries count `try` and `except`, `else` and `finally` statements combined in one method. The

Line of Code	Cyclomatic Complexity (CC) Func_Try_Except_Else_Finally.py					
	Lizard	Radon	MMet.	Metric	SCC	PyCGA
1 `def main():`	0	error	2	0	0	error
2 `x = 1`	1	1	2	0	0	1
3 `try:`	1	error	2	0	1	**error**
4 `x = 2`	1	error	2	0	1	error
5 `except:`	**2**	**error**	2	0	1	**error**
6 `x = 3`	2	2	2	0	1	3
7 `else:`	2	error	**3**	**1**	2	error
8 `x = 4`	2	3	3	1	2	3
9 `finally:`	**3**	error	3	1	**3**	error
10 `x = 5`	3	3	3	1	3	3
vs `Func_Baseline.py`	**+2**	**+2**	**+1**	**+1**	**+3**	**+2**

Table 28: Line by Line CC results of `Func_Try_Except_Else_Finally.py`

overall results shown in Table 28 are broken down by library. The line by line analysis of the results show that LIZARD counted `except` and `finally`, adding two to its baseline value. Radon also added two to its baseline value but counted `except` and `else`. Multimetric only counted the `else`. The Metric library also only counted the `else` while ignoring the other statements. SCCs result was three higher than its baseline. This resulted from counting `try`, `else` and `finally` while ignoring the `except`.

8.4 Halstead Metrics

RADON and MULTIMETRIC are the sole two programs among the 13 studied metric tools that offer support for the Halstead metrics for Python input.

8.4.1 Results

This section details the Halstead metric results of the following three real world Domain Generating Algorithms implementations: `Ramdo.py` , `Banjori.py` and `FobberV1.py` .

Halstead metric results The `Analysis Tool` was used to calculate the Halstead metrics for the python implementation of three Python DGA files. The results of the Halstead metrics of both RADON and MULTIMETRIC can be seen in Table 29 below. The table shows the raw output values for the two tools as well as a factor which is defined as the ratio of the RADON value and the MULTIMETRIC value. The table shows that the MULTIMETRIC tool produces values which are between 1.1 and 11.7 times higher than the RADON tool. The reasons for these large differences is explained in section 8.4.2.

	Radon				Multimetric				Factor			
	η_1	η_2	N_1	N_2	η_1	η_2	N_1	N_2	η_1	η_2	N_1	N_2
`Ramdo`	6	22	17	34	17	24	81	61	2.83	1.09	4.76	1.79
`Banjori`	4	20	10	20	18	22	117	77	4.5	1.1	11.7	3.85
`FobberV1`	9	23	14	27	20	29	100	74	2.22	1.26	7.14	2.74

Table 29: Halstead Results for `DGARamdo.py`, `DGABanjori.py` and `DGAFobberV1.py`

8.4.2 Analysis of Results

This section investigates the substantial disparities observed in the Halstead metric outcomes, as detailed in Section 8.4, between the RADON and MULTIMETRIC metric assessment tools. Significant differences and bugs exist within both tools, leading to vastly different Halstead metrics that are meaningless to compare. Using line by line comparisons on sample test code the reasons for these differences could be easily determined. Simply, differences stem from different definitions of what is counted for both operators and operands. A summary of this differences can be seen in Table 30. The table clearly shows that MULTIMETRIC tool counts many more operands and operators than RADON leading to vastly different results. Selected examples of these differences for the DGABanjori.py code are described in the next paragraph.

Assignment Handling Comparison

The results in Figure 12 uncover multiple differences between the two tools. First, RADON completely ignores "simple" assignments that do not contain any operators. Second, RADON never counts the variable being assigned to the left of the equals (=) operator in assignments or the equals operator itself, while MULTIMETRIC counts both. Third, RADON unlike MULTIMETRIC, does not count function names

Description	Radon	Multimetric
General:		
ignores "simple" assignments (e.g. `i = 1`)	✓	
Operator counts consider:		
parentheses (`( )`) and brackets (`{ }`)		✓
function and class definitions		✓
function calls		
assignment operator (=)		✓
coma (,) in list definitions		✓
Operand counts affected by:		
constants as global		✓
local vars as global		✓
left side of assignments		✓
typed return value		✓
parameters in functions		✓
list definitions		✓
imports (e.g "DGA" class)		✓
string operands denoted with single quotes		✓
array reference as multiple operands		✓

Table 30: Halstead Operator and Operand Counting Differences

or parenthesis (`"( )"`). Lastly, the MULTIMETRIC tool contains a bug concerning operand identification, which is why it does not count `"(2 + 3)"` as a operand of the multiplication.

```
1 def simpleAssign():
2   x = 51.5
```

	RADON	MULTIMETRIC
η_1	0	5|simpleAssign ( ) : =
η_2	0	2| x 51.5
N_1	0	5| η_1
N_2	0	2| η_2

```
1 def assign():
2     x = 1 * (2 + 3)
```

RADON	MULTIMETRIC
2|* +	7| assign( ) : = * +
4| 1 2 3 (2 + 3)	4| x 1 2 3
2| η_1	9| η_1 + ( )
4| η_2	4| η_2

Figure 12: Halstead Python Assignment Comparison

Imports, Function Calls and Lists

This test showcases the tools differences when handling imports, function calls and lists. The left test cases shows that RADON does not count imports or function parameters, while MULTIMETRIC does. The test on the right side shows that RADON does not count brackets (`[ ]`).Furthermore one can seeMULTIMETRIC contains a bug concerning operand identification, which results in list accesses being counted incorrectly.

```
1 from math import pow , sqrt
2 def imprt():
3     x = pow(1,2) + 3
```

	Radon	Multimetric
η_1	1|+	7|, imprt ( ) : = +
η_2	2| 3 pow(1,2)	6| pow sqrt x 1 2 3
N_1	1| η_1	10| η_1 + ( ) ,
N_2	2| η_2	7| η_2 + pow

```
1 def lst():
2     llist = [1,2,3]
3     llist = llist[1] + llist[2]
```

Radon	Multimetric
1|+	9| lst ( ) : = [ ] , +
2| llist[1] llist[2]	4| llist 1 2 3
1|η_1	15|η_1 + 2x[2x] , =
2|η_2	9|η_2 + 3xllist 1 2

Figure 13: Halstead Python Imports Comparison

DGABanjori.py Line by Line The biggest differences between the results of the three python DGA files where observed for `DGABanjori.py` created by Johannes Bader (see Table 31) [71].

It should be noted that all comments and blanks where removed to conserve space after assuring they did not affect the Halstead results. A value of "-" means that the metrics could not be calculated. This is often an error due to missing the next line(s) of code, e.g. function definition without body is not valid python code.

Line of Code	Radon η_1	Radon η_2	Radon N_1	Radon N_2	Multimetric η_1	Multimetric η_2	Multimetric N_1	Multimetric N_2
`1 from pr2.dga_classes.dga_abstract_class import DGA`	0	0	0	0	0	1	0	1
`2 class DGABanjori(DGA):`	-	-	-	-	4	1	4	2
`3 def __init__(self,seed:str="earnestnessbiophysicalohax.com"):`	-	-	-	-	7	6	11	8
`4  super().__init__()`	0	0	0	0	8	7	17	9
`5  self.seed = seed`	0	0	0	0	8	7	19	12
`6  self.lastdomain = seed`	0	0	0	0	8	8	21	15
`7 def map_to_lowercase_letter(self, s: int ) -> int:`	-	-	-	-	11	10	29	19
`8  return ord('a') + ((s - ord('a')) % 26)`	3	6	3	6	13	12	40	23
`9 def generate_domain(self) -> str:`	-	-	-	-	14	12	46	25
`10  dl = [ord(x) for x in list(self.lastdomain)]`	3	6	3	6	17	15	55	32
`11  dl[0] = self.map_to_lowercase_letter(dl[0] + dl[3])`	3	8	4	8	17	18	66	40
`12  dl[1] = self.map_to_lowercase_letter(dl[0] + 2 * dl[1])`	4	12	6	12	18	20	78	49
`13  dl[2] = self.map_to_lowercase_letter(dl[0] + dl[2] - 1)`	4	16	8	16	18	20	90	58
`14  dl[3] = self.map_to_lowercase_letter(dl[1] + dl[2] + dl[3])`	4	20	10	20	18	20	104	68
`15  self.lastdomain = ''.join([chr(x) for x in dl])`	4	20	10	20	18	22	114	75
`16  return(self.lastdomain)`	4	20	10	20	18	22	117	77

Table 31: Line by Line Halstead Results of `DGABanjori.py`

For example, line 1 of the table shows that the import directive DGA is counted as operand by Multimetric but not by Radon. Line 2 shows that the class name DJABanjori plus the characters (,) and : are all consideblue operators by Multimetric while the entire line is ignoblue by Radon. Similar differences can be seen for subsequent lines of code. The sum of these difference when analyzing an entire file lead to the significant overall differences when comparing the output of these two tools.

8.5 Maintainability Index

LIZARD and MULTIMETRIC are the only two studied metric tools that claim to provide functionality to calculate the Maintainability Index. There are several reasons why the results calculated by these two tools are not expected to be directly comparable. First, as previously stated in Section 7.4 the tools do not share a common formula to compute the MI. Second, the input to the formulas rely on the tools Halstead, Cyclomatic Complexity and Lines of Code related metric results. Section 8.4 and Section 8.3 clearly show that these tools do not calculate similar values for these metrics in most cases. Most significantly, the MULTIMETRICS LOC error which is described in Section 8.1 leads to a catastrophic calculation error for the MI metric in all cases.

8.5.1 Results

Table 32 shows the MI results of the two tools for the three Python DGA files.

	Radon	Multimetric		
	MI	Classic	SEI	Microsoft
`DGARamdo.py`	51.70%	100%	100%	100%
`DGABanjori.py`	75.29%	100%	100%	100%
`DGAFobberV1.py`	70.91%	100%	100%	100%

Table 32: Maintainability Results for Python DGAs

8.5.2 Analysis of Results

The MULTIMETRIC program returns 100 if the calculated MI result is greater than 100. The results in Table 32 show that MULTIMETRIC returned 100 for all 3 files and independent of MI settings, which we suspected indicated an error. Examining the input values used by MULTIMETRIC it was clear that the incorrect LOC value of 1 (see 8.1) causes an incorrect calculation. Furthermore, the other input values such as Comment Ratio, CC and Halstead Volume also significantly differ between the two tools. For this reason the Maintainability Index results for the two tools are incomparable.

8.6 COCOMO

A basic test was run to verify that the COCOMO results are directly dependent on the SLOC value. Therefore, when using COCOMO the limitations detailed in Section 6.1.1 can be expected to similarly impact the COCOMO results.

9 Transcribed C Results

In this section, we delve into the second research question, focusing on how input languages impact the results of metric tools. Specifically, we utilize the Analysis Tool to compare metrics and calculation methods for Python and its equivalent C code generated by the Transcriber. Additionally, we highlight essential considerations and problems when comparing metric results across the two languages and offer insights into why some metrics generally tend to be more comparable across different languages.

There are multiple ways to consider which source code files are included in the metric calculation process. C, for example, is a low-level language and does not include concepts like `list`s or `class`es and inheritance, which are supported by Python out of the box. In order to transcribe Python code, C implementations have to be provided for this Python functionality. It could be argued that these implementations should be included when computing metrics, as this code is required to execute the code. Additionally, it could be argued that this additional code demonstrates the advantages of using a higher-level language like Python over a lower-level language like C. However, it is unlikely that such code would be written more than once in a development environment and would, in fact, be plausibly considered as a standard library.

The C implementations of Python functionality (referred to as Helper Code), located in the `includes` folder and imported by the transcribed code, are significantly more complex than the DGA files themselves. Because the test cases used in this thesis are relatively short code snippets, these helper implementations have a great impact on the overall metric results, while their impact would decrease as the size of the overall code grows. Due to this, it was decided not to include these implementations in the analysis of the results.

9.1 Lines of Code (LOC)

This section compares the tools LOC metric results for Python and C input and highlights multiple factors that should be considered when comparing the LOC results for Python and C code.

9.1.1 Results

This section compares the LOC results of the three DGA files `DGARamdo.py`, `DGABanjori.py` and `DGAFobberV1.py`. In Section 8.1, which discusses the tools' treatment of Python input, it was observed that Multimetric mistakenly reported the number of input files, whereas the other four tools (OhCount, Radon, SCC, and Tokei) provided the correct lines of code (LOC) results for all the analyzed code snippets. A similar pattern can be observed for C input. Multimetric mistakenly reported the number of input files, whereas OhCount, SCC and Tokei provided the correct lines of code (LOC) results for all the analyzed code snippets. Radon only supports Python language input and therefore Multimetric and Radon are not included in these results. The transcription process of the DGA files is detailed in Section 5.2.5. To summarize, the three Python DGA files inherit and implement methods from their common abstract class `dga_abstract_class.py`. The transcribed C results `DGARamdo/export.c`, `DGABanjori/export.c` and `DGAFobberV1/export.c` contain a struct that represents the Python `DGA` class. These C files include `llist.c`

and `OrdChr.c` from the `includes` folder that contain implementations of Python functionality that C doesn't provide out of the box.

Table 33 provides a summary of the LOC results. As all three remaining metric tools yield the same values, OhCount, SCC and Tokei results are shown in a single column.

DGA	**Python**	**C**	
	inc. abstract class	w/o Helper Code	with Helper Code
`DGARamdo`	49	55	253
`DGABanjori`	50	56	254
`DGAFobberV1`	55	60	258

Table 33: LOC Results for Python and C DGA Files per OhCount, SCC and Tokei

9.1.2 Analysis of Results

The results in Table 33 confirm that the implementations of Python functionality in C, overshadow the results of the DGA classes themselves. The examination of test cases and the detailed analysis of the Lines of Code (LOC) metric results emphasize the significance of acknowledging syntax variations among languages and their implications on tool outcomes. Although tools are not required to account for input language when calculating the LOC metric, several factors merit consideration when comparing these results between Python and C language inputs.
Major observed disparities are:

- The C language utilizes braces (`{}`) rather than indentation to delineate code blocks, encapsulate function definitions and control statements. OhCount, SCC and Tokei all raise the LOC metric for lines solely containing curly braces (`{}`), thus the placement of braces greatly affects the overall C input result (see Test 34).
- Because C was designed as a very lean language, much standard functionality in Python has to be explicitly included in C, resulting in additional lines of code.

These disparities are demonstrated in the following test cases.

Handling of Curly Braces This test case investigates the impact of braces (`{}`) on the LOC metric of the studied tools within the C programming language. The findings presented in Table 36 demonstrate that all analyzed tools incorporate braces into their LOC metric. Consequently, varying coding styles can significantly influence the final LOC measurements. Additional test cases that are not shown here also demonstrated that both braces for `if`, `while`, `for`, `switch` etc. statements are counted in the LOC metric of OhCount, SCC and Tokei.

9.2 Source Lines Of Code (SLOC)

This section compares how the metric tools compute the Source Lines of Code (SLOC) metric for Python and C input files. SlocCount, CLOC, OhCount, Metrics, Tokei, GoCLOC, SCC support calculating the SLOC metric for both Python and C input. Additionally CCCC, which only supports C out of the two languages, was added for comparison because it is often used in scientific papers [11].

```c
int main(){
  long int z = 1;};
```

OhCount, SCC, Tokei: 2

```c
int main(){
  long int z = 1;
};
```

OhCount, SCC, Tokei: 3

```c
int main()
{
  long int z = 1;
};
```

OhCount, SCC, Tokei: 4

Table 34: Effects of Curly Braces Coding Style on LOC Results

9.2.1 Results

This section compares the tools SLOC metric results for Python and transcribed C code.

Comparing DGA Implementations Table 35 lists the SLOC results of the tools for both the Python and transcribed C code.

DGA	**Python** inc. abstract class			**C** w/o Helper Code	
	Sloc., CLOC, OhC., Met., GoC.	Tokei default|ini	SCC	Sloc., CLOC, OhC., Met., GoC., Tokei, SCC	CCCC
`DGARamdo`	35	41|41	43	49	43
`DGABanjori`	33	42|37	40	49	44
`DGAFobberV1`	36	46|43	49	54	49

Table 35: LOC Results for Python and C DGA Files per OhCount, SCC and Tokei

9.2.2 Analysis of Results

The difference between the SLOC and LOC metrics is that SLOC does not include blank lines and comment lines. For this reason, the SLOC result is significantly affected by the tools handling of comments syntax. For example in C, single-line comments are indicated by double slashes (`//`), while Python employs hashtags (`#`). Similarly, Python provides four (`"`,`"""`,`'`,`'''`) different ways to denote multi-line comments, C only provides a single syntax (`/* */`). The test cases and detailed analysis of the results highlights the importance of considering syntax differences between the languages and their effects on tool results.

Major observed differences are:

- As with the LOC metric, the SLOC metric is also affected by C language braces (`{}`) as described in Section 9.1.2. Thus coding (and transcription) style can greatly affect the SLOC results.
- With the exception of the situation outlined in Section 8.2.2, where Metrics incorporates hashtag comments between parameters, none of the tools accounted for single-line hashtag comments in Python code. This edge case does not occur in Metrics for C input, thus all tools always ignore `//`-comments.

- Some of the biggest discrepancies in the Python results in Section 8.2.2 where due to differces in the handling of Python multi-line comments, which where counted incorrectly by SLOCCOUNT, GOCLOC and TOKEI in some instances. In contrast, all of the studied tools handled C multi-line comments equally and no differences between the tools where uncovered.
- The CCCC metric tool ignores C language includes while calculating the SLOC, while all the other tools count the include statements.
- Section 8.2.2, describes two major errors that can occur in the SCC tool when calculating the SLOC metric for Python source code, which do not occur for C language input.

9.2.3 Test Cases

Multiple test cases where written to determine the exact differences between the tools results.

Handling of Curly Braces This test case investigates the impact of curly brackets ({}) on the SLOC metric of the studied tools within the C programming language. The findings presented in Table 36 demonstrate that all 7 analyzed tools incorporate curly braces into their SLOC metric. Consequently, varying coding styles can significantly influence the final SLOC measurements. Additional test cases that are not individually shown here also demonstrated that both curly braces for `if while for` ect. statements are counted in the SLOC metric of the 7 tools. Notably the CCCC metric tool subtracts one from the "expected" SLOC result in all experiments undertaken as part of this thesis.

```c
int main(){
  long int z = 1;};
```

CCCC: 1 Other Tools: 2

```c
int main(){
  long int z = 1;
};
```

CCCC: 2 Other Tools: 3

```c
int main()
{
  long int z = 1;
};
```

CCCC: 3 Other Tools: 4

Table 36: Effects of Curly Braces Coding Style on SLOC Results

Includes and Blanks Table 37 shows that both system header file inclusions (using <>) and user-defined or project-specific header file inclusions (using "") increased the SLOC result of all studied tools except CCCC. Further, one can observe that all 7 studied tools (including CCCC) ignore blank lines when calculating the SLOC measure.

```c
#include <stdio.h>
#include "includes/OrdChr.h"
```

CCCC: 0 Other ToolS: 2

```c
int main(){

};
```

CCCC: 1 Other ToolS: 2

Table 37: Handling of Include Statements and Blank Lines

Single Line Comments This tests cases compares how the tools handle single line comments when calculating the SLOC metric for Python and C language input.

```python
def ComTest():
    #SL comment
    "SL DQ"
    'SL SQ'
    """SL TDQ """
    '''SL TSQ '''

    pass
```

```c
int ComTest(){
    // SL comment
    // SL DQ
    // SL SQ
    /* SL TDQ */
    /* SL TSQ */

};
```

	Source Lines of Code Comparison							
	SloC.	CLOC	OhC.	Met.	Tokei default | ini	GoC.	SCC	CCCC
`CommentTest.py`	4	4	4	4	6|6	5	4	-
`CommentTest.c`	2	2	2	2	2|-	2	2	1

Table 38: Single Line Comments Test

Multi Line Comments While differences in the treatment of multi-line comments could be observed in Python code (see 8.2.2 and 8.2.2) this was not the case for C input files. None of the tools SLOC result was affected by either code, quotes or blank lines inside a multi-line comment denoted with `/* */`.

C Language Definitions The SLOC metric of all tools except CCCC is affected by C language `#define` statement lines. There is no equivalent to these statements in the Python language so no comparison between languages is possible.

Multi Line Inputs

```c
#include <stdio.h>
void main() {
    printf
    (
        "%s",
        " multi"
        " line"
        " string"
    );
}
```

9.3 Cyclomatic Complexity Metric

This section details differences in how the tools calculate their CC metric for Python and C code. The primary focus is to uncover language specific CC counting differences for comparable code in the two languages.

In theory, the CC metric should yield similar results for Python and C code, because one would not generally expect the number of decision points to vary because of language syntax.

9.3.1 Results

This section compares the Cyclomatic Complexity results of the LIZARD, RADON, MULTIMETRIC , METRIC and SCC libraries for C and Python code.

DGA Algorithms This experiment compares the CC metric results for both the Python and C implementation of the `Ramdo, Banjori` and `FobberV1` DGAs.

	Total Cyclomatic Complexity (CC)								
	Python				**C**				
File	Liz.	Met.	MMe.	SCC	Liz.	Met.	MMe.	SCC	CCCC
`DGARamdo`	6	3	5	0	6	2	1	2	5
`DGABanjori`	8	4	5	0	7	2	0	2	6
`DGAFobberV1`	7	3	4	0	7	2	0	2	6

Table 39: CC Results of the `Ramdo, Banjori` and `FobberV1` DGAs

The results presented in Table 39 show that only LIZARD produced comparable results for the three Python and C DGA implementations, while the other tools results where drastically different in most cases. The next section takes a closer look at these results, to expose the underlying causes of these variances.

9.3.2 Analysis of Results

Table 40 lists various scenarios and corner cases that lead to different CC results. Please note that any disparities noticed between the tools results for the Python and C implementations of a given test case are denoted with (!).

Depending on which tool is used and which of these scenarios are present in the input source code, the final CC metric results greatly differ by tool. For this reason, it is misleading to compare results without factoring in the effects of these corner cases. These cases are presented in specific test cases designed to showcase them in Subsection 9.3.3.

9.3.3 Corner and Test Cases

The *Transcriber* was used to transcribe the Python test cases from section 5.2 to C code. The *Analysis Tool* detailed in section 5.1 was then used to compare the metric tools results for the Python and C implementations of these test cases.

Starting Count Outside Function The goal of this test is to determine the Cyclomatic Complexity starting value of the different tools for code outside of a function. The source code of both implementations as well as the results can be seen in table 41:

Python code:

```python
print("cc1")
```

C code:

```c
#include <stdio.h>
printf("%s","cc1");
```

Case / Affected By	C Code				
	Lizard	Metric	MultiMetric	SCC	CCCC (.c)
Starting Count Outside Function	0	0	2	0	0
Starting Count Inside Function	1	0	2	0	0
Counts Code Outside of Functions	✓(!)	✓	✓	✓	
Func/Method adds +1	✓				
`if`-statement	✓	✓	✓	+2(!)	✓
`elif` (`else if`)	✓	+2(!)	+2(!)	+3(!)	✓
`else` (as part of `if`)		✓	✓	✓	
`while`	✓	✓	✓	(!)	✓
`and` / `or` in `if`/`while`	✓		(!)	+2(!)	✓
`?` operator (`if` and `else`)	✓			✓	✓
`assert()`		(!)	-1	✓(!)	
`return`			-1		
with open as file				✓	
strings (e.g. `s=="if"`)			✓		
`do-while`	✓	✓	✓		✓

Table 40: Overview of Cyclomatic Complexity Differences

Version	Cyclomatic Complexity (CC)				
	Lizard	Metric	MultiMetric	SCC	CCCC
Python	0	0	2	0	-
C	0	0	2	0	0

Table 41: Results of `No_Func_Baseline.py` and `No_Func_Baseline.c`

The results in table 41 show that Lizard, Metric and SCC returned a CC result of zero for both the Python and C implementation. CCCC also returned zero for the C implementation and lacks Python support. The Multimetrics tool returned a CC result of two for both implementations.

Starting Count Inside Function This test case is a simple variable definition within a function definition. It was transcribed from `Func\_Baseline.py`. Both implementations, aswell as the compared results can be seen below :

Python code:

```
1 def baseline():
2   print("cc_test2")
```

C code:

```
1 #include <stdio.h>
2 void baseline(){
3   printf("%s","cc_test2");};
```

Table 42 lists the calculated cyclomatic complexities returned by the different li-

Version	Cyclomatic Complexity (CC) Lizard	Metric	MultiMetric	SCC	CCCC
Python	1	0	2	0	-
C	1	0	2	0	0

Table 42: Results of `Func_Baseline.py` and `Func_Baseline.c`

braries for the source code above. The same results where returned for both the Python and C code.

Code Outside Functions This test case aims to check if code outside of functions affect the tools CC metric. This test was transcribed from `No_Func_3_If.py` . Both implementations, aswell as the compared results can be seen below :

Python code:

```python
u = 1
if u == 1:
    if u == 2:
      if u == 3:
        print(u)
```

C code:

```c
#include <stdio.h>
long int u = 1;
if (u == 1){
  if (u == 2){
    if (u == 3){
      printf("%i",u);
    }}}
```

Version	Cyclomatic Complexity (CC) Lizard	Metric	MultiMetric	SCC	CCCC (.c)
Python	0	3	5	3	-
C	3	3	5	6	0

Table 43: Results of `No_Func_3_If.py` and `No_Func_3_If.c`

Table 43 lists the calculated cyclomatic complexities returned by the different libraries for the source code above. There are multiple takeaways from this test. Firstly, that Lizard counts code outside of functions for C input but ignores it for Python files. SCC raises its CC by two for each `if` encountered in C code, while only raising it by one when analyzing Python code.

If Statements This tests goal is to see how the different tools count a simple `if`-statement inside a function.

Python code:

```python
def main():
    x = 1
    if x == 1:
        x = 2
```

C code:

```c
void main(){
long int x = 1;
if (x == 1){
      x = 2;
}};
```

The results in table 44 show that SCC returned two for the C-code while one was returned for the Python implementation. LIZARD, METRIC, MULTIMETRIC return the same result for both languages. CCCC returned one for the C-code and does not support Python.

Version	Cyclomatic Complexity (CC) Lizard	Metric	MultiMetric	SCC	CCCC
Python	2	1	3	1	-
C	2	1	3	2(!)	1

Table 44: Results of `Func_If.py` and `Func_If.c`

Else If Statements This tests goal is to see how the different tools count a simple `if-elif`-statement inside a function.

Python code:

```python
def ifelif(x: int):
    r = 0
    if x == 1:
        r = 1
    elif x == 2:
        r = 2
```

C code:

```c
void ifelif( long int x){
    long int x = 0;
    if (x == 1){
        i = 1;}
    else if (x == 2){
        i = 2;}
    }
```

Version	Cyclomatic Complexity (CC) Lizard	Metric	MultiMetric	SCC	CCCC
Python	3	2	4	2	-
C	3	3(!)	5(!)	5(!)	2

Table 45: Results of `Func_If_Elif.py` and `Func_If_Elif.c`

The results in table 44 show that SCC returned two for the C-code while one was returned for the Python implementation. The other tools return the same result for both languages.

Assert Statements This test cases focuses on how `assert` statements affect the CC metric. The code was transcribed from `Func_Assert.py`. Both implementations, aswell as the compared results can be seen below :

Python code:

```python
def main():
    x = 1
    assert(x==1)
    x = 2
```

C code:

```c
#include <assert.h>
void main(){
    long int x = 1;
    assert(x==1);
    x = 2;
};
```

Table 41 below lists the calculated cyclomatic complexities returned by the different libraries for the source code above.

Table 46 demonstrates that the Cyclomatic Complexity (CC) metric of the METRICS tool increases when encountering an `assert` statement in Python code, while METRIC's CC remains unaffected by `assert` statements in the C language.

Version	Cyclomatic Complexity (CC) Lizard	Metric	MultiMetric	SCC	CCCC (.c)
Python	1	1	1	0	-
C	1	0(!)	1	0	0

Table 46: Results of `Func_assert.py` and `Func_assert.c`

For Statements The sixth test case focuses on the for-statement. The goal is to see weather the for-statement is counted like an if-statement by the different libraries.

Python code:

```python
def main():
    x = 1
    for i in range(3):
        x = i
```

C code:

```c
void main(){
long int x = 1;
for (long int i = 0; i < 3; i++){
    x = i;
}
};
```

Table 47 below lists the calculated cyclomatic complexities returned by the different libraries for the source code above. No language related differences could be observed.

Version	Cyclomatic Complexity (CC) Lizard	Metric	MultiMetric	SCC	CCCC (.c)
Python	2	1	3	1	-
C	2	1	3	1	1

Table 47: Results of `Func_For.py` and `Func_For.c`

While Statements The sixth test case focuses on the while-statement. The goal is to see weather the while-statement is counted by the different libraries.

Python code:

```python
def main():
    x = 1
    while(x < 3):
        x = x + 1
```

C code:

```c
void main(){
long int x = 1;
while(x < 3){
    x = x + 1;
}
};
```

Table 48 below lists the calculated cyclomatic complexities returned by the different libraries for the source code above. All tools returned the same results for both languages.

If Statements with AND/OR Operators This tests goal is to see how the different tools count `if`-statements with additional checks using `or`/ `and`.

	Cyclomatic Complexity (CC)				
Version	Lizard	Metric	MultiMetric	SCC	CCCC (.c)
Python	2	1	3	1	-
C	2	1	3	0	1

Table 48: Results of `Func_While.py` and `Func_While.c`

Python code:

```python
def main():
    x = 1
    y = 2
    if x == 1 and y == 2:
        x = 3
```

C code:

```c
void main(){
long int x = 1;
long int y = 2;
if (x == 1 && y == 2){
     x = 3;
}
};
```

	Cyclomatic Complexity (CC)				
Version	Lizard	Metric	MultiMetric	SCC	CCCC
Python	3	1	4	2	-
C	3	1	3(!)	4(!)	2

Table 49: Results of `Func_If_and.py` and `Func_If_and.c`

The results in table 49 show that Lizard and Radon both count the additional check in the If-statement as a new path and thus return 3. Multimetric ignores the additional check in the C code but counts it in the python code. Metrics ignores the additional argument in the If-statement regardless of programming language.

While Statements with AND/OR Operators The sixth test case focuses on the and operator within a while-statement . The goal is to see weather the while-statement is counted by the different libraries.

Python code:

```python
def main():
    x = 1
    z = 5
    while(x < 3 and z == 5):
        x = x + 1
```

C code:

```c
void main(){
long int x = 1;
long int z = 5;
while(x < 3 && z == 5){
    x = x + 1;
}
};

```

Table 50 below lists the calculated cyclomatic complexities returned by the different libraries for the source code above.

Do While Statements (C Only) This test compares the tools results for a C language `do while` statement.

	Cyclomatic Complexity (CC)				
Version	Lizard	Metric	MultiMetric	SCC	CCCC (.c)
Python	3	1	4	2	-
C	3	1	3	2	2

Table 50: Results of `Func_While_And.py` and `Func_While_And.c`

		Cyclomatic Complexity (CC)				
	Func_Baseline_doWhile.c	Lizard	Metric	Multimetric	SCC	CCCC
1	`int main(){`	0	0	2	0	0
2	`long int x = 1;`	0	0	2	0	0
3	`do {x = 3;} while(0);`	0	1	3	0	1
4	`};`	2	1	3	0	1
	vs Func_Baseline.c	+1	+1	+1		+1

Table 51: Line by Line CC results of `Func_Baseline_doWhile.py`

Return Statements This test file contains a Method with a Return statement. Table 52 below shows that the results of the CC metrics varied by library. Lizard, Metric and SCC returned a CC of zero while Multimetric returned two and Radon returned an error.

		Cyclomatic Complexity (CC)				
	Func_Return.c	Lizard	Metric	Multimetric	SCC	CCCC
1	`int main(){`	0	0	2	0	0
2	`long int x = 1;`	0	0	2	0	0
3	`return x;`	0	0	1	0	0
4	`};`	1	0	1	0	0
	Final Result	**1**	**0**	**1**	**0**	0
	Final Python Result	**1**	**0**	**2**	**0**	

Table 52: Line by Line of `Func_Return.c` and comparison to Python results

9.4 Halstead

MULTIMETRIC is the only studied tool that supports calculating Halstead metrics for both Python and C input. The results in Section 8.4 show that MULTIMETRIC's Python Halstead results are deeply flawed.

9.4.1 Results

Table 53 show's MULTIMETRICS Halstead results for the Python and C implementations of the DGA files. For this test the Python results where calculated by passing the `dga_abstract_class.py` along with the main DGA implementation, to ensure the unique counts are considered for the project, rather than individual files.

DGA	Python				C			
	η_1	η_2	N_1	N_2	η_1	η_2	N_1	N_2
Ramdo	23	38	141	102	23	38	223	115
Banjori	22	35	177	118	23	35	304	151
FobberV1	26	44	160	115	24	46	256	139

Table 53: Multimetric Halstead for DGAs Python vs C

9.4.2 Analysis of Results

The results show that the unique counts of operators (η_1) and unique count of operands (η_2) are fairly similar for both C and Python implementations. Significant differences can be observed when comparing the total number of operators (N_1) and operands (N_2). Upon analysis we discovered that these differences are primarily caused by language syntax factors, such as the use of braces and semicolons in the C language. Additionally, the current *Transcriber* implementation is somewhat inefficient in its use of parenthesis, which further increases the C results.

9.4.3 Test Cases

C Language Syntax Influence on Results

The results in Figure 14 show that MULTIMETRIC ignores C language variable types but counts semicolons(`";"`), which results in a higher total number of parameters (N_1) compared to Python code.

```
1 long int i = 1;
2 long int u = 1;
```

	Result	Counted
η_1	2	`"="`, `";"`
η_2	3	`"i"`, `"1"`, `"u"`
N_1	4	η_1 + `"="`,`";"`
N_2	4	η_2 + `"u"`,`"1"`

```
1 double i = 1.1;
2 i = 1.1; ;;;;;
```

	Result	Counted
η_1	2	`"="`, `";"`
η_2	2	`"i"`, `"1.1"`
N_1	9	η_1 + `"="`, 6x`";"`
N_2	4	η_2 + `"i"`, `"1"`

Figure 14: C Language Syntax Influece Test

Additional Parenthesis Table shows that MULTIMETRIC calculates similar results for the number of unique operators (η_1), the number of unique operands (η_2) and the total number of operands (N_2) for both languages in this test. It can further be seen that the current version of the *Transcriber* implementation adds some unnecessary parenthesis (`"( )"`) that effect the number of total operators (N_1).

9.5 Maintainability Index

MULTIMETRIC is the sole studied tool that supports the MI for C input. Unfortunately, due to the incorrect LOC calculation as shown in 8.5 the MULTIMETRIC calculates a MI of 100 for all C inputs tested.

Python code:

```python
def calc() -> float:
  r = 1 + 5 * (3-10) / 2
  return r
```

	Result	Counted
η_1	10	`"calc","(",")","-",">",":", "=", "+", "*","/"`
η_2	7	`float,"r","1","5","3","10","2"`
N_1	13	η_1 + `"(",")","-"`
N_2	8	η_2 +`"r"`

C code:

```c
double calc(){
  double r =(1+((5*(3-10))/2));
  return r;}
```

	Result	Counted
η_1	11	`"calc","(",")","{","=","+", "*", "-","/", ";","}"`
η_2	6	`"r","1","5","3","10","2"`
N_1	20	η_1 + 4x`"(`, 4x`")","};"`
N_2	7	η_2 +`"r"`

Figure 15: Comparison of Python and C code

LIZARD and MULTIMETRIC are the only two studied metric tools that claim to provide functionality to calculate the Maintainability Index. There are several reasons why the results calculated by these two tools are not expected to be directly comparable. First, as previously stated in Section 7.4 the tools do not share a common formula to compute the MI. Second, the input to the formulas rely on the tools Halstead, Cyclomatic Complexity and Lines of Code related metric results. Section 8.4 and Section 8.3 clearly show that these tools do not calculate similar values for these metrics in most cases. Most significantly, the MULTIMETRICS LOC error which is described in Section 8.1 leads to a catastrophic calculation error for the MI metric in all cases.

10 Conclusion

This thesis concerns itself with the topic of software code metrics. Software code metrics quantify different aspects of code, including size, quality, complexity, maintainability, and development costs and are widely used in modern software development. First, we present an overview of the topic of software metrics. Second, we describe currently available open-source metric tools. Finally, we investigate two research questions regarding software code metrics.

The primary question is whether widely available open-source code metric tools calculate identical results when analyzing Python source code files. The second question asks to what extent implementation language impacts selected metric results.

To answer the first question, we first researched popular source code metrics via a literature search. Next, we selected 13 open-source metric tools, which claim to support some or all of the metrics for Python and/or C. For some metrics, such as LOC and SLOC, there is an abundance of available tools, but for others, such as the Halstead metrics, few options exist.

In order to efficiently run tests and gather results, an *Analysis Tool* was developed, which runs multiple tools on selected input files in various output modes, and outputs the results in an easy-to-analyze Excel file. Python source code test files were obtained from the Security and Privacy Research Group. Tests were run, and the results were analyzed. The analysis led to the development and execution of additional specific test scenarios in order to isolate the causes of anomalies uncovered in the initial testing.

To answer the second research question, a *Transcriber* was developed to automatically transcribe the class of Python code used in the first part of this thesis to C source code. This *Transcriber* was then used to automatically create C code from the Python source code. Next, the *Analysis Tool* was used to calculate and gather selected metrics for C code from selected metric tools. Following result analysis, specific tests were created and executed in order to determine the causes for observed differences in the results.

With respect to the first question, it can be said that the comparability of the tools' results varies significantly depending on the specific metric in question. For example, for the relatively simple LOC and SLOC metrics, the tools calculated the most similar values, although even with these metrics, some tools had significant errors. For instance, with one tool, it was necessary to set an obscure flag to obtain correct results when analyzing Python code. For the other studied metrics, the results were so different as to render them incomparable.

This thesis describes the main reasons for the observed differences, which are varying metric definitions, software bugs, and mishandling of syntax variants in the source code. Overall, we conclude that one cannot expect different metric tools to calculate the same result for identical input. Furthermore, even when using the same tool, significantly different results might be obtained for identical algorithms coded using slightly different syntax variants.

With respect to the second question, the results demonstrated that for a given algorithm, the studied tools calculated significantly different values depending on the input language. Detailed analysis of the results provided valuable insight into the reasons for these differences. First, some metrics such as LOC and SLOC are directly impacted by the syntax of the language. For example, C uses braces to define code blocks, while Python uses indentation. For this reason, coding (or transcription) style can significantly influence these metrics, as demonstrated by Jerry

Fitzpatrick (see 6.1.6). Second, for metrics that would be expected to be comparable, such as CC, some tools applied different metric calculation rules for the same code constructs in Python and C. For example, one tool raised its CC metric by one for every `if` statement encountered in Python code, while raising it by two in C. Finally, when comparing a high-level language like Python to a low-level language like C, one must decide whether code supporting higher-level capabilities should be counted in the results for the lower-level language. Overall, we conclude that the studied open-source tools should not be used to compare metric values between Python and C in their current state.

Source code metrics are a useful and important tool for software measurement and can be used to improve the quality of source code. Furthermore, metrics are applied in numerous different areas such as testing, quality assurance, as well as time and cost prediction. Because source code metrics can be automated into the development process, they cause minimal overhead and can easily be added to any development process. Additionally, the cost of developing software can be reduced by using software code metrics by producing better quality code that is easier to maintain. To conclude, software code metrics are an important tool and aid in producing better quality software, and their use should be increased. There is some valid criticism regarding code metrics, but one can still profit from using them while keeping their shortcomings in mind.

This thesis shows significant variance in the results calculated by the various studied tools when analyzing Python and C code. The thesis identifies several areas for future work which could improve the situation.

Future work This thesis lays the foundation for future work and provides a basis of knowledge about various source code metrics and their usage. First, there should be standardization for how each of the metrics should be calculated for different input languages. Second, software errors identified in this thesis should be corrected in the open-source software. Third, until the first two tasks are completed, when communicating software metric results, it is important to state which tool (and in some cases which options) were used to obtain the results. Finally, additional work could be done on the *Transcriber* to support the transcription of more complex Python code. Additionally, the *Transcriber* could be enhanced to generate C code optimized for specific metrics.

11 Appendix

Licensing Differences Originally gaining traction at the Massachusetts Institute of Technology (MIT) in the early 1980s, the MIT license is characterized by its permissiveness. The MIT license grants users the freedom to utilize, copy, modify, and distribute software licensed under MIT without requiring royalties or the publication of source code, provided that any modifications or derivative works are also released under the MIT license [72].

Published by Richard Stallman in 1989, the GPL license is more restrictive compared to the MIT license and was developed with the goal of supporting the development of free software [73]. GPL is a "copyleft" license, permitting users to freely use, copy, distribute, and modify GPL-licensed software without royalties. However, it mandates that any modified software must also be released under the GPL license and requires the publication of the corresponding source code [74]. The release of GPL version 2 in 1991 introduced clearer definitions and terminology and

clarified how patents should be handled concerning GPL-licensed software. Specifically, GPL2 aims to prevent patent holders from asserting patents against users of GPL-licensed software [75].

The Zlib license is a permissive software license similar to the MIT license, first released as part of the ZLIB compression library in 1995 [76]. The main difference between the two licenses is that the MIT License explicitly includes a patent grant clause, ensuring that the license grants users permission to use any patents held by the copyright holder, while the Zlib license typically does not include such a clause [76]. Thus, the Zlib license is slightly less permissive than the MIT license.

11.1 Metric Tool Installation

CCCC The same version of CCCC that is used in this thesis can be installed using:

```
apt-get install cccc=1:3.1.4-12build1
```

SLOCCount SLOCCOUNT was installed using the following *apt* command:

```
apt-get install sloccount=2.26-5.2
```

Flawfinder Flawfinder is available via the *apt* packet manager and can be installed with the following command:

```
apt-get install flawfinder=2.0.10-0.1
```

CLOC A variety of installation methods are listed on the projects Github [47]. CLOC is available via the *apt* packet manager and can be installed with the following command:

```
apt-get install cloc=1.82-1
```

Lizard Lizard is a official python package and available via *pip* [62]. The command to install the same version used in this thesis is:

```
pip install lizard==1.17.10
```

PyCGA's McCabe PyCQA's McCabe is available via *pip*. The version used in this thesis can be installed using the following command:

```
pip install mccabe==0.7.0
```

OhCount Ohcount is available via the *apt* packet manager and can be installed with the following command:

```
apt-get install ohcount=4.0.0-1
```

Radon Radon is a official python package and available via *pip* [62]. The command to install the version used in this thesis is:

```
pip install radon==5.1.0
```

Metrics Metrics is a official python package and available via *pip* [77]. The command to install the same version used in this thesis is:

```
pip install metrics==0.3.3
```

Tokei Tokei can be installed using multiple different package managers including pacman, apk, conda, homebrew and others mentioned on the packages Github page or be compiled and installed locally from the source files [58]. For this thesis Tokei was installed using homebrew:

```
brew install tokei
```

It is pertinent to acknowledge a minor issue encountered with Homebrew where specifying a version during installation, such as "`brew install tokei@12.1.2`", impedes Homebrew's ability to locate the Tokei package. Consequently, it is advisable to verify the installed version of Tokei using "`brew info tokei`" after installation and prior to reproducing any experiments, to ensuring consistency with the version (v12.1.2) used in this thesis.

Multimetrics Multimetric is a official python package and available via *pip* [62]. The command to install the same version as used in this thesis is the following:

```
pip install multimetric==2.0.3
```

GoCLOC The documentation suggests installing GoCLOC using the following command:

```
go install github.com/hhatto/gocloc/cmd/gocloc@latest
```

The compiled binaries for multiple operating systems are provided on GoCLOC's Github page [61]. The Linux x86/x64 release of GoCLOC (v0.5.2) was used for the experiments in this thesis. The exact binary is named "gocloc" and is located in the root folder of the project and has the following SHA-256 checksum:

```
4ba6de50c3f9f83542ac629ec3d2d716b8131163c98f1cf1c05eb70733e22541
```

A comprehensive list of SHA-256 checksums for all releases can be viewed on GoCLOC's GitHub page. These checksums can be used to verify and download the corresponding binary from GitHub [61].

SCC The SCC library can be installed using Homebrew, MacPorts, Scoop, Chocolatey, Snap, FreeBDS's package manager and a variety of other methods [46]. For this thesis SCC (v3.1.0) was installed using the Snap package manager with the following command:

```
snap install scc
```

11.2 Additional Test Cases

11.2.1 Source Lines of Code

	SLOC Comparison							
	SloC.	CLOC	OhC.	Rad.	Met.	Tokei default | ini	GoC.	SCC
Flask	9875	9976	9977	10013	9976	13327|10015	9976	14998
Scapy	47547	46682	46872	48399	46835	52046|47001	46830	55125
Matplotlib	139334	134649	135010	137697	137697	191667 |135833	135202	206558
SQLAlchemy	400220	399799	400724	413962	400710	472810|400645	400890	525870
Pandas	397871	389043	389028	405031	388990	477438|387867	389012	510059

Table 54: Overall SLOC Result of the Flask , Scrapy , Matplotlib, SQLAlchemy and Pandas Libraries.

Five Popular Python Libraries Output Numbers

11.3 DGA Implementations

11.3.1 DGAFobberV1

Python

```python
from pr2.dga_classes.dga_abstract_class import DGA

class DGAFobberV1(DGA):
    """
    Implementation of the "Fobber Version 1" DGA.
    """
    def __init__(self, seed: str = ""):
        super().__init__()
        self.r = 0xC87C8A78
        self.c = -1719405398
        self.ll = 17
        self.tld = '.net'
        self.domain = ""

    def ror32(self, v: int, n: int) -> int:
        return ((v >> n) | (v << (32 - n))) & 0xFFFFFFFF

    def generate_domain(self) -> str:
        self.domain = ""
        for _ in range(self.ll):
            self.r = self.ror32((321167 * self.r + self.c) & 0xFFFFFFFF, 16)
            self.domain += chr((self.r & 0x17FF) % 26 + ord('a'))
        self.domain += self.tld
        self.lastdomain = self.domain
        return(self.lastdomain)
```

11.3.2 DGABanjori

Based off Johannes Baders implementation [71]

Python

```python
'''This module contains an implementation of the Banjori DGA.

To ensure smooth operation it is implemented in a single class with the same name.

'''
from pr2.dga_classes.dga_abstract_class import DGA


class DGABanjori(DGA):
    '''Implementation of the "Banjori" DGA.'''

    def __init__(self, seed: str = "earnestnessbiophysicalohax.com"):
        super().__init__()
        self.seed = seed
        self.lastdomain = seed

    def map_to_lowercase_letter(self, s: int ) -> int:
        return ord('a') + ((s - ord('a')) % 26)

    def generate_domain(self) -> str:
        dl = [ord(x) for x in list(self.lastdomain)]
        dl[0] = self.map_to_lowercase_letter(dl[0] + dl[3])
        dl[1] = self.map_to_lowercase_letter(dl[0] + 2 * dl[1])
        dl[2] = self.map_to_lowercase_letter(dl[0] + dl[2] - 1)
        dl[3] = self.map_to_lowercase_letter(dl[1] + dl[2] + dl[3])

        self.lastdomain = ''.join([chr(x) for x in dl])
        return(self.lastdomain)
```

References

[1] D. E. Knuth, "Computer programming as an art," *Commun. ACM*, vol. 17, no. 12, pp. 667–673, Dec. 1974, ISSN: 0001-0782. DOI: 10.1145/361604.361612. [Online]. Available: https://doi.org/10.1145/361604.361612.

[2] E. Dijkstra, *A Short Introduction to the Art of Programming*. Technische Hogeschool Eindhoven, 1971. [Online]. Available: https://books.google.com/books?id=JE8ZAQAAIAAJ.

[3] www.geeksforgeeks.org (jagroopofficial), *Evolution of software engineering: From an art to engineering discipline*, https://www.geeksforgeeks.org/evolution-of-software-engineering-from-an-art-to-engineering-discipline/, Accessed: 13-05-2024.

[4] N. E. Fenton and M. Neil, "Software metrics: Successes, failures and new directions," *Journal of Systems and Software*, vol. 47, no. 2, pp. 149–157, 1999. DOI: https://doi.org/10.1016/S0164-1212(99)00035-7.

[5] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. DOI: 10.1109/TSE.1976.233837.

[6] B. Boehm, R. Valerdi, J. Lane, and A. Brown, "Cocomo suite methodology and evolution," *CrossTalk*, vol. 18, no. 4, pp. 20–25, 2005.

[7] M. H. Halstead, "Elements of software science.," *Elsevier North-Holland, Inc. ISBN 0-444-00205-7.*, vol. 14, no. 3, pp. 342–351, 1977.

[8] B. W. Boehm, "Software engineering economics," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4–21, 1984. DOI: 10.1109/TSE.1984.5010193.

[9] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings Conference on Software Maintenance 1992*, 1992, pp. 337–344. DOI: 10.1109/ICSM.1992.242525.

[10] P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994, Oregon Workshop on Software Metrics, ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(94)90067-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0164121294900671.

[11] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.03.044.

[12] *Why you need to know about code maintainability*, https://www.oreilly.com/content/why-you-need-to-know-about-code-maintainability/, Accessed: 27-11-2022.

[13] H. Zuse, *Software Complexity: Measures and Methods*. W. de Gruyter, 1991, ISBN: 9783110122268. DOI: 10.1515/9783110866087.

[14] J. Jones, *Abstract syntax tree implementation idioms*, https://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf, Accessed: 15-05-2023.

[15] Deepsource.com, *Glossary abstract syntax tree*, https://deepsource.com/glossary/ast/, Accessed: 16-05-2023.

[16] P. Arntz, *Explained: Domain generating algorithm*, https://www.malwarebytes.com/blog/news/2016/12/explained-domain-generating-algorithm, Accessed: 29-04-2023.

[17] S. Z. Aditya K. Sood, "A taxonomy of domain-generation algorithms," *IEEE Security & Privacy*, vol. 14, no. 4, pp. 46–51, 2016. DOI: 10.1109/MSP.2016.76.

[18] E. Mills, C. M. U. S. E. Institute, and U. S. D. of Defense, *Software Metrics: SEI Curriculum Module Sei-Cm-12-1.1* (Technical report (Carnegie-Mellon University. Software Engineering Institute)). Software Engineering Institute, 1988. [Online]. Available: https://books.google.at/books?id=sqOiNwAACAAJ.

[19] T. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*, NIST Special Publication 500-235 http://www.mccabe.com/pdf/mccabe-nist235r.pdf, Accessed: 13-06-2024.

[20] T. McCabe, M.
bibinitperiod Associates, and I. C. Society, *Structured Testing* (Ieee computer society). IEEE Computer Society Press, 1983, ISBN: 9780818604522. [Online]. Available: https://books.google.at/books?id=vtNWAAAAMAAJ.

[21] *Metrics definitions*, https://msquaredtechnologies.com/Metrics-Narration.html, Accessed: 22-09-2023.

[22] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. W. Boehm, "A sloc counting standard," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:16736634.

[23] Z. F. Hareton Leung, "Software cost estimation," *Handbook of Software Engineering and Knowledge Engineering, pp. 307-324*, 2002. DOI: 10.1142/9789812389701_0014.

[24] M. Halstead, "Advances in software science," in ser. Advances in Computers, M. C. Yovits, Ed., vol. 18, Elsevier, 1979, pp. 119–172. DOI: 10.1016/S0065-2458(08)60583-5.

[25] *Halstead's software metrics*, https://www.javatpoint.com/software-engineering-halsteads-software-metrics, Accessed: 29-11-2022.

[26] G. D. Greenwade, "The Comprehensive Tex Archive Network (CTAN)," *TUGBoat*, vol. 14, no. 3, pp. 342–351, 1993.

[27] *Software engineering | halstead's software metrics*, https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/, Accessed: 29-12-2023.

[28] *What is cyclomatic complexity?* https://www.perforce.com/blog/qac/what-cyclomatic-complexity, Accessed: 26-11-2023.

[29] G. J. Myers, "An extension to the cyclomatic measure of program complexity," *SIGPLAN Not.*, pp. 61–64, Oct. 1977. DOI: 10.1145/954627.954633.

[30] T. B. C. for Systems and S. E. (BCSSE), *Cocomo 81*, http://boehmcsse.org/tools/cocomo81/, Accessed: 05-05-2024.

[31] U. Center for Software Engineering, *Cocomo ii model definition manual*, https://www.rose-hulman.edu/class/cs/csse372/201310/Homework/CII_modelman2000.pdf, Accessed: 02-05-2024.

[32] geeksforgeeks.org, *Cocomo model – software engineering*, https://www.geeksforgeeks.org/software-engineering-cocomo-model/, Accessed: 05-05-2024.

[33] 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. DOI: https://doi.org/10.1002/(SICI)1096-908X(199705)9:3<127::AID-SMR149>3.0.CO;2-S.

[34] J. Fitzpatrick, “Applying the abc metric to c, c++, and java,” 2000. DOI: 10.5555/331120.331161. [Online]. Available: https://www.win.tue.nl/~wstomv/edu/2ip30/references/ABCmetric.pdf.

[35] V. R. Basili and T.-Y. Phillips, “Evaluating and comparing software metrics in the software engineering laboratory,” *SIGMETRICS Perform. Eval. Rev.*, 1981. DOI: 10.1145/1010627.807913.

[36] R. Lincke, J. Lundberg, and W. Löwe, “Comparing software metrics tools,” Association for Computing Machinery, 2008, ISBN: 9781605580500. DOI: 10.1145/1390630.1390648.

[37] B. Curtis, S. B. Sheppard, and P. Milliman, “Third time charm: Stronger prediction of programmer performance by software complexity metrics,” in *Proceedings of the 4th International Conference on Software Engineering*, IEEE Press, 1979, pp. 356–360. DOI: 10.5555/800091.802959.

[38] T. Littlefair, *Cccc - c and c++ code counter*, https://cccc.sourceforge.net/, Accessed: 27-11-2022.

[39] *User guide for cccc*, http://sarnold.github.io/cccc/CCCC_User_Guide.html, Accessed: 27-11-2022.

[40] M. Fink, *Metrics*, https://github.com/markfink/metrics, Accessed: 15-05-2023.

[41] D. A. Wheeler, *Sloccount*, https://dwheeler.com/sloccount/, Accessed: 13-12-2022.

[42] D. A. Wheeler, *More than a gigabuck: Estimating gnu/linux’s size*, https://dwheeler.com/sloc/redhat71-v1/redhat71sloc.html, Accessed: 13-12-2022.

[43] J. M. G.-B. M. A. O. P. P. de las Heras Quirós José Centeno González and V. M. Olivera, “Counting potatoes: The size of debian 2.2,” *European Online Magazine for the IT Professional,Vol. II, issue No. 6, December 2001*, 2001, Accessed: 13-12-2022 using archive: http://web.archive.org/web/20050129235335/http://www.upgrade-cepis.org/issues/2001/6/up2-6Gonzalez.pdf.

[44] J.-J. A.-I. J. M. G.-B. G. Robles-Martínez and I. Herráiz-Tabernero, “Measuring libre software using debian 3.1 (sarge) as a case study: Preliminary results,” *European Journal for the Informatics Professional,Vol. VI, issue No. 3, June 2005*, 2005, Accessed: 13-12-2022 using archive: http://web.archive.org/web/20060103134801/http://www.upgrade-cepis.org/issues/2005/3/up6-3Amor.pdf, ISSN: 1684-5285.

[45] D. A. Wheeler, *Counting source lines of code (sloc)*, https://dwheeler.com/sloc, Accessed: 13-12-2022.

[46] B. Boyter, *Sloc cloc and code*, https://github.com/boyter/scc, Accessed: 15-12-2023.

[47] AlDanial, *Cloc - count lines of code*, https://github.com/AlDanial/cloc, Accessed: 13-12-2022.

[48] AlDanial, *Recognized languages*, https://github.com/AlDanial/cloc#Languages, Accessed: 13-12-2022.

[49] T. Yin, *Lizard*, https://github.com/terryyin/lizard, Accessed: 15-05-2023.

[50] P. C. Q. Authority, *Mccabe complexity checker*, https://github.com/PyCQA/mccabe/, Accessed: 29-12-2023.

[51] Logilab and P. contributors, *Optional checkers*, https://pylint.pycqa.org/en/latest/user_guide/checkers/extensions.html, Accessed: 29-12-2023.

[52] B. D. Software, *About*, https://github.com/blackducksoftware/ohcount, Accessed: 13-12-2022.

[53] B. D. Software, *Package: Ohcount (4.0.0-4 and others)*, https://packages.debian.org/sid/ohcount, Accessed: 13-12-2022.

[54] I. openhub.net & Black Duck Software, *Ohcount*, https://openhub.net/p/ohcount, Accessed: 27-12-2023.

[55] SUSE, *Ohcount*, https://packagehub.suse.com/packages/ohcount/, Accessed: 27-12-2023.

[56] M. Lacchia, *Welcome to radon's documentation!* https://github.com/rubik/radon, Accessed: 15-05-2023.

[57] G. Brandl, *Languages*, https://pygments.org/languages/, Accessed: 15-05-2023.

[58] L. " Prima, *Tokei*, https://github.com/XAMPPRocky/tokei, Accessed: 27-12-2023.

[59] L. " Prima, *Tokei v12.1.2*, https://crates.io/crates/tokei, Accessed: 27-01-2024.

[60] docs.rs / Livia "XAMPPRocky" Prima, *Tokei: Count your code quickly.* https://docs.rs/tokei/latest/tokei/, Accessed: 27-12-2023.

[61] H. Hattori, *Gocloc*, https://github.com/hhatto/gocloc/, Accessed: 13-12-2022.

[62] K. Weihmann, *Multimetric*, https://github.com/priv-kweihmann/multimetric, Accessed: 15-05-2023.

[63] F. Sauer, *Metrics 1.3.6 - getting started*, https://metrics.sourceforge.net/, Accessed: 26-11-2022.

[64] *Code metrics values*, https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022, Accessed: 27-11-2022.

[65] *Codemetrics*, https://plugins.jetbrains.com/plugin/12159-codemetrics, Accessed: 27-11-2022.

[66] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, pp. 30–36, Apr. 1988. DOI: 10.1049/sej.1988.0003.

[67] N. Coulter, "Software science and cognitive psychology," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 2, pp. 166–171, 1983. DOI: 10.1109/TSE.1983.236461.

[68] M. Lacchia, *Introduction to code metrics*, https://radon.readthedocs.io/en/latest/intro.html, Accessed: 25-09-2023.

[69] D. A. Wheeler, *Sloccount user's guide*, https://dwheeler.com/sloccount/sloccount.html, Accessed: 13-12-2022.

[70] L. G. Thomas, *An analysis of software quality and maintainability metrics with an application to a longitudinal study of the linux kernel*, https://ir.vanderbilt.edu/bitstream/handle/1803/12852/Larry_Thomas-Dissertation-Final_Version-2008-07-07.pdf, Accessed: 02-05-2024.

[71] J. Bader, *The dga of banjori*, https://bin.re/blog/the-dga-of-banjori/, Accessed: 25-05-2024.

[72] D. W. Jerome H. Saltzer, *The origin of the "mit license"*, https://web.mit.edu/Saltzer/www/publications/MITLicense.pdf, Accessed: 29-12-2023.

[73] I. Free Software Foundation, *Gnu general public license, version 1*, https://www.gnu.org/licenses/old-licenses/gpl-1.0.html, Accessed: 29-12-2023.

[74] R. Stallman, *Copyleft: Pragmatic idealism*, https://www.gnu.org/philosophy/pragmatic.html, Accessed: 29-12-2023.

[75] I. Free Software Foundation, *Gnu general public license, version 2*, https://www.gnu.org/licenses/old-licenses/gpl-2.0.html, Accessed: 29-12-2023.

[76] J.-l. Gailly and M. Adler, *Zlib license*, https://www.zlib.net/zlib_license.html, Accessed: 29-12-2023.

[77] M. Fink, *Metrics 0.3.3*, https://pypi.org/project/metrics/, Accessed: 25-09-2023.