D2.1

Report on Security and Privacy Metrics

WP2 – Guidelines and Procedure for System and Software Security and Legal Compliance

SIFIS-HOME
Secure Interoperable Full-Stack Internet of Things for Smart Home

Due date of deliverable: 31/03/2021
Actual submission date: 30/03/2021

Responsible partner: POL
Editor: Luca Ardito
E-mail address: luca.ardito@polito.it

28/03/2021
Version 1.0

Project co-funded by the European Commission within the Horizon 2020 Framework Programme		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

The SIFIS-HOME Project is supported by funding under the Horizon 2020 Framework program of the European Commission SU-ICT-02-2020 GA 952652

Authors: Luca Ardito (POL), Luca Barbato (LUM), Marco Ciurcina (POL), Giacomo Conti (POL), Andrea Saracino (CNR), Michele Valsesia (POL)

Approved by: Marco Tiloca (RISE), Domenico De Guglielmo (MIND)

Revision History

Version	Date	Name	Partner	Section Affected Comments
0.1	18/12/2020	Tentative ToC and contents	POL, LUM, CNR	All
0.2	12/01/2021	Added software quality metrics	POL	Section 2
0.3	26/01/2021	Added security metrics	CNR	Section 3
0.4	26/01/2021	Added tools for gathering quality metrics	LUM	Section 2
0.5	28/02/2021	Added privacy metrics	POL	Section 3
1.0	02/03/2021	Ready for internal review	POL, LUM, CNR	All
1.1	18/03/2021	Changes after internal review	POL, LUM, CNR	All
1.2	29/03/2021	Ready for submission	CNR	All

Executive Summary

This document reports the theoretical and practical aspects of measuring the quality and security of source code produced in the SIFIS-Home project or by third-party developers developing apps expected to run on the SIFIS-Home framework.
The document is intended to provide a baseline for the definition of secure coding guidelines, which will be reported in D2.2 and D2.4. We will review here the main techniques and formalisms to evaluate the quality and reliability for general software, discussing how these measures are relevant for applications developed for the SIFIS-Home framework, presenting at first general quality metrics and then focusing on security and privacy-related indexes, as they are defined in the literature.
A subset of these metrics will also be used by the mechanisms to evaluate the quality and security of SIFIS-Home applications provided by third-party developers through the tools for evaluating the software defined in D2.3 and D2.5.

Table of contents

1 Introduction

Software Engineering has always been devoted to the issue of program quality, which, by definition, is seen as the extent to which a product meets a certain number of expectations concerning both its operation and its internal structure.
A more precise definition of code quality has been illustrated by [Kothapalli 2011]: the source code's ability to meet the stated and implied requirements for a given software project.
Software measurement is a process that assesses the manifestation of the size, quantity, amount, or dimension of particular attributes of a software product.
In the literature, parameters have been established against which software quality can be measured or defined. These are divided into two broad categories: *external* parameters, which refer to how end-users perceive the program, and *internal* parameters, which refer to how developers perceive software quality.
Internal parameters can be classified as follows:

- Testability: a software is defined as testable if its correctness and reliability properties are easily verifiable, i.e., if it effortlessly reveals its failures.
- Maintainability: the ability of a program to be modified. These modifications include corrections or adaptations of the system to changes in requirements, environments, and specifications. It includes the properties of:
    - Repairability: ease of eliminating defects and
    - Evolvability, ease of modifying the program to adapt it to a new environment or improve its quality.
- Reusability: the ability to reuse a piece of software in creating another program, in the case of minor modifications.
- Portability: the ability of the system to run on different hardware and software platforms. This parameter is facilitated by modular design.
- Readability: a software is defined as readable if there is an ease in understanding the reading of the code and its organization and implementation.
- Modularity: useful to measure how many modules that compose a software artifact. Modules are portions of source code containing instructions written to be reused multiple times in the same program.

External parameters can be classified as follows [McConnell 2004]:

- Correctness: The degree to which a system is free from faults in its specification, design, and implementation.
- Usability: The ease with which users can learn and use a system.
- Efficiency Minimal use of system resources, including memory and execution time.
- Reliability: A system's ability to perform its required functions under stated conditions whenever required having a long mean time between failures.
- Integrity: The degree to which a system prevents unauthorized or improper access to its programs and data. The idea of integrity includes restricting unauthorized user accesses and ensuring that data is accessed properly—that is, that tables with parallel data are modified in

parallel, that date fields contain only valid dates, and so on.

- Adaptability: The extent to which a system can be used, without modification, in applications or environments other than those for which it was specifically designed.
- Accuracy: The degree to which a system, as built, is free from error, especially with respect to quantitative outputs. Accuracy differs from correctness; it is a determination of how well a system does the job it’s built for rather than whether it was built correctly.
- Robustness: The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.

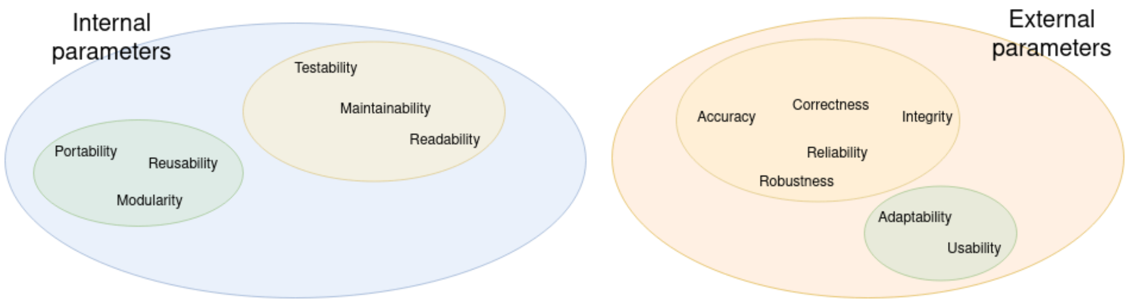

Figure 1 Software Parameters

In this work, the interest is focused on internal parameters since the final goal is strictly related to the developers’ point of view and not to the users’.
The external parameters are strongly related to the internal parameters:

- Correctness, Reliability, Accuracy, Integrity and Robustness are correlated to Testability, Maintainability and Readability.
- Adaptability and Usability are correlated to Modularity and Portability.

One of first the factors that can compromise a program comprehension is code readability. When a source code is hard to read, it is not easy to understand its flow and side effects. Poorly written code leads developers to introduce new bugs when fixing old bugs or adding new features. Readability measures the effort of the developer to access the information contained in the code. In contrast, understandability measures the complexity of such information [Nayrolles 2018]. However, code maintainability is the most critical part of software development. Being highly maintainable is the key to reducing approximately 75% of most systems’ life cycle costs [Welker 2001].
The IEEE Standard Glossary of Software Engineering Terminology defines software maintainability as the ease with which a software system or component can be modified to correct errors, improve performance or other attributes, or adapt to a changing environment.
Furthermore, according to the ISO standard IEC 9126, software follows the evolution of the organization, meaning that the program must adapt to all the boundary characteristics present in its development (environment, requirements, functionality).
Also, according to the ISO standard IEC 9126, the maintainability of the code has some attributes that allow its description entirely:

- Analyzability: ability to perform diagnosis on the software and identify the causes of errors and malfunctions.

- Changeability: ability to allow the development of changes to the original software. Implementation includes changes to code, design, and documentation.
- Stability: ability to avoid unwanted effects as a result of changes to the software.
- Testability: the ability to enable verification and validation of modified software, in other words, to perform testing.
- Maintainability compliance: ability to adhere to standards and conventions related to maintainability.

Software Engineering has dealt extensively with finding applicable models to measure the maintainability of software source code during its lifecycle. Through these models, it is possible to measure the source code's maintainability after any change to the code, checking whether the maintainability improves or worsens.

Over the years, it has been shown that measuring and improving code maintainability is very useful for managing technical debt; a definition used to describe all the complications that arise during the development of a software project [Cunningham 1992] . Besides, another more recent study has shown that analysis and measurement of source code maintainability are still the main methods used for the management of technical debt [Ernst 2015].

Software quality management is becoming a topic of absolute necessity as systems over the years are evolving in complexity and size. Using effective programs or tools to maintain them is critical for developers during the software lifecycle.

There are several types of tools in the literature that can be used to improve software quality [Krishnan 2007]:

- Static Analysis Tools: are useful for examining problems based on code analysis, such as the use of uninitialized variables, the possibility of memory leaks, dereferencing of null pointers.
- UT Tools: allows performing Unit Testing of the source code.
- Memory Leak Detection Tools: detect possible memory leaks at runtime.
- Code Browsing/Reverse Engineering Tools: help with code understanding so that improvements and troubleshooting can be applied appropriately.
- Profiling Tools: help understand and monitor performance aspects of the code.
- Coverage Tools: highlight which test cases cover parts of the code run to ensure test quality.

Software Quality is an aspect that has fundamental importance within the SIFIS-Home project together with Security and Privacy.

The purpose of this document is providing the theoretical and practical aspects of measuring the quality and security of source code produced in the SIFIS-Home project or by third-party developers developing apps expected to run on the SIFIS-Home framework.

SIFIS-Home will provide developers with software verification and evaluation tools to assess and communicate the overall quality of source code and produced applications to end-users in a user-friendly way.

SIFIS-Home developer must also consider software-based security metrics that aim to detect programming practices that might introduce, either by mistake or maliciously, dangerous behaviors or exploitable vulnerabilities.

In addition to that, a SIFIS-Home developer needs to evaluate a set of regulations and related measures to analyze privacy implications based on data management strategies.

Through these mechanisms, the SIFIS-Home project pushes developers to implement applications according to their best security and quality criteria, building over time a reputation score aimed at winning end-user trust.

2 Software Quality Assessment

Software Quality Assessment consists of several methodologies to evaluate the quality of various aspects and behavior of software through an assessment model. An assessment model contains quality criteria with straightforward methods to assess each quality criterion. The assessment method is often a mathematical model which aggregates product metrics to quality factors [Yan 2019].
Deissenboeck et al. in [Deissenboeck 2011] presented a toolchain for supporting, creating, and editing quality models and conducting automated quality assessments.
The software can be considered adequate if it satisfies its requirements.
The requirements can be grouped into functional requirements and non-functional requirements.
With *functional requirements,* we consider all the requirements specific to a given application.
With *non-functional* requirements we consider the primary attributes common in all software, notwithstanding their specific behavior and design.
Most of the *non-functional* requirements can be evaluated by automatic means. Tools can be used to produce reports and metrics with little to no human interaction.
A good deal of *functional* requirements cannot be automatically assessed and requires dedicated professional insights to be confirmed. The approach of declaring capabilities and another form of software contract allows some automatic assessment of the intended behavior.
The focus of the project WP2 is providing tools to automate the software assessment in order to minimize the effort of maximizing the software quality.
We will better describe a series of tools adopted in this project in *deliverable D2.3* and *D2.5*.

2.1 *Functional Requirements*

Even if assessing functional requirements is generally a manual process that often requires domain expertise, there are few opportunities for automation.

2.1.1 Documentation

It is not generally possible to ensure that the user documentation is in sync with the software itself without having developers and Quality Assurance experts cross-checking manually.
The developer documentation, though, can have a partial automatic assessment. While human intervention is needed to confirm that the documentation is in sync, it is easy to detect where the documentation is missing completely and make so that new code with no documentation is not accepted.

2.1.2 Behavior

2.1.2.1 Testing

Unit and integration testing are a widespread alternative to the more formal and cumbersome design by contract. Both allow some automatic verification of the software behavior.

In both cases, writing the tests requires a creative effort, but running the tests in a proper Continuous Integration environment does not require further human effort.

2.1.2.2 Sandboxing

Some platforms provide means to restrict the application to use the least amount of privileges; this is an easy and practical means to ensure that the application cannot misbehave.
iOS capabilities and Android Manifest Permission are good examples; macOS Gatekeeper, on the other hand, is a good case study on how badly set restrictions may cause more problems than the ones they are supposed to solve.

2.2 *Non-functional Requirements*

The most critical non-functional requirement is *maintainability*.
Software maintainability is defined as the ease of maintaining software during the delivery of its releases. Maintainability is defined by the ISO 9126 standard as
The ability to identify and fix a fault within a software component [ISO9126 1991] and by the ISO/IEC 25010:2011 standard as *degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers* [ISO/IEC2510 2011].
Maintainability is an integrated software measure that encompasses code characteristics, such as readability, documentation quality, simplicity, and understandability of source code [Krishan 2002].
Maintainability is also a crucial factor in the economic success of software products. It is commonly accepted in the literature that the most considerable cost associated with any software product over its lifetime is the maintenance cost [Zhou 2007]. The maintenance cost is influenced by many different factors, e.g., the necessity for code fixing, code enhancements, the addition of new features, poor code quality, and the subsequent need for refactoring operations [Lekshmi 2020].
The aspects of maintainability we focus on are the following:

- Code analysis: defect detection through static and dynamic code analysis.
- Code coverage: test coverage measurement and maximization through profiling.
- Code understandability: assessed through objective metrics

The three aspects are complementary to each other:

- maximizing the *test coverage* improves the results of the dynamic *code analysis*
- the computation of most of the *understandability objective metrics* is few orders of magnitude simpler than most of the **static code analysis** algorithms used to detect defects. Running the latter in the subset of the codebase deemed hard to understand by the former can provide useful results in a fraction of the time required to run the analysis over the full codebase corpus.

There are many valid models in the literature for measuring source code maintainability:

- The authors of [Aggarwal 2002] proposed a model based on three main characteristics: code readability (RSC), documentation quality (DOQ), and software understandability (UOS). The measures that are computed are transformed into *fuzzy* values, which will be processed and retransformed by domain experts.
- [Antonellis 2007] started from the characteristics of the ISO/IEC 9126 standard to propose a model for mapping *object-oriented* metrics in order to evaluate and measure the maintainability of a software system. This method has been applied to an OSS-type software, demonstrating the possibility to measure code maintainability through a systematic process.

- SIG Maintainability Model (SIG-MM): this model involves linking system-level maintainability characteristics with code-level measures in two steps. In the first pass, system-level characteristics are mapped to source code-level properties. In the second, one or more source code measures are determined for each property [Heitlager 2007].
- A probabilistic approach was adopted by [Bakota 2011] for high-level computing features by integrating expert knowledge while addressing ambiguity. The value of code maintainability is viewed as a probability distribution.
- SQUALE: this method is based on *Indices*, representing costs for evaluating various aspects of source code quality. There are two different models in the method: the *Quality Model* used to formulate and organize the non-functional requirements related to code quality, and the *Analysis Model* which contains both the rules that are used to normalize the measures and violations related to the code and the rules for aggregating the normalized values [Letouzey 2012].
- QUAMOCO: This approach involves the development of a meta-model for software quality that starts from structuring quality-related concepts to defining the operational methods for assessing its fulfillment in a specific environment. Also, an evaluation method is provided to integrate with the previous meta-model. This approach is used for integrating the abstract quality definitions provided in the quality taxonomies with concrete software quality assessment and measurement techniques [Wagner 2012].
- [Bauer 2012] proposed an alternative approach to the others analyzed so far, which involves using a framework that fits the needs of incremental quality and maintainability checks on the source code. This allows the incremental and distributed computation of quality metrics useful for software quality assessment and measurement, including both local and global metrics in the calculations.
- Delta Maintainability Model (DMM): this model measures the maintainability of a code change as a ratio of low-risk code to overall modified code. It also identifies source code risk factors by reusing software metrics and risk profiles from the SIG-MM, applying new aggregation and scoring for software delta metrics at the level of fine-grained code changes, such as commits or pull requests, instead of aggregating at the system level [Di Biase 2019].

2.2.1 Static Code Analysis

Static code analysis analyzes the code of computer software. It is usually performed without executing the relative program, in contrast with dynamic analysis, which is an analysis performed on programs while they are in execution. [Egele 2008] [Wichmann 1995]
Static analysis is mainly adopted by the industry for quality assurance purposes [Wichmann 1995]. It is typically used in safety-critical computer systems to locate potentially unsafe and insecure code [Livshits 2006].
Many industries have identified the use of static code analysis as a means of improving the quality of increasingly sophisticated and complex software:

- *Medical software*
- *Nuclear software*
- *Aviation software*

- *Automotive & Machines*

Automatic tools usually perform this analysis, and the produced results are then supervised through human intervention since the analysis may find false positives.
The semantics of a language strongly influences program analysis. The strength of the analysis may well depend on subtle features of the language, so it is fundamental to define a programming language in the most accurate way [Wichmann 1995]. Indeed, dynamic languages are more challenging to analyze than languages that include, for example, strong typing and range constraints. Hence, the nature of the input language needs to be taken into account in the static analysis specification to be undertaken.
The information obtained from the analysis of a code can be used just to highlight errors or define formal methods to mathematically prove whether the behavior of a code matches its specification.
In the sections below, we will present a series of open source and closed source solutions currently used in the academic and industrial worlds.

2.2.1.1 Open-Source Solutions

Open-source solutions can be particularly useful since they can be extended to support new platforms without the original authors' involvement, and they are not tied to a specific vendor. They may have a quite varying level of maturity. Below we provide some examples:

- Clang Static Analyzer: It leverages the clang parsers to perform static analysis over the languages supported by clang itself. It provides a detailed HTML report or provides the report within the IDE.
- GCC -fanalyze: It produces extended diagnostic messages and some GraphViz-compatible diagrams. As per GCC 10, it is in its infancy and under heavy development. Only C is currently supported.
- Infer: Infer is a static analysis tool developed by Facebook and written in the Ocaml programming language. It supports C, C++, objC, and Java. It offers integration with build-systems and provides reports in textual/diagnostic form, HTML, and JSON.

2.2.1.2 Proprietary Solutions

Proprietary solutions tend to be more feature-rich and more polished, and they offer extended support to their customers. It may be challenging to have them adopted and extended for specific needs. Below we provide some examples:

- PVS-Studio: It is a set of tools to run the on-premise static analysis. It has integrations for build-systems, IDEs, and Continuous Integration systems.
- Coverity Scan: It is an analysis *as-service* platform. It requires running a local scanner tool and then uploading its payload to run the remote platform's analysis. It supports out-of-box GitHub and Travis-CI.
- LGTM: It is a code analysis platform similar to Coverity Scan. It has better integrations with source hosting platforms such as GitHub. It offers a specific query language to dig deeper in the codebases and find structural similarities.

2.2.2 Dynamic Code Analysis

A *Dynamic* code analysis evaluates the code. In contrast, it is being executed, either by using specifically instrumented builds or by running unmodified code through special runtimes.
Differently, from a static analysis that is more focused on a software system's structural aspects, a dynamic analysis is more interested in detecting the behavioral aspects of a system.
Furthermore, a dynamic analysis provides more precise measures of the internal attributes of software, such as coupling, complexity, etc., based on the data collected during actual execution of the system, which have direct impact on quality factors of a software such as reliability, testability, maintainability, performance, and error-rates.
Below we present a simple comparative table to illustrate the differences between the metrics produced by static and dynamic analysis [Kumar 2010].

Static Metrics	Dynamic Metrics
Simpler to collect	Difficult to obtain
Available at the early stages of software development	Accessible very late in software development lifecycle
Less accurate than dynamic metrics in measuring qualitative attributes of software	Suitable for measuring quantitative as well as qualitative attributes of software
Deal with the structural aspects of the software system	Deal with the behavioral aspects of the system also
Inefficient to deal with object-oriented features such as inheritance, polymorphism, and dynamic binding	Dynamic metrics are capable of dealing with all object-oriented features
Less precise than dynamic metrics for the real-life systems	More precise than static metrics for the real-life systems

Table 1 Static and dynamic metrics comparison

Below we present a series of tools that can be used to obtain some dynamic metrics about software:

- Valgrind: It allows running unmodified binaries. It dynamically recompiles the binary as it runs on a simulation of the host CPU. The process tends to be slower than executing a custom binary with the instrumentation logic built-in. It requires platform-specific support, making it supporting new processors and operating systems more involving.
- DynamoRIO: It uses an approach similar to Valgrind, but it focuses on providing building blocks instead of a toolkit of ready-to-use tools.
- AddressSanitizer and MemorySanitizer: Introduced in LLVM and ported to GCC, they are a form of instrumentation and thus bound to a specific compiler and a set of helper libraries. They offer better execution speed, and their integration with debuggers such as GDB or RR makes them a good alternative to Valgrind.
- Miri: Miri is a specific tool to instrument and analyze the middle-level intermediate language currently used by Rust. It instruments the code and runs it on a generic platform abstraction

2.2.3 Code Coverage

Coverage test is a measure used to describe the degree to which a program's source code is executed when a particular test suite runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing, suggesting it has a lower chance of containing undetected software bugs than a program with low test coverage.
Basic coverage criteria:

- Function coverage
- Statement coverage
- Edge coverage
- Branch coverage
- Condition coverage (or predicate coverage)

Mutation and Fuzz Testing can be used for improving the effectiveness of software test cases and the coverage.
In Mutation Testing, some source code statements are changed (mutated) to check if the test cases can find errors in the source code. Mutation Testing aims to ensure the quality of test cases in terms of robustness that it should fail the mutated source code. Mutation Testing can be applied to many domains, including IoT. Parveen et al. [Parveen 2020] presented an automated framework that applies the mutation testing paradigm in the domain of IoT (Internet of things) apps.
Fuzz Testing is a software testing technique of putting invalid or random data (called FUZZ) into a software system to discover coding errors and security loopholes. The purpose of fuzz testing is inserting data using automated or semi-automated techniques and testing the system for various exceptions like system crashing or failure of built-in code. As described by Kumar et al. in [Kumar 2013] fuzzy techniques aim to reduce the number of test cases so that it is possible to achieve more efficient and accurate results. Fuzzy clustering is a class of algorithms for cluster analysis. The allocation of similar test cases is done to clusters that would help find out redundancy incorporated by test cases.
Code coverage tools' popularity and usage strictly depend on the language they support. Most code-coverage approaches rely on the running code's specific instrumentation; few rely on the normal debug information and runtime capabilities.
Here we present some examples of the most common ones.

- JACOCO: JaCoCo provides technology for code coverage analysis in Java VM-based environments. The focus is on providing a library for integration with various build and development tools.
- Coverage: Coverage is a tool for measuring code coverage of Python programs. It monitors a python program showing which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.
- gcov and llvm-cov: Compiler-specific tool to extract coverage information from binaries compiled with the profiling harness (e.g. `-fprofile-arcs -ftest-coverage`).
- lcov: Aggregates the coverage information generated by the `gcov` family of tools and produces reports in machine-parsable and human-readable formats.
- kcov: Relies on the DWARF debugging information and the platform-specific debug capabilities (e.g. ptrace) to extract coverage information from non-instrumented binaries.

- GCOVR and GRCOV: They collect and aggregate the coverage information provided by other tools. Both tools retain compatibility with `lcov` but support additional input and output formats.

2.2.4 Code Clarity

Code Clarity is clearness or lucidity as to perception or understanding of a code, freedom from indistinctness or ambiguity.[1]
The main goal consists of maximizing the amount of understanding conveyed in how a code is written, which needs to be easy to read, understand, and modify. Achieving clarity is about so much more than proper indentation. It requires code to be organized well, with careful planning and proper separation.
This concept impacts code maintainability also. Indeed, poorly written code can mean months of development later, while well written code can mean merely minutes or hours of maintenance in the future.
Code clarity can be evaluated through different standards and measures. As standards, we can consider the rules related to naming conventions and those that attempt to regulate the use of white spaces, so where comments, spaces, and braces should be put within a code.
Naming is important because it affects the readability of a code and the ease with which that code can be understood when it needs to be reviewed. Naming conventions are not meant to help the compiler or an interpreter. Indeed, a compiler or an interpreter has no trouble distinguishing names, no matter how long, short they are. However, a good name could help humans to get through a code in an easier way.
If a compiler or an interpreter does not consider at all the adopted naming convention, it considers even less the use of white spaces. There is a difference between the two, however. Formatting choices are relatively easy to change using a specific tool. At the same time, it is much harder to change a program to adhere to a different naming convention than the one the original programmer used, assuming, of course, one was used in the first place.
The names adopted by functions and identifiers impact the code clarity. However, it would be hard to capture this simple fact into a single rule or a simple naming convention that could be applied uniformly to all source codes. It comes down to the programmer's judgment whether a verb or a noun best captures the intent of a function.
An observed pattern states that very long names are pretty rare, while short names are best used for things that do not require much attention [Holzmann 2016].
For what concerns the approaches to evaluate the clarity of a code[2], three of them have been created over the time:

- Command-query separation
- Loose coupling
- High Cohesion

Except for the Command-query separation, the other ones can be applied only on object-oriented languages since they mainly focus on classes. We will explore in the following deliverables whether those kinds of approaches can also be applied to trait-based languages like Rust.

2.2.4.1 Command-query separation

Command-query separation provides a basis for safeguarding a code against unintended side effects

[1] https://gorails.com/blog/why-you-should-focus-on-writing-code-with-clarity Last visited 22/01/2021
[2] https://alistapart.com/article/coding-with-clarity/ Last visited 22/01/2021

when functions are called. Functions can be *commands*, which perform an action, and *queries*, which answer a question. A function should *not* be both of them simultaneously.
Thus, a *query* function answers a specific question returning a determined value, without altering the data state. Conversely, a *command* function runs a command which alters the data state, but it does not return any value. For maximum clarity, a function *should* never both return a value and alter the data state.
This kind of separation clearly reflects the intent and prevents errors. As functions and code bases become large, command-query separation becomes much more important, as hunting for the function definition to find out what it does is not an efficient use of time.

2.2.4.2 Loose Coupling

Coupling is a measure of how much one program unit relies on others. Too much coupling (or tight coupling) is rigid and should be avoided, so a code needs to be flexible enough to cover a wide variety of use cases.
Tight coupling consists of copying and pasting code, making minor changes to it, or rewriting code because it has been changed somewhere else in the codebase. This behavior is most prevalent in a group of functions and variables which could be better represented as a class. So when there are problems with inter-dependencies among functions, it is probably appropriate to break functions into a new class.
It is common for a developer to have either to use an excessive amount of function parameters or to create multiple copies of each function with the variables as hard-coded.
The problem above could be solved by using the *loose coupling* approach, which generally results in much greater clarity. It consists of breaking up the functions and variables into a reusable class. This results in fewer functions, with the variables stored only in one place, thus making updates much easier to perform.
Nevertheless, the tight coupling can also be present when a specific class needs to be modified because another one has changed. This usually happens when a class depends on methods or properties taken from other classes. For example, in order to not break the code when new parameters are added to a class, a *loose coupling* way could be passing the constructor parameters as an object with the receiving object having fallback default values.
Good code should be built as a series of independent blocks which are easily connectable with one another, rather than a series of intertwined pieces.

2.2.4.3 High Cohesion

Cohesion is a measure of how much the various program units belong together. A high level of cohesion is good and adds clarity to code blocks. Instead, a low level of cohesion is bad and leads to much confusion. Functions and methods in a code block should make sense together, in practice having a high level of cohesion.
High cohesion means keeping related things together and “close” to each other. For example, this means keeping database functions or functions related to a particular element in a same block or module. This helps not only with understanding how such things are laid out and where to find them, but also with preventing naming conflicts. If there are 30 functions, conflicting name chances are far greater than when there are 30 methods split over four classes.
If two or three functions use the same variables, they belong together. For example, a series of functions and variables that control a page element, like a slider, represents an excellent opportunity for high cohesion. It is possible to bundle them up into an object.
Repeated code is a sure sign of low cohesion. Similar lines of code should be broken into functions, and

similar functions should be broken into classes. The rule of thumb here is that a line of code should never be repeated twice. This is not always possible in practice, but it is always a good thing to think about how to cut down on repetition.
Similarly, the same bit of data should not exist in more than one variable. If the same bit of data is defined in multiple places, it is better to group it into a class. Alternatively, when references to the same element need to be passed to multiple functions, that reference should probably be a property in a class instance.
To further increase cohesion, objects can even be put inside other objects. Conversely, unrelated things should not be together in the same class. If multiple methods in a class do not use properties, this can be a sign of low or bad cohesion. Similarly, if methods cannot be reused in a few different situations, or if a method is not used at all, this can also be a sign of low or bad cohesion.
High cohesion helps alleviate tight coupling, and tight coupling is a sign that greater cohesion is needed. If the two ever come into conflict, though, choose cohesion. High cohesion is generally a greater help to the developer than loose coupling, although both can usually be accomplished together.

2.2.4.4 Code Maintainability

Code maintainability comprehends a series of different metrics to evaluate many aspects related to the maintainability of a code.
One of them is the *verbosity* and it is usually considered in terms of the number of code lines in a source file:

- **SLOC:** Source Lines of Code. It returns the total number of lines in a file.
- **PLOC:** Physical Lines of Code. It returns the number of instructions and comment lines in a file.
- **LLOC:** Logical Lines of Code. It returns the number of logical lines (statements) in a file.
- **CLOC:** Comment Lines of Code. It returns the number of comment lines in a file.
- **BLANK:** Blank Lines of Code. It returns the number of blank lines in a file.

The rationale behind using multiple measurements for the lines of code can be motivated by the need to measure different facets of the size of code artifacts and the relevance and content of the lines of code.
The measurement of physical lines of code (*PLOC*) does not consider blank lines or comments, however, the count depends on the physical format of the statements and programming style since multiple *PLOC* can concur to form a single logical statement of the source code. *PLOC* are sensitive to logically irrelevant formatting and style conventions, while *LLOC* are less sensitive to these aspects [Nguyen 2007].
In addition to that, the *CLOC* and *BLANK* measurements allow a finer analysis of the amount of documentation (in terms of used APIs and explanation of complex parts of algorithms) and formatting of a source file.

Another aspect is how a code is *structured*, so how the structure of a source code is analyzed in terms of the properties and functions that compose the source files. To that end, three metrics have been adopted:

- **NOM:** Number of Methods. It counts the number of methods in a file.
- **NARGS:** Number of Arguments. It counts the number of arguments of each method in a file.

- **NEXITS:** Number of Exit Points. It counts the number of exit points of each method in a file.

Nargs and *Nexits* are intuitively linked with the easiness in reading and interpreting a source code: a function with a high number of arguments can be more complex to analyze because of a higher number of possible paths, while a function with many exits may include higher complexity in reading the code for performing maintenance efforts.

To evaluate the *complexity* of a code, we have identified the following metrics:

- **CC:** McCabe's Cyclomatic Complexity. It calculates the code complexity by examining the control flow of a program.
- **COGNITIVE:** Cognitive Complexity. It is a measure which accurately reflects the relative difficulty of understanding, and therefore of maintaining methods, classes, and applications [Campbell 2018].
- **Halstead:** It calculates the Halstead suite. The Halstead Suite, a set of quantitative complexity measures originally defined by Maurice Halstead, is one of the most popular static code metrics available in the literature [Hariprasad 2017].

The details about the computation of all operands and operators are described in the table below:

Symbol	Description
$\eta 1$	Number of distinct operators
$\eta 2$	Number of distinct operands
$N1$	Total number of occurrences of operators
$N2$	Total number of occurrences of operands

Table 2 Halstead operators and operands

While this other table contains all the remaining metrics of the Halstead Suite computed from the operators and operands presented above:

Measure	Symbol	Formula
Program length	N	$N = N1 + N2$
Program vocabulary	η	$\eta = \eta 1 + \eta 2$

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

Table 3 Halstead formulas

Finally, we overview a metric to provide a single index of maintainability for software.

- **Maintainability Index (MI):** It is a composite metric to measure the maintainability of a source code [Oman 1992]. It is calculated both on files and functions.

Three distinct variants of this metric are considered. The original formula [Oman 1992], the one defined by the Software Engineering Institute (*SEI*) and promoted in the C4 Software Technology Reference Guide [Bray 1997], and finally the variant implemented for the Visual Studio IDE [Microsoft 2011].
The *SEI* formula adds to the original formula a specific treatment for the comments in the source code (i.e., the *CLOC* metric). Research is deemed more appropriate, given that the comments present in a source code can be considered correct and appropriate [Welker 2001].
Instead, the last formula resettles the MI value in the 0-100 range, without considering the distinction between *CLOC* and *SLOC* operated by the SEI formula [Molnar 2017].
The respective formulas are reported in the table below.

Variant	Formula
Original	$171.0 - 5.2 * ln(V) - 0.23 * CC - 16.2 * ln(SLOC)$
SEI	$171.0 - 5.2 * log_2(V) - 0.23 * CC - 16.2 * log_2(SLOC) + 50.0 * sin(\sqrt{2.4 * (CLOC/SLOC)})$
VS	$\max(0, (171 - 5.2 * ln(V) - 0.23 * CC - 16.2 * ln(SLOC)) * 100/171)$

Table 4 MI formulas

The measured MI interpretation varies according to the adopted formula to compute it, below the ranges for each of them.

Variant	Low Maintainability	Medium Maintainability	High Maintainability
Original	$MI < 65$	$65 < MI < 85$	$MI > 85$
SEI	$MI < 65$	$65 < MI < 85$	$MI > 85$
VS	$MI < 10$	$10 < MI < 20$	$MI > 20$

Table 5 MI interpretations

For the original and the *SEI* formulas of the *MI*, a value over 85 indicates an easily maintainable code. A value between 65 and 85 indicates average maintainability for the analyzed code. In contrast, a value under 65 indicates hardly maintainable code. The original and *SEI* formulas can also assume negative values. With the Visual Studio formula, the thresholds for medium and high maintainability are moved respectively to 10 and 20.

3 Security and Privacy Assessment Metrics

Unlike software quality assessment, security metrics are a far more recent field of study. The research community has started proposing objective measures to identify those pieces of code showing critical security issues. Many security metrics and standards have been proposed [Common 2017]. They are in use to measure the security and resilience of IT and software systems. WP2 is focused on providing metrics and tools to support SIFIS-Home aware app development, providing guidelines, and analyzing the source code to assess the level of security and the lack of vulnerabilities, which might imply security issues for the application, user, or the SIFIS-Home framework. In the following, this section reports an analysis of software-based security metrics proposed in the literature, based on detecting programming practices that might introduce, either by mistake or maliciously, dangerous behaviors or exploitable vulnerabilities. Furthermore, a set of regulations and related measures are introduced to analyze privacy implications based on data management strategies. The following metrics should not be taken as standalone, should instead be coupled with the previously defined code quality metrics, and should be used to have a more specific view on which issues might be brought by a low-quality source code.

3.1 ***Code Security Metrics***

As anticipated, a relatively large number of software quality attributes have been studied and measured, as discussed in the previous section. On the other hand, security received relatively small attention [Alshammari 2016] for what concerns the code. Security is mainly calculated at the system level. However, this kind of evaluation performed at the system level is not sufficient for a system where third-party software is installed at runtime. Such a model requires that instead, applications are evaluated at the code level. To this end, the following metrics and indexes have been defined to identify code structures and implementation language issues.

3.1.1 Stall Ratio

The Stall Ratio [Chowdhury 2008] is defined as the number of non-progressing code statements present in a block of code (e.g., in a loop statement), divided by the whole number of lines of code in that block of code. Examples of non-progressing statements are neutral operations such as *a=a+0*, empty loops, unused counters and tautologies (a==a). The rationale behind this metric's definition is to avoid attacks aimed at stalling a system in doing a long set of unnecessary operations, which might result, to a certain extent, in a Denial Of Service attack. Producing high numbers of logs or continuously asking to issue a connection to an external server are both examples of non-progressive operations, which might result in a DoS, either for the device running the code or an external device.

3.1.2 Coupling Corruption Propagation

Coupling between methods is the concept that two or more methods are reliant on each other due to one or more interwined elements. This could involve data sharing or decision-making in the child methods using one of the caller call parameters. The effects of content/control coupling can quickly ripple into other methods several levels down the call chain. Coupling corruption propagation is meant to measure the total number of methods affected by an erroneous originating method. Given a parameter *s* for a method *f*, the coupling corruption propagation [Chowdhury 2008] is defined as the number of children methods of *f* based on the parameter *s* of the original invocation. Thus, supposing to have a number of nested invocations, such as *f()*, which invokes *g()*, which invokes *k()*, if a variable *a* defined in *f()* is passed as parameter to invoke *g()*, directly or indirectly, uses the same parameter to invoke *h()*, the level of coupling corruption propagation is equal to 2, since a malicious alteration of *a* can potentially affect both *g()* and *h()*.

3.1.3 Critical Element Ratio

Critical element ratio measures the number of critical elements that are present in a specific block of code. A critical element [Chowdhury 2008] is defined as an element of a class that is not instantiated and is not used during the program execution, divided by the number of elements defined in a code block. Depending on the source code language, such elements might be maliciously changed during the program execution (buffer overflow) and might destabilize the whole execution. The critical element called ratio metric measures the ratio of elements that malicious user inputs can possibly corrupt to the total elements in a class or method. The more such user inputs enter the system, the more open the system is to the user. The more the system is open to the users, the more is the risk of getting attacked by malicious user inputs.

3.2 *Object Oriented Specific Security Metrics*

3.2.1 Information Flow and Data Accessibility

Security accessibility metrics statically measure the potential flow of information from an accessibility perspective for an individual object-oriented class [Alshammari 2009]. These metrics only consider attributes and methods declared as classified since they are the ones that need to be kept secret. The following measures, intended for object-oriented languages, are thus used to evaluate the data accessibility given by the availability to developers of methods to access private class data, which if not handled correctly might imply vulnerabilities and program misbehavior.

3.2.1.1 Classified Instances Data Accessibility (CIDA)

This metric measures the direct accessibility of classified instance attributes of a particular class. It helps to protect the classified internal representations of a class, i.e., instance attributes, from direct access. It is defined as "The ratio of the number of classified instance public attributes to the number of classified attributes in a class." Therefore, it is calculated by dividing the number of public classified instance attributes in a class to its total number of classified attributes. This gives us the ratio of classified instance attributes which have direct access from outside the class. Higher values indicate higher accessibility to these classified attributes and hence a larger 'attack surface.' This means a higher possibility for confidential data to be exposed to unauthorized parties. Aiming for lower values of this metric adheres to the security principle of *reducing the attack surface*.

3.2.1.2 Classified Class Data Accessibility (CCDA)

This metric measures the direct accessibility of classified class attributes of a particular class. This metric aims to protect the classified internal representations of a class. i.e., class attributes, from direct access. It is defined as follows: "The ratio of the number of classified class public attributes to the number of classified attributes in a class." This metric is calculated by dividing the number of public classified class attributes of a given class by its total number of classified attributes. The result shows the ratio of classified class attributes which are directly accessible from outside its class. Higher values mean that confidential data of that class has a higher chance of being exposed to unauthorized parties. This metric contributes towards measuring the attack surface size of a given program's classified class attributes. Thus, lower values of this metric enforce the security principle of *reducing the attack surface*.

3.2.1.3 Classified Operation Accessibility (COA)

This metric is the ratio of the accessibility of public classified methods of a particular class. It is defined as: "The ratio of the number of classified public methods to the number of classified methods in a class." It is calculated by dividing the number of classified methods which are declared as public in a given class by its total number of classified methods. This value also indicates the size of the attack surface of a given class. It aims to protect the internal operations of a class which interact with classified attributes from direct access. Lower values of this metric would reduce potential information flow of classified data which could be caused by calling public methods. This metric measures the potential attack surface size exposed by classified methods.

3.2.1.4 Classified Mutator Attribute Interactions (CMAI)

This metric measures the interactions of mutators (constructor, setters, getters) with classified attributes in a class. We define this metric as: "The ratio of the number of mutators which may interact with classified attributes to the possible maximum number of mutators which could interact with classified attributes." To calculate this metric, it is at first needed to find out in how many places in the design/program classified attributes could be mutated. Then, this number is divided by the total number of possible ways of mutating these classified attributes. The result is a ratio which can be used to indicate the potential interactions between mutators and classified attributes. Higher interaction means stronger cohesion between mutators and classified attributes within a given class, and consequently more privileges are given to mutators on classified attributes. Conversely, lower values indicate weaker cohesion between mutators and classified attributes which means a lower chance of classified information flow from mutators. With regard to the security principles, a lower value allows fewer privileges over confidential data and therefore adheres to the least privilege principle.

3.2.1.5 Classified Accessor Attribute Interactions (CAAI)

This metric measures the interactions of accessors with classified attributes in a class. We define this metric as: “The ratio of the number of accessors which may interact with classified attributes to the possible maximum number of accessors which could have access to classified attributes.” This metric is calculated in a similar way to the CMAI metric by first finding out in how many parts of the design/program classified attributes could be accessed. Then, this number is divided by the total number of possible ways of accessing these classified attributes. This results in a ratio which directly shows the potential interactions between accessors and classified attributes. Higher interaction means stronger cohesion between accessors and classified attributes within a given class. Similar to mutators, weak cohesion is desirable to reduce any potential flow of classified data caused by accessors. Weak cohesion also indicates fewer privileges are given to accessors over classified attributes. This would reduce the chance of potential flow of classified data to adversaries. Moreover, lowering the value of this metric would lower privileges of accessors over classified attributes and thus satisfy the security principle of least privilege.

3.2.1.6 Classified Attributes Interaction Weight (CAIW)

This metric is defined to measure the interactions with classified attributes by all methods of a given class. The metric is defined as: “The ratio of the number of all methods which may interact with classified attributes to the total number of all methods which could have access to all attributes.” This metric is calculated by finding the number of methods of a given class which may interact with classified attributes and dividing this number by the total number of potential interactions with all attributes in that class. The importance of this metric is that it shows how many potential class interactions are dependent on classified attributes. This is another metric which measures the privileges of class methods over classified data. However, this metric differs from the previous ones as it shows the overall privileges by a class’ methods over classified attributes. The higher the value of this metric for a given class the more privileges are given to this class’ methods over classified attributes, and therefore the less that class adheres to the security *principle of least privilege.*

3.2.1.7 Classified Methods Weight (CMW)

This metric is defined to measure the weight of methods in a class which potentially interact with any classified attributes in a particular class. We define this metric as: “The ratio of the number of classified methods to the total number of methods in a given class.” From this definition, we can calculate this metric by initially summing the number of methods which may interact in any form with classified attributes in a class. Then, this number is divided by the total number of methods in that class. This metric can directly measure the attack surface size of a given class based on its operations over confidential data. This differs from the previous attack surface metrics as it doesn’t focus on accessibility but instead it focuses on the interaction weight of classified methods. Higher values of this metric indicates that more classified operations are offered by the given class. This leads to a higher chance of information flow of classified data by calling the class's methods and violations of the security principle of reducing the attack surface.

3.2.2 Unhandled exceptions

Programs fail mainly for two reasons: logic errors in the code and exception failures. Exception failures occur when a program is prevented by unexpected circumstances from providing its specified service [Aggarwal 2017]. The following measures have thus been defined to measure the quality of exception handling in a specific class:

- The Number of Catch Blocks per Class (NCBS) is defined as the ratio of catch blocks in a class to the total number of possible catch blocks in a class. This ratio measures thus the percentage of handled exception on the total number of possible exceptions for a catch block. A low value of this index generally implies a poor work related to exception handling, where many conditions have not been considered and might thus represent an exploitable vulnerability.
- The Exception Handling Factor (EHF) is formally defined as the ratio of number of exception classes to the total number of possible *exception classes* in software, where the number of exception classes is the count of exceptions covered in a system. The exception class is passed as an argument to the catch construct as type of argument arg. This type of argument specifies types of exception classes.

These two metrics are semantically similar to the critical element ratio described in the previous subsection, representing the lack of error handling in object-oriented languages using exceptions as a construct.

3.3 *Vulnerability Assessment*

In the following, we report a list of measures and indexes to assess the potential threat brought by software, based on the presence of known vulnerabilities, due to usage of deprecated libraries or insecure software.

3.3.1 Common Weakness Enumeration (CWE)

CWE [Mellado 2010] provides a set of unified and measurable software weaknesses which facilitate a practical discussion, description, selection and use of software security services and tools, thus permitting these weaknesses to be discovered in the source code or in operational systems and facilitating a better understanding and management of those software weaknesses related to architecture and design. The severity of weaknesses can be scored using Common Weakness Scoring System (CWSS) and Common Weakness Risk Analysis Framework (CWRAF). CWSS enables organizations to score the severity of software coding errors found in their software applications to mitigate weaknesses in applications they are currently using and influence future purchases. In contrast, CWRAF enables organizations to apply CWSS to those CWEs that are most relevant to their specific businesses, missions, and deployed technologies.

3.3.2 Common Vulnerability Scoring System (CVSS)

CVSS is currently in the custody of the Forum for International Response Teams (FIRST). Among the benefits offered by the CVSS are Standardized punctuation of vulnerabilities, contextualized score, and open scoring system. The CVSS provides all the details concerning the parameters used to compute each score, thus permitting organizations to understand both the reasoning behind a score and the significance of differences between different scores. The scores assigned by the CVSS are derived from the following three groups of metrics:

- Base: This group represents the properties of a vulnerability that do not alter over time, specifically: the complexity of access, access vector, and the degree to which the system's confidentiality, integrity, and availability are compromised.
- Temporal: This group measures the properties of a vulnerability that alter over time, such as the existence of patches or code which could be exploited.
- Environmental: This group measures the properties of a vulnerability that are representative of the environment in which the IT is used, such as the prevalence of affected systems and potential losses.

The CVSS uses simple formulas along with the groups of metrics shown above to produce the final score associated with the vulnerability. The base metrics are used to derive a score from 0.0 to 10.0 as described in [Mellado 2010]. The CVSS was designed so that it would be understandable to the general public and permit any organization to prioritize the order in which it wishes to tackle computing vulnerabilities that affect it, regardless of the technology used by that organization in its computing systems. The overall CVSS for a specific software or system can be calculated by using freely available dedicated tools, like the one provided by FIRST itself.

3.3.3 Common Misuse Scoring System (CMSS)

CMSS [Mellado 2010] is an open scoring scheme standardized to measure the severity of software element misuse vulnerabilities. Software elements misuse vulnerabilities are those vulnerabilities in which the software elements provide a means to compromise the system's security. CMSS is derived from CVSS. The scores assigned by the CMSS are derived from three groups of metrics: base, temporal and environmental. The base metrics are used to evaluate the intrinsic exploitability of the vulnerability and the impact on confidentiality, integrity, and availability. The temporal measures measure the aspects of variation in time of the severity of the vulnerabilities, such as the preponderance of existing exploits. The environmental metrics measure those aspects of vulnerability related to the organization's specific vulnerability, such as the local implementation of countermeasures. The CMSS also includes a formula that combines these measures to provide a score for the severity of each vulnerability.

3.4 *Privacy Assessment Metrics*

Compliance with laws and regulations on privacy is most of all an issue of qualitative assessment of the adequacy of personal data processing. This is made clear by the rules of the GDPR.
According to art. 24 of the General Data Protection Regulation (EU) 2016/679 (hereinafter "GDPR"), the data controller (who determines the purposes and the means of the processing of personal data) is responsible for the correct processing of personal data. It has to implement appropriate technical and organizational measures (including appropriate data protection policies) to ensure and to be able to demonstrate that processing is performed in accordance with GDPR.

Moreover, articles 25 (Data protection by design and by default), 32 (Security of processing), and 35 (Data protection impact assessment) provide for assessment obligation on the data controller; art. 32 provides for assessment obligation on the processor (who processes personal data on behalf of the controller)[3].
From these rules, a common path to be followed arises. In assessing compliance, the controller must take into account: 1. the state of the art of technical and organizational measures, 2. the cost of implementation of technical and organizational measures, 3. the nature of the processing, 4. the scope of the processing, 5. the context of the processing, 6. the purposes of the processing, 7. the risks of varying likelihood and severity for rights and freedoms of natural persons posed by the processing. This list clarifies that GDPR compliance for IoT devices and the software installed on them implies a qualitative self-assessment to be performed by the controllers (and the processors).
Different methodologies to perform such assessments have been, and continue to be, proposed.
As a way of example, the DECODE project[4] adopted a series of privacy design strategies to comply with the obligation provided by art. 25 of GDPR to perform privacy by design and by default assessment (at the time of the determination of the means for processing and at the time of the processing itself) [Ciurcina 2017]:
1. Minimise
2. Separate
3. Abstract
4. Hide
5. Inform
6. Control
7. Enforce
8. Demonstrate.

The obligation to perform the data protection impact assessment (PIA) provided by art. 35 of GDPR implies a more accurate assessment than the assessment to be performed, according to art. 25 and 32 of the GDPR, and also it is provided only in special cases.
For example, CNIL (the French Privacy Supervising Authority) published[5], a method available in 3 documents to allow compliance with the obligation to perform the PIA.
The PIA methodology of CNIL, described in the first document [PIA Metodology 2018], allows to:

1. define and describe the context of the processing of personal data under consideration.
2. analyze the controls guaranteeing compliance with the fundamental principles: the proportionality and necessity of processing, and the protection of data subjects' rights.
3. assess privacy risks associated with data security and ensure they are properly treated.
4. formally document the validation of the PIA in view of the previous facts to hand or decide to revise the previous steps.

A document with templates to perform the Privacy Impact Assessments is also available on CNIL's

[3] It is worth mentioning that the IoT devices' seller is interested in supporting the controllers and the processors to perform the privacy assessments (to make it easier for customers that are controllers and processors to buy more easily its IoT devices). This makes available information to perform the assessments easily (including, if possible, a preconfigured assessment to be adapted by the controller) is an excellent way to achieve this.
[4] See https://decodeproject.eu/
[5] See https://www.cnil.fr/en/cnil-publishes-update-its-pia-guides

official website [PIA Template 2018].
Finally, a document with a knowledge base is also available [PIA Knowledge 2018].
CNIL also published a specific version of its PIA method applied to IoT devices [PIA IoT 2018]. CNIL also made available free software to perform the PIA2.
Performing GDPR privacy assessments can be supported by quantitative measures, including security measures.
Some approaches to complement security impact assessment and PIA in order to achieve an iterative and unified risk assessment process on-the-fly considering the interdependence of cybersecurity and privacy are starting to be proposed in the literature [Gouvas 2021].
It is, therefore, reasonable to expect that some of the security metrics to be produced by the SIFIS-Home project could be useful for the PIA and other privacy assessments to be performed by the controllers and the processors.

4 Conclusion

This document has treated the theoretical and practical aspects of measuring the quality and security of source codes produced in the SIFIS-Home project.
In Section 2, we have described the techniques to assess software quality through the use of static and dynamic methods. To do so, we have presented and listed the most common metrics present in the literature. This section has also introduced a set of modern and most used tools, both closed and open source, which perform this kind of analysis. Section 3 reports a high-level overview of methodologies to evaluate security aspects in source code to identify code blocks that might pose exploitable vulnerabilities.

5 References

[Common 2017]	2017. Common criteria for information technology security evaluation. Retrieved from https://www.commoncriteriaportal.org/cc/
[PIA Metodology 2018]	2018. Privacy impact assessment (PIA). methodology. Retrieved from https://www.cnil.fr/sites/default/files/atoms/files/cnil-pia-1-en-methodology.pdf
[PIA Templates 2018]	2018. Privacy impact assessment (PIA). templates. Retrieved from https://www.cnil.fr/sites/default/files/atoms/files/cnil-pia-2-en-templates.pdf
[PIA Knowledge 2018]	2018. Privacy impact assessment (PIA). Knowledge bases. Retrieved from https://www.cnil.fr/sites/default/files/atoms/files/cnil-pia-3-en-knowledgebases.pdf
[PIA IoT 2018]	2018. Privacy impact assessment (PIA). Application to IoT devices. Retrieved from https://www.cnil.fr/sites/default/files/atoms/files/cnil-pia-piaf-connectedobjects-en.pdf
[Campbell 2018]	Campbell G. A. 2018. Cognitive complexity. A new way of measuring understandability. (2018).
[Aggarwal, 2002]	K. K. Aggarwal, Y. Singh, and J. K. Chhabra. 2002. An integrated measure of software maintainability. In *Annual reliability and maintainability symposium. 2002 proceedings (cat. no.02CH37318)*, 235–241.

[Aggarwal 2017]	K. K. Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. 2007. Software design metrics for object-oriented software. *Journal of Object Technology* 6, 1 (January 2007), 121–138. DOI:https://doi.org/10.5381/jot.2007.6.1.a4
[Krishan 2002]	Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. 2002. An integrated measure of software maintainability. In *Annual reliability and maintainability symposium. 2002 proceedings (cat. No. 02CH37318)*, IEEE, 235–241.
[Alshammari 2009]	B. Alshammari, C. Fidge, and Diane Corney. 2009. Security metrics for object-oriented class designs. *2009 Ninth International Conference on Quality Software* (2009), 11–20.
[Alshammari 2016]	B. Alshammari, C. Fidge, and Diane Corney. 2016. Developing secure systems: A comparative study of existing methodologies.
[Antonellis 2007]	Panagiotis Antonellis, Antoniou Dimitris, Yiannis Kanellopoulos, Christos Makris, Evangelos Theodoridis, Christos Tjortjis, and Nikos Tsirakis. 2007. A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering–product quality standard. 1–11.
[Bakota 2011]	T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. 2011. A probabilistic software quality model. In *2011 27th IEEE international conference on software maintenance (ICSM)*, 243–252.
[Bauer 2012]	V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt. 2012. A framework for incremental quality analysis of large software systems. In *2012 28th IEEE international conference on software maintenance (ICSM)*, 537–546.
[Bray 1997]	Michael Bray, Kimberly Brune, David A Fisher, John Foreman, and Mark Gerken. 1997. *C4 software technology reference guide-a prototype.* Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
[Chowdhury 2008]	Istehad Chowdhury, Brian Chan, and Mohammad Zulkernine. 2008. Security metrics for source code structures. In *Proceedings of the fourth international workshop on software engineering for secure systems* (SESS ’08), Association for Computing Machinery, New York, NY, USA, 57–64. DOI:https://doi.org/10.1145/1370905.1370913
[Ciurcina 2017]	Marco Ciurcina, Shehar Bano, Eleonora Bassi. 2017. Privacy design strategies for the DECODE architecture. Retrieved from https://decodeproject.eu/publications/privacy-design-strategies-decode-architecture
[Cunningham 1992]	Ward Cunningham. 1992. The WyCash portfolio management system. In *Addendum to the proceedings on object-oriented programming systems, languages, and applications (addendum)* (OOPSLA ’92), Association for Computing Machinery, 29--30.
[Deissenboeck 2011]	F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner. 2011. The quamoco tool chain for quality modeling and assessment. In *2011 33rd international conference on software engineering (ICSE)*, 1007–1009. DOI:https://doi.org/10.1145/1985793.1985977
[Di Biase 2019]	M. Di Biase, A. Rastogi, M. Bruntink, and A. van Deursen. 2019. The delta maintainability model: Measuring maintainability of fine-grained code changes. In *2019 IEEE/ACM international conference on technical*

	debt (TechDebt), 113–122.
[Egele 2008]	Kirda Egele Scholte and Kruegel. 2008. A survey on automated dynamic malware-analysis techniques and tools. (2008).
[Ernst 2015]	Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (ESEC/FSE 2015), Association for Computing Machinery, 50--60.
[Hariprasad 2017]	T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software complexity analysis using halstead metrics. In *2017 international conference on trends in electronics and informatics (ICEI)*, IEEE, 1109–1113.
[Heitlager 2007]	I. Heitlager, T. Kuipers, and J. Visser. 2007. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, 30–39.
[Holzmann 2016]	G. J. Holzmann. 2016. Code clarity. *IEEE Software* 33, 02 (March 2016), 22–25. DOI:https://doi.org/10.1109/MS.2016.44
[ISO9126 1991]	ISO. 1991. ISO 9126 software quality characteristics.
[ISO/IEC2510 2011]	ISO/IEC. 2011. ISO/IEC 25010:2011 systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models.
[Kothapalli 2011]	Chaitanya Kothapalli, S. G. Ganesh, Himanshu K. Singh, D. V. Radhika, T. Rajaram, K. Ravikanth, Shrinath Gupta, and Kiron Rao. 2011. Continual monitoring of code quality. In *Proceedings of the 4th india software engineering conference* (ISEC ’11), Association for Computing Machinery, 175--184.
[Krishnan 2007]	R Krishnan, S Murali Krishna, and Nishil Bharill. 2007. Code quality tools: Learning from our experience. *SIGSOFT Softw. Eng. Notes* 32, 4 (2007), 5–es.
[Kumar 2010]	Gupta Kumar Chhabra J. 2010. A survey of dynamic software metrics. (2010), 1016–1029. DOI:https://doi.org/10.1007/s11390-010-9384-3
[Kumar 2013]	Gaurav Kumar and Pradeep Kumar Bhatia. 2013. Software testing optimization through test suite reduction using fuzzy clustering. *CSI Transactions on ICT* 1, 3 (2013), 253–260. DOI:https://doi.org/10.1007/s40012-013-0023-3
[Letouzey 2012]	J. Letouzey. 2012. The SQALE method for evaluating technical debt. In *2012 third international workshop on managing technical debt (MTD)*, 31–36.
[Livshits 2006]	Livshits. 2006. Improving software security with precise static and runtime analysis. (2006).
[McConnell 2004]	Steve McConnell. 2004. *Code complete, second edition*. Microsoft Press, USA.
[Mellado 2010]	Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. 2010. A comparison of software design security metrics. In *Proceedings of the fourth european conference on software architecture: Companion volume* (ECSA ’10), Association for Computing Machinery, New York, NY, USA, 236–242. DOI:https://doi.org/10.1145/1842752.1842797
[Microsoft 2011]	Microsoft. 2011. Code Metrics – Maintainability Index.
[Molnar 2017]	Arthur Molnar and Simona Motogna. 2017. Discovering maintainability

changes in large software systems. In *Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement*, 88–93.

[Lekshmi 2020] Lekshmi S Nair and J Swaminathan. 2020. Towards reduction of software maintenance cost through assignment of critical functionality scores. In *2020 5th international conference on communication and electronics systems (ICCES)*, IEEE, 199–204.

[Nayrolles 2018] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 8th working conference on mining software repositories* (MSR ’11), Association for Computing Machinery, New York, NY, USA, 73–82. DOI:https://doi.org/10.1145/3196398.3196438

[Nguyen 2007] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Citeseer, 1–16.

[Oman 1992] Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system’s maintainability. In *Proceedings conference on software maintenance 1992*, IEEE Computer Society, 337–338.

[Gouvas 2021] S. A.; Gouvas Papamartzivanos D.; Menesidou. 2021. A perfect match: Converging and automating privacy & security impact assessment on-the-fly. *Future Internet* 13, 30 (2021). DOI:https://doi.org/https://doi.org/10.3390/fi13020030

[Parveen 2020] S. Parveen and M. H. Alalfi. 2020. A mutation framework for evaluating security analysis tools in IoT applications. In *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)*, 587–591. DOI:https://doi.org/10.1109/SANER48275.2020.9054853

[Wagner 2012] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidi, A. Goeb, and J. Streit. 2012. The quamoco product quality modelling and assessment approach. In *2012 34th international conference on software engineering (ICSE)*, 1133–1142.

[Welker 2001] Kurt D Welker. 2001. The software maintainability index revisited. *CrossTalk* 14, (2001), 18–21.

[Wichmann 1995] Clutterbuck Wichmann Canning and Marsh. 1995. Industrial perspective on static analysis. (1995), 69–75.

[Yan 2019] Meng Yan, Xin Xia, Xiaohong Zhang, Ling Xu, Dan Yang, and Shanping Li. 2019. Software quality assessment model: A systematic mapping study. *Science China Information Sciences* 62, 9 (2019), 191101. DOI:https://doi.org/10.1007/s11432-018-9608-3

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

Annex A: Glossary

Acronym	**Definition**
CMSS	Common Misuse Scoring System
CVSS	Common Vulnerability System

CWE	Common Weakness Enumeration
DMM	Delta Maintainability Model
DOQ	Document Quality
IDE	Integrated Development Environment
ISO/IEC	International Organization for Standardization/International Electrotechnical Commission
LOC	Line of Code
MI	Maintainability Index
OSS	Open Source Software
SEI-MI	Software Engineering Institute Maintainability Index
SIG-MM	Software Improvement Group Maintainability Model
UOS	Understandability of Software
UT	Unit Testing
VM	Virtual Machine
VS	Visual Studio