D2.4

Final Developer Guidelines

SIFIS-HOME

Secure Interoperable Full-Stack Internet of Things for Smart Home

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

Due date of deliverable: 31/03/2023
Actual submission date: 31/03/2023

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

30/03/2023
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), Maurizio Morisio (POL), Marco Rasori (CNR), Andrea Saracino (CNR), Marco Torchiano (POL), Michele Valsesia (POL)

Approved by: Marco Tiloca (RISE), Teppo Ahonen, Marko Komssi (FSEC)

Revision History

Version	Date	Name	Partner	Section Affected Comments
0.1	17/02/2023	Tentative ToC and contents	POL, LUM, CNR	All
0.2	21/02/2023	Executive Summary and Introduction Release	POL, LUM, CNR	Executive Summary, Introduction
0.3	14/03/2023	Section 2 Release	POL, LUM	Section 2
0.4	15/03/2023	Section 3 Release	CNR, LUM	Section 3
0.5	16/03/2023	Section 4 Release	POL	Section 4
0.6	20/03/2023	Internal review draft	POL, LUM, CNR	All
1.0	30/03/2023	Final version after internal review	POL	All

Executive Summary

This document presents a series of guidelines to enhance software security and quality, structured into a workflow that developers can follow. We discuss the enforcement of this workflow through continuous integration, using GitHub Actions, and provide a practical example of applying this workflow on a WP6 component called libp2p-rust-dht.
We then address risks associated with the use of SIFIS-Home developer APIs, by introducing a system of API labels that provide information about potential safety, integrity, security, and privacy issues. This mechanism informs, at development time, third-party developers about the risks they are introducing within the application by using specific SIFIS-Home developer APIs. Moreover, it allows users to make informed decisions about installing applications based on their risk profiles and ensures that user-defined policies are respected at runtime.
Finally, we explore legal guidelines, focusing on GDPR compliance and its implications for software developers, application designers, and end-users. We also discuss licensing and the standardization of free and open-source software, proposing a traffic-light system for evaluating software with respect to privacy and licensing criteria.

Table of contents

1 Introduction

In Section 2, we propose a series of guidelines that a developer is strongly recommended to follow in order to improve the security and quality of software. These guidelines have been thought of and defined during the development of the various SIFIS-Home components, starting from software engineering concepts, and then structured into a precise workflow to simplify their usage. A workflow can thus be seen as a path that a developer should take in order to tackle specific issues, prevent determined problems from arising, and reduce well-known efforts. First, we will describe the workflow from a theoretical perspective and then decompose its structure into its main parts. Subsequently, we will show how the workflow has been effectively implemented through a continuous integration system - in our case, the GitHub Actions from GitHub. In the last part, we will apply our GitHub Actions scripts, which directly represent our workflow, on a WP5 and WP6 component called *libp2p-rust-dht*.

In Section 3, we deal with risks arising from the use of SIFIS-Home developer APIs. We define a set of labels representing safety, integrity, security, and privacy issues intrinsically related to the execution of each specific developer API. In other words, we associate each SIFIS-Home developer API with an API label that provides information about risks. The set of API labels associated with the APIs used in an application results in a general label called the app label. This label is shown to the user before the installation process of such an application. The advantage of this mechanism is twofold: (i) it informs the user about possible risks related to the application and allows the user to decide whether to proceed with the installation or not when some user-defined policies are in contrast with the risks displayed by the application; and (ii) it seamlessly integrates at runtime with user-defined policies, meaning that if the label of a given API is contained in the app label of an app, the user has decided to install the application anyway, and some risk contained in the app label violates a rule defined by the user, then the execution of such an API is automatically denied.

In Section 4, we provide legal guidelines. Collecting data is a necessary but challenging task. In the European Union, indiscriminate data collection is limited and regulated through the GDPR (General Data Protection Regulation). While there are some subjects, such as the Data Controller, that must be explicitly determined and are obliged to follow the GDPR's rules, others, like the software developer or the "application designer," do not have such an obligation but may be interested in programming along those lines, in a way that the resulting software is by default compliant with privacy laws. This allows the application to be better distributed, accepted and reviewed by the end user and the possible Service Provider. We try to clarify what rules must be followed by what subjects, and when it is mandatory or advantageous to follow them. This topic is also linked to licensing, free software use, and the legal obligations arising from its use. We explore the main requirements found in GDPR, specified in articles 13, 14, 25, 32, and 35, and the initiatives to make free and open software more standardized, such as in the OpenChain and ClearlyDefined projects. We propose a "traffic-light system" based on different criteria for evaluating software in privacy and licensing.

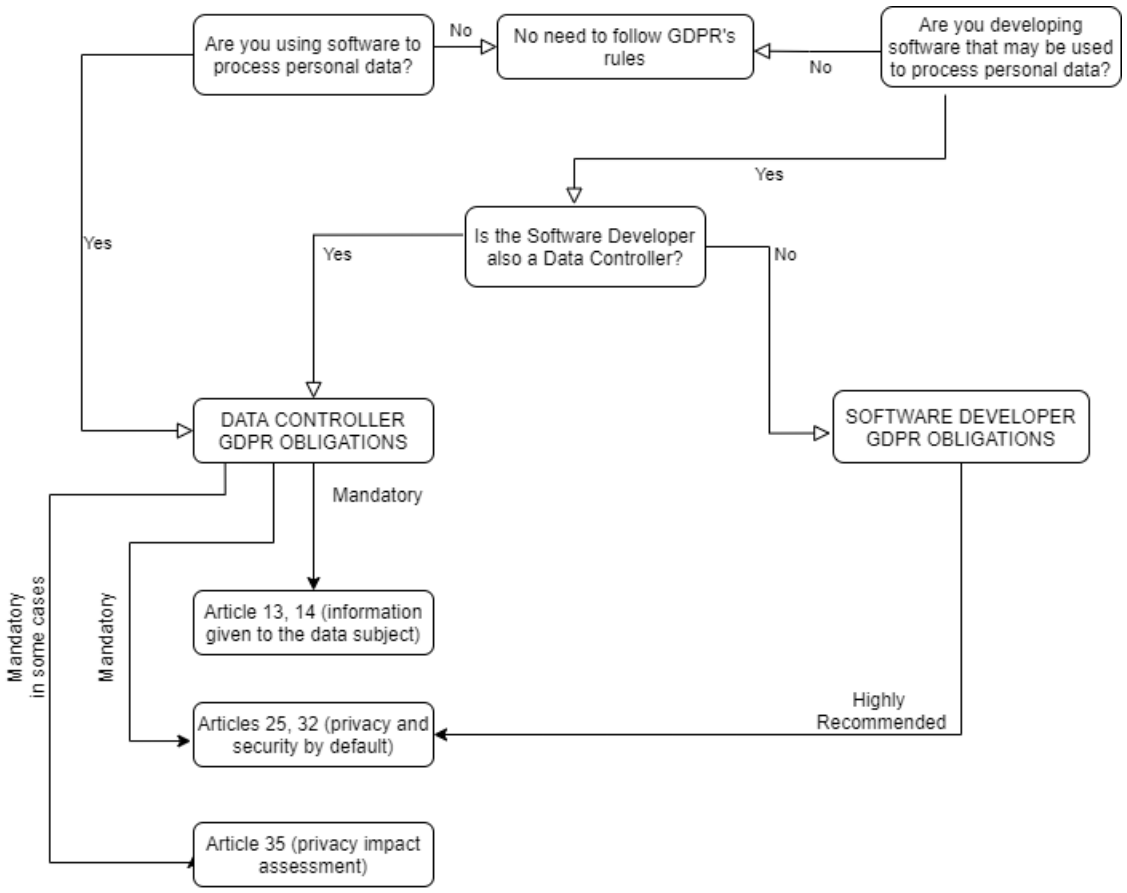


Figure 1 Main GDPR requirements and recommendations flowchart

2 Software Quality Guidelines

In this section, we propose some guidelines that a developer is **strongly** recommended to consider so that security and quality aspects are **always** guaranteed during software development.
The guidelines are structured as a workflow, representing the path that a developer should follow in order to avoid efforts related to software maintainability and, at the same time, prevent vulnerabilities and security issues. In order to design this workflow, we started from the questions raised during the development of our components, such as:

- How do we know if each single line of code has been tested?
- How does our software react in the case of an invalid input?
- How do we evaluate code quality and security of a determined component?

We have found the answers to these questions in a software engineering branch, called *Software Maintainability*, which groups various aspects related to the maintainability of software over time, from code to binary analysis. Starting from that, we have defined our workflow as a composition of some *Software Maintainability* areas:

- *Static analysis:* analyzes source code for defects and maintainability issues and provides information to improve code quality.
- *Dynamic analysis*: instruments and analyzes the software as it executes, in order to detect the vulnerabilities present in the binary in addition to checking against invalid inputs. It also provides information and graphs about the optimizations that are possible to adopt in order to improve the overall performance.
- *Code coverage*: determines the percentage of source code covered by a test suite.

In the rest of this section, we will explain as detailed as possible our journey through the definition and implementation of our workflow: the various parts that compose its structure, how it has been converted into GitHub Actions scripts to be run on a repository, and, at last, the process of integrating our workflow into a WP5 and WP6 component called **libp2p-rust-dht**.

2.1 *Workflow Structure*

The workflow structure is not a rigid one, indeed it can be modified according to the repository's needs; some phases can be removed if not necessary, or they can be moved forward and backward inside the workflow if priorities change. Most of the phases can be executed in parallel, since they include tasks that are independent from each other. This workflow tries to minimize the developer wait time: as soon as a mistake is detected, the workflow stops its execution, and the errors are immediately reported to the developer.

The first draft of our workflow has been thought of for projects which are created and developed only on a local machine. Starting from a commit, which represents the current development state, a developer follows a three-phased cycle to verify that everything works properly: static analysis, compilation test and software tests. All these phases provide rapid feedback, and are thus quick in detecting issues and

enabling developers to fix those. Dynamic analysis has not been considered in this initial workflow because it requires long execution times if it were to be performed for each single commit.

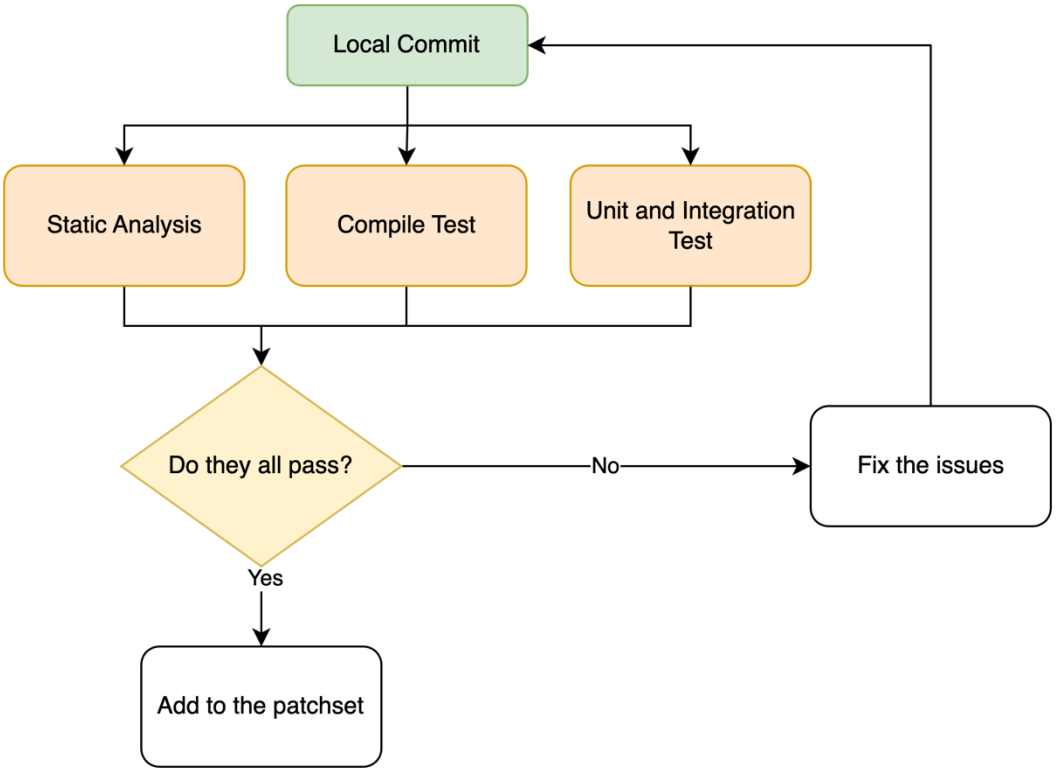


Figure 2 Local Workflow Structure

Subsequently, we have expanded our workflow to consider a **Pull Request** model. Once a set of local patchsets has been decided, a developer just needs to create a new pull request in the desired repository to start the workflow. Among the code changes included within a set of patchsets, there can be new features, bug fixes, and additional tests. As soon as a pull request has been created, it enters **in review** state. In this new workflow version, we have highlighted in light purple the most time-and resource-consuming phases, such as static fault analysis and dynamic fault analysis, which should run for **each** pull request change.

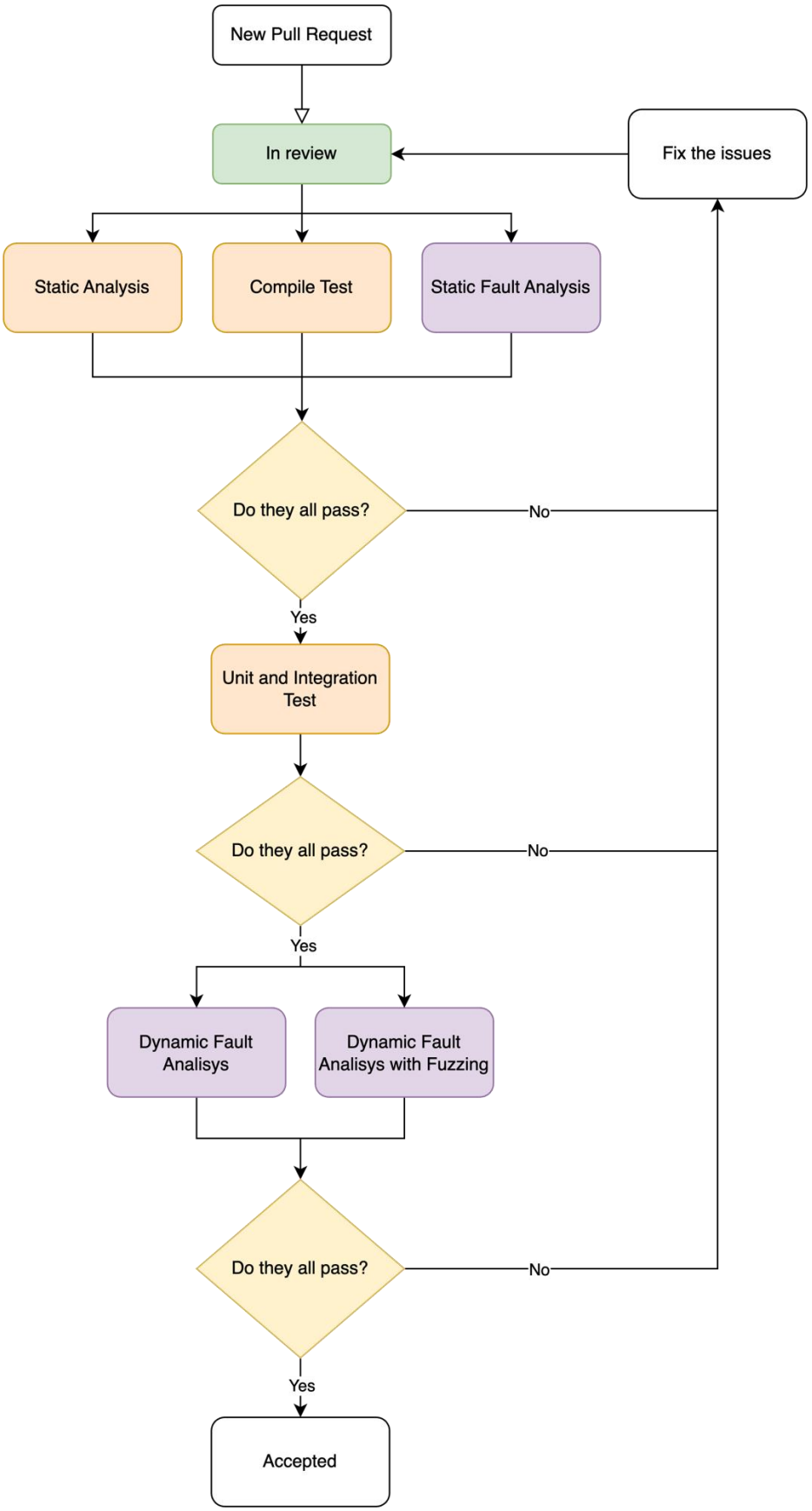

New Pull Request
In review
Fix the issues
Static Analysis
Compile Test
Static Fault Analysis
Do they all pass?
No
Yes
Unit and Integration Test
Do they all pass?
No
Yes
Dynamic Fault Analisys
Dynamic Fault Analisys with Fuzzing
Do they all pass?
No
Yes
Accepted

Figure 3 Pull Request Workflow Structure

When all workflow phases have been successfully passed, and reviewers have approved the respective code contributions, then a pull request can be merged. Code contributions should be reviewed by a third-party, since reliance on only automated tests can lead to logic mistakes as the same developer may make the same error both in tests and even in the code being tested.

The same workflow has been applied to package creation as well. Indeed, before deploying or distributing a software, it is **mandatory** to verify the same aspects that have been considered during the development ought to be verified. As soon as a new tag has been added, the workflow can start its execution.

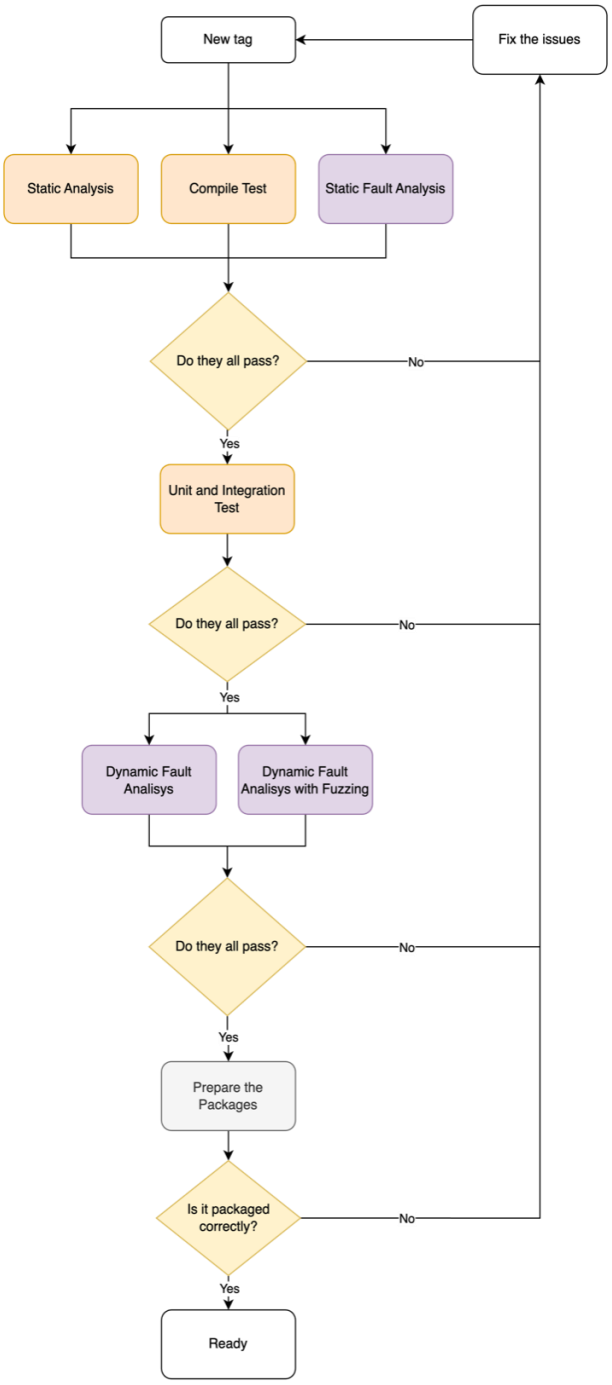


Figure 4 Deployment Workflow Structure

2.1.1 Static Analysis

Static analysis is the first phase of our workflow. It enforces code-uniformity through linters and provides a quick overview of the state of a project. A linter is a tool which analyzes source code to verify whether lints have been respected. A lintflags programming errors, bugs, stylistic errors, and bad-formatted constructs. The output of a linter are metrics and information which can help developers, other than reviewers, to highlight:

- If the project follows correct legal guidelines, more specifically the licenses associated to each file contained in the repository

- Which are the parts of the code with higher complexity, thus the ones that need more documentation and tests to be better understandable
- If code respects the lints conventions, set up by the owner through some configuration files, established for the project
- If the dependencies of a project are outdated, present vulnerabilities, or non-trusted sources

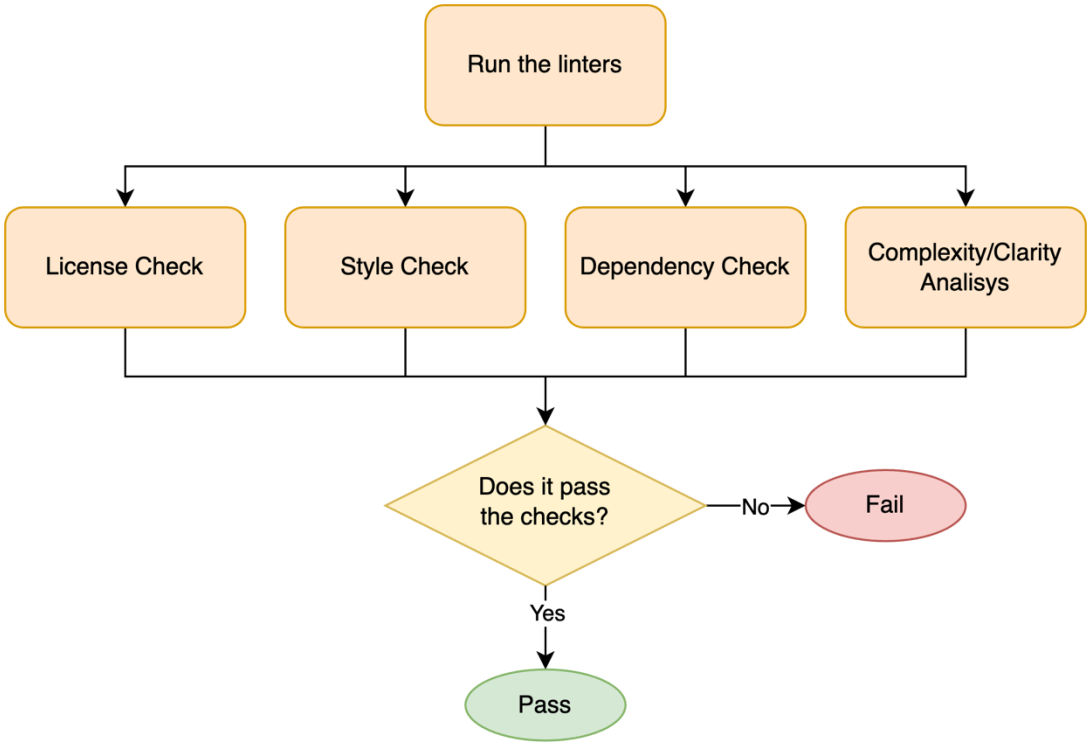


Figure 5 Static Analysis Flowchart

As a rule of thumb, these checks should be at least twice as fast as building the project. In some cases, though, this is not possible. For example, dependencies checks, which need to either look for security issues and identified vulnerabilities in huge online databases or build large and deep dependency trees, require a greater amount of time than simpler lints. A possible solution, but for some situations hard to implement physically, could be to not run dependency checks every time, but only when a dependency changes.

2.1.2 Compile Test

Software can run on different architectures, so making sure that the code builds for all supported targets is essential. Setting up and keeping operational an entire test environment for many architectures can be cumbersome, so having a cross-building setup is a good compromise. Indeed, this approach allows to immediately interrupt the workflow when code stops compiling on a specific architecture, notifying a developer to solve as soon as possible the issues for the problematic architectures.

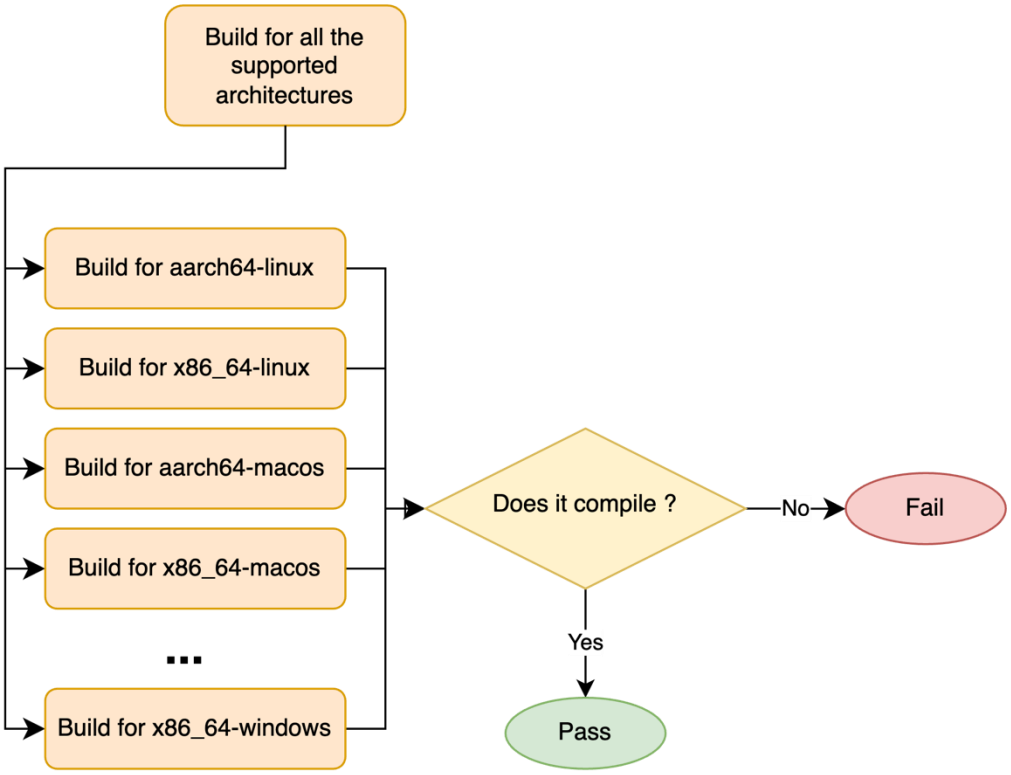


Figure 6 Compile Test Flowchart

2.1.3 Static Fault Analysis

Static fault analysis differs from static analysis because, in this phase, the code is scanned for security problems and vulnerabilities. It is usually as slow or a few times slower than building the software and executing linters, but it is still faster than the following phases, and, for many compiled languages, it is up to the compiler suite performing these checks. Among the main memory problems that can be detected during this phase, there are simple use-after-free and null-dereferences. This analysis can also identify some API misuse, thus errors, made by developers during the usage of an API, that could lead to bugs, system crashes, or security vulnerabilities.

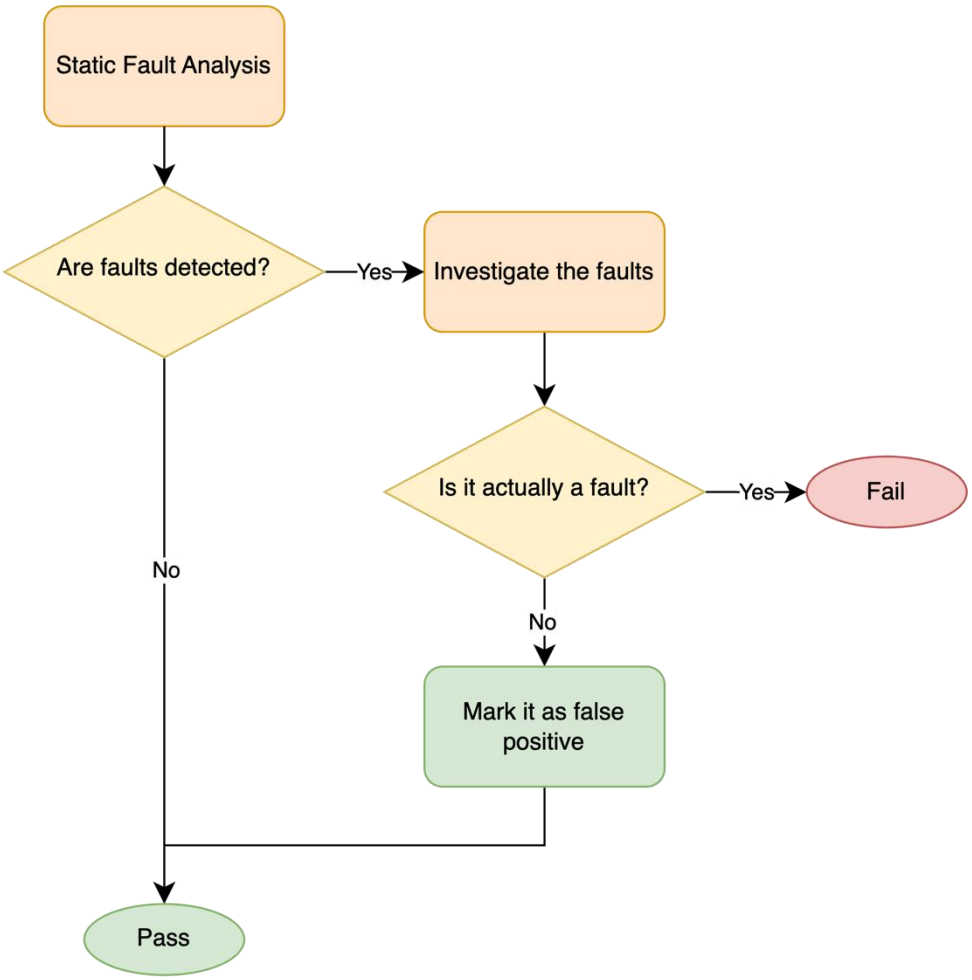


Figure 7 Static Analysis Flowchart

2.1.4 Unit and Integration tests

Unit and **integration** tests make sure that the behavior of a software is correct. Each unit test checks the behavior of a single software feature, and it is quick to write and execute. However, an entire unit test suite may require some time and resources to be completely executed.
Integration tests, instead, catch errors related to the cooperation among features within a software and they may be more cumbersome to write and slower to run compared to unit tests. In general, a test suite should **cover** as much of the codebase as possible.

2.1.5 Coverage Analysis

Code coverage is the percentage of code covered by a test suite. In order to compute this measure, a tool detects, through profiling instruments, which code lines are effectively executed by tests. When a new feature is introduced, and there are not enough tests to cover it, its code is highlighted in red, otherwise in green. If the partially covered option has been enabled, then its code is colored in yellow.

The concept of code coverage can be correlated to code certification, a procedure to guarantee that a software is reliable, secure, **well tested**, and with code written in a comprehensible way. We have focused on testability because the other properties of this procedure are already present within static

analysis and, still not explained, dynamic analysis. Since a percentage is a comparable value, we have thought of a mechanism that rejects any pull request with code coverage below a certain threshold, thus blocking the workflow.

Our procedure consists of assigning a colored stamp based on code coverage value. The colors are based on a traffic-light system model and for SIFIS-Home projects we have adopted the following thresholds:

- ***Red***: Code coverage is below 60%, which means that the repository is not well-covered, and the workflow **must** necessarily stop.
- ***Orange***: Code coverage is between 60% and 80%, which is an acceptable value, but still not a desirable one.
- ***Green***: Code coverage is greater than 80%, which means that the code is well-covered by tests.

The improvement introduced with this new procedure consists of blocking the execution of a workflow when the code coverage is low, thus the percentage is not only informational, but rather assumes an active role in the workflow execution.

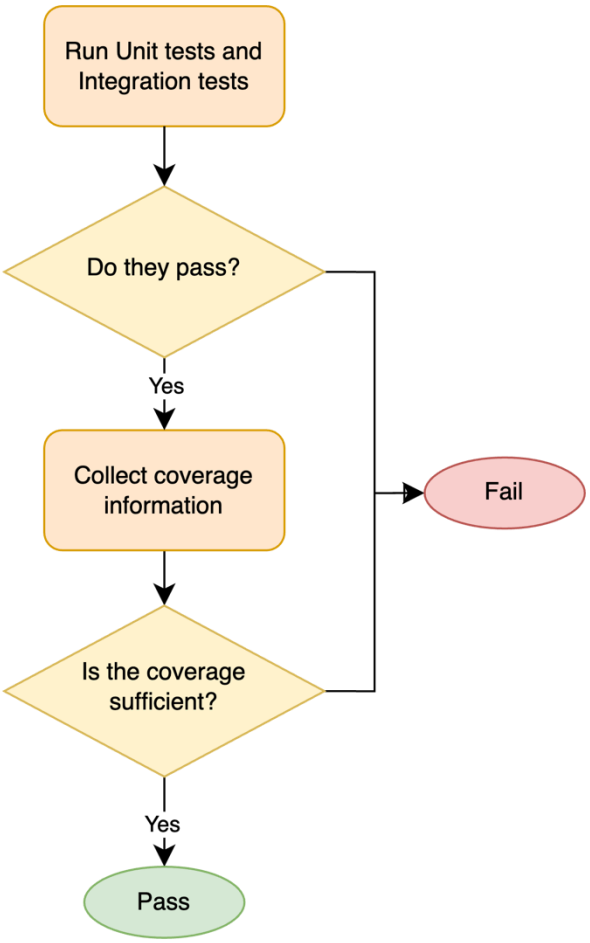


Figure 8 Code Coverage Analysis Flowchart

2.1.5.1 Code Coverage Model

Code complexity is a measure of the complexity of maintaining a codebase over an extended period. Associated metrics provide information on the difficulty of understanding the control flow of a program, in addition to the effort required to manage a repository. It also provides an estimate on the ease of introducing bugs and errors in code.

Reaching full code coverage can be time and resource-consuming and the code coverage measurement alone is not able to say how important is the code that is left uncovered.
Ideally, it would be more effective to write tests to cover the part of the codebase that would be more prone to hide defects. Distinct scenarios can happen, and, for each of them, different actions need to be taken. The table below summarizes the scenarios and the respective actions. We also have produced a series of rules to better comprehend what to do for each scenario:

- A **bold action** takes precedence over an *italic one*
- The **must** keyword makes an action **necessary** and **mandatory** in order to pass to the next workflow phase.

Code complexity	***Code Coverage***		
	High	***Normal***	***Low***
Low	**Maintain** this state	**Add more tests**	**Must add more tests**
Medium	**Simplify** complex code	**Simplify complex code** and *add more tests*	**Must add more tests** and *simplify complex code*
High	**Must simplify** complex code	**Must simplify** complex code and *add more tests*	**Must simplify** complex code and **must add more tests**

In order to better describe the definition of this model and how code coverage and code complexity are intertwined with one another, we need to present some notions.

A **code space** is any structure that incorporates a function, for example another *function,* or a *class*, a *struct*, a *namespace,* or more generally the *unit scope*. Therefore, source code must be divided into spaces to proceed.
Then it is necessary to set a threshold for code complexity. The minimum value for this metric is 1, which means that the code is **not** complex at all. Otherwise, as stated by an author of a certain type of code complexity metric[1], when the value of this metric is lower than or equal to 15, the code can be considered **not overly complex**. Any **higher** value leads instead to a hardly understandable code.
The last notion is the number of **physical lines (PLOC)**, thus the number of instructions and comment lines in a file.

[1] https://community.sonarsource.com/t/how-to-use-cognitive-complexity/1894/4

Now that all the notions have been defined, we present the structure of this new model as a series of steps that leads to the updated code coverage value, called **weighted code coverage**.

- Each line of a **code space** has a weight of 1 when covered by a test, otherwise its weight is 0. This weight is called **code coverage weight**.
- When the code complexity associated with a **code space** exceeds the threshold of 15, the relative **code complexity weight** is 2, otherwise it is 1.
- For each space, the **weighted code coverage** is obtained by multiplying the sum of its **code coverage weights** by its **code complexity weight.** If **weighted code coverage** is twice the original code coverage value, it should be discarded.
- The **global weighted code coverage** is obtained by dividing the sum of **accepted spaces weighted code coverage** by **the number of physical lines (PLOC)** in a source file**.**

2.1.5.2 Model Example

Let `foo` and `bar` be two functions. This yields two spaces of five lines each and is written in a simple pseudo-code with a code complexity value of 16 and 5, respectively. Therefore, **code complexity weights** are 2 and 1.

```
function foo() {
  instruction a
  instruction b
  instruction c
}

function bar() {
  instruction d
  instruction e
  instruction f
}
```

The number of covered lines is 5 for `foo` and 5 for `bar`.

The **weighted code coverage** value for the `foo` space is

```
5 * 2 = 10
```

which is doubled compared to the initial code coverage value and therefore it must be discarded.

The **weighted code coverage** value for the `bar` space is

```
5 * 1 = 5
```

which remains unaltered compared to the initial code coverage value.

The global **weighted code coverage** is then equals to:

```
5 / 10 = 0.5
```

where the numerator corresponds to the `bar` **weighted code coverage** while the denominator is the **PLOC** metric. In this specific case, only 50% of the source code lines have been weight-covered.

2.1.6 Dynamic Fault Analysis

Dynamic fault analysis is the last mandatory phase of our workflow whose goal is to look for vulnerabilities and security issues while software is executing. These faults can be detected at runtime with the usage of some tools which can define an instrumented build in order to extract data and information from tests and binaries. The tools in this class tend to not yield many false positives, and if they do, those are usually caused by either a miscompilation or limitations in their CPU/memory models. They also allow to find faults caused by unexpected interactions with external APIs, which are undetectable through static tools. Dynamic fault tools tend to execute between two times and tenfold times slower than a normal build time.

If any fault is detected during this phase, the workflow **immediately** stop its execution.

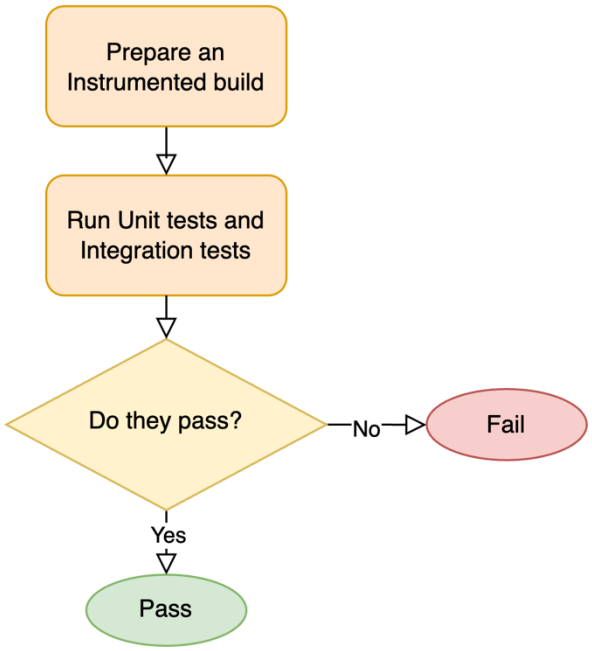
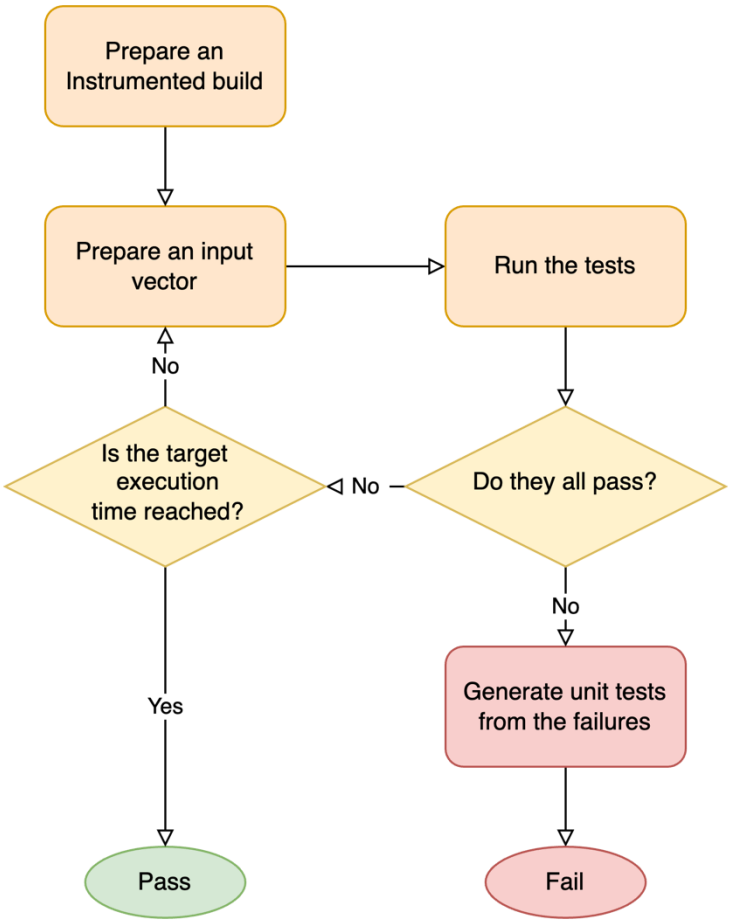


Figure 9 Dynamic Fault Analysis Flowchart

An additional check that we added to our workflow with the goal of detecting faults and expanding code coverage is *fuzzing*. That is, fuzzing tools construct random input vectors and pass them to tests and final binaries to verify the presence of crashes or potential memory leaks. If some tests fail, new unit tests are generated from the failure. However, if some tests are failing, but the target execution time has been reached, then the fuzzing phase can be considered passed. Since its execution time might be long,

it is highly recommended to not run it on pull requests if not deemed truly indispensable.



2.1.7 Figure 10 Fuzzing Analysis FlowchartPackaging and deployment

Software can be packaged and then distributed through software repositories. This process is usually cumbersome and prone to errors, so we have added to the workflow a specific phase dedicated to the creation of packages in an automatic way. Before proceeding with packaging, all the previous workflow phases **must** be completed with success. Then, for each supported architecture, libraries and executables should be produced considering the following aspects:

- The final binaries must be built with the best available optimization profile, in order to attain a software that runs as efficiently as possible and take up the less amount of memory as possible
- Depending on the architecture and programming language, having their debug symbols stripped to further reduce the final binary size
- Collect all their dependencies and either insert them in the final package, or define a procedure to install each dependency on the target machine
- Choose the most suitable archive file format for each architecture, in order to simplify the software deployment on hosting services such as, to name a few, GitHub or GitLab

Once a binary has been packaged, the conclusive phase is deployment, which can happen either on hosting services through the creation of a release or with the publication on software repositories. Both

actions can also be automatized by using scripts and appropriate tools.

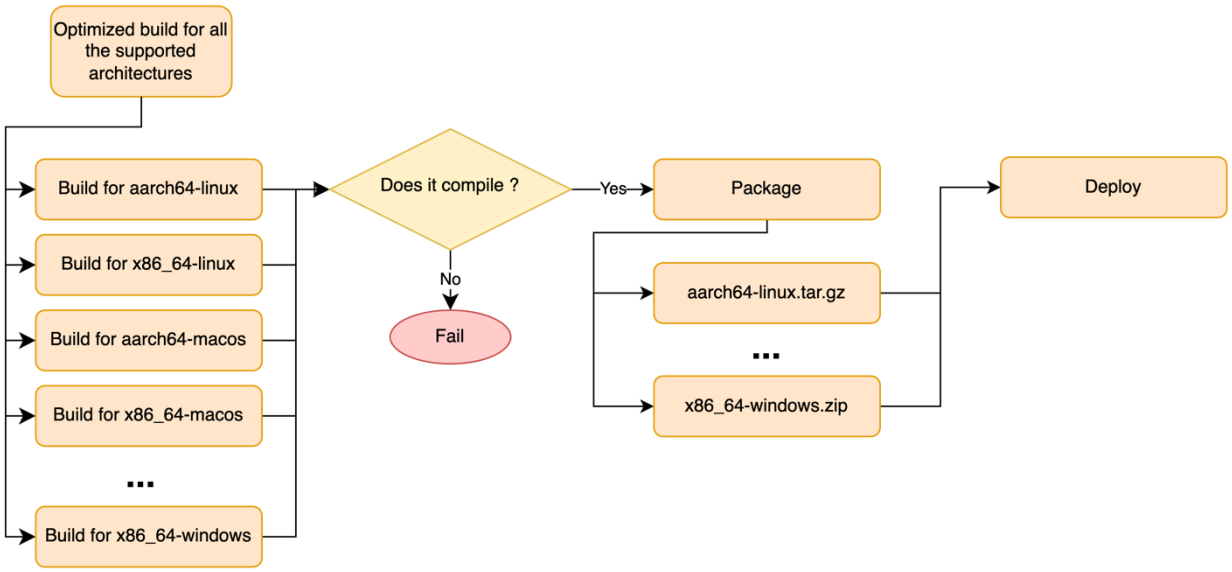


Figure 11 Packaging and Deployment Flowchart

2.2 *GitHub Actions Workflow*

After designing our workflow, we have mapped its structure into scripts, with the goal of performing its phases on SIFIS-Home's repositories. Also, since our codebases have been hosted on GitHub, we have adopted **GitHub Actions** as continuous integration system, better known as CI, to run these scripts.

In order to better separate the check phases from the deployment phase, we have defined two distinct scripts:

- The first script performs a series of quality and security checks related to code and binary analysis
- The second script deploys binaries on different architectures and, for some languages, even on package registries

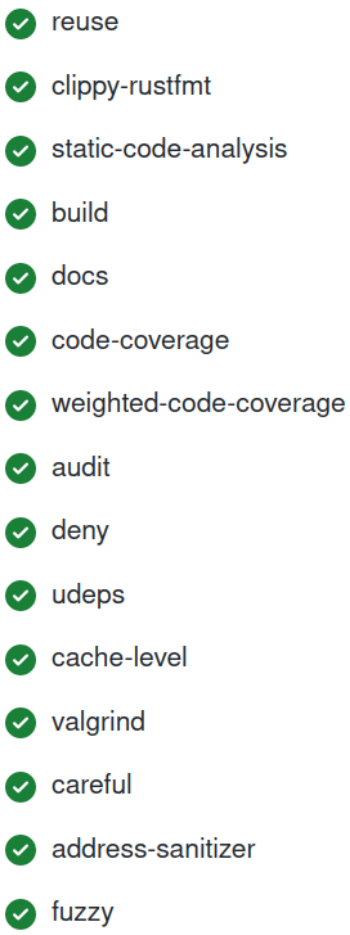


Figure 12 List of checks performed by the first GitHub Actions script

In the following, we explain the GitHub's continuous integration system structure and how our workflow has been physically implemented. We have chosen Rust as language for the workflow implementation since it already, intrinsically *covers* all phases, from static to dynamic fault analysis.

2.2.1 GitHub Actions

GitHub Actions is the continuous integration system developed by GitHub. It allows to automate build, test, and run deployment pipeline. It offers the possibility to create workflows that build and test the code of a repository or deploy a specific branch, usually a tagged one, to production.

It provides virtual machines for Linux, MacOS and Windows, but it is even possible to set up a self-hosted machine on personal data center or cloud infrastructure to run workflows.

A GitHub Actions *workflow* can be triggered when an *event* occurs in a repository, such as a pull request being opened, a commit being pushed on a branch and on a pull request, or an issue being created. A workflow is a configurable automated process, defined by a YAML file, which can run one or more jobs, either in sequential order or in parallel.

A *job* is a set of *steps* in a workflow that is executed on the same virtual machine, or more simply, on the same architecture. Each step is either a shell script or an *action*, which is a custom application

performing a complex but frequently repeated task. Steps are executed in order and are dependent on each other, thus data can be shared from one step to another.

GitHub Actions also offers a packaging step with the goal of producing a runnable or deployable *artifact*. The package can then be downloaded locally for manual testing, made available to users for download, published on a package registry, or deployed to a production environment.

Downloading a tool or setting up a specific configuration can be quite time consuming when performed many times for each event. In order to overcome this, GitHub Actions provides a *cache* action that saves data contained in one or more directories at the first execution and then restores all this information on subsequent continuous integration runs. This approach speeds up job computation and reduces network usage as much as possible.

There are cases where a particular event does not require the workflow execution, for example a commit that modifies only a textual file or the addition of a new image to the repository. Consistently, GitHub Actions allows a specific keyword, *paths*, to specify the directories or files that, if they are modified in some way, trigger a new workflow run.

2.2.2 Workflow composition

We have divided our workflow phases into distinct GitHub Actions jobs and established for them an order of execution introducing the concept of layer. A layer is a group of jobs that run in parallel on the same, or on a different, virtual machine, but they all must finish their execution before the next scheduled layer. This schema splits jobs depending on the context and allows to optimize the entire computation, deciding how many jobs can run in parallel at the same time. Thus, establishing beforehand the size of a layer helps to reduce the general execution time and set up the number of parallel jobs that GitHub Actions allows to run concurrently. For example, having more layers represents a uniform distribution of jobs and this might be caused by lack of resources present in virtual machines, or the impossibility of running more jobs concurrently. Both these cases lead to a higher execution time. For the Rust language workflow, seven layers have been defined:

1. Legal and format layer
2. Build and documentation layer
3. Code coverage layer
4. Dependency layer
5. Cache layer
6. Unsafe checks layer
7. Fuzzing layer

This workflow structure is not immutable since layers can be removed or split up into more layers when tools contained in jobs are computationally intensive. This could also lead to define a layer not coherent with a context but created to reduce the general workload.
Running the workflow only when some files and directories are changed allows a developer to save lots of hardware and software resources. We have used the paths keyword offered by GitHub Actions to specify which files and directories trigger the workflow:

```
    paths:
      - 'src/**'
      - 'crates/**'
      - 'fuzz/**'
      - '.github/**'
      - 'Cargo.toml'
      - 'Cargo.lock'
```

In the following, we describe each layer and its tools in detail. We also present the reasons behind its definition and composition. We have omitted how tools have been installed on machines, but usually their binaries are downloaded from their release pages and inserted into dedicated directories on those machines, to be immediately retrieved by the operating system when tools are called.

2.2.2.1 Legal and format layer

The legal and format layer verifies whether each file contained in a repository, both text and binary ones, is compliant with a specific license and whether the repository code respects the format standards of the Rust language.
To perform the legal check, we have adopted the *REUSE* specification, which provides a set of recommendations to make licensing projects easier. REUSE implemented a tool that analyzes each file in a repository to verify whether a license is contained inside that file or written in an external file with the same filename. It also provides a GitHub Actions action to perform this task.

This check runs on all three supported virtual machines.

```
    - name: REUSE Compliance Check
      uses: fsfe/reuse-action@v1
```

Format standards in Rust can be verified through two tools, *rustfmt* and *clippy*. The first one analyzes whether the code has been formatted according to style guidelines, while the second one determines whether the code can be improved or whether there are common mistakes in it through lints. Lints can highlight different code issues and they are divided into categories: code that is wrong or useless, code that does something simple but in a complex way, code that can be written to run faster and so on.
Even these checks run on all three supported virtual machines.

```
    - name: Run rustfmt
      run:
        cargo fmt --all -- --check --verbose

    - name: Run cargo clippy
      uses: giraffate/clippy-action@v1
      with:
github_token: ${{ secrets.GITHUB_TOKEN }}
clippy_flags: --all-targets -- -D warnings
```

The `github_token` allows to present lints as a series of messages in a dedicated job specifically created by the action through GitHub APIs. The `secrets.GITHUB_TOKEN` is the private key used to access repository data through these APIs.

The last job in this layer computes static analysis metrics. A Rust Mozilla-tool called *rust-code-analysis* first constructs an Abstract Syntax Tree starting from source codes and then, through its parsing, it computes a series of metrics, described in D2.1, to collect information that a developer can exploit to improve the code. Metrics are saved for each source code file as a json file and uploaded as artifacts on GitHub Actions to be later downloaded. We will explain our contributions to this tool and to its third-party crates in D2.5.
This check is run only on Linux and Windows virtual machines because rust-code-analysis has not been released on MacOS.

```
    - name: Run rust-code-analysis
      run: |
mkdir $HOME/rca-json
        rust-code-analysis-cli --metrics -O json --pr -o "$HOME/rca-json" -psrc

    - name: Upload rust-code-analysis json
      uses: actions/upload-artifact@v3
      with:
        name: rca-json-data
        path: ~/rca-json
```

The `metrics` option sets the tool to metrics computation, `pr` option outputs a formatted and readable json, while `p` sets input source codes directory.

2.2.2.2 Build and documentation layer

The jobs of this layer start their execution after all legal and format layer jobs are finished. The first job builds code and stops the workflow when build errors are found. The second job instead produces code documentation consultable on *docs.rs*, the documentation host for the Rust language. It does not upload the generated documentation on the host, but just executes the generator process to verify whether wrong-formatted documentation is present in the code.
These checks run on all the three supported virtual machines.

```
    - name: Build
      run: cargo build –verbose

    - name: Generate docs
run: cargo doc --verbose --no-deps
```

The `no-deps` option does not generate any documentation for project's third-party crates.

2.2.2.3 Code coverage layer

This layer's main task consists in running tests and computing the two kinds of code coverage. The first job extracts code coverage information running instrumented tests and collecting their outputs in an aggregated report. The overall execution time is much higher than a non-instrumented test suite execution. Once code coverage data have been retrieved for each source code file, and saved as `.profraw` files, another Mozilla tool, called grcov, collects and aggregates this code coverage information for multiple source files into a single file, *lcov.info*, that will be uploaded to a third-party website in order to ease its processing and analysis over time. This website is called *codecov* and we have adopted a specific action to perform the upload of the *lcov.info* file.

```
    - name: Run tests
```

```
      env:
        RUSTFLAGS: "-Cinstrument-coverage"
        LLVM_PROFILE_FILE: "name-%p-%m.profraw"
      run: |
        cargo test –verbose

    - name: Get coverage data for codecov
      run: |
grcov . --binary-path ./target/debug/ -s . -t lcov --branch \
             --ignore-not-existing --ignore "/*" --ignore "../*" -o lcov.info

    - name: Codecov upload
      uses: codecov/codecov-action@v3
      with:
        files: lcov.info
```

All *ignore* options sets grcov to ignore files contained in the specified directories.
The next step is the traffic light system implementation. We have used shell language to develop the system directly inside the continuous integration. Before that though, we had run grcov to aggregate code coverage data and used covdir as output format. We have chosen this format because it contains the total code coverage in a json field.

```
    - name: Get total coverage
      run: |
grcov . --binary-path ./target/debug/ -s . -t covdir --branch \
             --token YOUR_COVDIR_TOKEN --ignore-not-existing --ignore "/*" \
             --ignore "../*" -o covdir.json

    - name: Evaluate code coverage value
      run: |
        # Retrieve code coverage associated to the repository
        FLOAT_COVERAGE=$(jq'.coveragePercent' covdir.json)
        # Round the float value to the nearest value
        COVERAGE_OUTPUT=$(printf "%.0f" $FLOAT_COVERAGE)
        # If code coverage >= 80, green traffic light
        if [ $COVERAGE_OUTPUT -ge80 ]
        then
            echo "$COVERAGE_OUTPUT > 80 --> Green"
        # If code coverage is >=60 but < 80, orange traffic light
elif [ $COVERAGE_OUTPUT -ge60 ]
        then
            echo "60 <= $COVERAGE_OUTPUT < 80 --> Orange"
        # Otherwise, red traffic light
        else
            echo "$COVERAGE_OUTPUT < 60 --> Red"
            exit 1
        fi
```

The weighted code coverage model has been computed in another job. grcov aggregates code coverage data using coveralls as output format. We have chosen this format because it is much faster to be parsed by the weighted code coverage tool that we have developed, namely weighted-code-coverage. This tool requires as input a coveralls json file containing all code coverage data and code complexity metric. It produces as output a json file containing all weighted code coverage data that will then be uploaded as

artifacts through GitHub Actions.
Code coverage jobs run on all the three supported virtual machines.

```
    - name: Run grcov
      run: |
grcov . --binary-path ./target/debug/ -t coveralls -s . --token
YOUR_COVERALLS_TOKEN -o coveralls.json

    - name: Run weighted-code-coverage
      run: |
mkdir $HOME/wcc-output
        weighted-code-coverage -p src/ -j coveralls.json -c cyclomatic --json
$HOME/wcc-output/out.json

    - name: Upload weighted-code-coverage data
      uses: actions/upload-artifact@v3
      with:
        name: weighted-code-coverage-ubuntu
        path: ~/wcc-output/out.json
```

weighted-code-coverage supports more options, such as other weighted code coverage algorithms, but we will outline its structure and input arguments in D2.5. The `p` option defines the directory used as input.

2.2.2.4 *Dependency layer*

In a project, even dependencies crates could be affected by security issues and vulnerabilities. The first job of this layer runs a tool called *audit* that scans every third-party crate of a project and verifies, through a query to a database, whether a specific crate version is deemed dangerous or unsecure. As soon as a security problem is found, the workflow immediately stops its execution. We perform these checks using an action.

```
    - name: Run cargo-audit
      uses: actions-rs/audit-check@v1
      with:
        token: ${{ secrets.GITHUB_TOKEN }}
```

The `token` option allows the action to access the repository and open a new issue for each vulnerability found during the analysis. The `secrets.GITHUB_TOKEN` is the private key used to access a repository in write mode.

The second job runs the *deny* tool to ensure that all project dependencies conform to some requirements. It verifies that there are no duplicates among the dependencies through the *bans* command. A duplicated dependency could lead to some security problems if their versions differ, and they could increase the compilation time. It is possible to customize which dependencies need to be considered using a configuration file. However, since the deny configuration is complicated, this tool provides the *init* command to create a configuration file from a template as a starting point.

```
    - name: Run cargo-deny
```

```
      run: |
        cargo deny init
        cargo deny check bans
```

The last job of this layer runs *udeps*, a tool to find unused dependencies. It also allows to define a configuration file to ignore some dependencies from the analysis process.

```
    - name: Run cargo-udeps
      run: |
        cargo +nightly udeps --all-targets
```

All jobs run on all three supported virtual machines. We have also considered an optimization aspect for this layer, indeed, if no dependencies are added, removed, or modified by a commit, jobs are not run. To do so, we have used the following action

```
    - name: Check dependencies changes
      uses: dorny/paths-filter@v2
      id: changes
      with:
        filters: |
          cargo:
            - 'Cargo.toml'
            - 'Cargo.lock'
```

These two files, *Cargo.toml* and *Cargo.lock*, contain all dependencies used by a crate. The first file describes project dependencies in a broad sense, and is written by a developer, while the second file contains the exact information about dependencies, it is maintained by Cargo and should not be manually edited.

According to our workflow, this layer should be merged with the legal and format layer, since all these checks are also part of the static analysis, but we have preferred to insert that at this point in order to privilege code analysis over dependencies, hence taking a context-based decision. However, this behavior can be modified in the future without any problems, by moving this layer above in order.

2.2.2.5 Cache layer

This is an optimization layer used to install an unsafe-checker tool that will be used in the next layer: *cargo-careful*. Since this tool does not have a binary release, it takes a while to be installed because it needs to be compiled on virtual machines, thus the main task for this layer consists of building and installing cargo-careful and caching its binary using GitHub Actions for successive continuous integration executions. Jobs in the next layer must restore the cache to have the binary available for their computations. This job runs on all three supported virtual machines.

```
    - name: Cache produced data
      id: cache-data
      uses: actions/cache@v3
      with:
path: |
          ~/.cargo/bin/
          ~/.cargo/registry/index/
```

```
          ~/.cargo/registry/cache/
          ~/.cargo/git/db/
          target/
        key: ${{ runner.os }}-cargo-ci-${{ hashFiles('**/Cargo.toml') }}

- name: Install cargo-careful
      if: steps.cache-data.outputs.cache-hit != 'true'
      run: |
        cargo install cargo-careful
```

The `path` option sets all directories to be cached, while the `if` construct runs the step only when there is a cache miss. The `key` option represents the unique key associated to the cache. When the key value changes, the cache is updated, and all subsequent steps are automatic cache miss. In this case, the key modifies its value when there are differences in the Cargo.toml file.

2.2.2.6 Unsafe checks layer

A Rust project can have unsafe code in its source files. This kind of code does not guarantee memory safety enforced at compile time. Since underlying computer hardware is inherently unsafe, Rust allows to do low-level systems programming, such as directly interacting with the operating system or even writing an operating system. Working with low-level systems programming is one of the goals of the language. Another use for unsafe code is related to the speed of a program, using unsafe instructions and functions makes a program much faster because some runtime checks are completely removed. For these reasons, we have created this layer that can be deleted from the workflow when only safe code is being used.

The first job makes use of *cargo valgrind*, a cargo subcommand that invokes *valgrindmemcheck* under the hood. Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect memory management and threading bugs, and profile programs in detail. *Valgrindmemcheck* is one of these tools, more specifically a memory error detector. It can detect the following problems:

- Accessing memory after it has been freed
- Using undefined values, such as values that have not been initialised, or that have been derived from other undefined values
- Incorrect freeing of heap memory, such as double-freeing memory blocks
- Passing a fishy, presumably negative, value to the size parameter of a memory allocation function
- Memory leaks

Problems like these can be difficult to find, often remaining undetected for extended periods, then causing occasional, difficult-to-diagnose crashes.
This job can run only on Linux virtual machines, due to Valgrind restrictions for other operating systems. The tool can be set up to scan binaries or tests in search of memory errors. According to the number of tests and their dimensions, this check could take a great amount of time.

```
    - name: Run cargo-valgrind
```

```
      run: |
        cargo valgrind run -- --command
        # cargo valgrind test
```

The second job runs the *cargo careful* tool installed in the previous layer and then cached. This tool detects certain kinds of undefined behaviours and performs sanity checks while executing the software. It works on all code, is suitable for a continuous integration system due to its speed and can run both on tests and binaries. This job can run on all three supported operating systems.

```
- name: Restore cache data
      id: cache-data
uses: actions/cache@v3
      with:
path: |
          ~/.cargo/bin/
          ~/.cargo/registry/index/
          ~/.cargo/registry/cache/
          ~/.cargo/git/db/
          target/
        key: ${{ runner.os }}-cargo-ci-${{ hashFiles('**/Cargo.toml') }}

- name: Run cargo-careful
      run: |
        cargo +nightly careful test
        # cargo +nightly careful run
```

The first action restores the cache data, thus it provides cargo-careful binary, while the second step launches the command on tests.

The last job of this layer instruments tests with *AddressSanitizer*, a fast memory error detector developed by Clang. It can detect the following types of bugs:

- Out of bound accesses to heap, stack and globals
- Use after free
- Use after return
- Use after scope
- Double-free, invalid free
- Memory leaks

It is a tool contained in the Rust suite and can run only on Linux and MacOS virtual machines. According to the number of tests and their dimensions, this check could take a great amount of time.

```
    - name: Run AddressSanitizer
      env:
        RUSTFLAGS: -Zsanitizer=address -Copt-level=3
        RUSTDOCFLAGS: -Zsanitizer=address
      run: cargo test -Zbuild-std --target x86_64-apple-darwin
```

The `opt-level` option value means that all optimizations are enabled in the compiler, while `target` defines the architecture in use, in this case MacOS Darwin operating system. This memory error detector is also run on documentation tests.

2.2.2.7 Fuzzing layer

The last layer job is dedicated to fuzzing dynamic analysis. To perform this check, *cargo fuzz* has been used. This tool is a cargo subcommand that runs *libFuzzer* under the hood, a library that feeds fuzzed inputs to a software via a specific fuzzing entry point, then it tracks which areas of the code are reached and generates mutations on the corpus of input data to maximize the code coverage. Fuzz tests need to be defined in the *fuzz* directory to enable the library features. A fuzz test example is shown below.

```
#![no_main]

use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    // fuzzed code goes here
});
```

Once one or more fuzz tests have been defined, the `cargo fuzz build` command runs the fuzzing process.

```
    - name: Run cargo-fuzz
      run: cargo fuzz build
```

This job can run only on Linux and MacOS virtual machines due to limitations in libFuzzer library.

2.2.3 Workflow deployment

This last section describes the workflow part used to deploy binaries on different architectures. We have implemented the workflow deployment in another YAML file which starts its execution when a new tag has been added to a repository. We have considered tags in the form v*.*.* , thus a semantic versioning tag, but any other form is acceptable.
We have defined four jobs: the first three jobs produce binaries for different architectures and upload them as GitHub Artifacts, while the last one collects all these artifacts and publishes a new GitHub release with the chosen tag. The fourth job starts only when the other three have finished their execution. Each binary has been built using the release profile that enables all Rust time and memory optimizations. The first job builds the Windows binary, packages it into a .zip file using 7z, a Windows file archiver tool, and uploads this zip file as artifact.

```
    - name: Build binary
      run: cargo build –release

    - name: Create package
      run: |
        7z a binary.zip ./target/release/binary.exe

    - name: Upload artifacts
```

```
      uses: actions/upload-artifact@v2
      with:
        name: binary.zip
        path: binary.zip
```

The second job builds the Linux binary. It does the same operations as the Windows job, but it encapsulates the binary into a *.tar.gz* format, more known and used for Linux distributions, and strips debug symbols to reduce the final file size. This binary has been compiled using the *musl* library, an implementation of standard C library functionality described in the ISO C and POSIX standards.

```
    - name: Strip binary
      run: |
        strip target/release/binary

    - name: Create package
      run: |
        tar –czvfbinary.tar.gz ./target/release/binary
```

The third job builds the MacOS binary. It does the same operations as for the Linux job but executed on the Darwin operating system.

The fourth and last job downloads all the artifacts produced by the previous jobs and creates a new tagged release.

```
      - name: Download artifacts
        uses: actions/download-artifact@v2
        with:
          path: ./binaries

      - name: Create a release
        uses: softprops/action-gh-release@v1
        with:
          name: v${{ steps.tagName.outputs.tag }}
          files: |
./binaries/**/*.zip
./binaries/**/*.tar.gz
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

The `path` option sets the directory where binaries are downloaded, `name` sets the release name, `files` option selects which files are added to a release, while `GITHUB_TOKEN` sets the private key to access the functionalities necessary for publishing a new release.

2.3 *Workflow example*

The GitHub Actions workflow has been tested and implemented into a WP5 and WP6 Rust component called **libp2p-rust-dht**. This component offers a completely distributed publish/subscribe mechanism through which SIFIS-HOME applications can exchange messages.

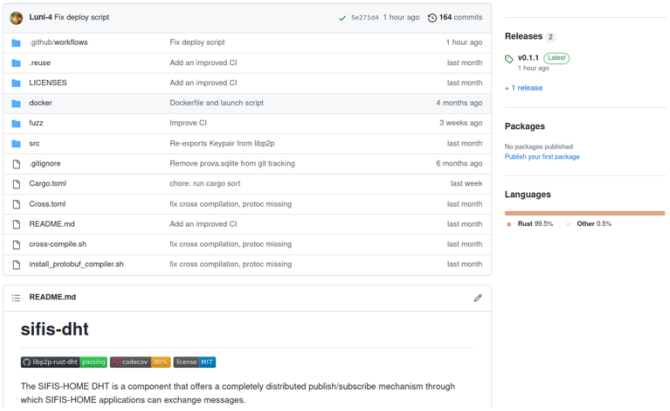


Figure 13libp2p-rust-dht GitHub repository

At first, we added our entire workflow, which considers all the three supported virtual machines, but we immediately faced a problem with the *protobuf-compiler* package on Windows and MacOS. This package was only available for Linux distributions, making the jobs in the other two architectures fail. However, since this component has been intended to run on Linux machines, we have fixed the issue by just deleting Windows and MacOS jobs from the workflow.

After this initial trial-and-error process which allowed to identify the error, the workflow ran smoothly, as shown in the image below:

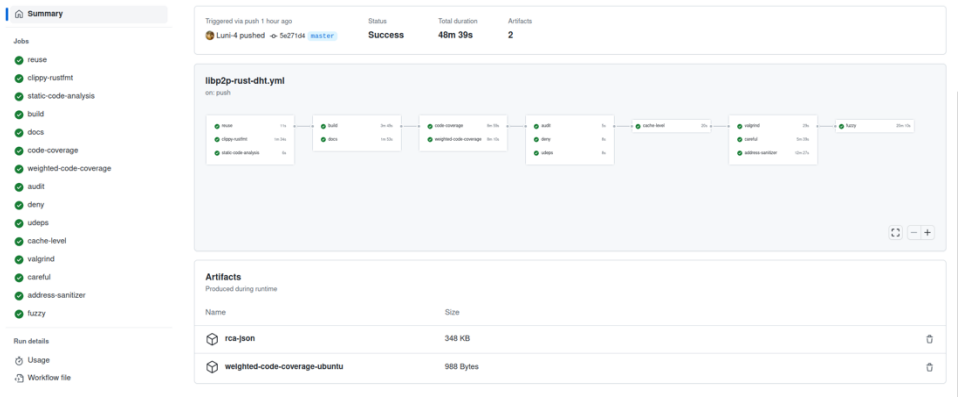


Figure 14A complete workflow with its jobs on the left and artifacts on the bottom

The *README.md* file, as visible in the repository image, contains two badges. The first badge is red-colored when the workflow has failed some jobs, or otherwise green-colored when it has successfully completed its execution. Instead, the second badge shows the repository code coverage percentage, and, when clicked, it redirects to the *codecov* website where one can browse code coverage data associated with each file.

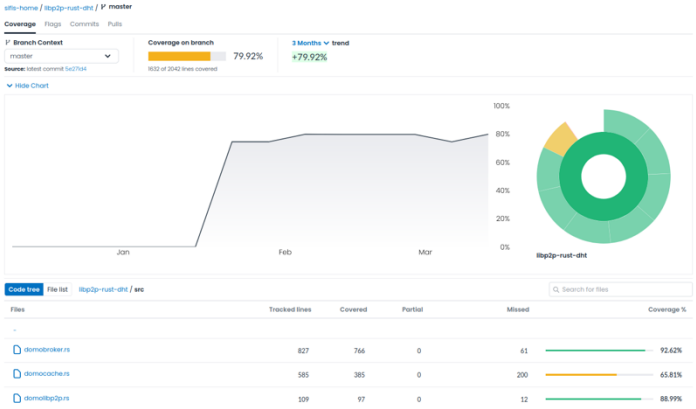


Figure 15Code coverage data shown by codecov

At some point, a new release has been published for this component, thus triggering the workflow deployment

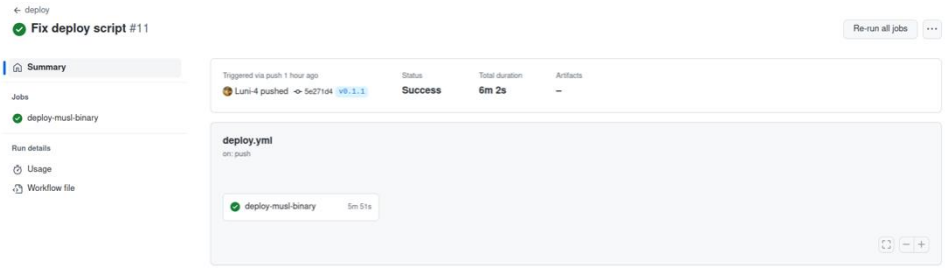


Figure 16A complete deployment

Once the deployment job has been completed, a new tagged release is automatically created in the GitHub release page with the associated, relative binary.

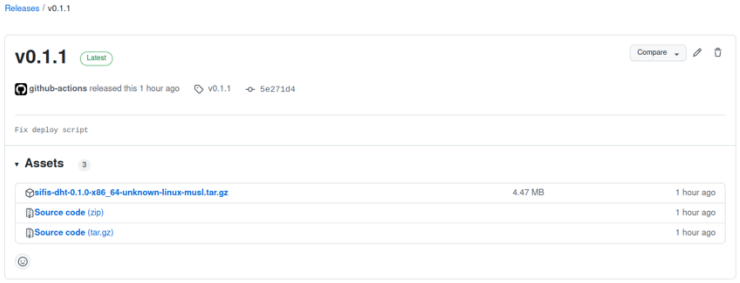


Figure 17A new libp2p-rust-dht tagged-release created automatically by the workflow deployment

2.4 *Additional Notes*

Some SIFIS-Home components have adopted a conventional commits specification during their development, with the intent of adding human and machine-readable meaning to commit messages. As a future work, it would be helpful to add a static analysis check to evaluate the correctness of commits based on the commits-convention established by the maintainer of a component. That would guarantee uniformity among commits in addition to a major simplicity in understanding code changes introduced in a project.

3 Security Label

A study conducted by researchers at the Carnegie Mellon University (Emami-Naeini et al., 2020) pointed out that security and privacy practices adopted by smart devices companies are rarely available to consumers before purchase. Following on from this, they proposed the idea of providing IoT devices with an explanatory label, which conveys information about security mechanisms, data practices, and other details about the device, e.g., manufacturer country, device compatibility, etc. According to their standard, the label can be printed on the device package or retrieved online by means of a QR code, in order to inform the customer about security features and data management. Transparency is good for business and is important as much for customers as it is for vendors. It is up to the vendors whether to adopt this new standard or not, but giving more information about a product can certainly boost a brand's reputation.

In the same spirit of this mechanism, but at a different level, we intend to provide SIFIS-Home-aware applications with labels describing security risks deriving from app's code execution. To this aim, we start at a fine grain by assigning labels to the APIs that developers use when writing their applications. Eventually, a label will be provided for the application, and it will be composed of all the labels of the APIs within the application codebase.

Our proposal is analogous to the permissions mechanism of Android operating system (Android Permissions, 2023), and our set of labels could resemble their Manifest file, but it differs in some crucial aspects. In SIFIS-Home, the user (or a maintainer on the user's behalf) defines policies to be enforced within their smart home. Policies definition is a simple and intuitive process which allows the user to declare which actions and operations can be performed by applications in their smart home environment and which cannot.

Our label is shown to the user during an app installation with an informative purpose, and it highlights if some of the risks go against the user-defined policies. If no risk contradicts the user's policies, the app is installed straightaway. Otherwise, the user can decide whether to edit their policies, and whether to finalize the app installation or not. On the contrary, the Android permissions mechanism just informs the user about the permissions that the app requires in order to run all its features. Permissions contained in the manifest file are just shown to the user, who is oftentimes not aware of what some entries pertain to. Lately, Android introduced runtime allowance for dangerous permissions, meaning that the user is asked to give permission within the app when first using a feature that requires such a permission. However, this approach is not convenient in our scenario because it requires user interaction with the app, which might be unfeasible in some cases.

3.1 *The SIFIS-Home Developer APIs*

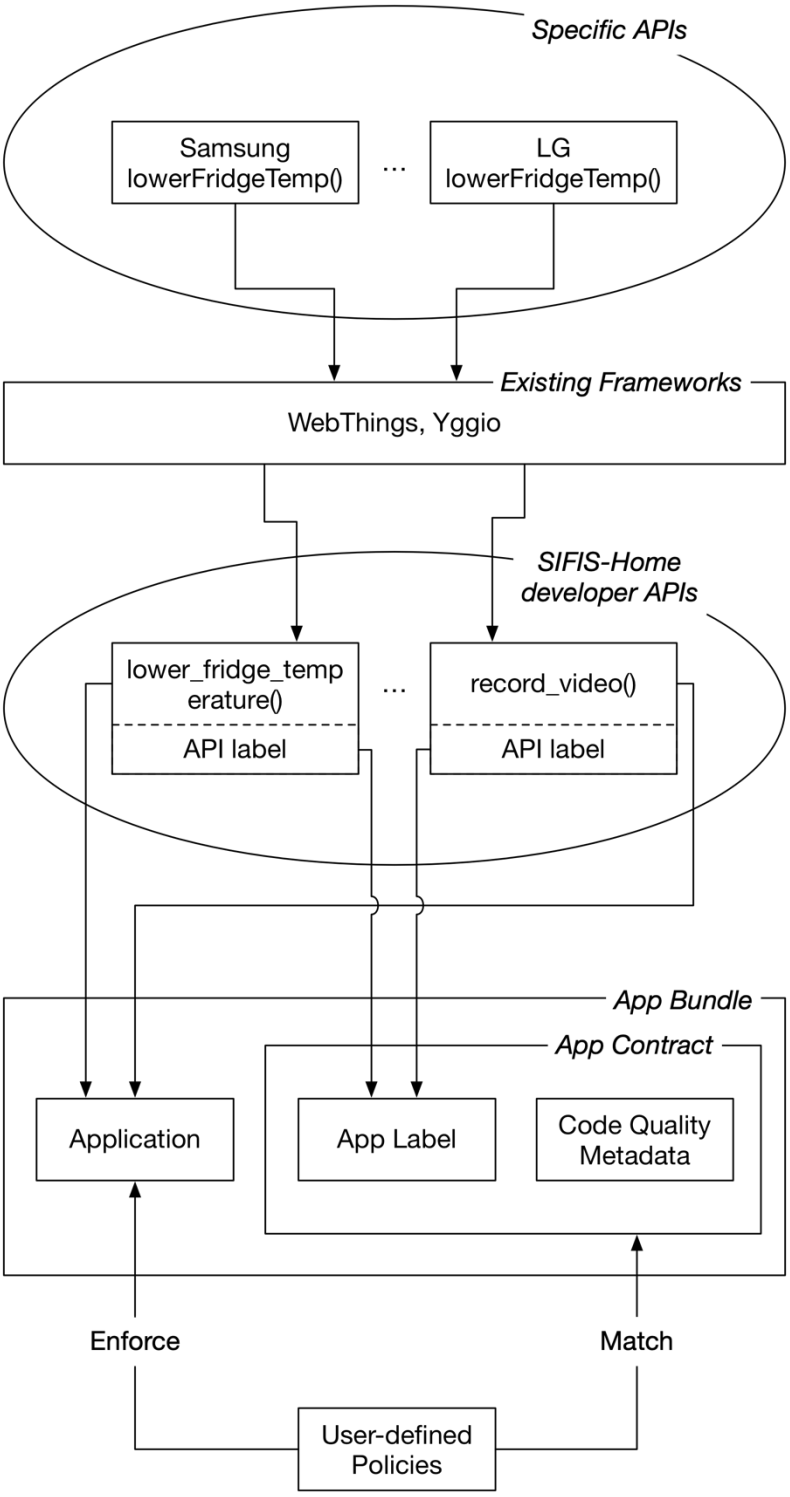


Figure 18 Architecture

As shown in Figure 18, the SIFIS-Home developer APIs are designed to extend and improve service level APIs such as those offered by Webthings or Yggio. The SIFIS-Home developers APIs build upon

this already existing model, which is used to abstract from the specific producer-based implementation of functionalities used to provide generic services, such as “Switch on Light,” “Open Lock,” “Increase Temperature,” etc. Following the Webthings terminology, we can name these services “Capabilities.” The Capabilities help developers of third-party applications to provide applications that can invoke these generic services, without having to be worried about the actual device-specific implementation. To clarify, let us suppose, for example, that two device manufacturers provide, for their device “Refrigerator,” two different API implementations for decreasing the current temperature in the refrigerator by 1 °C. In order to offer this API to third-party developers without having to foresee two distinct invocations (one for Manufacturer 1 and one for Manufacturer 2), the manufacturers describe the API as a capability “lowerFridgeTemp()”, exposed by SIFIS-Home. Thus, a developer can simply invoke the SIFIS-Home developer API call, with no need to determine which device they are talking to and invoke the device-specific implementation of it.

In order to define the SIFIS-Home developer APIs, we build upon the currently existing frameworks, focusing on Web of Things and FIWARE. In fact, SIFIS-Home is focused on the security and safety aspects of the smart home management. For this reason, developing a new standard is out of the scope of our activities. Still, both standards are still new and only few, basic capabilities have been defined. We are thus drawing from other non-standardized framework such as IFTTT, Home Assistant, and OpenHab to define some additional capabilities that are useful for representing desired features and functionalities, while still bringing potential security and safety issues. As it currently happens for Web of Things and FIWARE, new capabilities can be proposed by device producers and application developers, for representing functionalities that can be offered to third-party applications.

In order to be able to handle the privacy, safety, and security issues within the activities of Task T2.3, we have defined a set of “hazards” representing safety, integrity, security, and privacy issues intrinsically related to the execution of each specific developer API. Such risks are related to either a misuse or a malicious use of the functionality, e.g., excessively decreasing the refrigerator temperature to cause a greater energy consumption. The user must be informed of this possibility when installing an application on their SIFIS-Home devices, with the possibility of controlling the execution of risky operations, by means of security and safety policies. We recall from D1.2 that such policies can be defined either by the user themselves or by an external, expert maintainer. By binding the labels to specific APIs (*API label*, shown in Figure 18), we ensure that, if an API is invoked, the corresponding label is associated with the application, similarly to what happens with Android permissions. The application will thus carry an application label (*app label*) made by the combination of the API labels associated with the SIFIS-Home developer APIs included in the application source code. The app label, together with the code quality information provided by Task 2.1 and Task 2.2, together with other metadata, build the App Contract.

The App Contract is a structured document, which is both human readable and machine interpretable (being based on a markup language), which is bound to the application code by means of a digital signature. The contract provides information on the application quality, on the identity and reputation of the developer, on the resources that can be controlled by the application and on the correlated risk, which might stem from misuses of such resources. This document provides useful information to the user, for deciding whether to install or not the application. At the same time, the contract is analyzed by the SIFIS-Home framework, which, according to the enforced policies, will handle the privacy, security, and safety risks by limiting the application functionalities, and/or warning the user or maintainer about inconsistencies with the user decision of enabling the application functionalities, and about identified misbehavior.

In the following, we report the process of defining API labels for the SIFIS-Home developer APIs, also by discussing some proposed capabilities. We assume that for new proposed capabilities, the assignment of one or more API labels will be performed by a SIFIS-Home consortium. The certification system would behave similarly to what is in place regarding the CE conformance marking (Marking, 2023): depending on the API in use, a self-assessment may be sufficient to enter the SIFIS-marketplaces, while a dangerous API would require an independent party to confirm the safety of the API in use and that the software behind the API surface conforms to an adequate development standard.

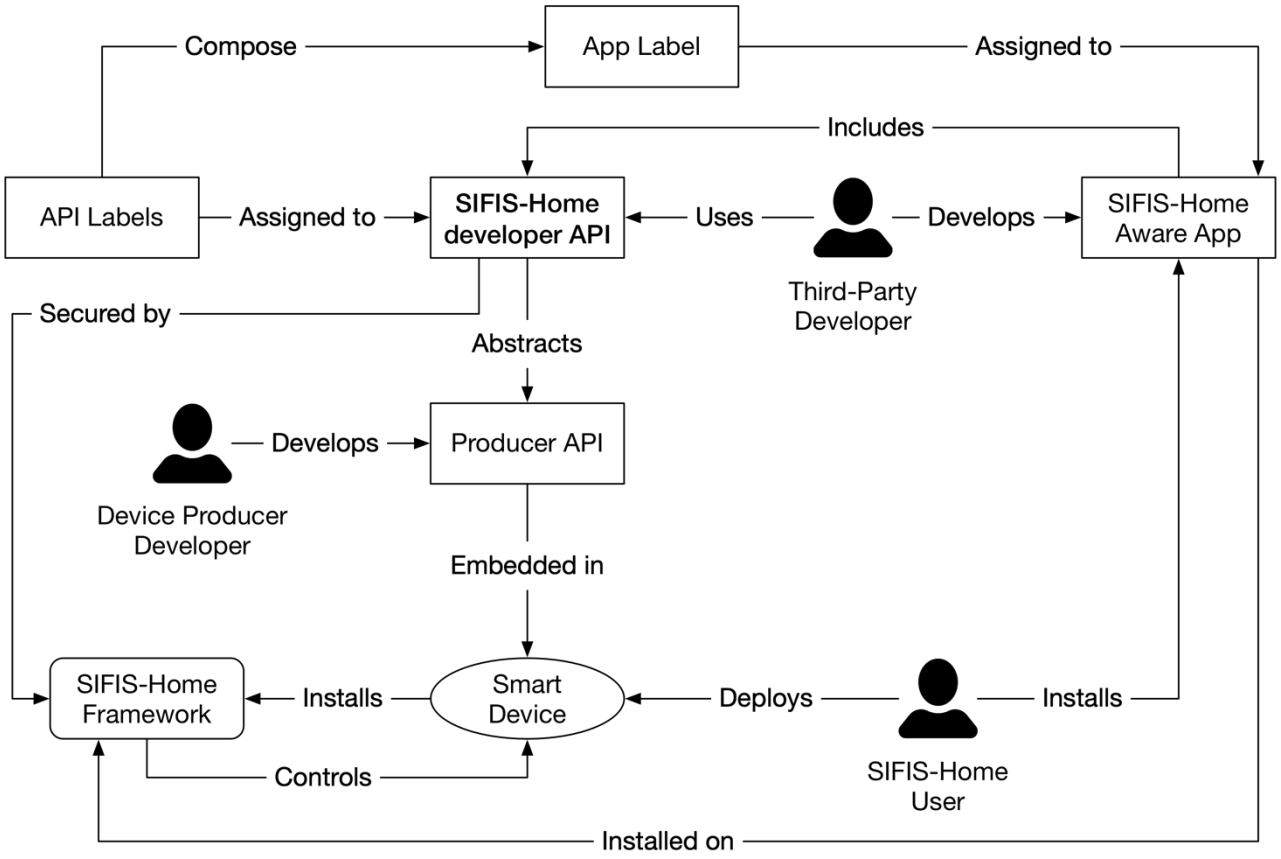


Figure 19 SIFIS-Home developer APIs integration and interaction with other components

In Figure 19 , we show how the SIFIS-Home developer API relates to various components of the architecture. We point out that an API label is assigned to a SIFIS-Home developer API; the SIFIS-Home aware app code includes SIFIS-Home developer APIs, whose API labels compose the app label. A SIFIS-Home developer API abstracts an API (producer API) that was written by the device producer developer.

The execution of a SIFIS-Home developer API is secured by the SIFIS-Home Framework, which is installed on smart devices. This means that the SIFIS-Home developer API includes some code that verifies whether such an API can be executed or not, according to the security policies defined by the user.

Since SIFIS-Home assumes that the system is fully distributed and redundant, the SIFIS-Home

developer API implementation uses a multi-layer approach.
The SIFIS-Home developer API itself tries to be idiomatic and easy to use:

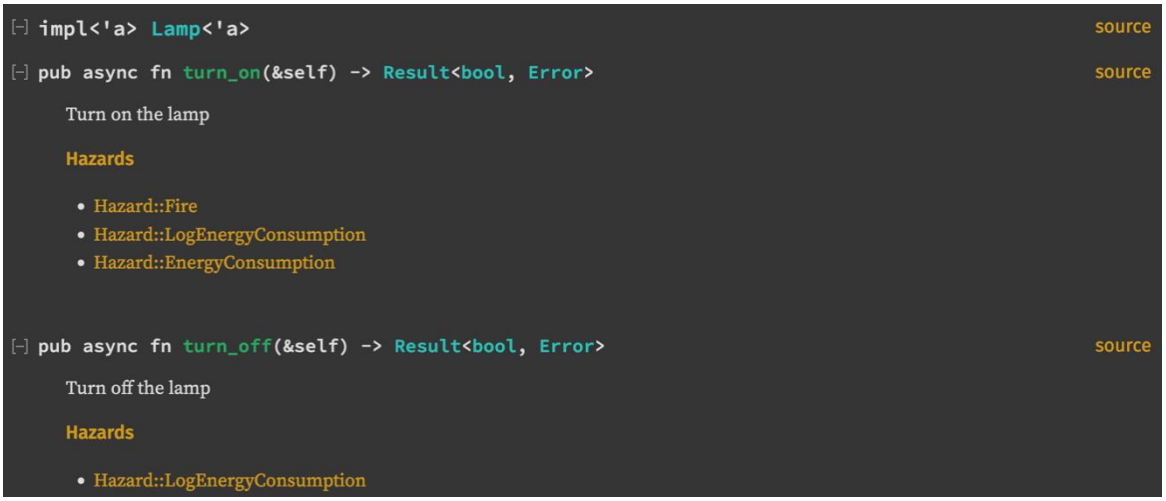


Figure 20 Rust autogenerated documentation via rustdoc

It relies on a simple RPC that uses messages matching 1:1 the API labels.

```
/// Turns a light off.
///
/// # Hazards
/// * [Hazard::LogEnergyConsumption]
async fn turn_lamp_off(id: String) -> Result<bool, Error>;
/// Get the current on/off status for a light
async fn get_lamp_on_off(id: String) -> Result<bool, Error>;
/// Change the brightness.
///
/// # Hazards
/// * [Hazard::Fire]
/// * [Hazard::LogEnergyConsumption]
/// * [Hazard::EnergyConsumption]
async fn set_lamp_brightness(id: String, brightness: u8) -> Result<u8, Error>;
/// Get the current brightness level.
async fn get_lamp_brightness(id: String) -> Result<u8, Error>;
```

Figure 21 Excerpt of the tarpc-based SIFIS-Home lower-level API

And a runtime daemon that maps the local RPC to the DHT system to interact with the other SIFIS-Home components.
The applications using the SIFIS-Home developer API can be segregated in a container with no I/O beside a single UNIX socket to communicate with the runtime daemon.

3.2 *Labelling Mechanism*

The SIFIS-Home framework defines APIs and makes them available for SIFIS-Home-aware application developers. A generic API implements functionalities of either a service or a device and operates on data. However, the execution of an API may imply obvious, as well as subtle, risks. We identified three categories of risks: *safety*, *privacy*, and *financial risks*.

Safety risks are due to events that produce a direct effect on the physical realm. Therefore, many APIs that trigger an actuator are associated with this kind of risk. Indeed, the smart home environment includes appliances that may cause injury, or even death, if misused. For instance, a smart cooktop could set the house on fire if unattended. Furthermore, safety risks regard all the threats that may put people and assets in danger. An undesired release of the door lock may lead to physical intrusion.

Privacy risks are related to operations that manage sensitive information. This kind of risk is associated with APIs that access resources and read data. APIs that get data from sensors, e.g., audio/video streams or temperature readings, as well as APIs that get the state from actuators, e.g., on/off state of a light bulb, are straightforward examples. Also, APIs that collect auxiliary data, such as logs, fall in this category.

Financial risks are related to operations that produce an expense. This kind of risk characterises APIs that lead to a monetary expense, either directly or indirectly. APIs that access user's "wallet" to place an order, e.g., to order food, or to pay for a subscription fee, e.g., periodic renewal of pay-per-view services, are examples of *direct financial risks*. On the contrary, with the term *indirect financial risks*, we refer to those operations that generate an indirect monetary cost for the user. In other terms, APIs that affect energy consumption (electricity, gas, water, etc.) bring an indirect financial risk. Of course, the extent of the risk differs from API to API and, more in general, from device to device.

In SIFIS-Home, we associate every SIFIS-Home developer API with a security label (the API label) that describes risks deriving from its execution. This security label consists of a list of *hazards*, each one identifying a risk. A hazard contains (i) its name, (ii) a description, and, optionally, (iii) the *risk score*, i.e., a value in the range [1,10] which indicates the gravity of the risk. The risk score could be assigned by the SIFIS-Home framework developers according to some risk assessment methodology, and depending on the type of device the API refers to. As practical examples, an API which has the effect of turning the oven on performs an operation that consumes a high instantaneous power and could potentially set the house on fire, so its label will include, among others, the hazard "FireHazard" and the hazard "ElectricEnergyConsumption". Another example is an API that acquires video signal from a video camera and stores it locally; this API may store frames containing children, which could represent a privacy concern for the end user; the hazard "ChildrenRecording" will be embedded in the API label. Again, an API that authorizes the payment of an asset will have a label embedding the hazard "SpendMoney."

Developers use our set of SIFIS-Home developer APIs to build SIFIS-Home-aware apps. When an app is ready for deployment, it is packaged in an *app bundle*. The app bundle contains the *application*, i.e., the executable, and the *app contract*, which consists of the *app label* and *code quality metadata*. The app label is automatically generated during the packaging phase and is populated with all the API labels associated with the SIFIS-Home developer APIs used within the app code.

When a user wishes to install an app from our marketplace, they can read the app label beforehand to be informed about risks deriving from the installation and usage of such an app. For each risk listed in the app label, a user-friendly description, and a risk score, when applicable, are provided. A short and simple description of the risks is required to promote the reading and comprehension by every class of end users. Moreover, the risk score, which is an integral number within a fixed range, could be mapped to keywords like "low," "medium," and "high" when shown to the user. This allows a more straightforward perception.

Besides informing the end user about the app's behavior and risks, the app label seamlessly integrates with user-defined policies. This means that if the user attempts to install an app which includes some risks that go against some user-defined policies currently in place, the user is notified of it, and is asked whether to proceed with the installation or instead abort it. In the former case, the application is installed, but, based on its labels, the execution of APIs that would violate the rules defined by the user are automatically denied at runtime. For example, if a user has defined a policy which reads as "No device that may cause fire can be turned on from outside the smart home," and the app label contains the `turn_oven_on` API, then the app can be installed, but the execution of that API is forbidden at runtime if the initiator cannot be ascertained to be within the local perimeter of the smart home. For a more detailed explanation of this workflow, please refer to Section 3.5 of deliverable D2.5.

3.3 *Hazards List*

In the following, we report a non-exhaustive list of hazards and their descriptions for the three risk categories. If present, the symbol ⚠ denotes that a risk score is associated with the hazard it is defined in.

Safety
FireHazard — ⚠ *The execution may cause fire.*
AirPoisoning *The execution may release toxic gasses.*
Burst *The execution may cause an explosion.*
Asphyxia *The execution may cause oxygen deficiency by gaseous substances.*
WaterFlooding *The execution allows water usage which may lead to flood.*
PowerOutage — ⚠ *The execution may cause an interruption in the supply of electricity.*
PowerSurge *The execution may expose to high voltage.*
UnauthorisedPhysicalAccess *The execution disables a protection mechanism and unaccredited individuals may physically enter the home.*
SpoiledFood *The execution may lead to rotten food.*
Burn *The execution allows usage of devices that may cause burns*
Scald *The execution allows usage of devices that may cause scalds*

Privacy
AudioVideoStream *The execution authorises the app to obtain a video stream with audio.*
AudioVideoRecordAndStore *The execution authorises the app to record and save a video with audio on persistent storage.*
LogUsageTime *The execution authorises the app to get and save information about app's duration of use.*
LogEnergyConsumption *The execution authorises the app to get and save information about app's energy impact on the device the app runs on.*
RecordUserPreferences *The execution authorises the app to get and save information about user's preferences.*
RecordIssuedCommands *The execution authorises the app to get and save user inputs.*
TakePictures *The execution authorises the app to use a camera and take photos.*
TakeDeviceScreenshots *The execution authorises the app to read the display output and take screenshots of it.*

Financial
SpendMoney *The execution authorises the app to use payment information and make a payment transaction.*
PaySubscriptionFee *The execution authorises the app to use payment information and make a periodic payment.*
ElectricEnergyConsumption — ⚠ *The execution enables a device that consumes electricity to operate.*
GasConsumption — ⚠ *The execution enables a device that consumes gas to operate.*
WaterConsumption — ⚠ *The execution enables a device that consumes water to operate.*

Note that the list of hazards is not exhaustive and is designed to be extended also externally, through third parties and/or developers proposing new tags for new specific operations related to smart home devices.

As described in deliverable D2.5, an ontology has been created to give a semantic meaning to the hazards. The ontology includes a complete list of currently defined hazards, which have been defined as named individuals.

3.4 *API Labels*

In the following, we report a brief list of sample APIs and their related labels.

API	Label
`turn_oven_on`	• FireHazard • PowerOutage (risk score: 8) • LogEnergyConsumption • ElectricEnergyConsumption (risk score: 8)
`record_video`	• AudioVideoRecordAndStore
`lower_fridge_temperature`	• PowerOutage (risk score: 5) • ElectricEnergyConsumption (risk score: 5)
`raise_fridge_temperature`	• SpoiledFood
`order_food`	• SpendMoney
`turn_air_conditioner_on`	• PowerOutage (risk score: 7) • ElectricEnergyConsumption (risk score: 7)
`disarm_alarm`	• UnauthorisedPhysicalAccess
`open_shutters`	• UnauthorisedPhysicalAccess
`unlock_door`	• UnauthorisedPhysicalAccess
`renew_subscription`	• PaySubscriptionFee

3.5 *Labels Format*

API labels and app label should be implemented so that they can be easily converted in other formats and exported, namely they need to be serializable. Formats that they can be serialized to are JSON, CBOR, XML, TOML, etc. In the following, we present and explain an instance of API label in JSON format, based on the JSON reference schema introduced and explained in Appendix A. Also, in Appendix A we provide an implementation of the JSON schema for the app label.

3.5.1 API Label Example (JSON Format)

The example is given for the `turn_oven_on` API. The JSON object in the following listing contains four properties: (i) `api_name`, which must match the SIFIS-Home developer API that the label refers to, i.e., `turn_oven_on`; (ii) `api_description`, which gives a brief explanation of the API behaviour; (iii) `behavior_label`, which specifies the type of device(s) associated with the API and the action(s);

and (iv) `security_label`, which specifies the risks associated with the API.

The `security_label` property is an object that contains three properties representing the three categories `safety`, `privacy`, and `financial`. Every property contains an array populated by objects with the same structure, each representing a hazard. These objects identify the risks associated with the API, and they are composed of the properties `name`, `description`, and, optionally, `risk_score`. In the example, the `safety` property is an array of size two containing the hazards PowerOutage, which also reports the risk score, and FireHazard.

```
{
  "api_name": "turn_oven_on",
  "api_description": "Activates the oven at the last selected temperature.",
  "behavior_label": [
    {
      "device_type": "oven",
      "action": "turn_on"
    }
  ],
  "security_label": {
    "safety": [
      {
        "name": "FireHazard",
        "description": "The execution may cause fire."
      },
      {
        "name": "PowerOutage",
        "description": "High instantaneous power. The execution may cause power o
utage.",
        "risk_score": 8
      }
    ],
    "privacy": [
      {
        "name": "LogEnergyConsumption",
        "description": "The execution allows the app to register information abou
t energy consumption."
      }
    ],
    "financial": [
      {
        "name": "ElectricEnergyConsumption",
        "description": "The execution enables the device to consume further elect
ricity.",
        "risk_score": 8
      }
    ]
  }
}
```

4 Legal Guidelines (Licensing and Privacy)

Different subjects may be obliged – or at least strongly interested – to comply with privacy laws and, therefore, adopt specific standards. In a situation where data is collected through smart-home systems, the Data Controller of processing performed with software interacting with IoT devices, according to the GDPR, could be the owner of the home (or the tenant), who willingly installed IoT devices in their home. However, it can also be a Software as a Service provider, who obtains personal data from IoT devices and stores it. While following the privacy rules is not mandatory for the software developer (or the "application designer", as it is usually called today), it is very advantageous for them to comply with privacy rules and follow current standards. That is, being compliant with privacy rules means that the software could be more easily reviewed and accepted by both the end-user and the Software as a Service Provider, who wants to provide services based on the application designed by the developer. For this reason, we propose some insight into following the best practices to ship software that is not only privacy-compliant but also based on the state-of-the-art approaches for data protection and self-evaluation. This section deals with a second goal that derives from the reuse and distribution of free and open-source software: compliance with a legal obligation arising from free and open-source software.

4.1 *Privacy*

Different rules provided by the GDPR are to be followed by software developers and publishers. In particular, the Software as a Service provider must satisfy accountability requirements, and must perform a quantitative risk assessment analysis. The main requirements are found in:

- **Article 13/14 GDPR**[2], that lists information to be provided to the data subject. This information is to be given also when not collecting personal data:
    - a) The identity and contact details of the Data Controller and the Data Protection Officer if present.
    - b) The purposes of the processing.
    - c) What kind of personal data is being processed.
    - d) If applicable, the intention of transferring personal data to a third-party country, external from the European Union.
    - e) The period in which personal data will be kept.
    - d) The right for the data subject to obtain a modification or cancellation of their data and the practical ways in which he can exercise this right.
    - f) the existence of automated decision-making, including profiling.

[2] Article 13: https://www.privacy-regulation.eu/en/article-13-information-to-be-provided-where-personal-data-are-collected-from-the-data-subject-GDPR.htm
Article 14: https://www.privacy-regulation.eu/en/article-14-information-to-be-provided-where-personal-data-have-not-been-obtained-from-the-data-subject-GDPR.htm

- **Article 25**[3], that asks the Data Controller to implement appropriate technical and organizational measures designed to implement data-protection principles, such as data minimization, subject to the cost of the implementation and the nature of processing. This is called "data protection by design" and is combined with the concept of "data protection by default," which mandates the controller to implement technical and organizational measures to ensure that, by default, only personal data which is necessary for each specific purpose of the processing is processed.
- **Article 32**[4], which sets requirements on the matter of security (obligation to implement appropriate technical and organizational measures to ensure a level of security appropriate to the risk).
- **Article 35**[5], mandates the controller to carry out a Privacy Impact Assessment when the processing is likely to result in an elevated risk to the rights and freedoms of natural persons, or if the processing is carried out automatically or on a large scale.

Of these requirements, the most complicated is the one in Article 35. To aid Data Controllers in building and demonstrating compliance with the GDPR, the French CNIL[6] created a useful tool[7] that has quickly become a reference standard. Using this tool can help the developer to understand the security and privacy risks posed by his software and may give him the chance to solve them before shipping his product to the public. It is therefore highly recommended to use this tool in order to ensure a software's compliance with GDPR rules.

4.2 *Licensing*

The goal of complying with legal obligations arising from reuse and distribution of free and open-source software stands at the base of both the OpenChain[8] and the ClearlyDefined[9] projects. These initiatives aim at helping free and open-source software to become more standardized and well defined, thus clearing doubts regarding legal compliance and providing a possible subsequent developer with clear and comprehensive information, about the limits and obligations that the various free and open-source licenses impose on the use or modification of the original software. More specifically, the OpenChain project wants to "establish requirements to achieve effective management of free/open-source software for software supply chain participants, such that the requirements and associated collateral are developed collaboratively and openly by representatives from the software supply chain, open-source community, and academia."OpenChain has become an international standard (ISO 5230) that allows software developers to obtain compliance regarding open-source licenses. After following its guidelines, a software developer can send to the OpenChain organization a document affirming that their software satisfies all the requirements of the specification and is therefore compliant to the

[3] Article 25: https://www.privacy-regulation.eu/en/article-25-data-protection-by-design-and-by-default-GDPR.htm
[4] Article 32: https://www.privacy-regulation.eu/en/article-32-security-of-processing-GDPR.htm
[5] Article 35: https://www.privacy-regulation.eu/en/article-35-data-protection-impact-assessment-GDPR.htm
[6] CNIL is France's independent administrative regulatory body to ensure that data privacy law is enforced in the French territories.
[7] Downloadable on the official CNIL's website: https://www.cnil.fr/en/open-source-pia-software-helps-carry-out-data-protection-impact-assesment
[8] See https://www.openchainproject.org/resources/faq
[9] See https://clearlydefined.io/about

OpenChain standard[10]. It is important to notice that an OpenChain compliance badge can only be obtained if all the requirements are satisfied, and not just some of them. Following the OpenChain specification, the software developer creates, and therefore can make available, the compliance artifacts, that is "a collection of artifacts that represent the output of a compliance program and accompany the supplied software.." that "..may include (but is not limited to) one or more of the following: attribution notices, source code, build and install scripts, copy of licenses, copyright notices, modification notifications, written offers, open source component bill of materials, and SPDX documents". The ClearlyDefined project, on the other hand, is still on its early days, but already offers some suggestions on how to distribute a clearly defined open source software, so that users and other developers alike are precisely informed on important things such as the type of open source license used, or where to find the components used for bug fixing or new versions (such as a GitHub page), and how these are made. It also offers a security forum so that developers can ask questions and receive answers in the matter of security and vulnerabilities that may be present in their software[11]. Following both the OpenChain and the ClearlyDefined practices gives the software an important certification that can aid its diffusion.

4.3 *Highlights*

In order to summarize the privacy and licensing legal requirements and consider the usefulness of following some standard practices, we propose this "traffic light system" for assessing whether one's software is compliant with the concepts exposed in the previous paragraphs.

4.3.1 Green Light

1. The software developer successfully performed a Privacy Impact Assessment based on reasonable assumptions for at least a standard use case and the documentation it accompanies the software[12].
2. The software developer has produced information to be used for compliance with articles 13, 14, 25 and 32 of GDPR and the document containing the information accompanies the software.
3. The software developer followed the OpenChain specification or other public specification for licensing compliance and the software is accompanied by compliance artifacts.
4. The methodology used and the standard followed in creating the privacy impact assessment, the document produced according to point 2 and the compliance artifacts, is publicly available, free of any right of third party, so that everyone can assess compliance and use it.

The Data Controller can easily assess if the software is compliant with the GDPR and Free and Open-

[10] These requirements are found in the Supplier Education Pack (see https://github.com/OpenChain-Project/Reference-Material/raw/master/OpenChain-ISO-5230-Supplier-Education-Pack/OpenChain%20ISO%205230%20Supplier%20Education%20Pack.zip) downloadable on the official OpenChain website, and in particular in the "basic-open-source-education" .pdf file in the .zip pack.

[11] The ClearlyDefined "checklist" can be found in the ClearlyDefined by clearlydefined webpage (see https://docs.clearlydefined.io/clearly#secure), on the official ClearlyDefined website.

[12] For example using the CNIL's tool as stated before

Source license obligations and can therefore be used in the EU to process personal data.

4.3.2 Yellow Light

1. The software developer states that they can make available all the information required to perform a Privacy Impact Assessment.
2. The software developer declares that they gave produced information to be used to comply with articles 13, 14, 25 and 32 of the GDPR.
3. The software developer declares that they have compliance artifacts.
4. The software developer declares that the methodology used and the standard followed in creating the above documentation can be made available to assess compliance.

The Data Controller should be in the position to assess if the software is compliant with the GDPR and free and Open-source license obligations and can therefore be used in the EU for processing personal data, although some additional work will be required.

4.3.3 Red Light

One or more of the points provided for Yellow Light is not satisfied. This software must be carefully analyzed to assess if it is compliant with the GDPR and free and open-source license obligations before using it in the EU to process personal data.

4.4 *Implementation of the "traffic light" system in the Dashboard*

The "traffic light system" described above has been implemented in the Privacy Dashboard described in detail in D2.7 (Final report on legal and ethical aspects), Chapter 8 (Implementation of the Privacy Dashboard) through a "questionnaire view". The data controller can use this tool designed to evaluate the GDPR and license compliance and privacy-friendliness of apps, which uses a traffic light system to indicate compliance levels.
As described in the previous paragraphs, this relates to GDPR obligations, recognized privacy standards, and OpenChain or other public specifications. If a developer complies with all the criteria, the app will be assigned a green light, thus indicating compliance with the GDPR and free and open-source license obligations. If the responses are incomplete but indicate a willingness to comply, the app will be assigned a yellow light, while if the responses do not meet the criteria, the app will be assigned a red light.
The color-coded overview provides an immediate way for data controllers to assess the compliance status of all their applications, which is shown in the "app" view for each application.
This tool encourages data controllers to evaluate their apps and identify areas where they can improve their compliance, thus promoting privacy-friendliness, GDPR compliance and open-source licenses compliance.

5 Conclusion

This document has presented comprehensive guidelines for software developers, aiming at facilitating the creation of secure, privacy-aware, and policy-based IoT software that aligns with the latest standards and best practices. The guidelines are structured into three main sections, each focusing on a crucial aspect of software development and management within the context of smart home environments.

Section 2 offers an extensive set of guidelines designed to ensure security and quality during software development introducing a multi-layered workflow that covers various aspects, such as static and dynamic analysis, code coverage, dependency management, and deployment. This adaptable workflow enables parallel execution of independent tasks to reduce developer wait time and provide prompt feedback on potential issues. Starting from this multi-layered workflow, a specular continuous integration workflow for Rust language projects, based on GitHub Actions, has been created with the goal of running tools that guarantee code quality, security, and performance aspects.

Section 3 emphasizes the significance of transparency and easy accessibility of information regarding the security and privacy aspects of IoT applications. It presents the SIFIS-Home framework, which associates developer APIs with security labels that encompass safety, privacy, and financial risks. These labels are combined into an app label, providing users with essential information about the application's risks and compliance with their policies. The SIFIS-Home framework offers an innovative approach for managing smart-home applications, allowing users to make informed decisions while ensuring that their user-defined policies are respected.

Section 4 underscores the importance of compliance with privacy laws and open-source software licensing requirements for developers and publishers of smart-home applications. By adhering to the GDPR and open-source licensing standards such as OpenChain and ClearlyDefined, developers can ensure that their software is more easily accepted by end-users and service providers and enhance the software's credibility and diffusion. The section also describes a "traffic light system" to help developers assess their software's compliance with privacy and licensing requirements, making it easier for data controllers to evaluate the software and determine its suitability for use in the EU.

In conclusion, the guidelines presented in this document provide developers with a solid foundation to create secure, privacy-aware, and policy-compliant IoT software that meets the latest standards and practices. By following these guidelines, developers can not only create a more secure and compliant software environment but also contribute to a broader ecosystem that values transparency, privacy, and open collaboration in developing smart home applications.

6 References

[Emami-Naeini et al., 2020] Emami-Naeini, P., Agarwal, Y., Cranor, L. F., & Hibshi, H. (2020, May). Ask the experts: What should be on an IoT privacy and security label? *In 2020 IEEE Symposium on Security and Privacy (SP)* (pp. 447-464). IEEE.

[Android Permissions, 2023] Retrieved from Permissions on Android: https://developer.android.com/guide/topics/permissions/overview

[Marking, 2023] CE conformance marking. (2023). Retrieved from https://ec.europa.eu/growth/single-market/ce-marking/manufacturers_en

Glossary

Acronym	Definition
API	Application Programming Interface
CBOR	Concise Binary Object Representation
CE	Conformité Européene
CI	Continuous Integration
CNIL	Commission Nationale de l'Informatique et des Libertés
CPU	Central Processing Unit
DHT	Distributed Hash Table
EU	European Union
GDPR	General Data Protection Regulation
IFTTT	If This Then That
IoT	Internet of Things
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
QR	Quick Response
PLOC	Physical Line of Code
RPC	Remote Procedure Call
SaaS	Software As A Service
SIFIS-Home	Secure Interoperable Full-Stack Internet of Things for Smart Home
SPDX	Software Package Data Exchange
TOML	Tom's Obvious, Minimal Language
WP	Work Package
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Appendix A: API Label and App Label Schemas

In the following, we present and describe the API label and the App label JSON schemas. Both schemas can be found in the GitHub repository "json-schemas", within the sifis-home organization.

API Label Schema

The reference JSON schema for an API label is presented in the following listing. This schema includes the *hazard* object and the *functionality* object. The `hazard` object is a structure representing a hazard and is used by all the risk categories (`safety`, `privacy`, and `financial` properties) of the schema. The `hazard` object contains two required properties: (i) `name`, which must match one of the hazards defined in the hazard list, and (ii) `description`, which gives a brief explanation of the risk. Additionally, the `hazard` may contain the `risk_score` property, indicating the gravity of the risk and defined as a number in the range [1,10] with a step size of 1.

The `functionality` object includes the properties `device_type,` i.e., the category of the device, and `action`, i.e., the operation that such a device would perform. Since an API may operate on more than one device and action, the `behavior_label` is defined to be an array of `functionality` objects.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://raw.githubusercontent.com/sifis-home/json-schemas/master/api-
label.jschema",
  "title": "SIFIS-Home API label schema.",
  "description": "JSON schema defining the API behavior label and the API
security label structures for the SIFIS-Home framework in the context of the
SIFIS-Home developer APIs.",
  "type": "object",
  "properties": {
    "api_name": {
      "type": "string"
    },
    "api_description": {
      "type": "string"
    },
    "behavior_label": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/functionality"
      }
    },
    "security_label": {
      "type": "object",
      "properties": {
        "safety": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/hazard"
          }
```

```
      },
      "privacy": {
        "type": "array",
        "items": {
          "$ref": "#/definitions/hazard"
        }
      },
      "financial": {
        "type": "array",
        "items": {
          "$ref": "#/definitions/hazard"
        }
      }
    },
    "required": [
      "safety",
      "privacy",
      "financial"
    ]
  }
},
"required": [
  "api_name",
  "api_description",
  "behavior_label",
  "security_label"
],
"definitions": {
  "hazard": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string"
      },
      "description": {
        "type": "string"
      },
      "risk_score": {
        "type": "number",
        "minimum": 1,
        "maximum": 10,
        "multipleOf": 1
      }
    },
    "required": [
      "name",
      "description"
    ]
  },
  "functionality": {
    "type": "object",
    "properties": {
```

```
        "device_type": {
          "type": "string"
        },
        "action": {
          "type": "string"
        }
      },
      "required": [
        "device_type",
        "action"
      ]
    }
  }
}
```

The app label can be a JSON object that contains an array of API labels in JSON format, as the one defined in the example above.

App Label Schema

The *App label schema* is defined in the following listing. The schema declares three required properties, i.e., `app_name`, `app_description`, and `api_labels`. The latter is an array of API label objects, defined according to the API label schema.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://raw.githubusercontent.com/sifis-home/json-schemas/master/app-
label.jschema",
  "title": "SIFIS-Home app label schema.",
  "description": "JSON schema defining the app label structure within the SIFIS-
Home framework.",
  "type": "object",
  "properties": {
    "app_name": {
      "type": "string"
    },
    "app_description": {
      "type": "string"
    },
    "api_labels": {
      "type": "array",
      "items": {
        "$ref": "/sifis-home/json-schemas/master/api-label.jschema"
      }
    }
  },
  "required": [
    "app_name",
    "app_description",
    "api_labels"
  ]
```

```
}
```