D2.2

WP2 – Preliminary Developer Guidelines

SIFIS-HOME

Secure Interoperable Full-Stack Internet of Things for Smart Home

Due date of deliverable: 30/09/2021
Actual submission date: 30/09/2021

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

24/09/2021
Version 1.0

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

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

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

Approved by: Joni Jämsä (CEN), Marko Komssi (FSEC)

Revision History

Version	Date	Name	Partner	Section Affected Comments
0.1	14/05/2020	Tentative ToC and contents	POL, LUM, CNR	All
0.2	11/06/2021	Added software quality metrics	POL	Section 2
0.3	26/06/2021	Added labels	CNR	Section 3
0.4	16/07/2021	Added workflow	LUM	Section 2
0.5	28/07/2021	Added privacy and licensing	POL	Section 3
0.6	18/08/2021	Document proofread	POL, LUM, CNR	All
1.0	20/09/2021	Changes after internal review	POL, LUM, CNR	All

Executive Summary

This document presents a set of developer guidelines for the creation of secure, privacy-aware, policy-based IoT code for the SIFIS-Home project and third-party applications expected to run within the SIFIS-Home framework. The guidelines presented in this deliverable are preliminary. The completed guidelines will be presented in deliverable D2.4.

This document also reviews best practices, techniques, and formalisms designed to increase the quality and reliability of IoT software and discusses how SIFIS-Home developer APIs can help developers write more reliable and secure software.

Finally, this deliverable presents legal guidelines for complying with regulations when handling personal data. A more specific report on Legal and Ethical Aspects will be presented in deliverable D2.5.

Table of contents

1 Introduction

One of the main objectives of the SIFIS-Home project is to provide developers with some guidelines for writing secure, privacy-aware, and policy-based IoT software. Developers will benefit from a new labelling method, which will evaluate IoT software and infrastructure levels based on security and privacy metrics. To achieve this labelling system, it is required to define security and privacy metrics for measuring software and infrastructure.

The SIFIS-Home deliverable D2.1 "Report on Security and Privacy Metrics" has provided a set of metrics to be used for assessing the quality, security, and privacy of the IoT software.

This document will give developers an initial set of guidelines for writing secure, privacy-aware, and policy-based IoT software and is structured as follows.

Section 2 proposes guidelines for software development and software quality evaluation. The proposed guidelines are structured as a workflow that describes step-by-step procedures and methods a developer should follow to produce high-quality software. The described workflows include practical examples. Also presented are notions of software quality from a developer perspective and definitions of software quality to be used for evaluation. Section 2 also includes additional notes that a developer may consider for improving their software further.

Section 3 covers possible risks deriving from the execution of SIFIS-Home developer APIs. A set of labels representing safety, integrity, security, and privacy issues intrinsically related to the execution of each specific developer API are presented in this section. The section links each SIFIS-Home developer API with an *API label* that describes its possible risks. *API labels* linked with the APIs used within an application code form a general label called *App Label*. This label is designed to be shown to a user during the installation process. The advantage of this mechanism is twofold: (i) it informs the user about possible risks related to the application, and (ii) it seamlessly integrates with user-defined policies, meaning that if the label of a given API violates some rule defined by the user, its execution is automatically prevented.

Section 4 covers legal guidelines concerning the SIFIS-Home system. Collecting data is a necessary but challenging task. In the European Union, indiscriminate data collection is limited and regulated through GDPR (General Data Protection Regulation). Some subjects, such as the Data Controller, are obliged to follow GDPR's rules. Others, such as software developers or "application designers" are not. However, these other parties may still be interested in following GDPR guidelines so that the resulting software is by default compliant with privacy laws. Following GDPR guidelines allows an application to be better distributed, accepted, and reviewed by the end user and the potential Service Provider. We, therefore, clarify which rules must be followed, by what subjects, and when it is mandatory or optional to follow them. This topic is also closely linked to licensing, the use of free and open-source software, and legal obligations arising from the software's use. This section explores requirements found mainly in GDPR articles 13, 14, 25, 32, and 35, and corresponding initiatives aimed at making free and open-source software more standardized (such as the OpenChain[1] and ClearlyDefined[2] projects). A "traffic

[1] https://www.openchainproject.org/
[2] https://clearlydefined.io/

light system” is proposed based upon different criteria through which software can be evaluated based on privacy and licensing regulations.

2 Software Quality Guidelines

This section proposes some guidelines that developers are recommended to follow while developing software and evaluating software quality. These guidelines are structured as workflows that have been created starting from the concepts and mechanisms described in D2.1. Of the mechanisms presented, the following are most notable from a developer's perspective:

- **Static analysis**: mechanisms for the analysis of source code to find defects and provide information to improve code quality.
- **Dynamic analysis**: mechanisms for the analysis of running software to find possible memory faults and security issues. Such analysis can also detect parts of a program that can be further optimized.
- **Code coverage**: mechanisms for determining the percentage of source code covered by tests.

Each workflow presented in this section includes practical examples depicted using the C programming language and notes about software quality evaluation procedures necessary to better explain the subsequent certification process and the content of some workflow steps. Some additional notes related to the *Rust* programming language are also included towards the end of this section. The guidelines presented in this section are primarily for developers. System integrators and software distributors may note that it is possible to find some information about software packaging within the various subsections tests for the final binary.

2.1 *Workflow Structure*

While multiple tools may fit the same role within a good workflow, this section does not mention specific tools or software. For specific examples, refer to the C workflow below.

2.1.1 Main Components

The workflows presented assume that existing projects, build systems, continuous integration, and continuous delivery phases have already been implemented and properly configured.

2.1.2 Lifecycle

The workflow, as illustrated in Figure 1, assumes that the software is developed using a pull request model:

- A patch set is prepared, containing features and/or fixes as well as tests covering all code changes.
- The patch set is put up for review.
- The continuous integration automation will run a set of fast static analyses, which as a rule of thumb, should be at least twice as fast as building the project. Static analysis checks:

  - Coding style
  - Code quality
- If the previous phase passes, the continuous integration system will run more resource-intensive tasks, including:
  - Compile tests
  - Static fault analysis
  - Unit tests
  - Integration tests
  - Code coverage evaluation
  - Dynamic fault analysis
- Once those phases pass, it is possible to prepare packages and ensure that the software is ready for distribution.
- If all phases pass and the reviewers approve the changes, the patch set is merged.

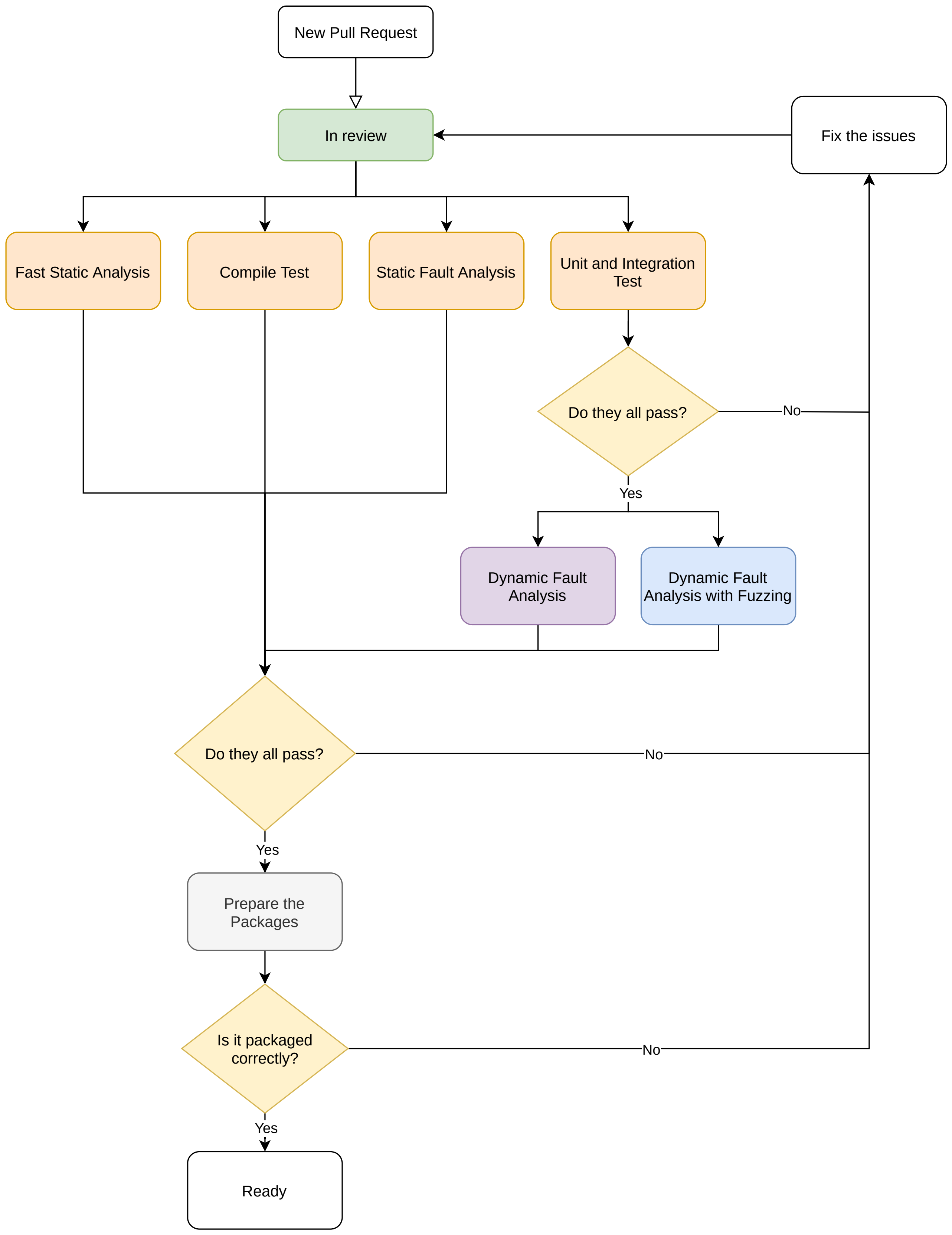

Figure 1: Global flowchart

The described workflow should attempt to minimize developer wait time. As soon as a mistake is detected, it should be reported. When possible, most of the faster tests should be run by the developer while writing the software. Ideally, all checks should be integrated into the build system, making it more practical to execute every test locally when needed.

2.1.3 Fast Static Analysis

Lint, or a linter, is a static code analysis tool used to flag programming errors, bugs, stylistic errors, and suspicious constructs. The term originates from a Unix utility that examined C language source code. Fast static analysis is designed to enforce uniformity through linters and provide a quick overview of the project state. Code quality analysis mechanisms should execute quickly and assist developers and reviewers by highlighting parts of the code that have higher complexity and thus require more documentation and additional tests. Figure 2 illustrates the fast static analysis flowchart.

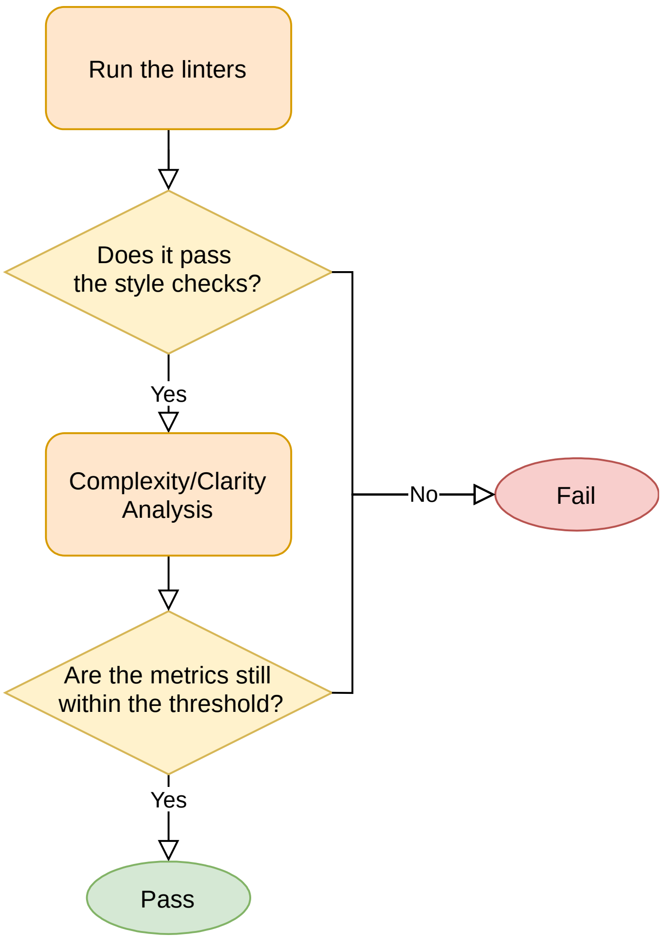
Run the linters
Does it pass the style checks?
Yes
Complexity/Clarity Analysis
No
Fail
Are the metrics still within the threshold?
Yes
Pass

Figure 2: Fast static analysis flowchart

2.1.4 Compile Test

Making sure the code builds for all supported targets is essential, even if the developers are not running tests on all of them. Setting up and keeping an entire test environment operational for many architectures can be cumbersome; having a cross-building setup is a good compromise. If the code stops compiling on a specific architecture, the problem must be resolved as soon as possible. Figure 3 illustrates the compile test flowchart.

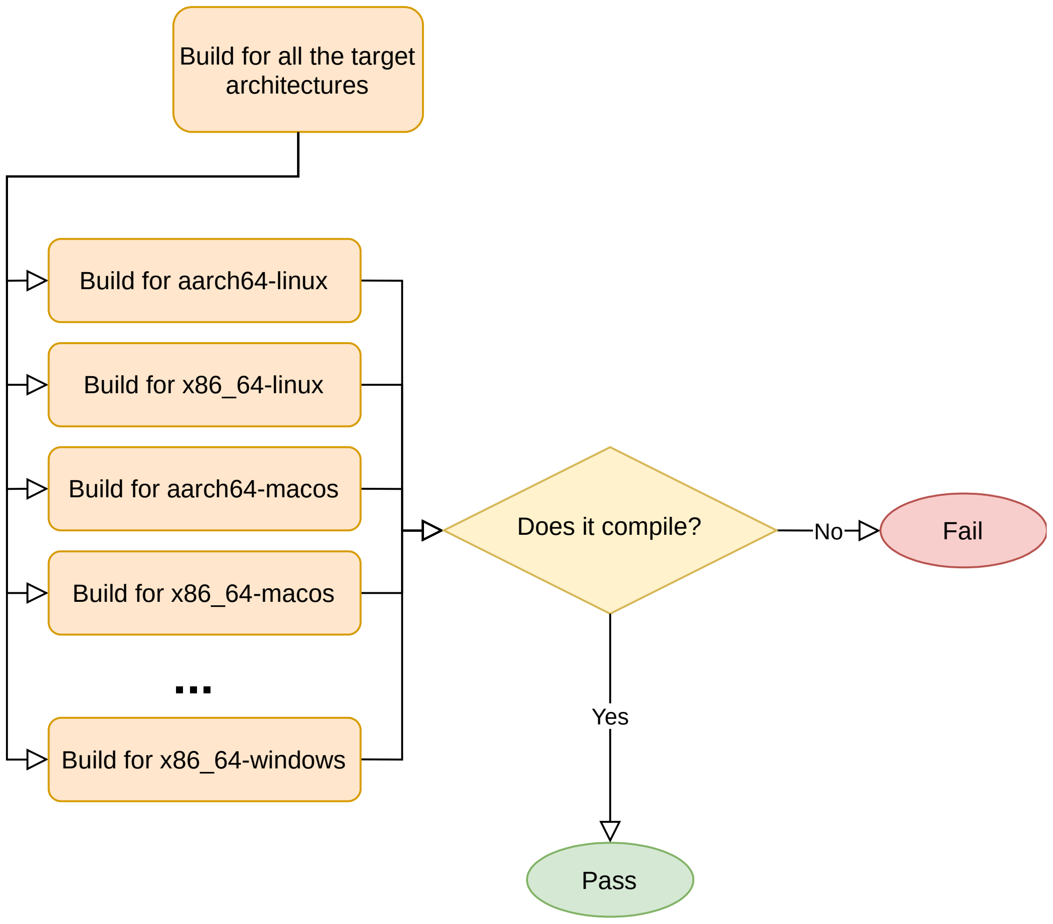

Figure 3: Compile test flowchart

2.1.5 Unit and Integration Testing

Proper unit and integration tests ensure that the behaviour of the software is correct. A single unit test is quick to write and usually quick to execute. However, many individual tests add up quickly, and completing the unit test suite may require significant time and resources. Integration tests may be more cumbersome in general. However, they consider a bigger picture and catch mistakes that unit tests do not. In general, tests should cover as much of the code base as possible. Figure 4 illustrates the testing flowchart.

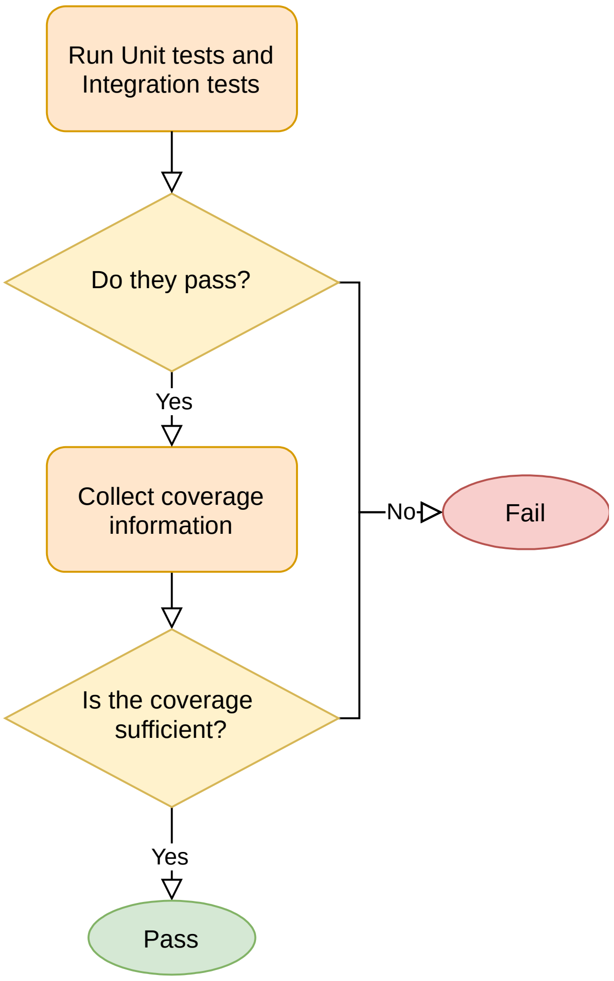
Run Unit tests and Integration tests
Do they pass?
Yes
Collect coverage information
No
Fail
Is the coverage sufficient?
Yes
Pass

Figure 4: Testing flowchart

2.1.6 Coverage Analysis

Many tools designed to measure how much code unit tests cover exist. Some require specifically instrumenting the build, adding another compilation phase. Others use non-intrusive profiling to obtain less precise data but at a fraction of the time expenditure.
It is possible to use a code complexity/quality map and a code coverage map to decide which areas of the code should be prioritized and when the coverage is adequate. If a new feature is introduced without enough tests covering it, such mechanisms will highlight the issue. Any pull request reducing

the code coverage below a set threshold should be rejected.

2.1.7 Static Probable Fault Analysis

Static analysis typically takes as long as, or longer than compiling software. For many languages, the compiler suite itself may include static analysis capabilities. Static analysers can detect many mistakes that may have been overlooked during a code review, and they are usually still faster than some later build phases. Depending on the tool, static analysers can detect simple use-after-free or null-dereferences or actual API misuse such as locking faults using pthreads. Figure 5 illustrates the static fault analysis flowchart.

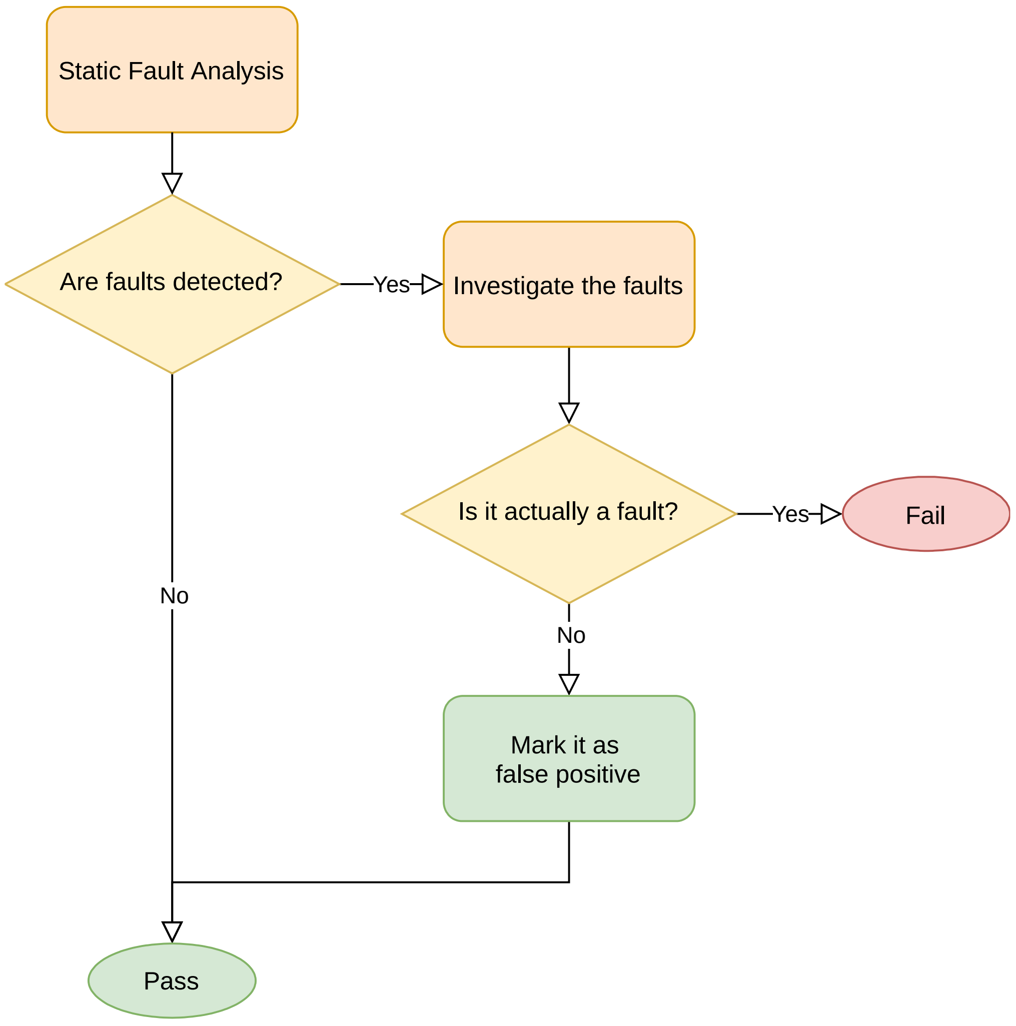

Figure 5: Static fault analysis flowchart

2.1.8 Dynamic Fault Detection

The tools available to dynamically detect faults in code can be split into two groups:

- those that require custom builds and instrumentation;
- those that rely on the non-intrusive profiling features provided by the platform.

Dynamic analysis execution can take between two and ten times the duration of the execution of a normal debug build. They may include options for fuzzing (to detect faults and expand code coverage), but due to the time-consuming nature of such functionality, these options should not be enabled on a per-pull request. Dynamic analysis tools tend not to hit many false positives. When they do, they are usually caused by a problem with the compilation or due to limitations in their CPU/memory models. These tools can easily find faults caused by unexpected interactions with external APIs that static analysis mechanisms cannot detect. Figure 6 illustrates the dynamic fault detection flowchart.

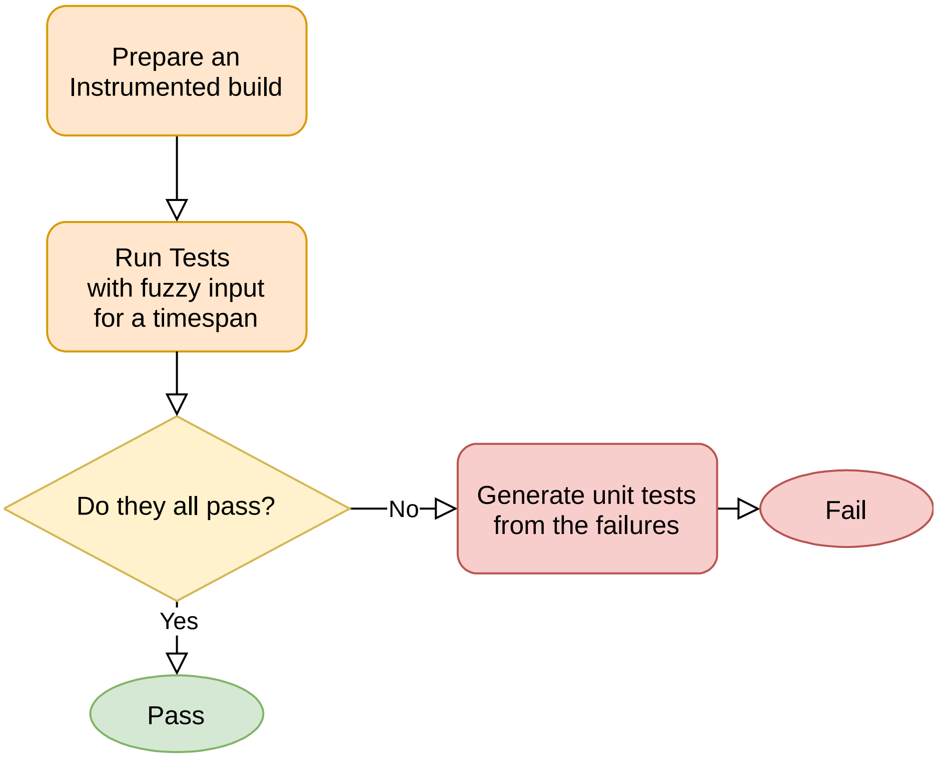
Prepare an Instrumented build
Run Tests with fuzzy input for a timespan
Do they all pass?
No
Generate unit tests from the failures
Fail
Yes
Pass

Figure 6: Dynamic fault detection flowchart

2.1.9 Packaging Check

Once all builds and tests have been completed, artefacts are ready to be packaged into an installer or
update. The process of creating packages should also be automated and scheduled to run based on the
organization's needs (per pull request or at specific time intervals). Packages destined to be shipped to
2.2 customers may need to be digitally signed, which may require additional (and possibly manual) steps.
Since code-signing is a highly secure process, signing keys should be stored separately from build
environments when possible.

Example Workflow - C

The build process for C code differs per platform. On Windows, compilers are provided with each IDE. On Linux and Unix systems, makefiles and *configure* helper scripts are most widely used. MacOS uses a mix of the two, usually utilizing mechanisms provided by XCode, the default IDE on that platform.

Nowadays, Ninja (Ninja, 2021) has gained popularity, with CMake[3] and Meson (Meson, 2021) replacing traditional *configure* scripts.

For the following examples, we will use Meson since it provides excellent yet minimalistic test integration that includes support for test coverage out of the box. CMake's testing support is richer and more complex, making it less suited as an illustrative example.

2.2.1 Fast Static Analysis

2.2.1.1 Linting

A popular linter format for the C language is clang. **Meson** integrates with it out of the box.

2.2.1.2 Code Quality Metrics

To obtain code quality metrics, we suggest the rust-code-analysis tool, which provides a good report, despite the fact that it is not integrated directly with Meson. Calling the rust-code-analysis-cli while passing the source root directory is enough to get relevant information. The metrics provided by this tool can help guide developers to test more complex code in a more thorough manner. Depending on the project, it is possible to consider automatically blocking a patch that introduces too much code complexity until enough tests have also been added.

In Task T2.2, better integration with the build system and the code-coverage evaluation will be explored.

2.2.2 Tests

The C language does not have a built-in unit test concept. However, several third-party libraries exist to automate unit test processes, all of which have a standardized output. In our examples, unit tests can be built as normal Meson executable() targets. The test() function allows developers to run a test executable, parse its output and report the result.

```
e = executable('prog', 'testprog.c');
test('name of test', e);
```

2.2.3 Code Coverage

Meson integrates with a code coverage tool called gcovr. As explained in the manual, `-Db_coverage=true` is a shorthand to instrument the build and then:

```
$ meson compile
$ meson test
$ meson compile coverage # or coverage-text, coverage-xml
```

[3] https://cmake.org

The generated XML file can be parsed to implement continuous integration blockers if code coverage is not found to be adequate. Therefore, having a good code coverage report is important. Configuring patch blockers that prevent reducing test coverage ensures that all analysis run on the test corpus stays meaningful.

2.2.4 Probable Fault Analysis

It is possible to automate the detection of probable faults in a code base using static and dynamic analysis tools. Running tests under those tools takes between 2x, and 10x the normal execution of a debug build.

2.2.4.1 Static Analysis

Meson integrates with clang-analyzer scan-build, and it has some partial support for clang-tidy. Any tool that can consume the compile_commands.json can be successfully used.

2.2.4.2 Dynamic Analysis

Analysis tools that work on non-instrumented binaries can use the `--wrap` option for the test runner:

```
$ meson test --wrap=valgrind testname
```

Meson supports the sanitize family of tooling available with GCC and clang out of the box through the `-Db_sanitize=option`, e.g., to use AddressSanitizer

```
$ meson <other options> -Db_sanitize=address
$ meson test <other options> <testname>
```

2.2.5 Packaging

Meson has minimal built-in support to generate RPM specfiles but no built-ins for other common targets. During Task T2.2, we will evaluate strategies to automate package creation for common distributions.

2.3 *Software Quality*

Software quality methods provide information on the safety, security, reliability, and maintainability of a codebase. Such methods provide metrics that can be computed by analysing a program's code or execution flow. The following sections contain recommendations for some programs for computing software quality metrics. The ultimate choice of software to be used is at the discretion of the developer.

2.3.1 Static Analysis

Static code analysis methods analyse code quality and detect faults before a program has been run.

2.3.1.1 Code Quality

Code quality is a set of metrics that establish the quality of a piece of code through the verification of specific properties. One such property is verbosity. Verbose code can take a long time to be read and comprehended, wasting mental energy. It is usually measured in terms of the number of code lines in a source file. The following represent common metrics in this domain.

SLOC - Source Lines of Code. The total number of lines in a file.

PLOC - Physical Lines of Code. The number of instructions and comment lines in a file.

LLOC - Logical Lines of Code. The number of logical lines (statements) in a file.

CLOC - Comment Lines of Code. The number of comment lines in a file.

BLANK - Blank Lines of Code. The number of blank lines in a file.

Reducing *SLOC*, *PLOC*, and *LLOC* metrics through refactoring guarantees less verbose code, which results in better understandability of a codebase, while a higher value for *CLOC* indicates good documentation and clarity in the most difficult parts of a code.

Another property is the structure of a code - functions and closures are analysed to evaluate their lengths, number of arguments, and the number of exit points. Metrics in this domain include:

NOM - Number of Methods. The number of methods in a file.

NARGS - Number of Arguments. The number of arguments in each method in a file.

NEXITS - The number of Exit Points. The number of exit points of each method in a file.

NARGS and *NEXITS* are intuitively linked with the ease of reading and interpreting source code - a function with a high number of arguments can be more difficult to analyse because of the higher number of possible paths. In contrast, a function with many exit points may be difficult to read.

For metrics computation, we recommend an open-source tool developed by Mozilla (Ardito, et al., 2020), called *rust-code-analysis*, because it is fast on large codebases and covers some of the most widely used programming languages.

2.3.1.2 Code Complexity

Code complexity is a measure of the complexity of maintaining a code base over a long period of time. Associated metrics provide information on the ease or difficulty of understanding the control flow of a program and the effort required to manage a codebase. Some tools even provide an estimate on the ease of introducing bugs and errors in a code.

As explained in detail in D2.1, the most well-known metrics created for these purposes include:

- *Cyclomatic Complexity*: a measure of the complexity of a method’s control flow, originally intended to identify software modules that are difficult to test or maintain (McCabe, 1976).
- *Cognitive Complexity*: evaluates the control flow of code through mathematical models that reflect programmers’ intuitions about the mental, or *cognitive* effort required to understand those flows (Campbell, 2018).
- *Halstead Suite*: After having retrieved all operands and operators present in a source code,

Halstead Suite computes a set of complexity measures that quantify, for example, the effort to manage a codebase of a determined size and volume or an estimate on the ease of introducing bugs and errors in a code.

We recommend using *rust-code-analysis* to compute code quality metrics.

2.3.2 Dynamic Analysis

Dynamic analysis (or dynamic code analysis) methods analyse software that is running. The goal of dynamic analysis is to find errors in a program while it is executing (instead of examining the code itself). Dynamic analysis techniques can identify lack of code coverage, errors in memory allocation and leaks, fault localization according to failing and passing test cases, concurrency errors (race conditions, exceptions, resource & memory leaks, and security attack vulnerabilities), performance bottlenecks and security vulnerabilities.

Software	**Analysis**	**Description**
Valgrind	Memory, Thread	Virtual Machine with in-memory binary patching
miri	Memory, Thread, Undefined Behaviour, Soundness	Rust-specific instrumentation and virtual machine
Clang/Gcc AddressSanitizer	Memory	Compiler instrumentation
Clang/Gcc UndefinedBehaviourSanitizer	Undefined Behaviour	Compiler instrumentation
Clang/Gcc ThreadSanitizer	Thread	Compiler instrumentation
kcov	Code Coverage	Relies on DWARF debugging information and kernel-specific debugging features
Clang/Gcc/Rustc gcov output and grcov/gcovr/lcov analysis	Code Coverage	Compiler instrumentation, and offline analysis

Table 1 Dynamic analysis tools

2.3.3 Code Coverage

Code coverage is a metric that can help developers understand how much source code is covered by unit tests. It can also be used to assess the quality of existing tests. A simple code coverage measure is statement coverage, which records the lines of code that were executed. Many commercial tools also analyse multiple condition coverage which is a measure of whether each logical condition in the code has been evaluated as both true and false (across multiple executions of the program). For example, the following pseudo-code:

```
if (a > 0)
  do_something();
```

should be tested with `a > 0` and with `a <= 0`.

As pointed out by R.Hamedy in[4], some common aims of code coverage analysis are:

- find out which parts of the codebase are covered by tests and which are not;
- find out which code execution paths are missed;
- a high code coverage score indicates well-written and testable code;
- a developer is more likely to write a unit test if the coverage drops;
- enforce a culture of writing unit tests using code coverage rules;
- high code coverage leads to confidence in code;
- high code coverage matters to some potential customers;
- low code coverage scores can indicate the need for code refactoring;
- code coverage can verify whether tests are executed or not.

Many commercial code coverage tools (open source or proprietary) are available for different programming languages. The following is a list of popular code coverage tools, along with their supported programming language and open-source status.

- Cobertura, Java, open-source
- Coverage.py, Python, open-source
- JaCoCo, Java, open-source
- OpenClover, Java and Groovy, open-source
- Bullseye Coverage, C/C++, proprietary
- NCover, .NET suite, proprietary
- Vector Cast C++, C/C++, proprietary
- Devel:Cover, Perl, open-source
- dotCover, .NET, proprietary
- Visual Studio, .NET, proprietary
- Istanbul, Javascript, open-source

[4] https://codeburst.io/10-reasons-why-code-coverage-matters-9a6272f224ae

2.4 *Code Certification*

Software quality certification is a procedure that a developer must undertake to guarantee to users, third parties, and other developers that software is reliable, secure, well tested, and containing readable code. To do so, we have created a traffic light-based system to assign scores to code quality as follows:

- *Red*: The software is dangerous and unreliable. Its use is not recommended.
- *Orange*: The software can be used, but it is not fully certified.
- *Green*: The software has been entirely certified, so its use is recommended.

The minimal set of requirements that the software must satisfy to obtain the *orange* colour is:

- no memory faults detected;
- no undefined behaviours;
- no known security issues;
- a specified level of code coverage.

To obtain a *green* colour, the code must have:

- no memory faults detected;
- no undefined behaviours;
- no known security issues;
- a specified level of code coverage.

Failing to meet the above requirements will cause a *red* colour to be assigned.

Software that is certified is not necessarily free of faults - it just means that the tools used for certification have not detected any problems. Some programming languages may implement out-of-the-box features that prevent certain classes of faults from being detected during certification. In those cases, developers should provide details about how the programming language specification might cause specific workflow steps to be skipped.

2.4.1 Code Coverage Mechanism

In this section, a mechanism for scoring code based on both testing coverage and complexity is defined. This mechanism is based on the following observations found while performing code coverage tests:

- Lines of code that are covered by tests, but are awarded a high code complexity score can be considered relatively safe. If bugs are introduced to these lines during refactoring, due to the high complexity score, tests will likely fail. Obviously, for better maintainability, developers should reduce the code complexity score.
- Lines of code that are not covered with tests, and are awarded high complexity scores are the

worst-case scenario. Bugs could already be present in the code, and no tests exist to verify code functionality. Additional bugs can be easily introduced during future refactoring.

- Missing cases including (i) code not covered by tests and low code complexity and (ii) code covered by tests and with low code complexity are already considered by common code coverage metrics. The former can be solved by adding tests to the uncovered lines, while the latter represents the best possible outcome.

In order to measure code complexity, source code must first be divided into *spaces.* Space is defined as any structure that can incorporate a function. The following list represents space kinds that can be found in C, C++, and Rust source files:

- *functions*
- *classes* (C++)
- *structs* (Rust, C, C++)
- *traits* (Rust)
- *impl* (Rust)
- *unit* (all languages)
- *namespace* (C++)

The described mechanism implements both cognitive and cyclomatic complexity measurements, which are intuitive and well-documented. As stated by the cognitive complexity authors[5], acceptable values for cognitive complexity scores are usually between 1 and 15, and for cyclomatic complexity scores, usually between 1 and 10, although these thresholds may vary depending on the programming language. In fact, for the C language is recommended a value less than 25. The minimum value for both instead is 1. Any other value **must** be considered as a high code complexity value and thus a complex code.

Next, we define a strategy for incorporating both code coverage and complexity scores together:

- Each covered line has a weight of 1. Lines with no coverage receive a weight of 0.
- Code complexity score is then calculated on each space. If it exceeds the code complexity threshold (either cognitive or cyclomatic), the block receives a weight of 2. If not, the block receives a weight of 1.
- The new code coverage value for a space is obtained by multiplying the sum of the code coverage weights by the code complexity weight associated with that space.
- The global code coverage value is obtained by dividing the sum of the new code coverage values by the number of physical lines in a source file (PLOC).

2.4.2 Mechanism Example

Let `foo` and `bar` be two functions (two spaces) of five lines each and written in simple pseudo-code.

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

These spaces have code complexity values of 16 and 5, and thus the code complexity weights of 2 and 1.

```
function foo() {
  instruction 1
  instruction 2
  instruction 3
}

function bar() {
  instruction 1
  instruction 2
  instruction 3
}
```

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

The new 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 new code coverage value for the `bar` space is:

```
5 * 1 = 5
```

which remains unaltered compared to the initial code coverage value.
The global code coverage value is then equal to:

```
5 / 10 = 0.5
```

where the numerator corresponds to the `bar` new code coverage value, while the denominator is the
2.5 *PLOC* metric. In this case, only 50% of the source code lines are covered.

Additional notes

This section contains notes that a developer may consider to further improve software quality and maintainability.

2.5.1 Rust

We advise considering Rust (Rust, 2021) as the primary language for new projects. Rust is a new programming language whose focus is on developing reliable and efficient systems that exploit parallelism and concurrency. Conciseness, expressiveness, and memory safety are among the principal properties that guided Rust development (Matsakis & Klock, 2014). According to a report on security vulnerabilities published by the Microsoft Security Response Centre (MSRC), about 70% of vulnerabilities are memory safety issues caused by developers who accidentally included memory corruption bugs into their C and C++ code. Rather than investing in additional tools for addressing those flaws, the use of a programming language that prevents the introduction of memory safety issues directly during feature development would be of benefit both developers and security engineers. In this way, the onus of software security is removed from the feature developer. It is put in the hands of the language developer.

Some programming languages regarded as safe from memory corruption vulnerabilities produce sub-optimal code that wastes hardware resources. Rust prevents these problems in an efficient way, as referenced by its main design goals:

- fast and memory-efficient;
- no runtime or garbage collector;
- easily integrates with other programming languages;
- guarantee memory-safe and thread-safe code, eliminating many classes of bugs at compile-time;
- useful methods to manage errors and print the relative messages in a comprehensible way;
- good documentation.

Along with the *rustc* compiler, Rust also provides a package manager called Cargo, which performs the following tasks:

1. download the dependencies of a program;
2. call *rustc* to compile the dependencies. Each dependency is compiled independently;
3. call the linker to link together all the produced objects in order to obtain the final artefact.

A Rust project can easily integrate with an existing codebase through a C API/ABI, making it easy to use the language to create new stand-alone components or rewrite old ones: -system-deps streamlines linking to external libraries, -bindgen can consume C headers to generate low-level bindings automatically, and -cargo-c provides a simple way to build rust code into a library that any C-ABI consumer can use.

Ongoing work with autocxx makes it easy to consume strictly idiomatic C++ libraries, and uniffi aims to provide automatic bindings for Swift (Swift, 2021) and Kotlin (Kotlin, 2021), targeting mobile app developers. The additional guarantees provided by the language and the work to formally prove them can provide a great starting point to build safe and trustworthy applications with less effort spent on testing. Tools such as rudra and miri are being developed to ensure that even unsafe code is automatically validated. During Task T2.2, we will actively compare similar codebases and provide automation to reduce further the setup of workflows based around the Rust ecosystem.

3 Security Label

A study conducted by researchers at Carnegie Mellon University (Emami-Naeini, Agarwal, Cranor, & Hibshi, 2020) concluded that details about security and privacy practices adopted by smart device companies are rarely made available to consumers before purchase. The study suggested attaching labels to IoT devices designed to convey information about security mechanisms, data practices, and other details such as manufacturer's country and device compatibility. Such a label could be included on a device's packaging or retrieved online by means of a QR code. Transparency is important for both customers and vendors, but it is up to vendors to decide whether they want to adopt new standards or not. Ultimately, providing more information about a product can help boost brand reputation, especially if the vendor adheres to good practices.

In a similar fashion, we would like to provide SIFIS-Home-aware applications with labels that describe security risks derived from an application's code execution. We propose a mechanism to label individual APIs. A label for the application as a whole will be thus composed of all labels associated with the APIs used within the application's code.

Our proposal is analogous to the permissions mechanism in the Android operating system (Android Permissions, 2021). Indeed, our proposed labelling mechanism resembles the Android manifest file. In SIFIS-Home, the user (or a maintainer on the user's behalf) defines policies to be enforced within the smart home. Defining policies is a simple and intuitive process - the user declares which actions and operations can be performed by applications in the smart home environment and which cannot. In SIFIS-Home, this process is conducted using novel mechanisms based on artificial intelligence and natural language processing, as described in work package WP4.

Our application label is displayed to the user during installation with an informative purpose, and it highlights if some of the risks go against the user's defined policies. If no risk contradicts the user's policies, the application is automatically installed. Otherwise, the user can decide whether to edit their policies, proceed with installation anyway, or cancel installation. This differs from Android's permission mechanisms that simply inform the user about the permissions the application requires to run all of its features. Permissions contained in the Android manifest file are just shown to the user, who oftentimes is unaware of what some of the entries mean. Recently, Android introduced runtime allowance for dangerous permissions, meaning that the user is asked to give permission within the application when first using a feature that requires such a permission. However, this approach is not convenient in our scenario since it requires user interaction which might be unfeasible in some cases.

3.1 *The SIFIS-Home Developer APIs*

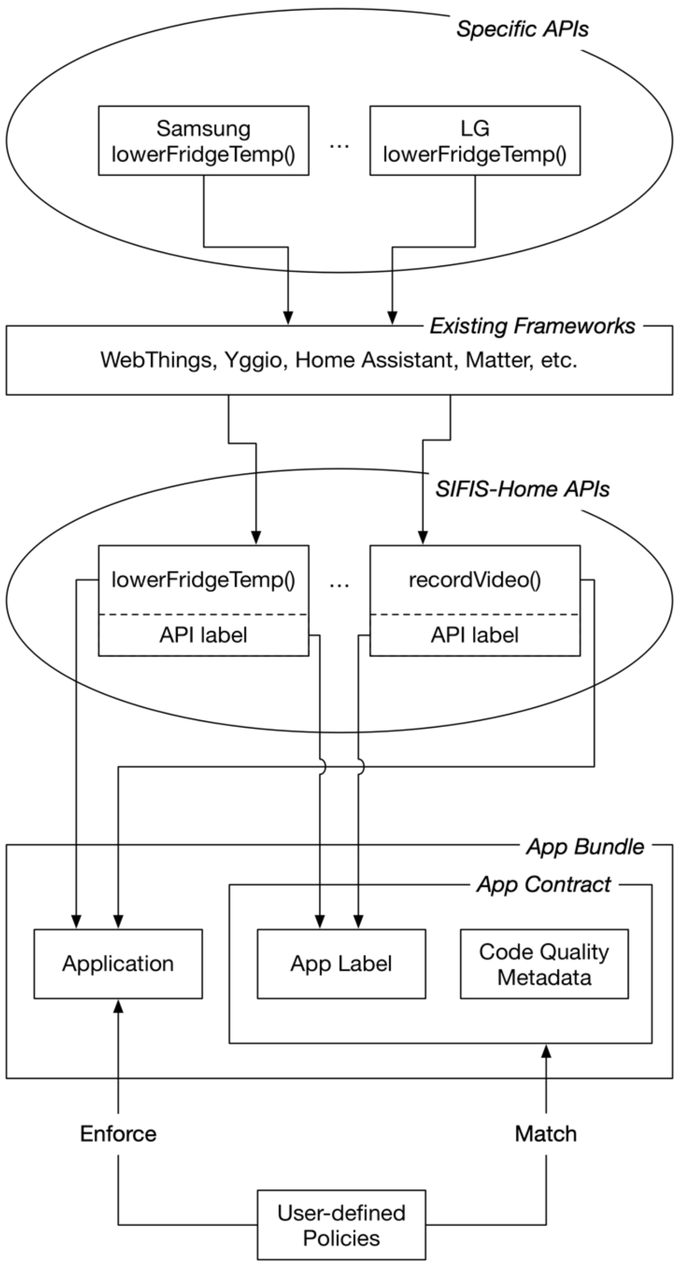

Figure 7: Architecture

As shown in Figure 7, the SIFIS-Home Developer APIs are designed to extend and improve service level APIs such as those offered by WebThings and Yggio. The SIFIS-Home developers APIs build upon this 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”, and “Increase Temperature”. Following the Web of Things terminology, we name these services “Capabilities”. Capabilities help developers of third-party applications 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 refrigerator manufacturers provide two different API implementations to decrease the current temperature in the refrigerator by 1 °C. To offer this API to third-party developers, not 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 API call and without needing to determine which device they are talking to and invoke the device-specific implementation of it.

To define the SIFIS-Home developer APIs, we build upon currently existing frameworks, focusing on Web of Things and FIWARE[6]. Since SIFIS-Home is focused on the security and safety aspects of smart home management, it is not in the scope of our activities to develop new standards. Since many IoT standards are new and only a few, basic capabilities have actually been defined, SIFIS-Home draws from other non-standardised frameworks such as IFTTT[7], Home Assistant, and OpenHab[8] to define some additional capabilities useful for representing desired features and functionalities. As defined by Web of Things and FIWARE, new capabilities can be proposed by device producers and application developers, to represent functionalities that can be offered to third-party applications.

To be able to handle the privacy, safety, and security issues, within the activities of Task T2.3, we have defined a set of "tags" representing safety, integrity, security, and privacy issues intrinsically related to the execution of each specific developer API. Such risks are generally related to either misuse or malicious use of functionality, e.g., decreasing the refrigerator temperature excessively to cause greater energy consumption. The user must be informed of this possibility when installing an application on SIFIS-Home devices, and they must have the opportunity of controlling the execution of such risky operations, by means of security and safety policies. As described in deliverable D1.1, this can be achieved by means of security and safety policies, which can be defined either by the user themself or by an external, expert maintainer. By binding the labels to specific APIs (*API label* in Figure 7) we ensure that if an API is invoked, the corresponding *API label* is associated with the application, in a similar way to how Android permissions are handled. The application will thus have an application label (*App Label*) associated with it that is derived from the combination of the API labels invoked by the application's source code. The *App Label*, together with the code quality information provided by Tasks 2.1 and 2.2, and other metadata, define an *App Contract*.

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

The following describes the process of defining *API labels* for SIFIS-Home developer APIs and

[6] https://www.fiware.org
[7] https://ifttt.com
[8] https://www.openhab.org

discusses 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 in a similar way to CE conformance marking (CE conformance marking, 2021) - depending on the API in use, a self-assessment would be sufficient to enter the SIFIS-marketplaces. Dangerous APIs 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.

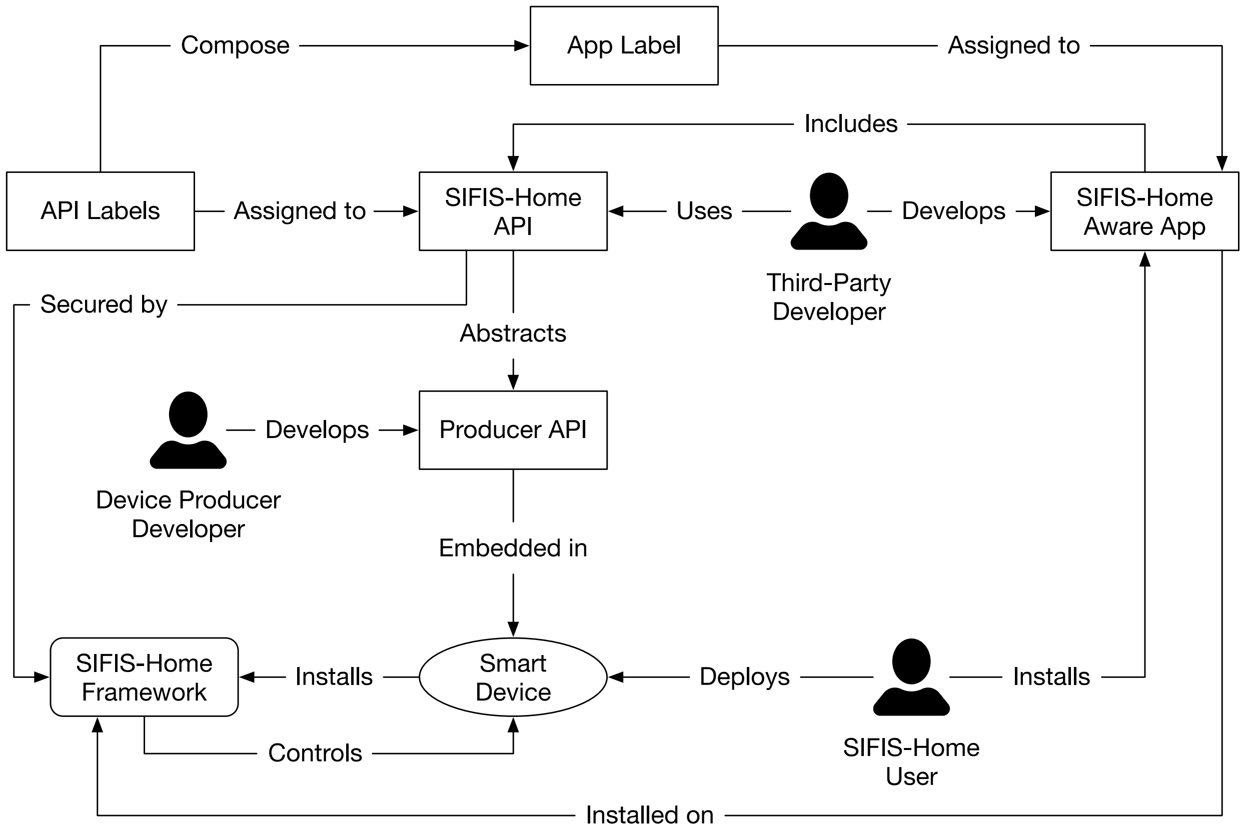

Figure 8: SIFIS-Home APIs integration and interaction with other components

Figure 8 illustrates how the SIFIS-Home APIs relate to various components of the architecture. An *API label* is assigned to a SIFIS-Home API; the SIFIS-Home aware app code includes SIFIS-Home APIs, whose *API labels* contribute to the *App Label.*

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

The following demonstrates a pseudocode example of the SIFIS-Home API `SIFIS-LowerFridgeTemp()`.

```
SIFIS-LowerFridgeTemp(){
  makeSecure();
  WoT-LowerFridgeTemp(){
    linkToProducerLowerFridgeTemp();
  }
}
```

The `makeSecure()` method implements security checks that are performed before executing the actual capability.

3.2 *Labelling Mechanism*

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

Safety risks occur when events produce a direct physical effect. APIs that trigger actuators are associated with this kind of risk. Indeed, smart home environments may include appliances that can 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 a 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 retrieve actuator states, e.g., on/off state of a light bulb, are straightforward examples. APIs that collect auxiliary data, such as logs, also fall into this category.

Financial risks are related to operations that generate a monetary expense, either directly or indirectly. APIs that access a user's "wallet" to place an order or pay a subscription fee are examples of direct financial risks. Indirect financial risks refer to operations that generate an indirect monetary cost for the user. These can include operations that affect the consumption of electricity, gas, or water. The extent of the risk differs from API to API and from device to device.

In SIFIS-Home, we add a *security label* to every API to describe possible risks deriving from its execution. The security label consists of a list of *tags*, each identifying a risk. A tag contains (i) the risk name, (ii) a description, and, optionally, (iii) a *risk score*. *Risk score* is a decimal value between 0 and 1. For example, an API that can be used to turn on an oven performs an operation that consumes high instantaneous power and could potentially set the house on fire, so its label will include, among others, the tag "FIRE_HAZARD" and the tag "ELECTRIC_ENERGY_CONSUMPTION" with a risk score of 0.8. An API that acquires feed from a video camera and stores it locally may store images of children, which could represent a privacy concern for the end user; and will thus receive a tag "CHILDREN_RECORDING". An API that authorises the payment of an asset will have a label including the tag "SPEND_MONEY".

Developers use SIFIS-Home 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 an application (executable) and an app contract, which consists of an app label and code quality metadata. An *App Label* is automatically generated during the packaging phase and is populated with all *API labels* associated with APIs used.

When a user wishes to install an app from the SIFIS-Home marketplace, the *App Label* will inform them about possible risks deriving from the installation and usage of the app. For each risk listed in the *App Label*, a user-friendly description, and a *risk score*, when applicable, is provided. A short and simple description of all risks is required to promote the reading and comprehension by every class of end users. Moreover, *risk scores*, which are decimal values, can 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 an app's behaviour and possible risks, the *App Label* seamlessly integrates with user-defined policies. This means that if the label of a given API would violate the rules defined by the user, its execution will be automatically denied. For example, if a user has defined a policy which reads as "No device that may cause a fire can be turned on remotely", and the *App Label* contains the `turnOnOven` API, the app can be installed, but the execution of that API is forbidden at runtime if the initiator is outside the local perimeter.

Tags

3.3 Table 2 illustrates a non-exhaustive list of tags and their descriptions for the three risk categories. The symbol ⚠ denotes that a risk score is associated with the tag it is defined in.

Safety
FIRE_HAZARD *The execution may cause fire.*
AIR_POISONING *The execution may release toxic gases.*
EXPLOSION *The execution may cause an explosion.*
ASPHYXIA *The execution may cause oxygen deficiency by gaseous substances.*
WATER_FLOODING *The execution allows water usage which may lead to flood.*
POWER_OUTAGE — ⚠ *The execution may cause an interruption in the supply of electricity.*
POWER_SURGE *The execution may lead to exposure to high voltages.*
UNAUTHORISED_PHYSICAL_ACCESS *The execution disables a protection mechanism and unauthorised individuals may physically enter home.*
SPOILED_FOOD *The execution may lead to rotten food.*

Privacy
AUDIO_VIDEO_STREAM *The execution authorises the app to obtain a video stream with audio.*
AUDIO_VIDEO_RECORD_AND_STORE *The execution authorises the app to record and save a video with audio on persistent storage.*
LOGGING_USAGE_TIME *The execution authorises the app to get and save information about the app's duration of use.*
LOG_ENERGY_CONSUMPTION *The execution authorises the app to get and save information about the app's energy impact on the device the app runs on.*
RECORD_USER_PREFERENCES *The execution authorises the app to get and save information about the user's preferences.*
RECORD_ISSUED_COMMANDS *The execution authorises the app to get and save user inputs.*
TAKE_PICTURES *The execution authorises the app to use a camera and take photos.*
TAKE_DEVICE_SCREENSHOTS *The execution authorises the app to read the display output and take screenshots of it.*

Financial
SPEND_MONEY *The execution authorises the app to use payment information and make a payment transaction.*
PAY_SUBSCRIPTION_FEE *The execution authorises the app to use payment information and make a periodic payment.*
ELECTRIC_ENERGY_CONSUMPTION — ⚠ *The execution enables a device that consumes electricity.*
GAS_CONSUMPTION — ⚠ *The execution enables a device that consumes gas.*
WATER_CONSUMPTION — ⚠ *The execution enables a device that consumes water.*

Table 2: Sample lists of tags for the three categories

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

API Labels

Table 3 illustrates some sample APIs and their own labels.

API	Label
`turnOnOven`	• FIRE_HAZARD • POWER_OUTAGE (risk score: 0.8) • LOG_ENERGY_CONSUMPTION • ELECTRIC_ENERGY_CONSUMPTION (risk score: 0.8)
`recordVideo`	• AUDIO_VIDEO_RECORD_AND_STORE
`lowerFridgeTemperature`	• POWER_OUTAGE (risk score: 0.5) • ELECTRIC_ENERGY_CONSUMPTION (risk score: 0.5)
`raiseFridgeTemperature`	• SPOILED_FOOD
`orderFood`	• SEND_MONEY
`turnOnAirConditioner`	• POWER_OUTAGE (risk score: 0.7) • ELECTRIC_ENERGY_CONSUMPTION (risk score: 0.7)
`turnOnVacuumCleaner`	• POWER_OUTAGE (risk score: 0.8) • LONG_LASTING_RESOURCE_LOCK • ELECTRIC_ENERGY_CONSUMPTION (risk score: 0.8)
`disarmAlarm`	• UNAUTHORISED_PHYSICAL_ACCESS
`openShutters`	• UNAUTHORISED_PHYSICAL_ACCESS
`streamMicAudio`	• TENANTS_VOICE_STREAM • CHILDREN_VOICE_STREAM
`unlockDoor`	• UNAUTHORISED_PHYSICAL_ACCESS
`renewSubscription`	• PAY_SUBSCRIPTION_FEE

Table 3: Sample list of APIs and their API labels

3.5 *Label Format*

Both *API labels* and *App Labels* should be implemented so that they can be easily converted into other formats and exported, namely they need to be serializable. Possible serialization formats include JSON, XML, and TOML. The following describes an implementation of a JSON schema for both *API labels* and *App Labels*.

3.5.1 JSON Format

This section introduces an *API label* JSON format via an example, and then it defines an *API label* schema and an *App Label* schema.

The example is given for the `turnOnOven` API. The following JSON object contains three properties: (i) `api_name`, which must match the API the label refers to, i.e., `turnOnOven`; (ii) `description`, which gives a brief explanation of the API behaviour; and (iii) `security_label`, which specifies the risks associated with the API.

The `security_label` property is an object that contains three properties representing the `safety`, `privacy`, and `financial` categories. Each property contains an array of objects with the same structure, each representing a tag. These objects identify 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 tags POWER_OUTAGE, which also reports the risk score, and FIRE_HAZARD.

```
{
  "api_name": "turnOnOven",
  "description": "Activates the oven at the last selected temperature.",
  "security_label": {
    "safety": [
      {
        "name": "FIRE_HAZARD",
        "description": "The execution may cause fire."
      },
      {
        "name": "POWER_OUTAGE",
        "description": "High instantaneous power. The execution may cause power o
utage.",
        "risk_score": 0.8
      }
    ],
    "privacy": [
      {
        "name": "LOG_ENERGY_CONSUMPTION",
        "description": "The execution allows the app to register information abou
t energy consumption."
      }
    ],
    "financial": [
      {
        "name": "ELECTRIC_ENERGY_CONSUMPTION",
        "description": "The execution enables the device to consume further elect
ricity.",
        "risk_score": 0.8
      }
    ]
  }
}
```

3.5.1.1 API Label Schema

The reference JSON schema for an *API label* is presented next. This schema bundles [JSON Bundle 2021] the *API label* schema and a tag subschema into a single schema. The tag subschema defines the `tag` object, which is used by all the risk categories (`safety`, `privacy`, and `financial` properties).

The `tag` object contains two required properties: (i) `name`, which must match one of the tags defined in the tags list, and (ii) `description`, which gives a brief explanation of the risk. Additionally, the `tag` may contain the `risk_score` property, indicating the gravity of the risk, defined as a number between 0 and 1, with a step size of 0.1.

```
{
  "$schema":"http://json-schema.org/draft-07/schema#",
  "$id":"https://raw.githubusercontent.com/sifis-home/wp2-documents/master/wp2-de
liverable-2.2/schemas/api-label.jschema",
  "title":"SIFIS-Home API label schema.",
  "description":"JSON schema defining the API security label structure within the
SIFIS-Home framework.",
  "type":"object",
  "properties":{
    "api_name":{
      "type":"string"
    },
    "description":{
      "type":"string"
    },
    "security_label":{
      "type":"object",
      "properties":{
        "safety":{
          "type":"array",
          "items":{
            "$ref":"/schemas/tag"
          }
        },
        "privacy":{
          "type":"array",
          "items":{
            "$ref":"/schemas/tag"
          }
        },
        "financial":{
          "type":"array",
          "items":{
            "$ref":"/schemas/tag"
          }
        }
      },
      "required":[
        "safety",
        "privacy",
        "financial"
      ]
    }
  },
  "required":[
    "api_name",
    "description",
    "security_label"
  ],

  "definitions":{
    "tag":{
      "$id":"/schemas/tag",
      "type":"object",
      "properties":{
        "name":{
          "type":"string"
        },
        "description":{
          "type":"string"
        },
        "risk_score":{
          "type":"number",
          "minimum":0,
          "maximum":1,
          "multipleOf":0.1
        }
      },
      "required":[
        "name",
        "description"
      ]
    }
  }
}
```

3.5.1.2 App Label Schema

The *App Label* JSON contains an array of *API labels*, such as the one defined in the example above. The *App Label schema* is defined below. This schema declares three required properties, i.e., `app_name`, `description`, and `api_labels`. The latter is an array of `api-label` objects.

```
{
  "$schema":"http://json-schema.org/draft-07/schema#",
  "$id":"https://raw.githubusercontent.com/sifis-home/wp2-documents/master/wp2-de
liverable-2.2/schemas/app-label.jschema",
  "title":"SIFIS-Home app label schema.",
  "description":"JSON schema defining the app label structure within the SIFIS-Ho
me framework.",
  "type":"object",
  "properties":{
    "app_name":{
      "type":"string"
    },
    "description":{
      "type":"string"
    },
    "api_labels":{
      "type":"array",
      "items":{
        "$ref":"/sifis-home/wp2-documents/master/wp2-deliverable-2.2/schemas/api-
label.jschema"
      }
    }
  },
  "required":[
    "app_name",
    "description",
    "api_labels"
  ]
}
```

4 Legal Guidelines (Licensing and Privacy)

Different subjects may be obliged to, or interested in complying with privacy laws and, therefore, may need to adopt specific standards. In a situation where data is collected through smart-home systems, the *Data Controller* of processing performed by software interacting with IoT devices, according to the GDPR, could be the owner of the house (or the tenant), who willingly installed IoT devices in the house. However, *the Data Controller* may also be a Software as a Service (SaaS) provider, who obtains and stores personal data from IoT devices. While following privacy rules is not mandatory for software developers (or application designers), it is advantageous for them to comply with privacy rules and follow current standards, since compliance to privacy rules means that the software can be more easily reviewed and accepted by both the end users and SaaS providers who want to provide services based on the application designed by the developer. For this reason, we propose some insight into following best practices for shipping software that is not only privacy-compliant but also based on state-of-the-art approaches to 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 legal obligations arising from free and open-source software.

4.1 *Privacy*

Rules provided by GDPR are designed to be followed by software developers and publishers. In particular, SaaS providers must satisfy accountability requirements and must perform quantitative risk assessment analyses. The requirements described in articles 13 (EU GDPR, Art.13, 2021) and 14(EU GDPR, Art.14, 2021) of the GDPR require information to be provided to the data subject, even if personal data is not collected. The required information includes:

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 of time in which personal data will be kept;

f) the right for the data subject to obtain a modification or cancellation of their data and the practical ways in which they can exercise this right;

g) the existence of automated decision-making, including profiling.

Additionally, the following guidelines should be followed:

- **Article 25** (EU GDPR, Art.25, 2021) asks that the Data Controller implement appropriate technical and organisational measures designed to enforce data-protection principles, such as data minimisation compatible with the cost of the implementation and the nature of the processing. This “data protection by design” principle combines with the concept of “data protection by default”, which mandates the controller to implement other technical and organisational measures to ensure that. By default, only personal data which is necessary for each specific purpose of the processing is processed.

- **Article 32** (EU GDPR, Art.32, 2021) sets requirements on the matter of security (obligation to implement appropriate technical and organisational measures to ensure a level of security appropriate to the risk).
- **Article 35** (EU GDPR, Art.35, 2021) mandates the controller to carry out a Privacy Impact Assessment when the processing is likely to result in a high 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 probably that found in Article 35. To aid Data Controllers in building and demonstrating compliance to the GDPR, the French CNIL[9] created a useful tool[10] that has quickly become a reference standard. Using this tool can help the developer to understand the security and privacy risks posed by their software and may give them the chance to solve any identified issues before shipping the product to the public. It is therefore recommended to use this tool to ensure software's compliance with GDPR rules.

Figure 9 shows the flowchart to obtain a privacy-compliant software by following GDPR requirements and recommendations.

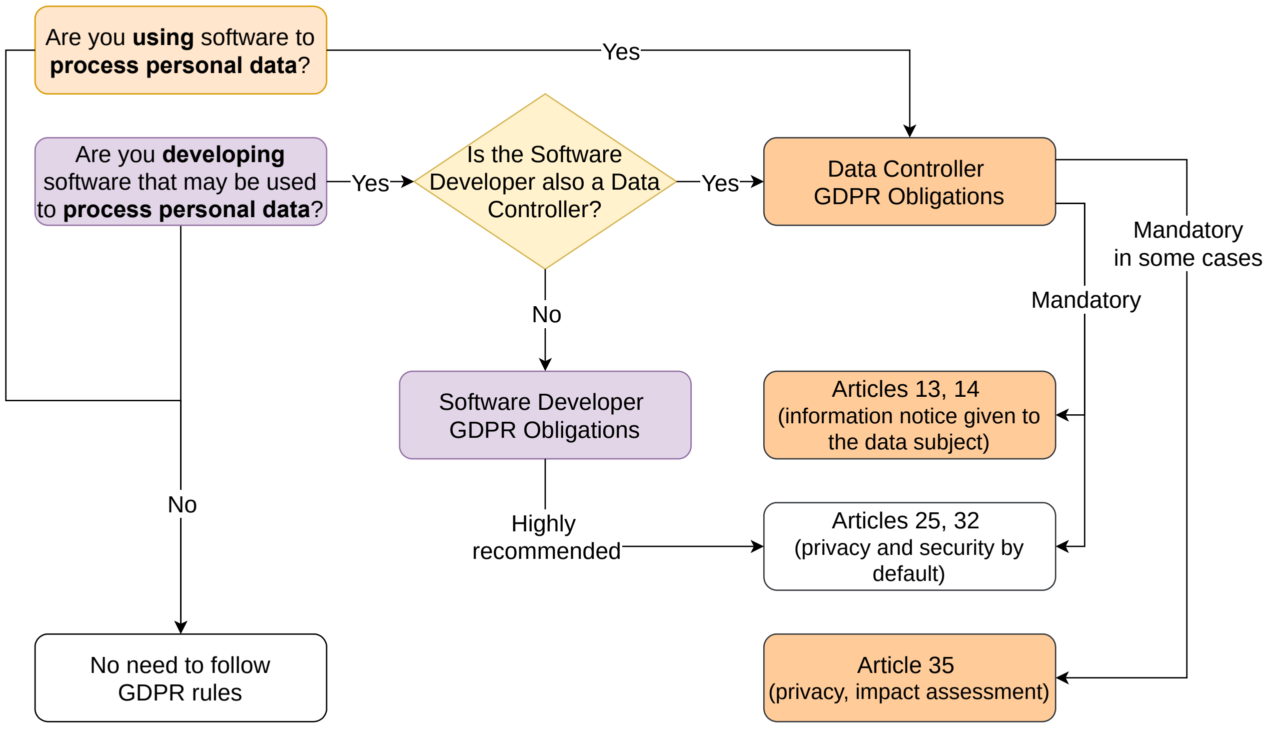
Are you **using** software to **process personal data**?
Yes
Are you **developing** software that may be used to **process personal data**?
Yes
Is the Software Developer also a Data Controller?
Yes
Data Controller GDPR Obligations
Mandatory in some cases
Mandatory
No
Software Developer GDPR Obligations
Articles 13, 14 (information notice given to the data subject)
No
Highly recommended
Articles 25, 32 (privacy and security by default)
No need to follow GDPR rules
Article 35 (privacy, impact assessment)

Figure 9: Main GDPR requirements and recommendations flowchart

4.2 *Licensing*

[9] CNIL is France's independent administrative regulatory body to ensure that data privacy law is enforced in the French territories.

[10] Downloadable on the official CNIL's website: https://www.cnil.fr/en/open-source-pia-software-helps-carry-out-data-protection-impact-assesment

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[11] and the ClearlyDefined[12] projects. These initiatives aim at helping free and open-source software to become more standardized and well defined, clearing doubts regarding legal compliance and providing developers with clear and comprehensive information that will inform them of 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 aims to "establish requirements to achieve effective management of free and 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 these guidelines, a software developer can send a document to the OpenChain organisation affirming that their software satisfies all the requirements of the specification and is therefore compliant with the OpenChain standard[13]. 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 artefacts - "a collection of artefacts 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 relatively new, but already offers some suggestions on how to distribute clearly defined open-source software, so that users and other developers are clearly informed about aspects such as the type of open-source license used, 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 on the matter of security and vulnerabilities that may be present in their software[14]. Following both the OpenChain and the ClearlyDefined practices gives the software important certification that can aid in its diffusion.

4.3 *Highlights*

To summarise the privacy and licensing legal requirements and consider the usefulness of following some standard practices, we propose the following "traffic light system" to assess whether one's software is compliant with the concepts exposed in the previous paragraphs.

4.3.1 Green Light

[11] See https://www.openchainproject.org/resources/faq

[12] See https://clearlydefined.io/about

[13] These requirements are found in the Supplier Education Pack (permanent link: https://github.com/OpenChain-Project/Reference-Material/blob/eebf7cdc873691f89a1765425de4f456f0f41988/OpenChain-ISO-5230-Supplier-Education-Pack/en/OpenChain%20ISO%205230%20Supplier%20Education%20Pack.zip) downloadable on the official OpenChain website, and in particular in the "basic-open-source-education" pdf file within the zip archive.

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

1. The software developer successfully has performed a privacy impact assessment based on reasonable assumptions for at least a standard use case, and the documentation of this assessment accompanies the software[15].
2. The software developer has produced information to be used for compliance to articles 13, 14, 25, and 32 of GDPR, and a document containing the information accompanies the software.
3. The software developer followed the OpenChain specification or other public specifications for licensing compliance, and the software is accompanied by compliance artefacts.
4. The methodology used and standards followed in creating a privacy impact assessment, the document produced according to point 2 and the compliance artefacts, is publicly available, free of any right of third party, so that everyone can assess compliance and use it.

Green-lighted products ensure that a Data Controller can assess if the software is compliant with GDPR and free and open-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 information required to perform a privacy impact assessment.
2. The software developer declares that they have produced information to be used to comply with articles 13, 14, 25, and 32 of GDPR.
3. The software developer declares that they have compliance artefacts.
4. The software developer declares that the methodology used, and the standards followed in creating the above documentation can be made available in order to assess compliance.

Yellow-lighted products allow the Data Controller 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 to process personal data but some additional work will be required.

4.3.3 Red Light

One or more of the points provided for the Yellow Light is not satisfied. This software must be carefully analysed 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.

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

5 Conclusions

This document introduced a series of developer guidelines for the production of secure, privacy-aware and policy-based IoT software.

The first section of this document described a workflow to assist developers in improving the quality of their software. It also provided an example of this workflow based on the C language. It finally presented procedures for automatically certifying software quality, and some notes that a developer might consider for further improving software quality and maintainability.

The second section of this document described the SIFIS-Home developer APIs and their relation to other architectural components. The concept of an *API label*, which describes safety, privacy, and financial risks associated with an API was introduced, along with a sample list for each category, and a concrete example of an *API label*. The concept of an *App Label*, derived from *API labels* within the application code, and designed to integrate with user-defined policies so that the execution of an API can be allowed or denied at runtime was introduced. Finally, the schemes of both *API labels* and *App Labels* in JSON format were described.

In the third and final section of this document, EU legal guidelines for compliance with privacy laws and standards were presented. These laws and guidelines are essential for developers of smart home applications that collect personal data.

6 References

[*Android Permissions*, 2021] Retrieved from Permissions on Android: https://developer.android.com/guide/topics/permissions/overview

[Ardito, 2020] Ardito, L., Barbato, L., Castelluccio, M., Coppola, R., Denizet, C., Ledru, S., & Valsesia, M. (2020). rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes. *SoftwareX*.

[Campbel, 2018] Campbell, G. A. . Cognitive Complexity - A new way of measuring understandability. *SonarSource SA*, 10.

[Marking, 2021] *CE conformance marking*. (2021). Retrieved from https://ec.europa.eu/growth/single-market/ce-marking/manufacturers_en

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

EU GDPR, Art.13. (2021). Retrieved from https://www.privacy-regulation.eu/en/13.htm

EU GDPR, Art.14. (2021). Retrieved from https://www.privacy-regulation.eu/en/14.htm

EU GDPR, Art.25. (2021). Retrieved from https://www.privacy-regulation.eu/en/25.htm

EU GDPR, Art.32. (2021). Retrieved from https://www.privacy-regulation.eu/en/32.htm

EU GDPR, Art.35. (2021). Retrieved from https://www.privacy-regulation.eu/en/35.htm

[*JSON Schemas*, 2021] Retrieved from Structuring a complex schema: https://json-schema.org/understanding-json-schema/structuring.html#bundling

[*Kotlin*. 2021] Retrieved from https://kotlinlang.org

[Matsakis, N. D., & Klock, F. S., 2014]. The rust language. *ACM SIGAda Ada Letters*, 103-104.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*.

[*Meson*, 2021] Retrieved from The Meson Build system: https://mesonbuild.com

[*Ninja*, 2021] Retrieved from https://ninja-build.org

[*Rust*, 2021] Retrieved from https://www.rust-lang.org

[*Swift*. 2021]. Retrieved from https://swift.org/about/#swiftorg-and-open-source

Glossary

Acronym	Definition
ABI	Application Binary Interface
API	Application Programming Interface
CE	Conformité Européene
CNIL	Commission Nationale de l'Informatique et des Libertés
CPU	Central Processing Unit
DWARF	Debugging With Arbitrary Record Formats
EU	European Union
GCC	GNU Compiler Collection
GDPR	General Data Protection Regulation
IDE	Integrated Development Environment
IFTTT	If This Then That
IoT	Internet of Things
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
MSRC	Microsoft Security Response Center
QR	Quick Response
RPM	RPM Package Manager
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