D2.5

Final Version of Developer Tools

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

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

Due date of deliverable: 31/03/2023
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), Maurizio Morisio (POL), Marco Torchiano (POL), Marco Rasori (CNR), Michele Valsesia (POL)

Reviewers: Andrea Saracino (CNR), Valerio Frascolla (INT)

Revision History

Version	Date	Name	Partner	Section Affected Comments
0.1	24/02/2023	Tentative ToC and contents	POL, LUM, CNR	All
0.2	11/03/2023	Software Lifecicle Tools description	POL, LUM	Section 2
0.3	13/03/2023	API Labelling	CNR, LUM	Section 3
0.4	17/03/2023	WoT Example	LUM	Section 4
0.5	20/03/2023	Conclusions	POL	Section 5
0.6	23/03/2023	Final proofread	POL	All
1.0	30/03/2023	Final Release	POL	All

Executive Summary

This document is the companion to the final release of the 'developer tools' code, developed in the context of Work Package (WP) 2, available on the SIFIS-Home project GitHub repository https://github.com/sifis-home. All the source code is released under the MIT licence and has documentation directly from the repository.

This document provides the reader with detailed information about the released code, the API labelling, and an example of use.

Table of contents

1 Introduction

In software development, finding the various issues that could affect code and binaries is extremely important. Catching bad code or security vulnerabilities needs setting up a workflow running a series of tools; each focused on detecting a specific problem that could arise while coding or producing the final binary. This workflow has been explained in detail in deliverable D2.4, but it requires a developer to make a remarkable effort to build that up, thus subtracting time to the fundamental code phase. For this reason, we implemented a tool that accompanies a developer in building this workflow automatically through the creation of an initial set-up that is different depending on the chosen programming language. This set up also allows to retrieve information about code status in addition to metrics related to the security of software. Tools that compose this workflow are very varied and we have contributed to their development starting from our needs, but at the same time, fixing their issues or cleaning up their code a bit. We have also implemented a new tool that extracts complex code snippets using metrics which are able to compute code complexity. Another tool, instead, allows to detect which code parts are both not covered by tests and complex, thus it implements the code coverage model explained in deliverable D2.4. This helps a developer to add tests to the most critical zones. The main goal of these tools would be that of streamlining processes, lower debugging costs, and achieve more secure, functional, usable, available, efficient, and maintainable software.

In the context of the hazards related to the execution of risky operations, such as turning an oven on, we developed some effective tools. As described in deliverable D2.4, the SIFIS-Home developer APIs are associated with API labels which include information about the risks brought by each API. Such information is shown to third-party developer as part of an autocompletion plug-in within their IDE.
All the API Labels of the APIs used within the application code contribute to the generation of the App Label. To this aim, a tool called *manifest* which operates directly on the application binary, has been developed. Another tool, called *sifis-xacml* extracts XACML requests from the App Label; this tool has been developed as part of a workflow to enforce a Security by Contract methodology executed during application installation performed by the user. Moreover, we defined an ontology to associate the hazards with the properties of the Thing Descriptions, which have a one-to-one mapping with the SIFIS-Home developer APIs. Since each device is different from another, the specific hazards and levels of risk may vary per device basis.

As part of the process of testing our practices and tools we applied them while implementing our Web of Things implementation in Rust, we used **sifis-generate** to generate the continuous integration workflow, experimented with **conventional commits** and **REUSE** and used the **weighted-code-coverage** and **complex-code-spotter** tools. As part of the assessment of dependencies process, we found a weak/orphaned component and we decided to incubate it renaming it as **datta**.
In the following table, we list the status of the released tools, and in the following chapters we provide further information about their usage.

Table 1. Status of released tools.

Project name	**Status**
wot-td	**Release 0.3**
wot-serve	**Release 0.3**
wot-discovery	**Release 0.2**
sifis-generate	**Release 0.5**
weighted-code-coverage	**Release 0.2**

complex-code-spotter	**Release 0.1**
datta	**Release 0.1**
The SIFIS-Home Hazards Ontology	**Release 1.0.3**
sifis-api	**Release 0.1**
demo-things	**Release 0.1**
wot-test	**Release 0.1.0**
sifis-xacml	**Release 0.1.0**

2 Software Lifecycle Tools

When we interviewed our project partners, they declared that our workflow, with its Continuous Integration system, is often too time-consuming to be set manually in their projects. Since our main goal consists in accompanying a developer during software development, thus reducing the effort of performing checks as much as possible, we have developed tools to automatize the entire process and some of its steps.

Among the tools, we have made to make our workflow more time efficient and less resources-hungry, but also to provide some information about code status, we have implemented:

1. **sifis-generate**: A project creation tool that automatically sets up an initial project equipped with our Continuous Integration workflow. Supported project types, and their features, have been chosen depending on the programming languages used and requested by SIFIS-Home partners.
2. **complex-code-spotter**: A tool that helps detecting the most complex parts of a codebase through complex code metrics. A code with high complexity is difficult to maintain and could hide unseen bugs.
3. **weighted-code-coverage:** A tool that implements various weighted code coverage algorithms, the one described in D2.4, in addition to other ones taken from the Ruby world. It can identify code parts which are both complex and without any code coverage, thus allowing a developer to choose which code zones to put effort into and reducing the time to write tests.

In addition to these new tools, we have contributed to some of the open-source projects integrated into our workflow, implementing features and fixing bugs. Notably, we have not just limited ourselves to them. Still, we have supported third-party libraries used by the workflow's tool adding scripts to their Continuous Integration systems to deploy binaries on different architectures and publish libraries into the Rust package registry called *crates.io*. In the next sections, we will explain the meaning of these contributions and how we have done that. Still, to name one, we have performed a conspicuous refactoring of tests of a Mozilla project for static code analysis computation called **rust-code-analysis**.

2.1 *Implemented tools*

All implemented tools have been written in Rust because of the advantages of this language, which, considering our needs, can be listed as:

- Memory safety without the need of a garbage collection
- Eliminating many classes of bugs at compile-time
- Possibility to optimise software both in time and memory
- Useful methods to manage errors and print the relative messages in a comprehensible way
- Writing parallel code in an easier way
- Package and deploy software in a few steps
- Easily integrate with other programming languages
- Support the most known platforms, such as Linux, macOS, Windows and bare-metal targets

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 software dependencies. Each dependency is compiled in an independent way from the other ones.
3. Call the linker to link together all the produced objects and obtain the final artifact.

Cargo can also define subcommands, which are external executables, directly callable from its command line interface.

Another consistent feature of the Rust language is its good documentation that guarantees a better software maintainability and an easier possibility for contributors to add features to the main code.
For what concerns interoperability, a Rust project can easily integrate with an existing codebase through a C API/ABI, making it easy to use the language for creating new stand-alone components or rewrite old ones.

As a glimpse into the world, according to the report on the security vulnerabilities published by the Microsoft Security Response Centre (MSRC), about 70% of those vulnerabilities are memory safety issues caused by developers who inadvertently insert memory corruption bugs into their C and C++ code.
Rather than investing in more tools for addressing those flaws, the use of a programming language that prevents the introduction of memory safety issues into a feature work directly during its development would help both the feature developers and the security engineers. In this way, the onus of software security is removed from the feature developer and put in the hand of the language developer. These observations have also helped choose Rust as a language for implementing our tools.

Each tool is composed by a library and a binary. The library contains all tool features as public APIs, while the binary exposes them in a straightforward way through shell. This kind of structure allows to interact with both in a modular way, thus the addition of new features to one side does not require any change in the other. Furthermore, this model favours external developers to construct their own binary starting from library’s API. Having two crates also helps in building them with different optimizations in time and memory depending on the context to be considered.

We have designed *sifis-generate* after some interviews with our partners which have pointed out the difficulty of implementing our workflow manually. As for *complex-code-spotter* instead, we have noticed, through an analysis of our repositories, how hard it might be the comprehension of code at the first impression, so a tool to identify the most complex code areas was a necessity. For what concerns *weighted-code-coverage*, we needed a tool that was able to detect both uncovered and complex functions to focus most of a developer effort in creating tests for those unsecure and hiding-bugs parts.

To better describe our tools, we have divided their descriptions into different sections, so to have a better understanding of the rationale we have used to implement them. The contributions made to third-party components of a crate are represented by light violet boxes in the following figures, and they will be explained in a specific section of the document.

2.2 *sifis-generate*

sifis-generate primary purpose consists in reducing the conceptual effort requested by developers to set up a project and the GitHub Actions workflow described in deliverable D2.4.
When an automatized project generator is either not available or too complex to use, *sifis-generate* generates a whole new project from scratch, while in all other cases, it adds GitHub Actions scripts to an existing project. GitHub Actions scripts contain our workflow adapted according to the programming language and build system in use, so some layers might not be present or be different due to some optimizations and context choices.
This tool must be considered as a starting point for a developer who wishes to implement new software, so the simplest programming language patterns, notions and paradigms have been used for projects definition. Its structure is visible in Figure 1 alongside with third-party library and cli components.

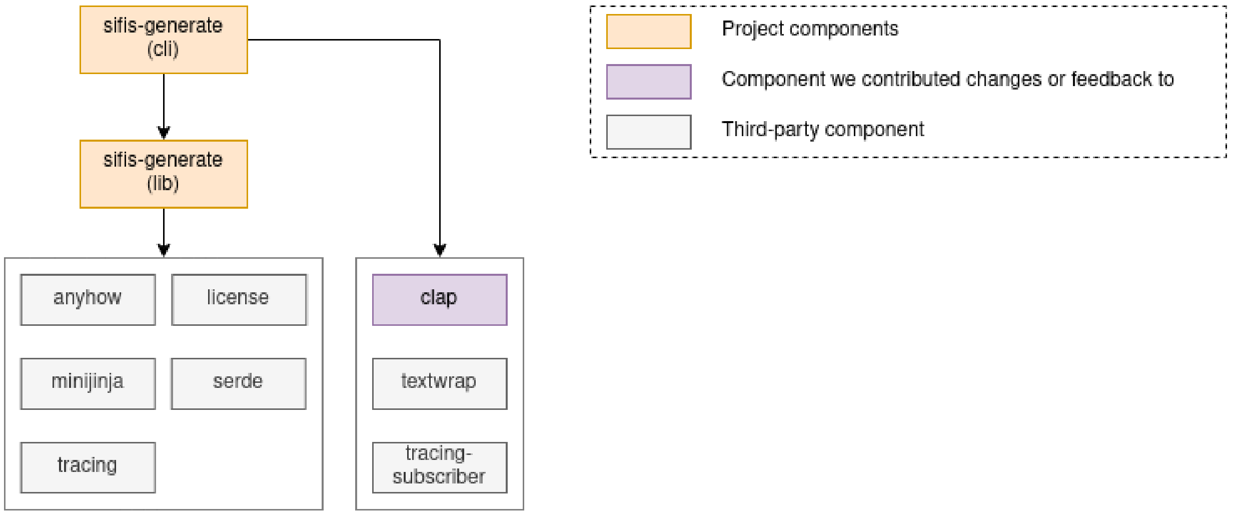
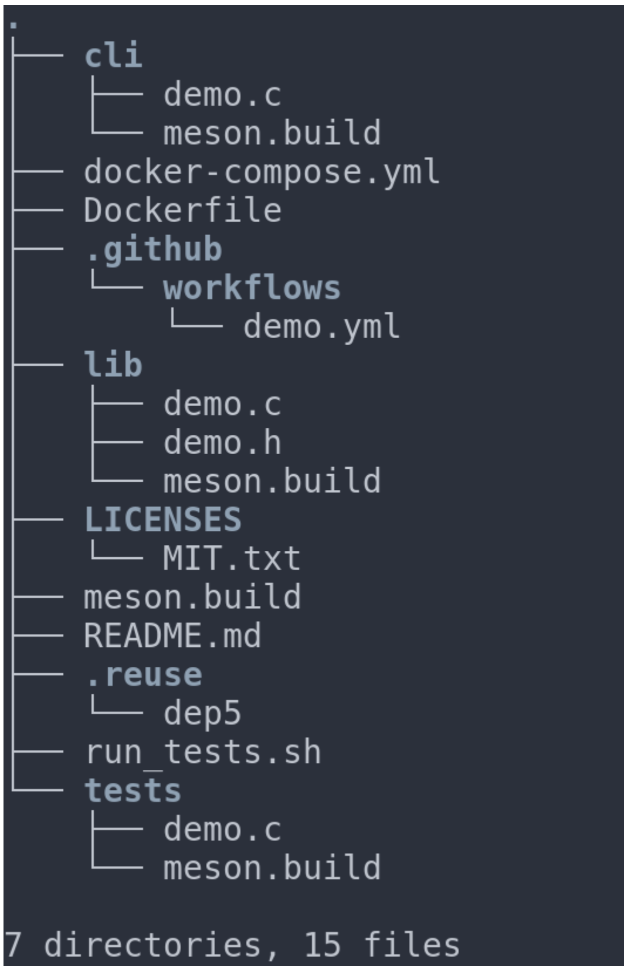


Figure 1: sifis-generate cli and library with their third-party components

sifis-generate generates project files starting from templates of various kinds: source codes, build systems, YAML files, shell scripts, and Dockerfiles. Missing templates parts are then filled up at runtime with the information passed in input by a developer, such as project name, license and build systems arguments. Sometimes a project requires many inputs for filling up its templates, making the command line interface cumbersome and difficult to use. To overcome this issue, we have defined a *toml* file divided into sections, one for each supported project, with the goal of passing to templates all necessary information in a straightforward and manageable way.

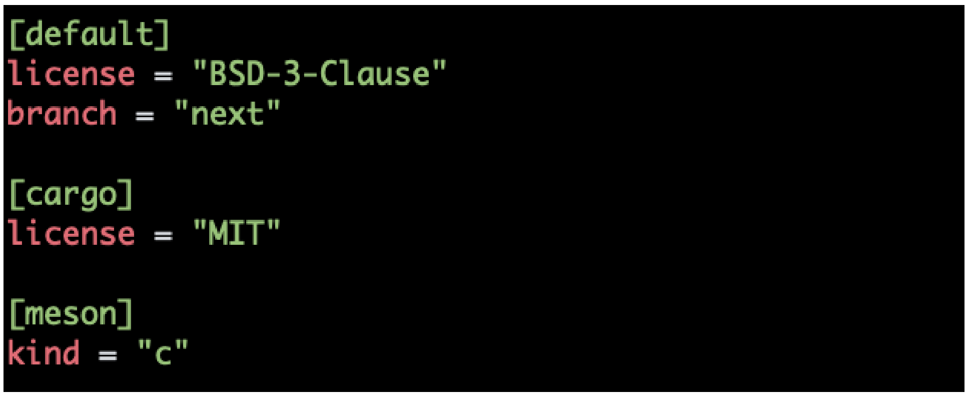
As an example, Figure 2 illustrates a directory containing a new-generated project.



```
.
├── cli
│   ├── demo.c
│   └── meson.build
├── docker-compose.yml
├── Dockerfile
├── .github
│   └── workflows
│       └── demo.yml
├── lib
│   ├── demo.c
│   ├── demo.h
│   └── meson.build
├── LICENSES
│   └── MIT.txt
├── meson.build
├── README.md
├── .reuse
│   └── dep5
├── run_tests.sh
└── tests
    ├── demo.c
    └── meson.build

7 directories, 15 files
```

Figure 2: A simple C-demo project with its build configuration files

Figure 3 shows instead an example of configuration file



```
[default]
license = "BSD-3-Clause"
branch = "next"

[cargo]
license = "MIT"

[meson]
kind = "c"
```

Figure 3: sifis-generate configuration file

2.2.1 Creating projects

We have defined five kinds of projects, one for each programming language used for implementing SIFIS-Home components. By default, each project has a README containing badges, that show some information about software quality, and other details such as license and an acknowledgment heading. Below we briefly explain the build systems and package managers associated to each of the supported projects whose we generate files for.

C/C++
Meson is a new open-source build system meant to be extremely fast and, even more importantly, as user-friendly as possible. This tool has become popular in many C/C++ projects for its simplicity, so the decision to adopt it and generate its configuration files.

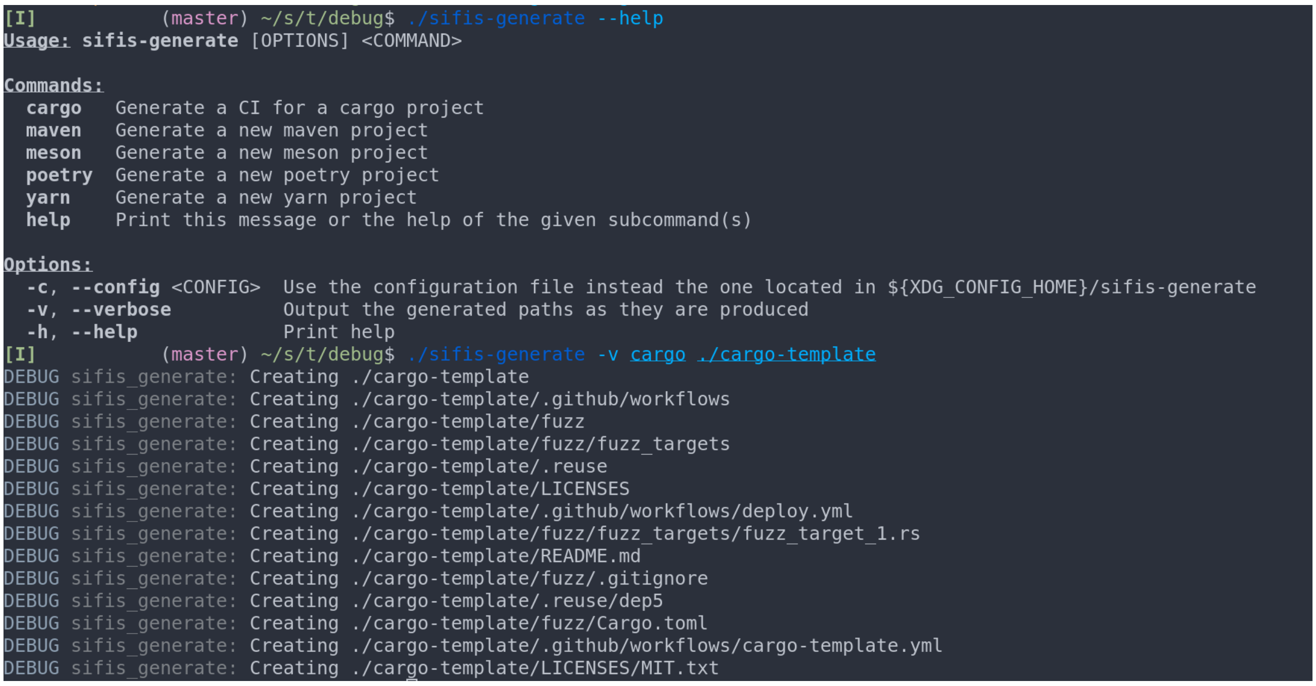
Python
The innovative packager and dependency management *Poetry* has been used to build and deploy Python packages.

Java
Apache Maven is a software project management and comprehension tool for Java.

Rust
Cargo is the official package manager for Rust. It already implements a command to generate all building files: *cargo new*. This command also allows to choose between a library or an executable as output. Figure 4 first shows the helper for *sifis-generate* and then the command to create GitHub Actions scripts for the Rust language.

Javascript
Yarn is a JavaScript package manager. As Cargo, it already implements a command to generate a new project: `yarn init`.

The execution flow of *sifis-generate* is the following: once a project has been chosen, selecting the respective subcommand, the list of its templates, contained in the library as binaries data, are taken and filled up with the information passed as input either in the form of cli options or retrieved from the configuration file. Filled-in templates are then written in the respective output directories which are created when not present on the system. Each subcommand code is contained in a different directory of the repository, and all of them form the *sifis-generate toolchain*.
In Figure 6, the first *sifis-generate* execution prints the list of *sifis-generate*'s subcommands, while the second one executes the *cargo* subcommand, in charge of generating files and directories for the Rust workflow. To visualize the debug information shown below, it is necessary to enable the `-v` option.



```
[I]           (master) ~/s/t/debug$ ./sifis-generate --help
Usage: sifis-generate [OPTIONS] <COMMAND>

Commands:
  cargo   Generate a CI for a cargo project
  maven   Generate a new maven project
  meson   Generate a new meson project
  poetry  Generate a new poetry project
  yarn    Generate a new yarn project
  help    Print this message or the help of the given subcommand(s)

Options:
  -c, --config <CONFIG>  Use the configuration file instead the one located in ${XDG_CONFIG_HOME}/sifis-generate
  -v, --verbose          Output the generated paths as they are produced
  -h, --help             Print help
[I]           (master) ~/s/t/debug$ ./sifis-generate -v cargo ./cargo-template
DEBUG sifis_generate: Creating ./cargo-template
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows
DEBUG sifis_generate: Creating ./cargo-template/fuzz
DEBUG sifis_generate: Creating ./cargo-template/fuzz/fuzz_targets
DEBUG sifis_generate: Creating ./cargo-template/.reuse
DEBUG sifis_generate: Creating ./cargo-template/LICENSES
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/deploy.yml
DEBUG sifis_generate: Creating ./cargo-template/fuzz/fuzz_targets/fuzz_target_1.rs
DEBUG sifis_generate: Creating ./cargo-template/README.md
DEBUG sifis_generate: Creating ./cargo-template/fuzz/.gitignore
DEBUG sifis_generate: Creating ./cargo-template/.reuse/dep5
DEBUG sifis_generate: Creating ./cargo-template/fuzz/Cargo.toml
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/cargo-template.yml
DEBUG sifis_generate: Creating ./cargo-template/LICENSES/MIT.txt
```

Figure 4: sifis-generate tool in action. At first, the list of subcommands is shown through the --help option; then, a Rust continuous integration workflow is created by using the "cargo" subcommand.

2.3 *Complex-code-spotter*

Detecting pieces of complex code is the main goal of ***complex-code-spotter***. Complex code might hide some bugs and tends to be difficult to comprehend at first glance, so its extraction, in the form of snippet, helps a developer to determine its real complexity and spot its position in the code.

This tool structure is visible in Figure 5 alongside with third-party library and cli components.

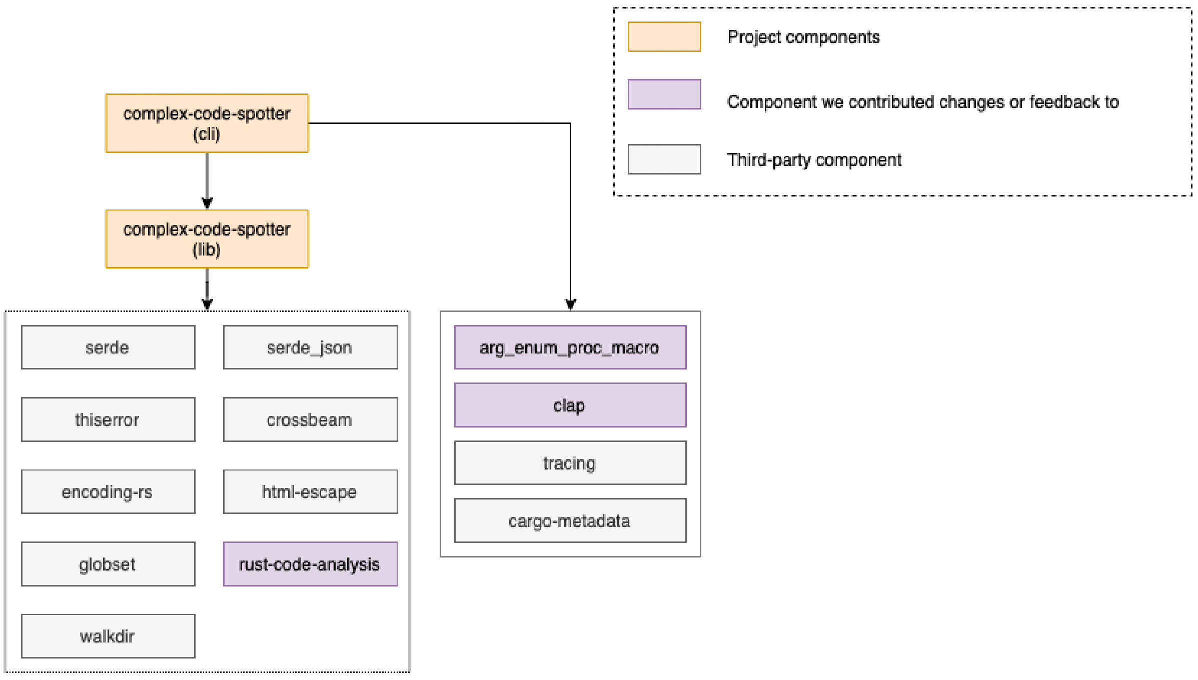


Figure 5: complex-code-spotter cli and library with their third-party components

It requires only two mandatory arguments as input:

- The path to the source code, which is parsed in search of complex snippets.
- The output directory containing all code snippets which exceed the thresholds of the implemented code complexity metrics.

A snippet is any piece of code identified by a *rust-code-analysis* space. A space is any scope which contains a function, and this construct have been employed to have a more defined granularity because it allows to retrieve both functions and closures, other than the complete file.
This tool supports the same programming languages defined in *rust-code-analysis* and makes use of this crate’s APIs to compute code complexity metrics, next it extracts code snippets whether one or more complexity metrics exceed a determined threshold, and finally saves these snippets into one of the supported formats, which can be: *markdown, html, and JSON*.
The default configuration extracts code snippets for *cyclomatic* and *cognitive* metrics, with an empirical threshold of *15*, and defines *markdown* as the default output format.

Other optional arguments are:

- The kinds of code complexity metrics with their own thresholds.
- The number of computational threads.
- An option to visualize the operations performed by the tool.

It is also possible to filter input source files using *I* and *X* options. The *input* -I option is a glob filter that considers **only** files with a determined file extension. The *exclude* -X option is a glob filter that excludes **only** files with a determined file extension.

For example, to consider only *Rust* `*.rs` files:

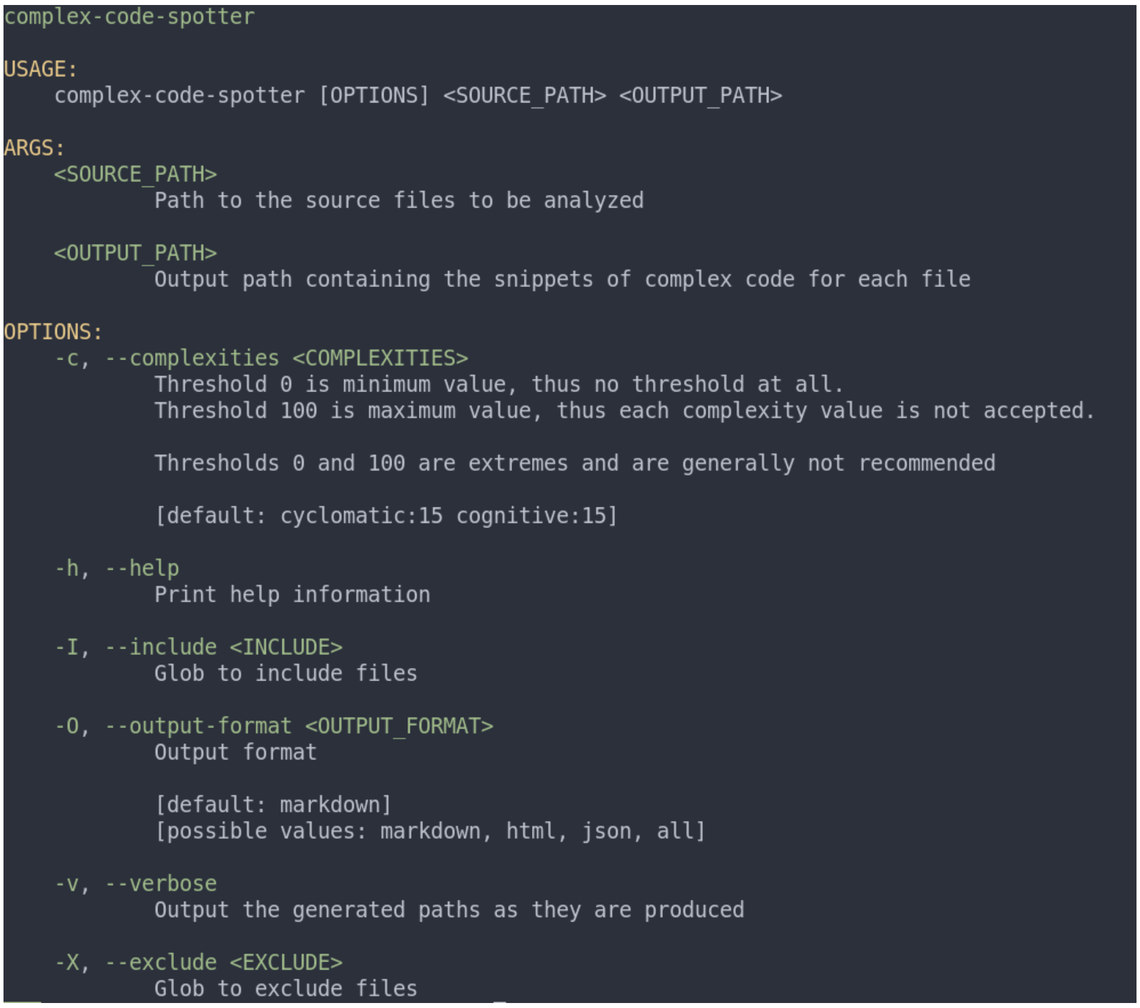
```
complex-code-spotter -I "*.rs" /path/to/file/or/directory /output/path
```

To exclude only *Rust *.rs* files instead:

```
complex-code-spotter -X "*.rs" /path/to/file/or/directory /output/path
```

Both these options can be used more than once.

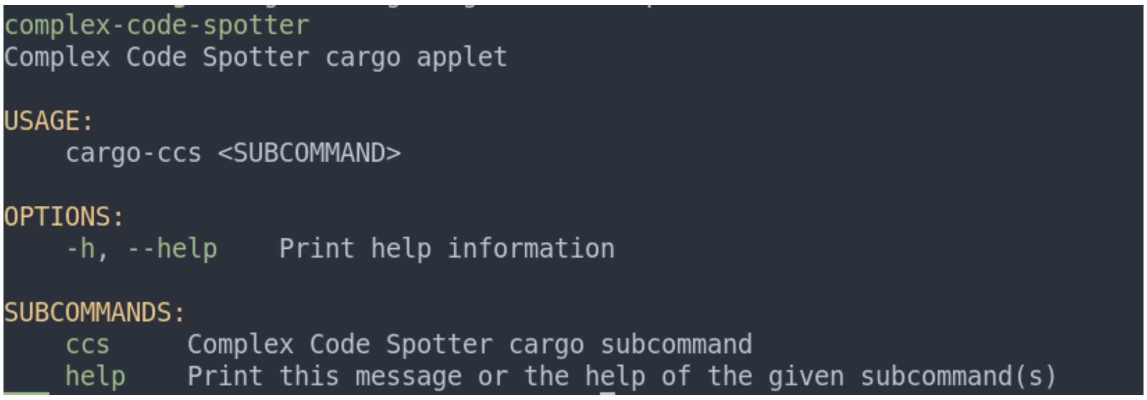
The execution flow of *complex-code-spotter* is the following: each source code is assigned to a thread which at first computes complexity metrics and then extracts snippets.
The results computed by every thread are then sent to a collector thread which, as first step, creates the structure of the output directory and then writes every snippet saved in memory into the output file formatted as requested. In Figure 6, the list of arguments and options for *complex-code-spotter*.



```
complex-code-spotter

USAGE:
    complex-code-spotter [OPTIONS] <SOURCE_PATH> <OUTPUT_PATH>

ARGS:
    <SOURCE_PATH>
            Path to the source files to be analyzed

    <OUTPUT_PATH>
            Output path containing the snippets of complex code for each file

OPTIONS:
    -c, --complexities <COMPLEXITIES>
            Threshold 0 is minimum value, thus no threshold at all.
            Threshold 100 is maximum value, thus each complexity value is not accepted.

            Thresholds 0 and 100 are extremes and are generally not recommended

            [default: cyclomatic:15 cognitive:15]

    -h, --help
            Print help information

    -I, --include <INCLUDE>
            Glob to include files

    -O, --output-format <OUTPUT_FORMAT>
            Output format

            [default: markdown]
            [possible values: markdown, html, json, all]

    -v, --verbose
            Output the generated paths as they are produced

    -X, --exclude <EXCLUDE>
            Glob to exclude files
```

Figure 6: complex-code-spotter interface

We have also added a *cargo* subcommand to extract snippets using *cargo* itself. Figure 7 shows the list of arguments and options for this *cargo* subcommand.



```
complex-code-spotter
Complex Code Spotter cargo applet

USAGE:
    cargo-ccs <SUBCOMMAND>

OPTIONS:
    -h, --help    Print help information

SUBCOMMANDS:
    ccs     Complex Code Spotter cargo subcommand
    help    Print this message or the help of the given subcommand(s)
```

Figure 7: cargo subcommands

Below, we report a snippet with a cyclomatic complexity of 26 written in Rust language. We can note that is difficult to comprehend this code at first glance because of its flow complexity.

```
loop {
    if let Some(subtype) = subtype.as_ref() {
        match subtype {
            Array(array) => {
                match (array.min_items, array.max_items) {
                    (Some(min), Some(max)) if min > max => return
Err(Error::InvalidMinMax),
                    _ => {}
                };

                if let Some(items) = array.items.as_deref() {
                    stack.extend(items.iter());
                }
            }
            Number(number) => {
                match (number.minimum, number.maximum) {
                    (Some(x), _) | (_, Some(x)) if x.is_nan() => return
Err(Error::NanMinMax),
                    (Some(min), Some(max)) if min > max => return
Err(Error::InvalidMinMax),
                    _ => {}
                }

                match number.multiple_of {
                    Some(multiple_of) if multiple_of <= 0. => {
                        return Err(Error::InvalidMultipleOf)
                    }
                    _ => {}
                }
            }
        } // match
    } // if let
} // loop
```

2.4 *Weighted Code Coverage*

The weighted code coverage algorithm explained in deliverable D2.4 has been implemented in a tool called **weighted-code-coverage**. In the same deliverable, we have integrated this tool inside our GitHub Actions workflow to retrieve information about the parts of code which could be complex or not covered by tests, or both.
It also contains two additional weighted code coverage algorithms from the Ruby language, called *Skunk* and *Crap*.

This tool's structure is shown in Figure 8 alongside with third-party libraries and cli components.

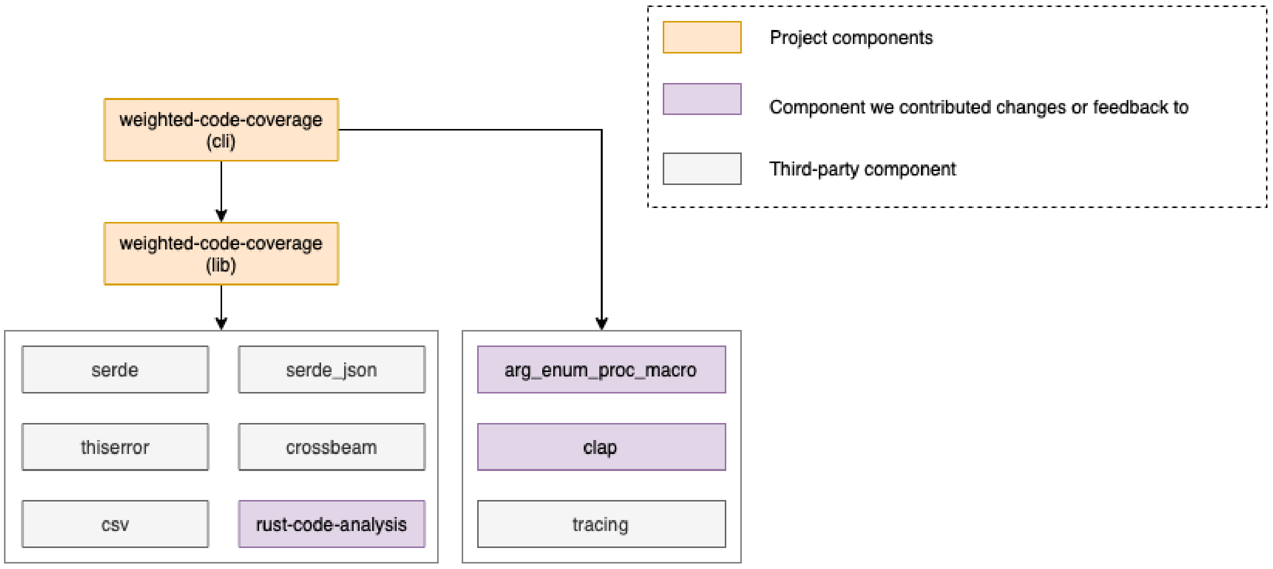


Figure 8: weighted-code-coverage cli and library with their third-party components

It requires only two mandatory arguments as input:

- The path to the source code directory, needed to compute code complexity metrics.
- The JSON file produced by *grcov* that, for each source code file, contains information about covered and uncovered lines.

It is possible to export the results of all four algorithms in two different formats: JSON and CSV. If an output path has not been specified, results are printed on the terminal. We have chosen the JSON format to upload the artifacts in our GitHub Actions workflow.

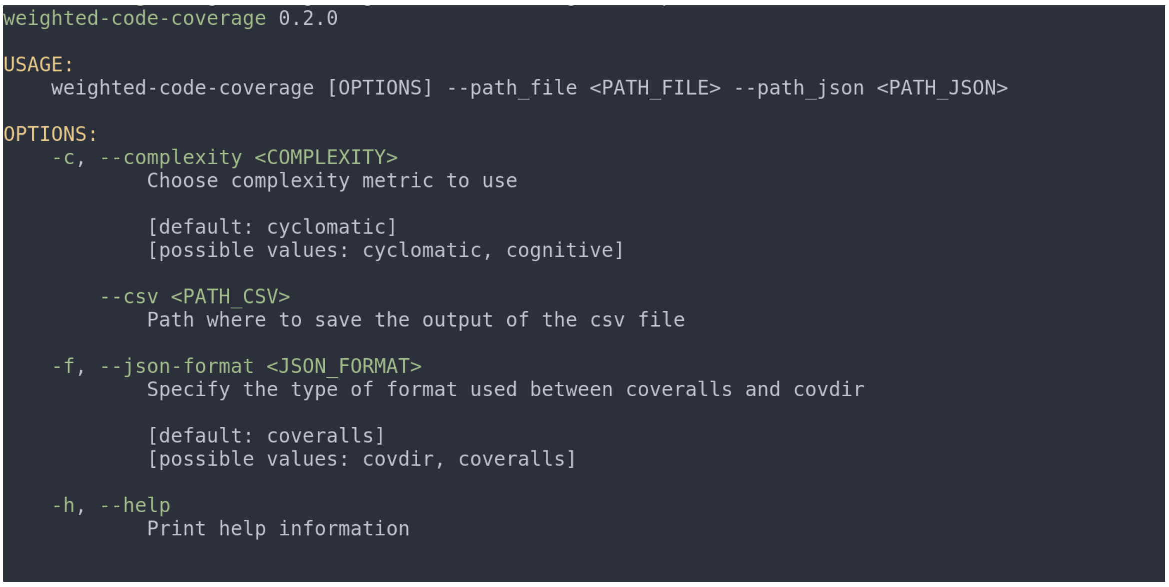
Other optional arguments are:

- The kinds of code complexity metrics with their thresholds.
- The number of computational threads.
- An option to visualize the operations performed by the tool.

The execution flow for *weighted-code-coverage* is the following: source code files are grouped in chunks, and then each chunk is assigned to a thread that computes every algorithm for each file contained in the chunk. The results computed by every thread are then sent to a collector thread which merges all of them into a final output file.

In addition to single files computation, the tool can be more granular and consider functions too. In this way, it is possible to retrieve uncovered functions and those which contain a difficult code to comprehend at first glance.

Currently, *weighted-code-coverage* analyses only Rust files, but it might be expanded to other programming languages. Figure 9 shows the list of arguments and options for *weighted-code-coverage*.



```
weighted-code-coverage 0.2.0

USAGE:
    weighted-code-coverage [OPTIONS] --path_file <PATH_FILE> --path_json <PATH_JSON>

OPTIONS:
    -c, --complexity <COMPLEXITY>
            Choose complexity metric to use

            [default: cyclomatic]
            [possible values: cyclomatic, cognitive]

        --csv <PATH_CSV>
            Path where to save the output of the csv file

    -f, --json-format <JSON_FORMAT>
            Specify the type of format used between coveralls and covdir

            [default: coveralls]
            [possible values: covdir, coveralls]

    -h, --help
            Print help information
```

Figure 9: A portion of weighted-code-coverage command line interface

2.5 *Third-party Contributions*

As part of the activities carried out within WP2, we have contributed to some open-source projects with the purpose of implementing features, fixing bugs, and refactoring code, so that to produce a new software version which could be easily integrated into our workflow or used as third-party dependencies in our tools. Some features required to develop external libraries as well.
Sometimes we have also filed issues on repositories to signal bugs or ask some clarifications about a certain matter.
In the next sections, we explain each of such software contributions we have accomplished during the SIFIS-Home project.

2.5.1 Rust-code-analysis

A Rust Mozilla-tool that computes a series of metrics from source codes. It covers all programming languages present in Firefox codebase. These metrics help a developer identifying the parts of code that need a refactor or structural changes. It has been created to attach more information to patches sent to *mozilla-central* mailing list.
This tool computes metrics in parallel, distributing each file to the threads available on the system. For each file, an Abstract Syntax Tree (AST) is built from a third-party dependency called *tree-sitter*, which also provides some functions to interact with its nodes. So, through a tree search algorithm, it is possible to identify tokens and constructs of the considered programming language and then compute metrics starting from there.

The contributions we made to this tool are the following:

- Refactored the general code to make it more comprehensible. We split the initial library into different modules, such as metrics, AST computation, and general methods which extract information related to programming languages. We also refactored command line and web server code updating their dependencies and their general structure.
- Added more metrics and unit tests to verify whether their results are valid. We also created unit tests for metrics already present in the codebase.
- Completely replaced integration tests with snapshots tests. A snapshot test, or approval test, is a test that asserts values against a reference value: the *snapshot*. Old integration tests do not allow to compare all metrics values, and they were difficult to update when a new language grammar was introduced or upgraded. The snapshot crate, called *insta*, provides a series of commands to simplify and automatize the replacement of snapshots, contained in an external repository and downloaded during test execution. We have tested *rust-code-analysis* on three large repositories written in different programming languages: *serde* for Rust, *DeepSpeech* for C/C++ and *pdf.js* for JavaScript.
- Improved continuous integration system, based on *Taskcluster*, adding tasks that run tests on Windows, check whether cli does not crash when executed on huge *mozilla-central* repository before the deployment phase, deploy Linux and Windows *rust-code-analysis* binaries through a GitHub release.
- Helped developers in reviewing pull requests made by external contributors. We are still helping in maintaining *rust-code-analysis* repository.

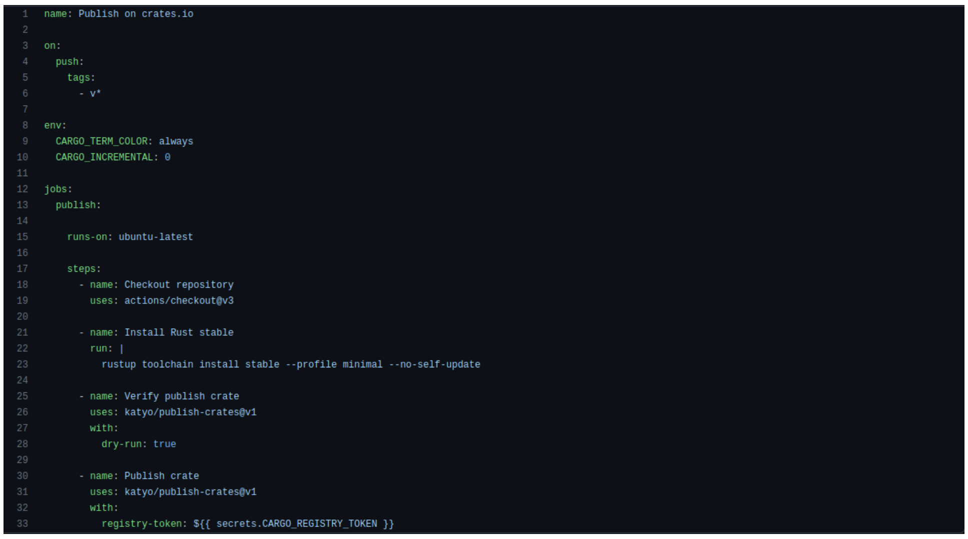
2.5.2 Tree-sitter

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited. This library has been used in *rust-code-analysis* to create the Abstract Syntax Tree.

For each programming language, it provides a grammar which parses the code and produces the associated concrete syntax tree. Rust grammars were not published with the same version on *crates.io*, the Rust package registry, because it was difficult to publish them manually every time using the *cargo publish* command. For this reason, we have created a GitHub Actions script that automatically publishes grammars on *crates.io* when a new tag is added.
We have added this script to eight grammars:

- tree-sitter-java
- tree-sitter-kotlin
- tree-sitter-typescript
- tree-sitter-javascript
- tree-sitter-python
- tree-sitter-rust
- tree-sitter-c
- tree-sitter-cpp

We have also filed some issues about erroneous code parsing or crashes in grammars.



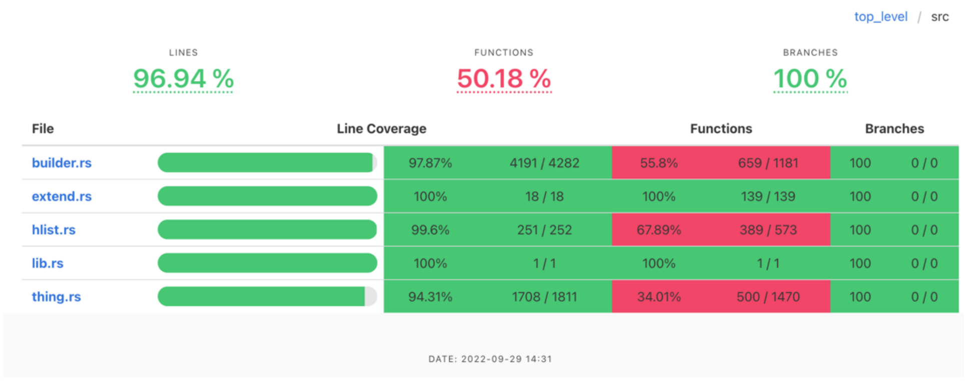
```yaml
name: Publish on crates.io

on:
  push:
    tags:
      - v*

env:
  CARGO_TERM_COLOR: always
  CARGO_INCREMENTAL: 0

jobs:
  publish:

    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Install Rust stable
        run: |
          rustup toolchain install stable --profile minimal --no-self-update

      - name: Verify publish crate
        uses: katyo/publish-crates@v1
        with:
          dry-run: true

      - name: Publish crate
        uses: katyo/publish-crates@v1
        with:
          registry-token: ${{ secrets.CARGO_REGISTRY_TOKEN }}
```

Figure 10: GitHub Action script to publish a tree-sitter grammar on crates.io package registry

2.5.3 Grcov

This tool collects and aggregates code coverage information for multiple source files. We have employed this tool in our workflow and explained its usage in deliverable D2.4.

We have implemented, using the *Bulma* front-end framework, a more consultable report generated by the information extracted by this tool. It visualizes, for each source code file, the percentage and the number of covered lines in addition to the percentage and number of covered functions lines. It also displays the number of branches and the relative coverage percentage. A branch is one of the possible executions paths the code can take after a decision statement, such as an if statement. Figure 11 shows an example of this HTML report.



top_level / src

LINES 96.94 % — FUNCTIONS 50.18 % — BRANCHES 100 %

File	Line Coverage			Functions		Branches	
builder.rs		97.87%	4191 / 4282	55.8%	659 / 1181	100	0 / 0
extend.rs		100%	18 / 18	100%	139 / 139	100	0 / 0
hlist.rs		99.6%	251 / 252	67.89%	389 / 573	100	0 / 0
lib.rs		100%	1 / 1	100%	1 / 1	100	0 / 0
thing.rs		94.31%	1708 / 1811	34.01%	500 / 1470	100	0 / 0

DATE: 2022-09-29 14:31

Figure 11 Grcov HTML report

We have also filed an issue about wrong Windows paths which makes the program crash and improved the overall testing system by introducing snapshot testing using **insta**.

2.5.4 Insta

Insta is a snapshot testing harness that integrates with the standard cargo test system. While using it to address some grcov testing shortcomings, we also found some minor issues that were reported and quickly fixed by upstream.

2.5.5 Cargo-fuzz

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 input data corpus to maximize code coverage.

We have added a GitHub Actions script to automatically deploy *cargo-fuzz* on Linux, MacOS and Windows architectures once a new tagged-release is created. We have done this contribution to avoid building this tool from scratch in our GitHub Actions workflow, thus reducing computational time a little.

2.5.6 Cargo-valgrind

Another cargo subcommand that runs *valgrind memcheck* under the hood. *Valgrind memcheck* is a memory error detector that can detect security vulnerabilities such as accessing memory after it has been freed, using undefined values, i.e., values that have not been initialised or that have been derived from other undefined values, and memory leaks.

We have added a GitHub Actions script to automatically deploy *cargo-valgrind* on Linux, MacOS and Windows architectures once a new tagged-release is created. We have done this contribution to avoid building this tool from scratch in our GitHub Actions workflow with the aim of reducing the overall computational time a little.

2.5.7 Cargo-careful

This cargo subcommand detects certain kinds of undefined behaviours and performs sanity checks while executing a software.

Even for this tool, we have added a GitHub Actions script to automatically deploy *cargo-careful* on Linux, MacOS and Windows architectures once a new tagged-release is created. We have done this contribution to avoid building this tool from scratch in our GitHub Actions workflow with the aim of reducing computational time a little. At the time of writing this document, the pull request is still pending and with only a summary review.

2.5.8 Clap

It is the most known **C**ommand **L**ine **A**rgument **P**arser for Rust language. It can create a command-line parser both declaratively and procedurally.

We extended its functionality to support a use case we incurred in our tools: the possibility to merge two different sources of information at the same time. In our case, it happens when a command line argument needs to replace an argument defined in a preset that could be a default or passed as a command line input as well.

2.5.9 Arg-enum-proc-macro

We have developed this tiny crate to provide a mean to integrate our libraries, that do not need to use

Clap, with our tools command lines that make use of *Clap* in declarative mode.

2.5.10 REUSE

Free Software Foundation Europe has defined a specification, called *REUSE*, to provide 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.

We have contributed to this tool by opening an issue on its main GitHub repository asking to produce better error messages in case of missing or not correct licenses.

2.5.11 mdns-sd

The Web of Thing Discovery specification uses mDNS/DNS-SD to advertise Things existence. We selected the **mdns-sd** crate to add mDNS/DNS-SD support in **wot-serve** and **wot-discovery** and contributed some bugfixes and support for DNS subtypes.

2.5.12 webthing-arduino

Webthings.io provides an arduino-compatible implementation of Web of Things Servient. We contributed with code patches to have the implementation compatible with the W3C Thing Description 1.0.

2.5.13 webthings-rust

We also contributed with fixes and specification updates to the Rust implementation before the decision of building our own complete stack, as explained later in this document.

2.5.14 node-wot

We routinely cooperate with the *node-wot* community to make sure **wot-rust** and **node-wot** implementations interoperate acceptably.

2.5.15 cargo-c

Use mainly to showcase how the rust code may be easily used via a C-API, patches and improvements provided.

2.5.16 maturin

Similar to cargo-c, it makes easy to consume rust code, but via Python. We reported some issues as it was evaluated.

2.5.17 datta

An implementation rfc6570. Used in **wot-serve** and as showcase of the developer handbook as detailed in 4.2.1.

3 API Labelling Tools

The tools we show in this section use the concepts of API Label and App Label described in Section 3 of deliverable D2.4.

We recall that an API Label is associated with a SIFIS-Home developer API to describe possible risks deriving from its execution. The API Label consists of a list of hazards, each identifying a risk.
On the other hand, the App Label is a label associated with an application written by a third-party developer; it is derived from the combination of the API Labels related to the SIFIS-Home developer APIs invoked by the application's source code.

The list of hazards presented in deliverable D2.4 has been formally defined in an ontology introduced in the following section.

3.1 *The SIFIS-Home Hazards Ontology*

An ontology called *SIFIS-Home Hazards Ontology* (SHO) has been created to formally define the hazards we identified for the smart home environment. Notably, the SHO can be used to extend the representation of WoT's smart devices, called Thing Descriptions (TDs).

A TD is a JSON-LD representation of a connected device called Thing. The TD ideally provides all the information to control the device in a structured way. Every possible interaction is mapped through three categories: Properties, Actions, and Events. A client consuming the description can set or read a Property, subscribe/unsubscribe for future Events or issue a complex order and then wait for it to happen (Action).

Since the TD is a JSON-LD, it is possible to extend it and add semantic meaning to every element of it; the SHO is used to bind the risk information to every interaction described. For example, a Property of an oven's TD could be "on". Of course, turning an oven on may pose some risks, especially if this function is called remotely and the home is unattended. Within a TD, such risks can be expressed precisely by mapping the related hazards to the "On" state, i.e., when the "on" property value is "true".

```
"property" : {
  "on" : {
    "@type" : "OnOffProperty",
    "type" : "boolean",
    "hazards" : [
      {
        "@id": "sho:ElectricEnergyConsumption",
        title": "Electric energy consumption",
        description" : "The execution enables a device that consumes
electricity",
        "risk_score" : 5,
        "type" : "boolean",
        "const" : true
      },
      {
        "@id": "sho:FireHazard",
        "title": "Fire hazard",
        "description" : "The execution might cause fire",
        "risk_score" : 8,
        type" : "boolean",
        "const" : true
      }
    ]
  }
}
```

Figure 12 shows a graphical representation of the SHO by means of a force-directed graph layout.

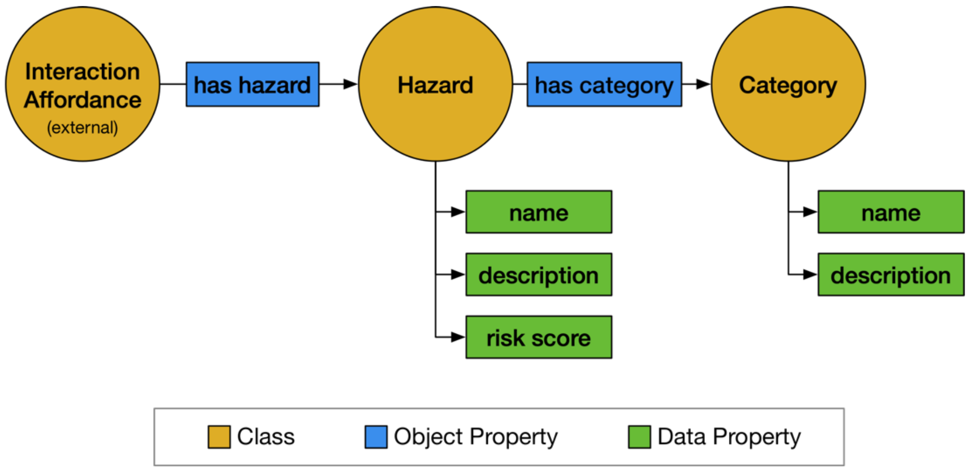


Figure 12: Visualization of the SIFIS-Home Hazards Ontology

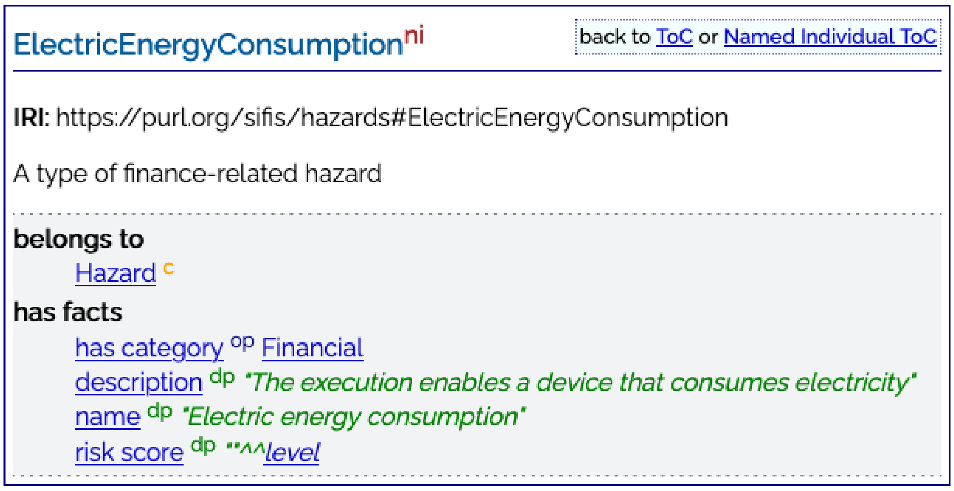
The SHO defines the classes Hazard and Category, and it uses the external class Interaction Affordance (defined in the Thing Description Ontology [TD, 2022]) to link to the WoT world. In particular, the object property hasHazard connects the Interaction Affordance class to the Hazard class with a one-to-many relationship, meaning that a Property, an Action, or an Event can be associated with one or more objects of class Hazard.

The Hazard class is characterized by the data properties "name" and "description" of type string, and, optionally, "risk score" of type level, defined as an integer in the range from 0 to 10; Also, the Hazard class is connected to the Category class through an object property named hasCategory, indicating that a hazard belongs to some category.

The Category class is characterized by the data properties "name" and "description" of type string. Moreover, the ontology defines three Named Individuals for the Category class:

- Safety: category identifying hazards that may lead to physical harm to people and/or assets.
- Privacy: category identifying hazards that may compromise privacy.
- Financial: category identifying hazards that lead to an expense.

Currently, the ontology defines also 24 Named Individuals for the Hazard class. As an example, we report in Figure 13 the complete description of a hazard named ElectricEnergyConsumption.


ElectricEnergyConsumption ni

back to ToC or Named Individual ToC

IRI: https://purl.org/sifis/hazards#ElectricEnergyConsumption

A type of finance-related hazard

belongs to
Hazard c

has facts
has category op Financial
description dp "The execution enables a device that consumes electricity"
name dp "Electric energy consumption"
risk score dp ""^^level

Figure 13: Example of hazard defined as Named Individual within the SHO.

The screenshot above has been taken from the SIFIS-Home Hazards Ontology specification webpage (SHO, 2022), where the whole ontology is described and can be downloaded in various serialized formats, among which JSON-LD.

3.2 *Ontology Translation into Different Programming Languages*

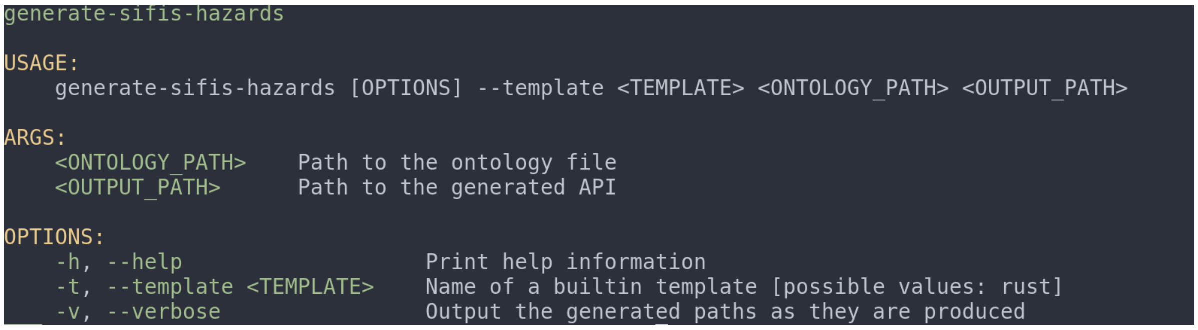
A serialized version of the SIFIS-Home Hazards Ontology has to be translated into specific programming languages to be used for programming purposes. This is needed in many cases, such as for creating the API Labels and the App Label and deserializing Thing Descriptions.

To this aim, we created a tool called *generate-sifis-hazards* which reads the SHO's JSON-LD serialized file and translates it into a programming-language-specific file containing the structures to represent the ontology. Currently, the only programming language supported is Rust.

This tool fills a given template, different for each programming language, with information extracted from the ontology, generating in the output the programming-language-specific file, which allows to interact with hazards and get their data. A template is composed of a series of APIs and structures representing concepts and information contained in an ontology, so the same goes for the output file produced by the tool.

The `generate-sifis-hazards` is run from the command-line interface, as shown in Figure 14, and its inputs are:

- `<ONTOLOGY_PATH>`: the path to the input JSON-LD file containing the ontology.
- `<TEMPLATE>`: the template to be used, which determines the programming language the ontology will be translated into.
- `<OUTPUT_PATH>`: the path where the output programming-language-specific file will be stored.



```
generate-sifis-hazards

USAGE:
    generate-sifis-hazards [OPTIONS] --template <TEMPLATE> <ONTOLOGY_PATH> <OUTPUT_PATH>

ARGS:
    <ONTOLOGY_PATH>    Path to the ontology file
    <OUTPUT_PATH>      Path to the generated API

OPTIONS:
    -h, --help                   Print help information
    -t, --template <TEMPLATE>    Name of a builtin template [possible values: rust]
    -v, --verbose                Output the generated paths as they are produced
```

Figure 14: generate-sifis-hazards command line interface

Below we show a portion of the output produced by *generate-sifis-hazards* representing a Rust enumerator, including all the hazards extracted from the ontology.

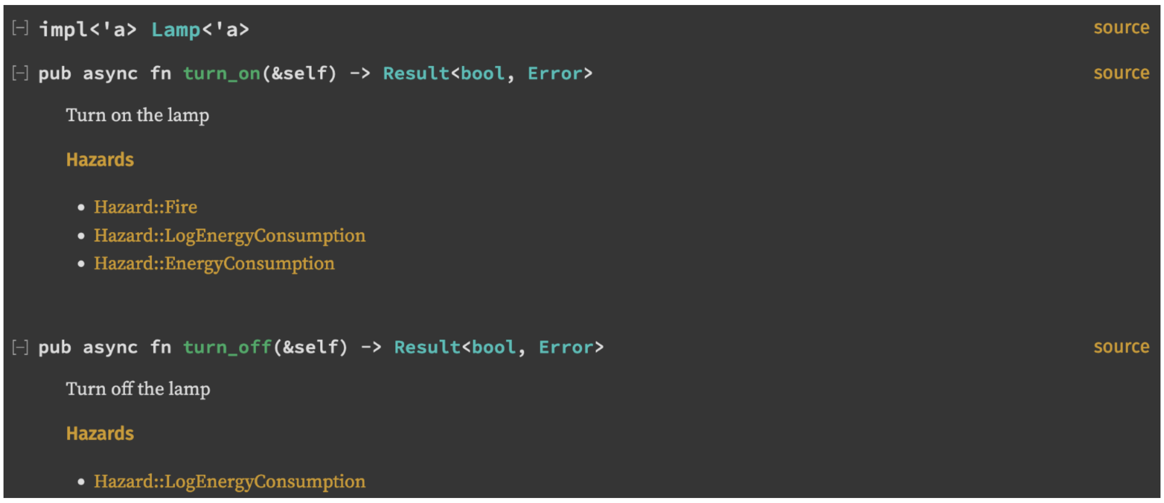
```
/// Hazards type.
pub enum Hazard {
    /// The execution may release toxic gases
    AirPoisoning,
    /// The execution may cause oxygen deficiency by gaseous substances
    Asphyxia,
    /// The execution authorises the app to record and save a video with audio on
persistent storage
    AudioVideoRecordAndStore,
    /// The execution authorises the app to obtain a video stream with audio
    AudioVideoStream,
    /// The execution allows usage of devices that may cause burns
    Burn,
    /// The execution enables a device that consumes electricity
    ElectricEnergyConsumption,
    /// The execution may cause an explosion
    Explosion,
    /// The execution may cause fire
    FireHazard,
    /// The execution enables a device that consumes gas
    GasConsumption,
    /// The execution authorises the app to get and save information about the
app's energy impact on the device the app runs on
    LogEnergyConsumption,
    /// The execution authorises the app to get and save information about the
app's duration of use
    LogUsageTime,
    /// The execution authorises the app to use payment information and make a
periodic payment
    PaySubscriptionFee,
    /// The execution may cause an interruption in the supply of electricity
    PowerOutage,
    /// The execution may lead to exposure to high voltages
    PowerSurge,
    /// The execution authorises the app to get and save user inputs
    RecordIssuedCommands,
    /// The execution authorises the app to get and save information about the
user's preferences
    RecordUserPreferences,
    /// The execution allows usage of devices that may cause scalds
    Scald,
    /// The execution authorises the app to use payment information and make a
payment transaction
    SpendMoney,
    /// The execution may lead to rotten food
    SpoiledFood,
    /// The execution authorises the app to read the display output and take
screenshots of it
    TakeDeviceScreenshots,
    /// The execution authorises the app to use a camera and take photos
    TakePictures,
    /// The execution disables a protection mechanism and unauthorised
individuals may physically enter home
    UnauthorisedPhysicalAccess,
    /// The execution enables a device that consumes water
    WaterConsumption,
    /// The execution allows water usage which may lead to flood
    WaterFlooding,
}
```

The generated structures are then packaged into a crate so they can be reused by the Developer API implementation and other components that reason about Hazards.

3.3 *Labelling for Application Developers*

Third-party application developers are provided with a high-level API (SIFIS-Home Developer API) that abstracts away the device-specific protocols and forces some constraints on what can be represented:

- The device must provide only the set of interaction affordances linked to their category (e.g., a Lamp must provide an OnOff property)
- The devices are exposed using concrete classes with self-explanatory methods (e.g., Lamp has a *turn_lamp_on* method)
- The high-level API is mapped to a flat RPC that is all a SIFIS-Home-aware Application can use to communicate outside its container/sandbox.

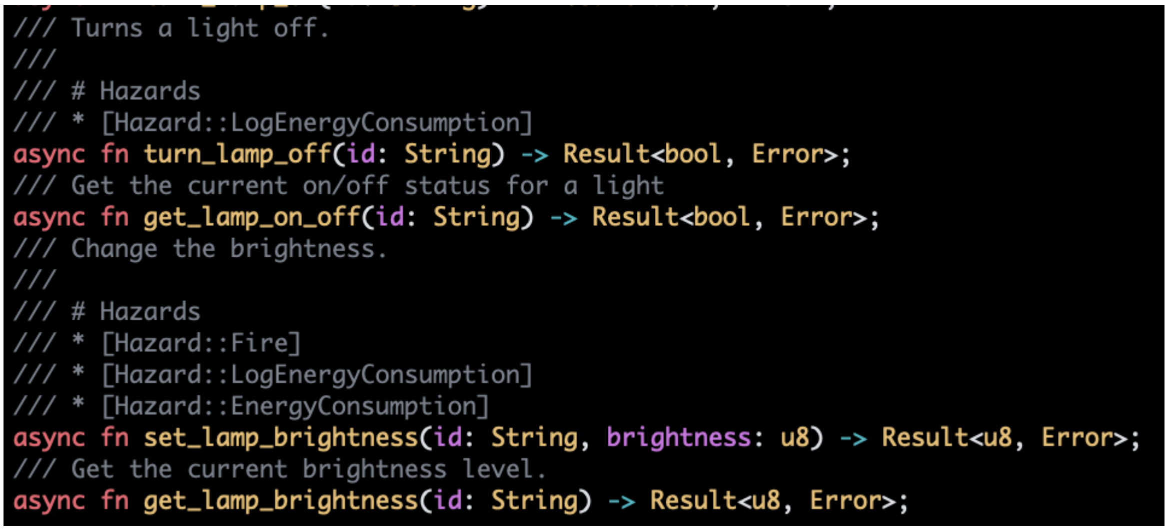


```
[-] impl<'a> Lamp<'a>                                        source
[-] pub async fn turn_on(&self) -> Result<bool, Error>       source
      Turn on the lamp
      Hazards
        • Hazard::Fire
        • Hazard::LogEnergyConsumption
        • Hazard::EnergyConsumption

[-] pub async fn turn_off(&self) -> Result<bool, Error>      source
      Turn off the lamp
      Hazards
        • Hazard::LogEnergyConsumption
```

Figure 15 rustdoc-generated documentation

The RPC uses messages with a regular formatting: {verb}_{object}_{property}

- **Verb** is either {get} to read or {set} to write. Shortcut verbs for the widespread properties can also be used, e.g., turning on/off is mapped to a *turn_lamp_on* and *turn_lamp_off*.
- **Object** is the specific kind of device/resource, e.g., {lamp} or {sink}.
- **Property** is a property of the object that is accessed, e.g., {brightness} or {flow}.



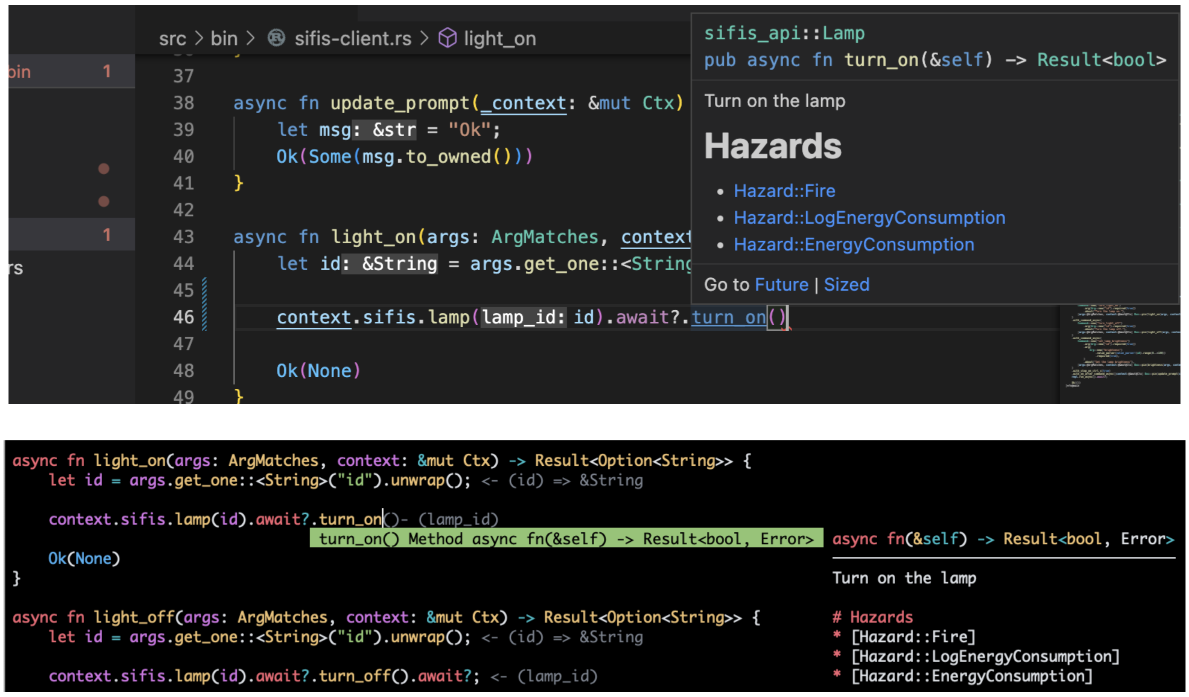
```rust
/// Turns a light off.
///
/// # Hazards
/// * [Hazard::LogEnergyConsumption]
async fn turn_lamp_off(id: String) -> Result<bool, Error>;
/// Get the current on/off status for a light
async fn get_lamp_on_off(id: String) -> Result<bool, Error>;
/// Change the brightness.
///
/// # Hazards
/// * [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 16 Example implementation of the rpc using tarpc

Since API Labels are also embedded in the method comments, they can be represented as popup notes that appear as part of an autocomplete feature in a development tool. This task is nowadays performed by an implementation of a Language Server Protocol.

Autocomplete, go-to definition or documentation on hover features are usually implemented similarly for each development tool providing different APIs for the same goal. A Language Server Protocol has been invented to reduce this attitude by defining a standardized protocol to interact with these development tools through inter-process communication. The protocol delineates a series of guidelines to build a server that can be reused in multiple development tools, providing the features described above and supporting various programming languages with minimal effort. Many development tools implement this protocol, particularly code editors like VS Code, Emacs and Vim.

In SIFIS-Home, an overview of a typical implementation session conducted by a third-party developer has been performed using *rust-analyzer*, a Language Server Protocol implementation for the Rust language, and VS Code as a code editor. Figure 17 shows the autocompletion plug-in in action within VS Code.



```
src > bin > sifis-client.rs > light_on

async fn update_prompt(_context: &mut Ctx)
    let msg: &str = "Ok";
    Ok(Some(msg.to_owned()))
}

async fn light_on(args: ArgMatches, context
    let id: &String = args.get_one::<String

    context.sifis.lamp(lamp_id: id).await?.turn_on()

    Ok(None)
}
```

```
sifis_api::Lamp
pub async fn turn_on(&self) -> Result<bool>

Turn on the lamp

Hazards

 • Hazard::Fire
 • Hazard::LogEnergyConsumption
 • Hazard::EnergyConsumption

Go to Future | Sized
```

```
async fn light_on(args: ArgMatches, context: &mut Ctx) -> Result<Option<String>> {
    let id = args.get_one::<String>("id").unwrap(); <- (id) => &String

    context.sifis.lamp(id).await?.turn_on()- (lamp_id)
                            turn_on() Method async fn(&self) -> Result<bool, Error>
    Ok(None)
}

async fn light_off(args: ArgMatches, context: &mut Ctx) -> Result<Option<String>> {
    let id = args.get_one::<String>("id").unwrap(); <- (id) => &String

    context.sifis.lamp(id).await?.turn_off().await?; <- (lamp_id)
```

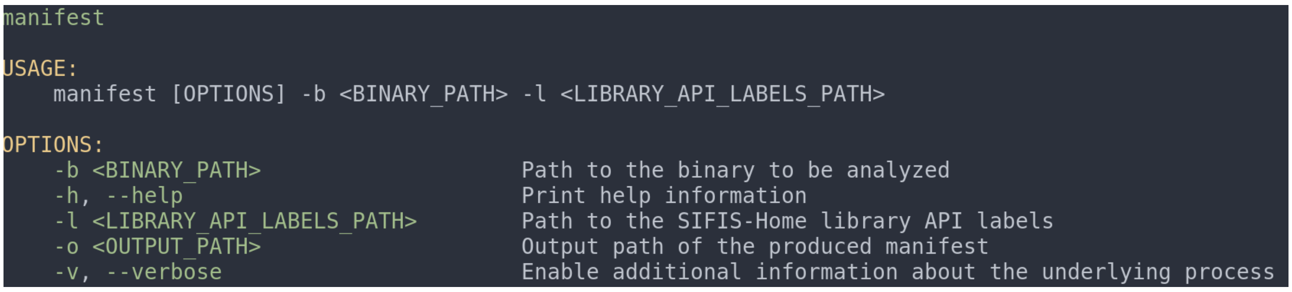
```
async fn(&self) -> Result<bool, Error>

Turn on the lamp

# Hazards
* [Hazard::Fire]
* [Hazard::LogEnergyConsumption]
* [Hazard::EnergyConsumption]
```

Figure 17: The language server protocol plugins in action in different editors. It allows autocompletion of SIFIS-Home Developer APIs and shows the description of a SIFIS-Home Developer API and its related hazards.

3.4 *Labelling for Users*

The usefulness of security labels regarding users becomes clear when we reason about applications users wishes to install onto their smart devices. Indeed, every third-party application includes an App Label, which is used to notify users of all the risky operations carried by the app.
Besides informing the end user about app's behaviour and possible 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 it is asked whether it wants to proceed with the installation or abort it. In the former case, the application is installed, but, based on their labels, the execution of APIs that would violate the rules defined by the user are automatically denied at runtime.

To extract every hazard contained in an application, and thus to create the App Label, a tool called *manifest* has been developed in Rust. Figure 18 shows its usage and options printed at the command-line interface. The *manifest* tool analyses the binary format of an application (multiple binary formats, e.g., ELF and Mach-O, are supported) to find all the SIFIS-Home Developer APIs contained in it and subsequently discover which are the hazards and behaviours paired with each retrieved API. The tool needs only two input parameters: the application binary path and the *Library API labels path.*



```
manifest

USAGE:
    manifest [OPTIONS] -b <BINARY_PATH> -l <LIBRARY_API_LABELS_PATH>

OPTIONS:
    -b <BINARY_PATH>                    Path to the binary to be analyzed
    -h, --help                          Print help information
    -l <LIBRARY_API_LABELS_PATH>        Path to the SIFIS-Home library API labels
    -o <OUTPUT_PATH>                    Output path of the produced manifest
    -v, --verbose                       Enable additional information about the underlying process
```

Figure 18: manifest command line interface

In particular, the Library API labels path is a JSON file formed by an array of API Labels, including all the SIFIS-Home Developer APIs contained in a specific version of a SIFIS-Home library. Each API Label within this file is defined according to the API Label Schema introduced in Section 3.5.1.1 of deliverable D2.4. Note that since new APIs can be added or removed over time, this file is likely to be different for each version of the SIFIS-Home library.

Below, the content of a Library API labels file is shown. Within the file, an API Label is included for each SIFIS-Home Developer API. In this oversimplified example, the number of all the SIFIS-Home Developer APIs defined within the SIFIS-Home library is two.

```
{
  "version": "0.1",
  "api_labels": [
    {
      "api_name": "turn_lamp_on",
      "api_description": "Turns on a lamp.",
      "behavior_label": [
        {
          "device_type": "lamp",
          "action": "turn_on"
        }
      ],
      "security_label": {
        "safety": [
          {
            "name": "FireHazard",
            "description": "The execution may cause fire.",
            "risk_score": 2
          }
        ],
        "privacy": [
          {
            "name": "LogEnergyConsumption",
            "description": "The execution allows the app to register information
about energy consumption."
          }
        ],
        "financial": [
          {
            "name": "ElectricEnergyConsumption",
            "description": "The execution enables the device to consume further
electricity.",
            "risk_score": 5
          }
        ]
      }
    },
    {
      "api_name": "turn_oven_on",
      "api_description": "Turns on an 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": "AudioVideoStream",
            "description": "The execution authorises the app to obtain a video
stream with audio."
          },
          {
            "name": "PowerOutage",
            "description": "High instantaneous power. The execution may cause
power outage.",
            "risk_score": 8
          }
        ],
        "privacy": [
          {
            "name": "LogEnergyConsumption",
            "description": "The execution allows the app to register information
about energy consumption."
          }
        ],
        "financial": [
          {
            "name": "ElectricEnergyConsumption",
            "description": "The execution enables the device to consume further
electricity.",
            "risk_score": 8
          }
        ]
      }
    }
  ]
}
```

The *manifest* tool produces in output a JSON file composed of the array of the API labels related to the SIFIS-Home developer APIs invoked within the application code. Below is an example of an App Label extracted from a binary.

```
{
  "app_name": "app_name",
  "app_description": "app_description",
  "sifis_version": "0.1",
  "api_labels": [
    {
      "api_name": "turn_lamp_on",
      "api_description": "Turns on a lamp.",
      "behavior_label": [
        {
          "device_type": "lamp",
          "action": "turn_on"
        }
      ],
      "security_label": {
        "safety": [
          {
            "name": "FireHazard",
            "description": "The execution may cause fire.",
            "risk_score": 2
          }
        ],
        "privacy": [
          {
            "name": "LogEnergyConsumption",
            "description": "The execution allows the app to register information
about energy consumption."
          }
        ],
        "financial": [
          {
            "name": "ElectricEnergyConsumption",
            "description": "The execution enables the device to consume further
electricity.",
            "risk_score": 5
          }
        ]
      }
    }
  ]
}
```

As any other software developed in WP2, even this one is subject to the CI checks described in Section 2, in addition to a script for deployment and release.

3.5 *Contract-based Security Methodology*

The App Label is deterministically derived from the application code and contains behavioural and security-related information, based on the specific SIFIS-Home developer APIs invoked within the application code. At installation time, the App Label is used to verify whether the application violates some user-defined policies. If this is the case, the user is informed of the reason why the application is not compliant with his policies, and he is asked whether to abort the installation or proceed with it anyway.

This approach for application policy evaluation and enforcement falls in the set of the Contract-based security methodologies and is derived as an optimization of the Security-by-Contract approach. In particular, an application A is linked to a contract C –in our case, the App Label– describing the behaviour of the application. The level of behaviour representation comes from the set of invoked SIFIS-Home developer APIs and their related risks. On the other hand, the policy P is written in XACML and provided by the user. The evaluation is performed at deploy time, converting the App Label (contract) in an XACML request which is then evaluated against an Access Control policy. The compliance between the application behaviour and contract A ⊨ C is ensured by the automated derivation of the API sets from the source code or binary, i.e., by the *manifest* tool, while the contract-policy matching C ⊨ P is performed through the policy enforcement engine, which is an XACML policy evaluation engine.

To convert the App Label in XACML requests, we created a tool called *sifis-xacml*, which takes the App Label as input and returns a set of XACML requests. This tool runs on the Application Manager component and is part of the chain of events triggered by the user pressing the Install button on the Marketplace. The generated XACML requests are then individually submitted to the policy enforcement engine, to be evaluated against user-defined policies. An evaluation of Permit means that the corresponding request is allowed, and therefore it does not go against any user-defined policy. On the contrary, an evaluation of Deny means that a user-defined policy does not allow the requested behaviour or the execution of an operation carrying some risk.

More in detail, an XACML request is extracted from each API Label included in the App Label and contains information about the API behaviour as well as the hazards associated with such an API.
For example, the API label of the `turn_lamp_on()` API, included in the App Label shown in Section 3.4, would result in the generation of one XACML request to be submitted to the policy enforcement engine. The resulting XACML request contains the attribute "subject-id" with value "marketplace", the attribute "resource-id" with value "app_name", the attribute "action-id" with value "install", and the resource-related attributes "device_type" and "action" with value "lamp" and "turn on", respectively. Moreover, it contains the attributes related to the hazards. In particular, the request contains the environment-related attribute "hazard" with possibly multiple values; in this case, the values are "FireHazard", "LogEnergyConsumption", and "ElectricEnergyConsumption".

The policies used at installation time are called "Installation Policies" and are specifically designed to be evaluated at installation time. The <Target> to which they apply includes the attribute "subject-id" with value "marketplace" and the attribute "action-id" with value "install". Sticking with the previous example, an installation policy stating "Do not install applications that turn lamps on" would include a rule with effect Deny, containing a condition defined as a Boolean AND between the resource-related attributes "device_type" and "action" with value "lamp" and "turn_on", respectively.

The evaluation of previous request and installation policy produces an evaluation of Deny, and this determines the non-compliance of the contract with the policy. When this happens –after the evaluation of all the XACML requests is complete– the user is informed of the reason for the non-compliance, and he can decide either to abort the installation or to install the application regardless of the non-compliance. Suppose the user decides to install the application anyway. In that case, a monitor is attached to the specific API that caused the non-compliance, and such a monitor will prevent its execution. Therefore, policy enforcement will be performed at runtime.

Figure 19 shows the flow of the procedure just described.

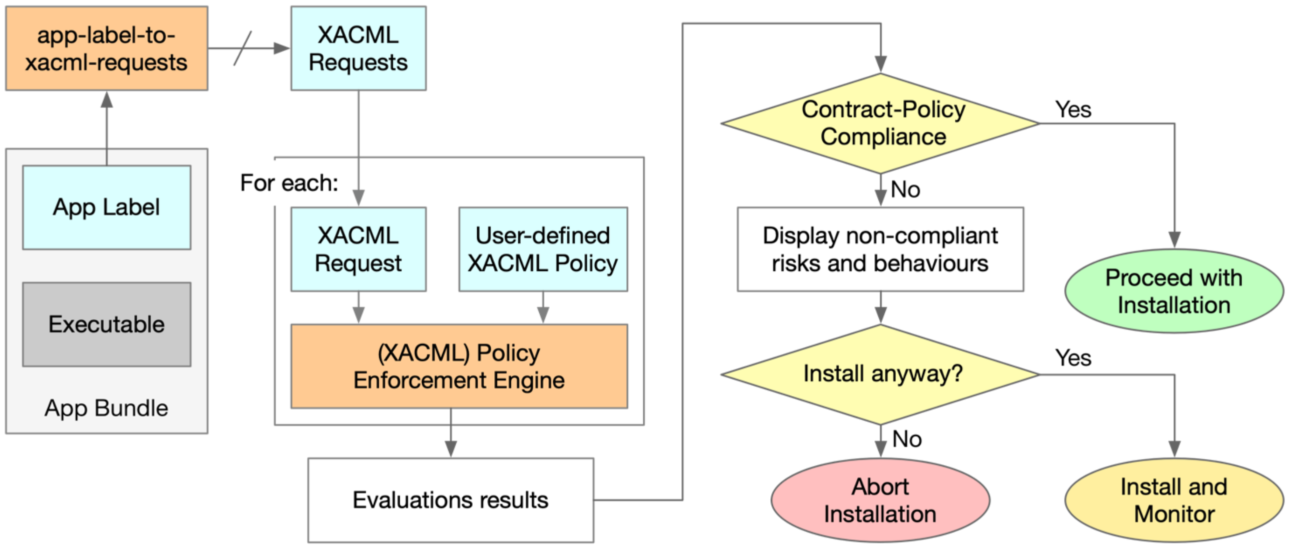

app-label-to-xacml-requests
XACML Requests
App Label
Executable
App Bundle
For each:
XACML Request
User-defined XACML Policy
(XACML) Policy Enforcement Engine
Evaluations results
Contract-Policy Compliance
Yes
No
Display non-compliant risks and behaviours
Proceed with Installation
Install anyway?
Yes
No
Abort Installation
Install and Monitor

Figure 19 Security by Contract workflow.

4 A Complete Working Example

We focused on WoT and started working with the Webthings.io community, extending their *webthings-rust* and *webthings-arduino* implementations by writing an initial proof of concept of Thing extended with the SIFIS-Home Hazards ontology.

From this initial experience, we developed a set of separate crates to support the WoT 1.1 standard and the next WoT 2.0. As result of our W3C Web of Thing community interaction, now a consortium partner joined the W3C as member, directly contributing to the specification.

4.1 *WoT Implementation in Rust*

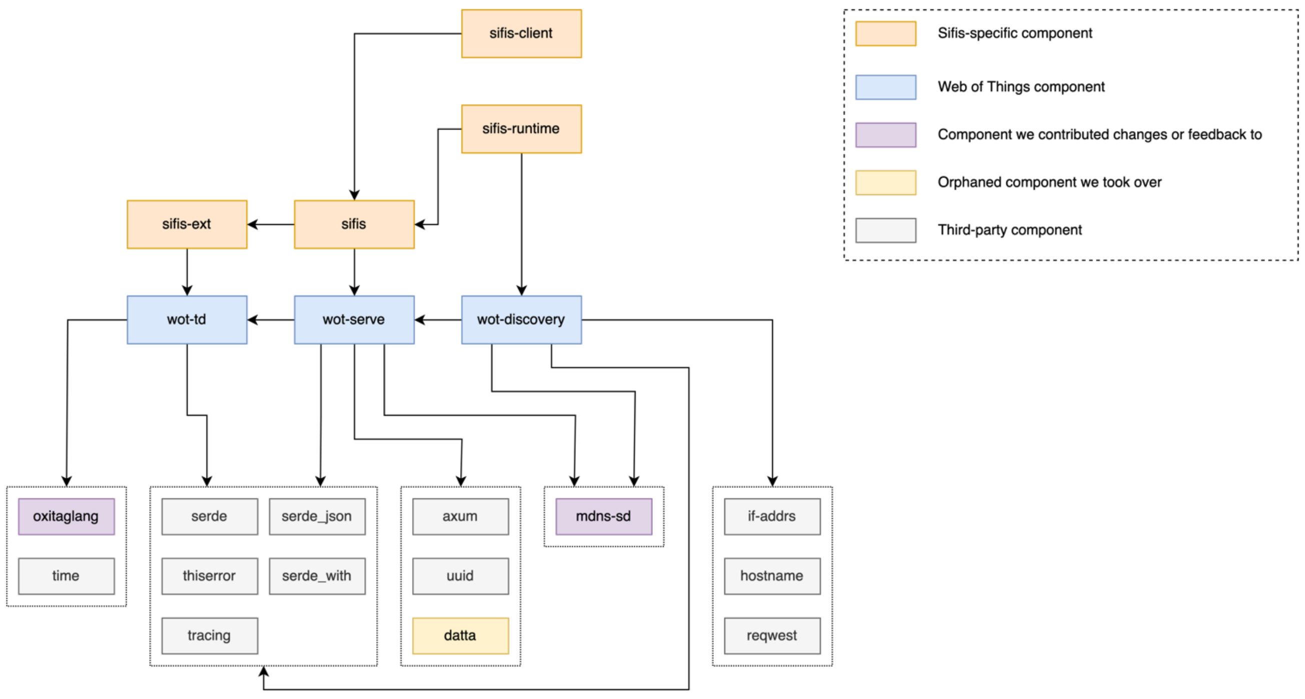


Figure 20: dependency diagram for the SIFIS-Home Rust framework

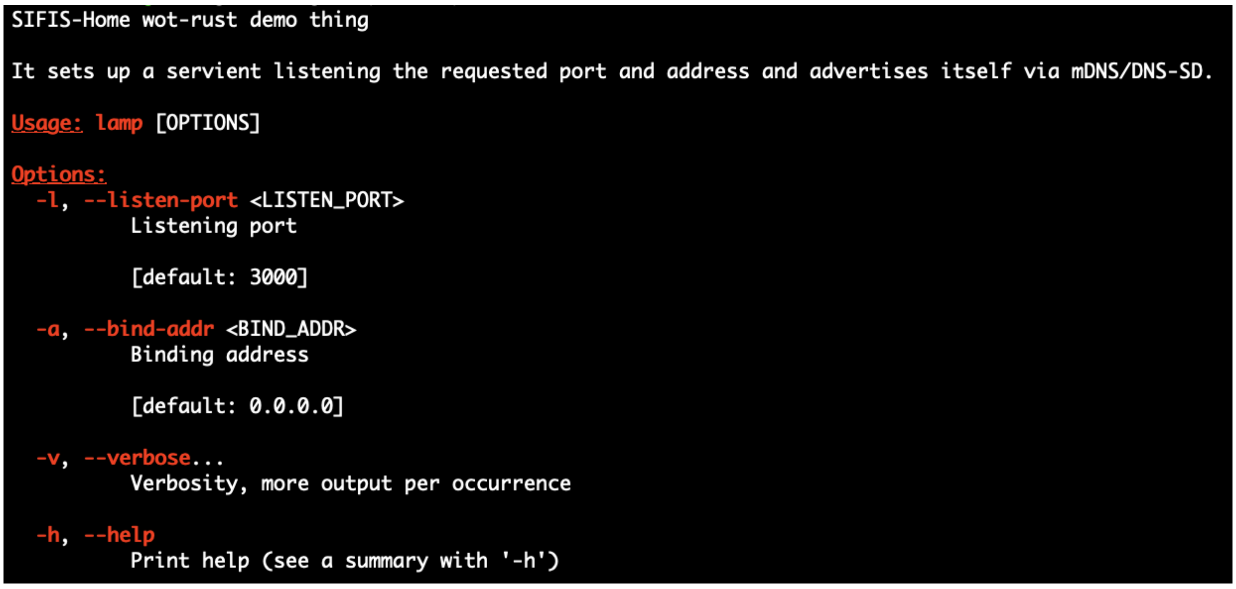
We use WoT as a foundation layer and build on top of it an implementation of the SIFIS-Home framework.

We split the WoT implementation into the following crates:

- **wot-td** that works on serializing, deserializing and extending the **Thing Description** in a type-safe way.
- **wot-serve** provides the building blocks to implement **Servients**, initially supporting building HTTP servients via the web application framework **axum** and advertising its existence via multicast mDNS/DNS-SD.
- **wot-discovery** that provides the components to discover Things in the local network and build a directory.

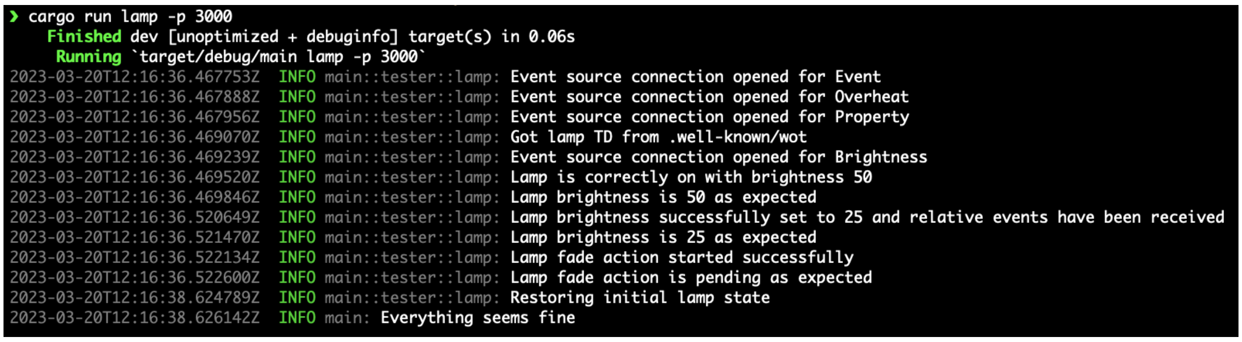
Using them, we will reimplement our SIFIS-Home Hazards ontology as an Extension to the Thing Description and build on top of it the high-level SIFIS-Home API described. The full dependency graph is shown in Figure 20.

In preparation for the testing activities in WP5 and WP6 we prepared a set of simulated devices in the **demo-things** repository using **wot-serve** and a proof of concept of behavioral tester in **wot-test,** porting the Webthings.io python example tester to Rust and expanding it further.



```
SIFIS-Home wot-rust demo thing

It sets up a servient listening the requested port and address and advertises itself via mDNS/DNS-SD.

Usage: lamp [OPTIONS]

Options:
  -l, --listen-port <LISTEN_PORT>
          Listening port

          [default: 3000]

  -a, --bind-addr <BIND_ADDR>
          Binding address

          [default: 0.0.0.0]

  -v, --verbose...
          Verbosity, more output per occurrence

  -h, --help
          Print help (see a summary with '-h')
```

Figure 21 Simulated lamp using the Webthings.io Schemas



```
❯ cargo run lamp -p 3000
    Finished dev [unoptimized + debuginfo] target(s) in 0.06s
     Running `target/debug/main lamp -p 3000`
2023-03-20T12:16:36.467753Z  INFO main::tester::lamp: Event source connection opened for Event
2023-03-20T12:16:36.467888Z  INFO main::tester::lamp: Event source connection opened for Overheat
2023-03-20T12:16:36.467956Z  INFO main::tester::lamp: Event source connection opened for Property
2023-03-20T12:16:36.469070Z  INFO main::tester::lamp: Got lamp TD from .well-known/wot
2023-03-20T12:16:36.469239Z  INFO main::tester::lamp: Event source connection opened for Brightness
2023-03-20T12:16:36.469520Z  INFO main::tester::lamp: Lamp is correctly on with brightness 50
2023-03-20T12:16:36.469846Z  INFO main::tester::lamp: Lamp brightness is 50 as expected
2023-03-20T12:16:36.520649Z  INFO main::tester::lamp: Lamp brightness successfully set to 25 and relative events have been received
2023-03-20T12:16:36.521470Z  INFO main::tester::lamp: Lamp brightness is 25 as expected
2023-03-20T12:16:36.522134Z  INFO main::tester::lamp: Lamp fade action started successfully
2023-03-20T12:16:36.522600Z  INFO main::tester::lamp: Lamp fade action is pending as expected
2023-03-20T12:16:38.624789Z  INFO main::tester::lamp: Restoring initial lamp state
2023-03-20T12:16:38.626142Z  INFO main: Everything seems fine
```

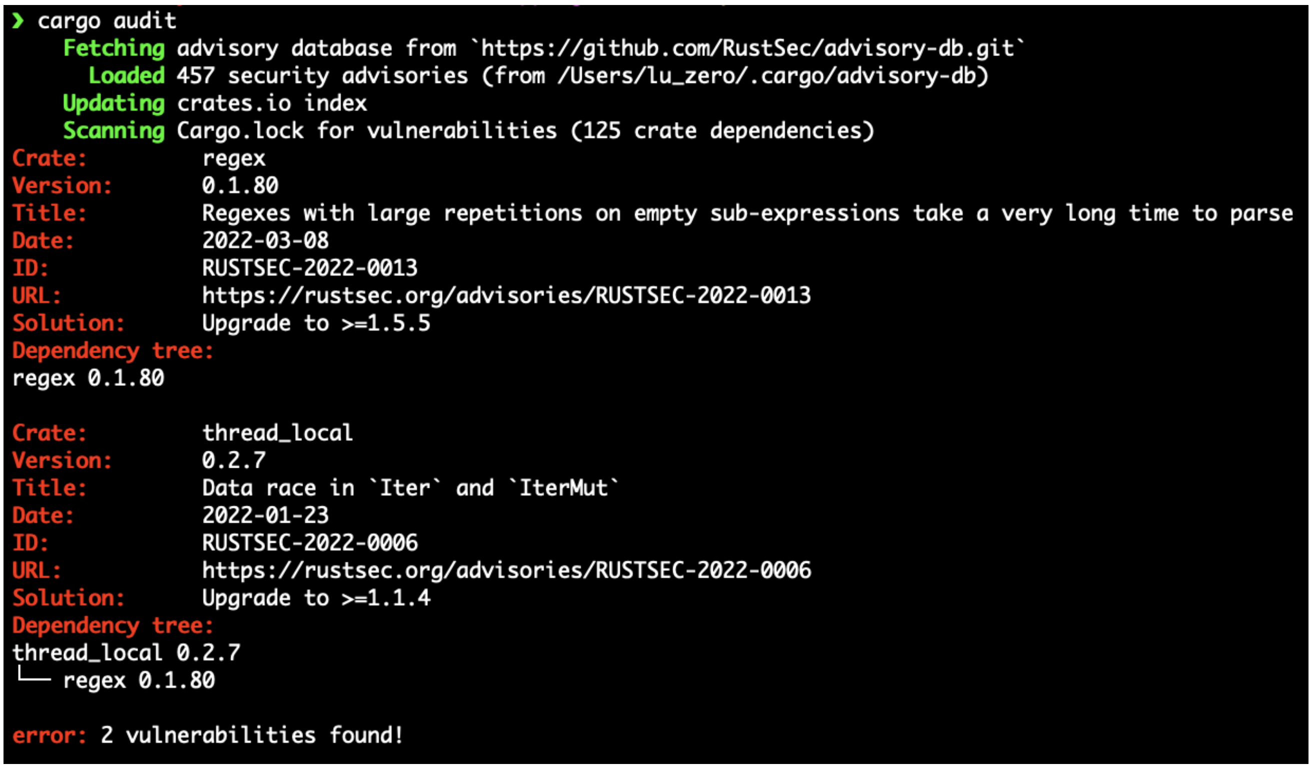
Figure 22 Tester for the lamp above

As the standardization work at the W3C proceeds we will update the codebase to match the latest specification and provide an implementation of our standardization proposals.

4.2 *Tools in Action*

We used **sifis-generate** to create all the WoT projects, and we refined its Rust template from the experience of using it in this scenario.
We tried to keep the code coverage above 85%, aiming to stay well above 90% and ensure every pull request landing is clean of **clippy lints** (as described in deliverable D2.4). All the dependencies do not bring problems thanks to ***cargo audit*** and ensure that our implementation is spec-compliant as much as possible.

When we had to introduce **uritemplate** as a dependency, it triggered a good number of warnings, as shown in Figure 23.



```
❯ cargo audit
    Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
      Loaded 457 security advisories (from /Users/lu_zero/.cargo/advisory-db)
    Updating crates.io index
    Scanning Cargo.lock for vulnerabilities (125 crate dependencies)
Crate:         regex
Version:       0.1.80
Title:         Regexes with large repetitions on empty sub-expressions take a very long time to parse
Date:          2022-03-08
ID:            RUSTSEC-2022-0013
URL:           https://rustsec.org/advisories/RUSTSEC-2022-0013
Solution:      Upgrade to >=1.5.5
Dependency tree:
regex 0.1.80

Crate:         thread_local
Version:       0.2.7
Title:         Data race in `Iter` and `IterMut`
Date:          2022-01-23
ID:            RUSTSEC-2022-0006
URL:           https://rustsec.org/advisories/RUSTSEC-2022-0006
Solution:      Upgrade to >=1.1.4
Dependency tree:
thread_local 0.2.7
└── regex 0.1.80

error: 2 vulnerabilities found!
```

Figure 23: cargo-audit output for uritemplate

4.2.1 datta

To implement a part of wot-serve, we had to implement a mapping between the uritemplates used in the WoT Forms and the axum Paths.
The uritemplate would fit our needs, but it has not been updated for six years, with short-winded tries to update it by other parties, and as shown above, it is showing its age.

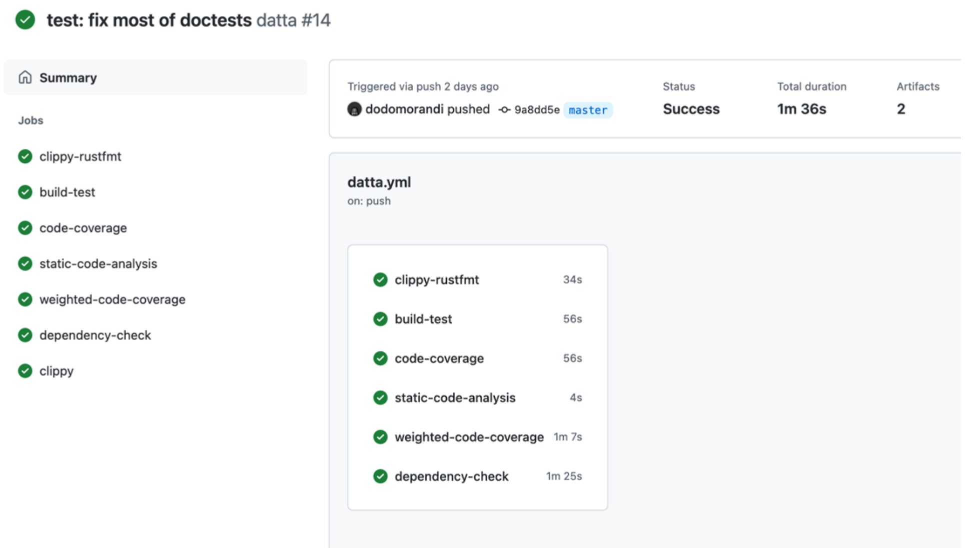
4.2.1.1 Shortcomings

- Pre-2018 codebase.
- Plenty lints from the default rust linter clippy trigger.
- Missing CI.
- Stale dependencies:
    - Regex and thread_local faults were caught by cargo audit.
    - Since we wanted to test how our sifis-generated CI behaved, we had Miri catch at least one problem while running the test suite.

4.2.1.2 Mitigation

- Since the original developer is not responsive and the crate is left in full neglect, we created a full fork of it.

- We made sure to update it to the Rust edition 2021.
- We addressed all the lints clippy found.
- We updated the dependencies, so the cargo audit report is clear.
- We set up the CI using the sifis-generate, as shown in Figure 24.



test: fix most of doctests datta #14

Summary

Jobs
- clippy-rustfmt
- build-test
- code-coverage
- static-code-analysis
- weighted-code-coverage
- dependency-check
- clippy

Triggered via push 2 days ago	Status	Total duration	Artifacts
dodomorandi pushed 9a8dd5e master	Success	1m 36s	2

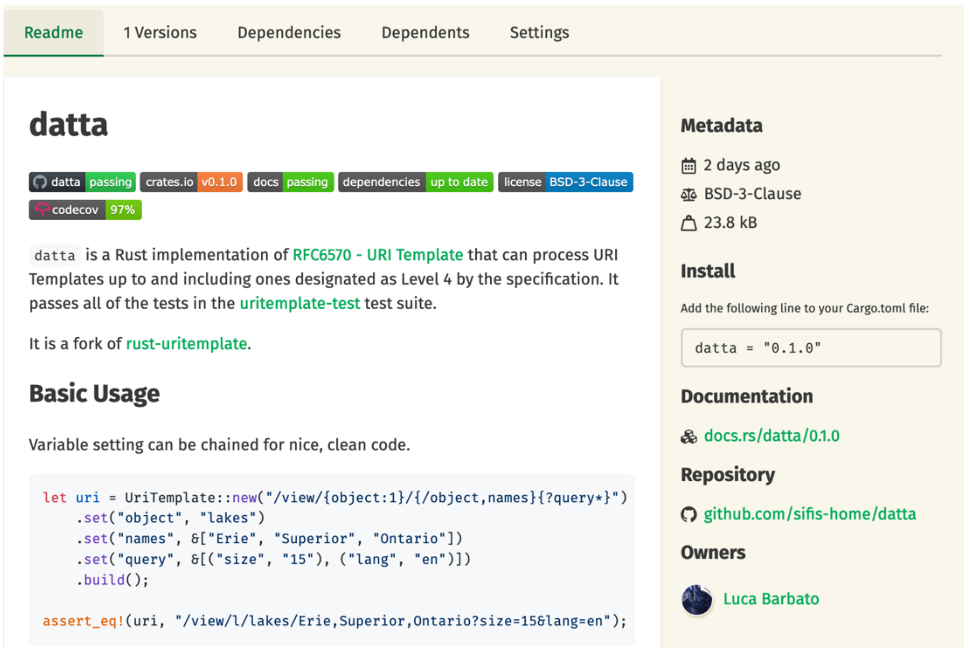
datta.yml
on: push

clippy-rustfmt	34s
build-test	56s
code-coverage	56s
static-code-analysis	4s
weighted-code-coverage	1m 7s
dependency-check	1m 25s

Figure 24: CI for datta

4.2.1.3 Updates and Release

We extended datta API to fit the wot-serve needs, updated documentation and released it with a new name, as shown in Figure 25.



datta

`datta` is a Rust implementation of RFC6570 - URI Template that can process URI Templates up to and including ones designated as Level 4 by the specification. It passes all of the tests in the uritemplate-test test suite.

It is a fork of rust-uritemplate.

Basic Usage

Variable setting can be chained for nice, clean code.

```
let uri = UriTemplate::new("/view/{object:1}/{/object,names}{?query*}")
    .set("object", "lakes")
    .set("names", &["Erie", "Superior", "Ontario"])
    .set("query", &[("size", "15"), ("lang", "en")])
    .build();

assert_eq!(uri, "/view/l/lakes/Erie,Superior,Ontario?size=15&lang=en");
```

Metadata

2 days ago

BSD-3-Clause

23.8 kB

Install

Add the following line to your Cargo.toml file:

```
datta = "0.1.0"
```

Documentation

docs.rs/datta/0.1.0

Repository

github.com/sifis-home/datta

Owners

Luca Barbato

Figure 25: crates.io for datta (https://crates.io/crates/datta)

5 Conclusions and Future Works

This deliverable presents a comprehensive set of tools and methodologies designed to facilitate the development of secure, efficient, and maintainable IoT software within the SIFIS-Home project. This document and the accompanying code and documentation provide developers with the necessary resources to apply the developer guidelines for creating secure, privacy-aware, policy-based IoT source code introduced in D2.4.

Throughout this deliverable, we have introduced a range of software lifecycle tools, including sifis-generate, complex-code-spotter, and weighted code coverage. We have also detailed our contributions to the open-source projects we included in implementing our workflows described in D2.4. In addition to software lifecycle tools, this document has presented API labelling tools and methodologies, such as the SIFIS-Home Hazards Ontology and contract-based security methodology, to ensure a secure and controlled environment for IoT applications.

As the complete working example demonstrates, the developer tools have been successfully applied in a real-world setting.

Moving forward, we envision several areas of potential improvement and expansion for the developer tools:

- Integration of additional programming languages and platforms to further extend the applicability of the tools and methodologies.
- Enhancement of the API labelling tools to provide more granular and customizable hazard assessments, accommodating a wider range of IoT devices and use cases.
- Extension of the SIFIS-Home Hazards Ontology to encompass a broader range of potential hazards and associated mitigation strategies.
- Expansion of the educational resources and documentation to facilitate the adoption and implementation of the developer tools by a wider audience, including more comprehensive tutorials, sample projects, and case studies.

In conclusion, the developer tools presented in this deliverable mark a significant contribution in the SIFIS-Home project, providing a strong foundation for the secure and efficient development of IoT software.

6 References

[TD, 2022] Thing Description (TD) Ontology. URL: https://www.w3.org/2019/wot/td

[SHO, 2022] The SIFIS-Home Hazards Ontology Specification. URL: https://purl.org/sifis/hazards

Glossary

Acronym	Definition
ABI	Application Binary Interface
API	Application Programming Interface
AST	Abstract Syntax Tree
CI	Continuous Integration
DNS	Domain Name System
DNS-SD	DNS-based Service Discovery
CSV	Comma Separated Values
GCC	GNU Compiler Collection
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
mDNS	Multicast DNS
MIT	Massachusetts Institute of Technology
MSRC	Microsoft Security Response Centre
RPC	Remote Procedure Call
SHO	SIFIS-Home Hazards Ontology
SIFIS-Home	Secure Interoperable Full-Stack Internet of Things for Smart Home
TD	Thing Description
WoT	Web of Things
WP	Work Package
XACML	eXtensible Access Control Markup Language
YAML	YAML Ain't Markup Language