Getting Started

This getting started guide explains how to get started with Prusti – a verifier for the Rust programming language. We provide two options for testing Prusti: a virtual machine and a Docker image. While the virtual machine is easier to use, we observed that Prusti inside a VM is up to 10 times slower. Unfortunately, the slow down not only causes problems for reproducing the performance measurements, but can also lead to spurious verification errors that would not be generated otherwise (see the ASSERT_TIMEOUT parameter in section V). Therefore, we also provide a Docker image that enables testing Prusti without VM's performance overhead in case a Linux machine is available.

In this guide we explain how to set up the virtual machine, how to use Prusti in general and how to set up the Docker container with Prusti. Then, the step by step guide has instructions on how to check the main parts of our evaluation in VM and the full evaluation in the Docker container.

I. Virtual Machine

You can check whether the ZIP file with the artefact was downloaded without any errors by checking its md5 checksum, e.g. on Linux/Mac:

```
md5sum VM.zip
```

The output should be “67c4ad32346df59e64b9688f762bb92b VM.zip”

The ZIP file contains a VirtualBox image with Ubuntu that has Prusti installed as a command line tool and as an IDE plugin. To start the virtual machine, follow these steps:

1. Download and install VirtualBox from https://www.virtualbox.org/
2. Extract the ZIP archive.
3. Start VirtualBox.
4. Choose File --> Import VM… (on Linux) or Import Appliance… (on Windows/Mac).
5. Select the extracted Prusti.ova file and follow the instructions. It is highly recommended to provide at least 4 GB of RAM.

If the import was successful, there should be *Prusti* in the list of virtual machines. Select it and click Start. The OS should log in without asking for a password. In case you need it, the user is called `user` and the password is also `user`.

II. Content of the Virtual Machine

On the desktop there is a folder called `prusti`. Inside it:

- the `prusti-dev` folder contains Prusti's source code and pre-compiled binaries;
- the `viper` and `z3` folders contain the Viper and Z3 dependencies needed to run Prusti;
- the `demo` folder contains small Rust programs and a configuration file for Prusti, which are used as support material for this guide;
- the `evaluation-1`, `evaluation-2` and `evaluation-3` folders contain the material used in Prusti's evaluation (Sec. 7.2 of the paper).

III. Trying Examples

There are two ways to run Prusti: from the command line or from the Visual Studio Code editor. The former can be easily integrated in scripts (e.g. benchmarks or tests), while the latter makes it easy to change and iterate over different versions of a program and its specification (e.g. when verifying a program from scratch).

Using Prusti from the command line

1. Open a terminal.

2. Change the current directory to the demo directory:

   ```
   cd "Desktop/prusti/demo"
   ```

3. Run Prusti on `demo.rs`:

   ```
   prusti-rustc demo.rs
   ```

   Note that Prusti reports a verification error, because the assertion `assert!(max(1, 2) == 1)` in the main function would cause a panic if executed.

   Prusti's output, indeed, reports `Verification failed`, pointing out that at line 19 the condition in the `assert!(..)` statement might not hold.

4. To fix the reported error, open the program in an editor (not Visual Studio Code), e.g.:

   ```
   gedit demo.rs
   ```

5. Replace line 19 with `assert!(max(1, 2) == 2)`, save the change and close the editor.

6. Run Prusti on `demo.rs` as in step 3. Now the verification of the program succeeds and the last line in the output is `Successful verification of 3 items` indicating that Prusti successfully verified the three functions in `demo.rs`.

7. Replace line 19 with `assert!(max(1, 2) == 1)`, to bring the program back to its original version, as required for the steps in the next section.

Using Prusti from Visual Studio Code

1. Double-click on the `prusti` folder on the desktop and open the `demo` folder.

2. Double-click on the `demo.rs` file to open it in the Visual Studio Code editor. As soon as the editor starts, the Prusti extension for Visual Studio Code will start verifying the program. If the editor was already open, save the file to start verifying the program with Prusti.

   Note that Prusti reports a verification error, because the assertion `assert!(max(1, 2) == 1)` in the main function would cause a panic if executed.

3. For a few seconds the editor will show `Running Prusti...` in the status bar at the bottom of the editor. Please wait until the status bar shows `Verification failed`.

4. In the menu, click on `View --> Problems`. You can see that Prusti reports a verification error for the assertion at line 19. The error is listed in the `Problems` tab and the corresponding source code location is underlined with a red line in the code.

5. To fix the program, replace line 19 with `assert!(max(1, 2) == 2)` and save the change. The Prusti extension will automatically start, and at the end the status bar will show `Verification succeeded`.

Enabling overflow checks

Prusti can be configured via a `Prusti.toml` configuration file or via some environment variables (whose name always begins with `PRUSTI_`). For example, you can configure Prusti to check for absence of integer overflow errors in two different ways:

1. By creating a file named `Prusti.toml` containing `CHECK_BINARY_OPERATIONS=true` in the folder from which Prusti is run. We already provide a `Prusti.toml` file that you can edit in the `demo` folder. If Prusti is run using the Visual Studio Code extension then the `Prusti.toml` file must be placed in the same folder as the programs to be verified.

   An example `Prusti.toml` file that enables overflow checking can be also seen in the `prusti/evaluation-3/with-overflow-checks` folder.

2. By setting the `PRUSTI_CHECK_BINARY_OPERATIONS` environment variable to `true`:

   ```
   PRUSTI_CHECK_BINARY_OPERATIONS=true prusti-rustc program-to-be-verified.rs
   ```

   This overrides any setting specified in the `Prusti.toml` file.

If you now try to verify `demo.rs` again with the overflow checks enabled, Prusti will report `attempt to add with overflow` in the following function:

```
fn average(x: u32, y: u32) -> u32 {
    (x + y) / 2
}
```

The problem is that `x + y` may overflow when executed with big values of `x` and `y`. A possible solution would be to add a precondition that requires the sum of `x` and `y` to be no larger than the maximum value of `u32` type:

```
#[requires="x + y <= std::u32::MAX"]
fn average(x: u32, y: u32) -> u32 {
    (x + y) / 2
}
```

This way, Prusti checks that in the function implementation the requirement is sufficient to avoid any overflow. Prusti will also check that any call to `average` satisfies the specified requirement.

More advanced examples

We will now try more advanced examples that we obtained from the Rosetta website (https://rosettacode.org). You may open the following programs in Visual Studio Code, to run Prusti in the IDE. Remember that you need to save the file (`CTRL+S`) in order to verify it with Prusti. If you instead prefer the command line version, note that Prusti needs to be run from the same folder of the program to be verified in order to use the correct configuration file that we prepared in the VM.

For many examples in the `demo` and `prusti/evaluation-3` folders Prusti will warn either about failing lints (e.g. unused functions) or uses of some partially supported Rust features. All such warnings can be ignored for the examples in these folder: the currently implemented syntactic checks are too coarse grained to detect that the examples are actually supported.

With overflow checks enabled, try verifying `prusti/demo/buggy_binary_search_1.rs`. The implementation (obtained from Rosettacode: https://rosettacode.org/wiki/Binary_search#Rust) contains a bug, which we discovered trying to verify the correctness of the program. Indeed, the verification fails with an error:

```
error: [Prusti] loop invariant might not hold at the end of a loop iteration.
```

The error message also points out the location of the failing loop invariant. The IDE highlights it if you click on the error message *the failing assertion is this one* in the tab that you can open by clicking on *View → Problems*.

```
note: the failing assertion is this one -->
tests/verify/pass-overflow/rosetta/Binary_search_shared_monomorphised.rs:95:18
    |
95  | #[invariant="result.is_none() ==>
    |              ^^^^^^^^^^^^^^^^
96  | (forall k: usize :: (base + size <= k && k < arr.len()) ==> *elem != arr.lookup(k))"]
...
```

The failing loop invariant expresses that the element searched-for is not contained in the range above `base + size` index. The implementation has a bug and this invariant actually does not hold: for example, the function would return that 6 is not contained in array `[1, 2, 3, 4, 5, 6]`. An executable version of the example that demonstrate the bug can be found here: https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=ca6b3abd252c819cec76d9b12de22645.

Try now verifying `prusti/demo/buggy_binary_search_2.rs`: a second implementation of binary search taken from Rosettacode. Also this implementation has an issue, which is correctly caught by Prusti: the addition in `mid = (high + low) / 2;` may overflow.

You can find the fixed (and verified) implementation of binary search in the file `prusti/evaluation-3/with-overflow-checks/Binary_search_shared_monomorphised.rs`. An attempt to verify it should succeed: the IDE should display `Verification succeeded with warnings` in the status bar, while the command line version of Prusti should report `Successful verification of 5 items`.

You can try more examples from the `prusti/evaluation-3` folder. For many examples, Prusti will warn either about unused functions or uses of some partially supported Rust features. All such warnings can be ignored for the examples in the `demo` and `evaluation-3` folder: as mentioned above, the currently implemented syntactic checks are too coarse grained to detect that the examples are actually supported.

IV. (Linux only) Docker image for performance evaluation

The following instructions are only for Linux OS because on Windows and Mac OS Docker containers are executed inside a virtual machine and, therefore, are unlikely to perform better than the provided Prusti VM.

To start the Prusti Docker container, follow these steps:

1. Install Docker by following the instructions at https://docs.docker.com/v17.09/engine/installation/linux/docker-ce/ubuntu/.

2. Open a terminal and check that Docker is installed and usable:

   ```
   docker --version
   ```

   This command should successfully show the Docker version. If not, Docker is not installed correctly.

3. Download the Docker image for the Prusti artefact

   ```
   sudo docker pull fpoli/prusti-artefact
   ```

4. Check the hash of the downloaded image

   ```
   sudo docker inspect --format='{{index .RepoDigests 0}}' fpoli/prusti-artefact
   ```

The output should contain

```
sha256:8f1d9d079653cb70fc420e642a16bac676c8d70051e1e1e3336f2ebad26534bc
```

5. Start a Docker container and open a terminal in it:

```
sudo docker run -it fpoli/prusti-artefact /bin/bash
```

This should open a terminal in the `tmp/prusti-dev` folder of the Docker image, where we placed Prusti's source code.

6. Check that Prusti is available:

```
prusti-rustc --version
```

7. Try to verify an example from the test suite:

```
prusti-rustc evaluation/artifact/examples/Selection_sort.rs
```

For this example, Prusti should report two warnings that can be ignored and show a message `Successful verification of 2 items`.

8. You can exit the container by using the `exit` command.

V. Additional Notes

- Since we observed that, for a few test cases, the presence of non-linear arithmetic caused slow running of the SMT solver, we incorporated a per-assertion timeout. The downside of using the timeout is that on slower machines (for example, when executed inside a VM) Prusti may report spurious errors. The timeout can be controlled via the `ASSERT_TIMEOUT` configuration parameter (value is in milliseconds). For example, to increase the timeout to 1 minute, set `ASSERT_TIMEOUT` to 60000 (the default is 10 seconds). This parameter can be set by adding e.g. a line `ASSERT_TIMEOUT = 60000` to the `Prusti.toml` configuration file (in the folder where Prusti is run), or by setting an environment variable with the command `export PRUSTI_ASSERT_TIMEOUT=60000` (note the `PRUSTI_` prefix).

- You may notice that all Rust programs containing Prusti specifications (the `requires`, `ensures`, `invariant` annotations) begin with an `extern crate prusti_contracts` line. This is a technical requirement to let Prusti use special functions in the specifications (e.g. `old`, `before_expiry`, `after_expiry`). The `prusti_contract` crate is currently not distributed over https://crates.io/, but is automatically provided by Prusti.

- When it runs, Prusti creates two temporary folders named `log` and `nll-facts`. They can be safely deleted after Prusti terminates.

- It is also possible to run Prusti directly on your computer. You can find the setup instructions in the README file in the Prusti GitHub repository: https://github.com/viperproject/prusti-dev. We have successfully tested Prusti on Ubuntu 18.04 and using the Windows Linux Subsystem. The VM and the Docker image contains tools built from the following commits:

  - Prusti (https://github.com/viperproject/prusti-dev): 205019dcd21e61521eeef03ca9153aa938b8b256
  - Viper Symbolic Execution backend (https://bitbucket.org/viperproject/silicon/): 00186aee346aa824505f7f0816062180c21a2d7e
  - Viper language (https://bitbucket.org/viperproject/silver/): cff1dc4ea21c48097ffd16e4a98ab67a35fb2795
  - Z3 (https://github.com/Z3Prover/z3/): 53514281d21ab8e396e199b125ee0d33837c36b4

Step by Step Guide

This guide describes how to reproduce the evaluation presented in Sec. 7 of the paper. In particular:

- Contribution (5), Sec 7.1: We provide an implementation of our technique as a plugin for the Rust compiler. The source code can be found in the virtual machine.
- Sec. 7.2.1: We provide a test suite of more than 300 correct and incorrect Rust programs annotated with expected verification errors. Supported by sections I and V.
- Contribution (5) and Sec. 7.2.1: We construct core proofs on several thousand unannotated Rust functions from the 500 most popular Rust crates that fall within our supported language subset. Supported by sections II and VI.
- Sec. 7.2.2: We prove absence of overflows in examples that check for overflows at runtime. Supported by sections III and VI.
- Contribution (5) and Sec. 7.2.3: We verify a range of stronger properties (via our specification language) for selected Rust implementations. Supported by section IV.

This guide is divided into two parts: the first part covers checking the evaluation in the VM while the second part covers in the Docker container.

Virtual Machine

I. Development tests (Sec. 7.1)

1. Open a terminal and move to the `prusti-dev` folder:

   ```
   cd ~/Desktop/prusti/prusti-dev
   ```

2. Prusti provides two programs: `prusti-rustc`, which can be used as a replacement of `rustc` to verify Rust programs, and `cargo-prusti`, which can be used as a replacement of `cargo check` to verify Rust crates. They come already compiled in the virtual machine. If you want to recompile them run

   ```
   make release
   ```

   This command should take less than one minute, because no change has been made to the code. Compiling from scratch typically takes around 25 minutes.

3. Prusti has a development test suite of more than 300 correct and incorrect Rust programs. To list them:

   ```
   find prusti/tests/verify/ -name '*.rs'
   ```

   The tests are divided amongst the following folders:

   i. the `pass` folder contains the programs that should verify successfully.
   ii. the `pass-overflow` folder contains the programs that should verify successfully with overflow checks enabled.
   iii. the `fail` folder contains the programs that should cause at least one verification error, annotated as a special comment beginning with `//~ ERROR`.

iv. the `fail-overflow` folder contains the programs that should cause at least one verification error when overflow checks are enabled. The expected error is again annotated as a special comment.

4. You can run the whole development test suite (about 300 small programs) using the following command:

   ```
   make test-examples
   ```

   Note that running all of the tests typically takes around 2 hours. The test suite checks for each example program that the verification outcome and error messages match the expected ones. A single test (fail/unsupported_attribute.rs) is currently disabled because it is invalid: an `ignored` message is reported for it in the output.

5. To only run the tests matching a particular name (e.g. `pass/enum/basic.rs`) you can use:

   ```
   TESTNAME=pass/enum/basic.rs make test-examples
   ```

   The test above on a virtual machine should take around 2 minutes to run.

   The output of the command above may be more verbose than expected: it reports multiple times that `0` tests failed, that many tests were filter out and that a few tests were successfully executed. All tests are expected to pass. If instead some test fails the last line in the output would contain `error: test failed`.

II. Evaluation 1 (Sec. 7.2.1)

We estimate that running the full evaluation described in the paper in a virtual machine would take several days, because we experienced that Prusti runs up to 10 times slower when run inside a virtual machine than on bare metal. To run the full evaluation outside the virtual machine please refer to section V of this guide.

To avoid compiling all 500 most popular crates and running the automatic filtering of supported functions on them, we provide in the virtual machine the intermediate files that can be used to manually run Prusti on a chosen crate:

- `prusti/evaluation-1/supported-crates.csv` contains the list of the 352 crates over (out of 500) that compile successfully within 15 minutes (on bare metal) using the standard compiler and the compilation flags described in the paper.
- for each of the 352 crates, we provide the list of functions supported by Prusti and a `Prusti.toml` file which configures Prusti to verify only the supported functions in the crate.

The following steps describe how to verify the supported functions in one of the 352 crates. The verification of all such crates should succeed with the message `Successful verification`.

1. Open a terminal and move to the `~/Desktop/prusti/evaluation-1` folder:

   ```
   cd ~/Desktop/prusti/evaluation-1
   ```

2. Choose a random crate from the list of those that compiled successfully within 15 minutes:

   ```
   cat supported-crates.csv | sort -R | head -n 1
   ```

   If you also want to avoid the crates that contain no supported functions use the following command instead of the previous one:

```
grep -l '^"' */Prusti.toml | uniq | cut -d/ -f1 | sort -R | head -n 1
```

In the following steps we'll assume that the chosen crate is `027_semver`.

3. Move to the crate's folder

```
cd 027_semver
```

4. You can inspect the list of supported functions in the `Prusti.toml` file:

```
cat Prusti.toml
```

In the case of the `027_semver` crate, the file will contain the definition of a whitelist of three elements. If the chosen crate has no supported functions, the last two lines in `Prusti.toml` will be

```
Whitelist = [
]
```

5. Run Prusti on the crate:

```
cargo clean
cargo-prusti
```

This command will compile all the dependencies and will generate the core proof for the supported functions of the crate, checking the core proof with Viper. In total it may take several minutes.

When finished, the output should contain a line starting with `Successful verification`, meaning that Viper successfully verified the core proofs.

6. If you wish, you can go back to step 1 and repeat the steps to run Prusti on another crate.

III. Evaluation 2 (Sec. 7.2.2)

For the same reason explained in the previous section, we provide the intermediate files that can be used to manually run Prusti on a chosen crate, without having to compile and filter the supported functions from the 500 most popular crates. To run the full evaluation outside the virtual machine please refer to section VI of this guide.

The `evaluation-2` folder contains a folder for each function that contains operations that could panic due to an overflow or assertion failure. Each of those folders contain a `Prusti.toml` file that configures Prusti to check absence of panics and of integer overflows for one particular function.

The list of folders in which the command at step 6 is expected to succeed with no verification errors can be shown using the following command in a terminal:

```
cat ~/Desktop/prusti/evaluation-2/successful.csv | cut -d, -f1
```

These folders correspond to the 52 cases of functions for which Prusti verifies absence of panics and overflows without need of program annotations. In all other cases Prusti should report a verification error.

1. Open a terminal and move to the `~/Desktop/prusti/evaluation-2` folder:

   ```
   cd ~/Desktop/prusti/evaluation-2
   ```

2. You can check the number of functions classified as `definitely supported` by automatic filtering that might panic due to overflows:

   ```
   cat */Prusti.toml | grep '^"' | wc -l
   ```

   This should show 532. The number is slightly bigger than what reported in the paper (520) because of some improvements to Prusti in the meantime.

3. Choose a random folder

   ```
   ls | sort -R | head -n 1
   ```

   In the following steps we'll assume that the chosen folder is `111_num-complex_2`.

4. Move to the chosen folder

   ```
   cd 111_num-complex_2
   ```

5. You can see the function that is going to be verified in the `Prusti.toml` file:

   ```
   cat Prusti.toml
   ```

6. Run Prusti on the crate:

   ```
   cargo clean
   cargo-prusti
   ```

   This command will compile all the dependencies (you may notice that on one dependency Prusti tries to verify 0 items; we plan to avoid this unnecessary call in the future) and will check for absence of panics and of integer overflows. In total it may take several minutes.

   In case of `111_num-complex_2` Prusti reports a verification error and terminates with messages `aborting due to previous error` and `could not compile`:

   ```
   error: [Prusti] assertion might fail with "attempt to multiply with overflow"
      --> src/lib.rs:804:51
       |
   804 |                     Complex::new(self * other.re, self * other.im)
       |                                                 ^^^^^^^^^^^^^^^^
   ...

   Verification failed

   error: aborting due to previous error

   error: Could not compile `num-integer`.
   ```

The error message correctly points to a potential overflow in the method `mul`, which multiplies the components of two complex numbers. In this case the caller of the method should ensure that the components are small enough, and Prusti makes it possible to explicitly write and check this assumption by adding a precondition to the method.

7. If you wish, you can go back to step 1 and repeat the steps to run Prusti on another function.

IV. Evaluation 3 (Sec. 7.2.3)

The following steps were used to generate the performance measurements of the table in Figure 7 of the paper. The same examples are also available in the folder `~/Desktop/prusti/evaluation-3` and can be manually verified: each line in the table corresponds to a `*.rs` files with a self-descriptive name. As was already mentioned above, Prusti inside a VM is significantly slower. To obtain reliable measurements, see at the end of this section how to run the benchmark in the Docker image.

1. Open a terminal and move to the `~/Desktop/prusti/prusti-dev` folder:

   ```
   cd ~/Desktop/prusti/prusti-dev
   ```

2. Use the following commands to prepare Prusti and run the benchmark:

   ```
   make release
   rm -f evaluation/benchmark/bench.csv
   python3 evaluation/benchmark/benchmark.py
   python3 evaluation/benchmark/analyse.py
   ```

   The benchmark should take about 30 minutes.

3. The last 16 lines in the output represent a table of 5 columns:

   - the name of the program (first column in Fig. 7 of the paper)
   - the average time required by Prusti to verify the example (fifth column in Fig. 7 of the paper)
   - the standard deviation of the measurement in the previous column
   - the average time required by Viper to verify the encoding of the example (sixth column in Fig. 7 of the paper)
   - the standard deviation of the measurement in the previous column

   The measurements may differ from the table in Fig. 7 of the paper for two reasons:

   - The virtual machine slows down the tool, in some cases by a factor of 10. To run the measurements directly on your computer, please see the two options below.
   - Since submitting the paper, we implemented some optimizations in Prusti that in some cases decrease the time required by Prusti to verify the encoding.

To run the measurements above outside the virtual machine there are two options:

1. On your computer, start the Prusti Docker container with

   ```
   sudo docker run -it fpoli/prusti-artefact /bin/bash
   ```

   and run the commands described in step 2 of this section.

2. On your computer, set up Prusti as described in the README file in the Prusti GitHub repository (https://github.com/viperproject/prusti-dev) and run the commands described in step 2 of this section from the root of the repository.

Docker Container

V. Development tests using the Docker image (Sec. 7.1)

1. Create a container and open a terminal in it:

   ```
   sudo docker run -it fpoli/prusti-artefact /bin/bash
   ```

2. Compile Prusti and run the development test suite (about 300 small programs)

   ```
   make build
   make test-examples
   ```

   The test suite is expected to terminate successfully. If instead some test fails the last line in the output would contain `error: test failed`.

VI. Evaluation 1 and 2 using the Docker image (Sec. 7.2.1, 7.2.2)

To run Prusti without the overhead caused by a virtual machine we provide a Docker image, which should be executed on a non-virtualized Linux OS.

1. Create a container and open a terminal in it:

   ```
   sudo docker run -it fpoli/prusti-artefact /bin/bash
   ```

2. Start the script which will run from scratch the evaluation described in Sec. 7.2.1 and 7.2.2. of the paper:

   ```
   ./evaluation/script/artifact-evaluation.sh
   ```

Note that the script should take about 24 hours to terminate, generating about 11 GB of intermediate files. Please ignore the output of the script: it is expected to contain many error messages, coming e.g. from crates that fail to compile with the standard Rust compiler.

The script will perform the following steps, of which you can check the output files:

1. Download the source code of the 500 most popular crates (as of November 2, 2018).

   This step will produce 500 folders (one per crate) in the `/tmp/crates` folder.

2. Filter the crates that compile successfully within 15 minutes using the standard compiler and the compilation flags described in the paper.

   This step produces the file `/tmp/crates/supported-crates.csv`. The file should contain exactly 352 lines. To check the number of lines in it use:

   ```
   cat /tmp/crates/supported-crates.csv | wc -l
   ```

   The expected content of the file is available at `/tmp/prusti-dev/evaluation/crates/supported-crates.csv`.

3. Filter the functions that are definitely supported by Prusti.

   This step produces the file `/tmp/crates/filtering-report.csv`. The second column in the CSV should only contain `true` values. A `true` value indicates that the filtering succeeded as expected without errors or crashes. To check if there is any `false` value use:

   ```
   cat /tmp/crates/filtering-report.csv | grep false
   ```

   The output of the previous command should be empty.

4. Prepare the whitelists of functions to be verified with Prusti.

   This step produces the file `/tmp/crates/whitelist-report.csv`. The sum of the values in the numeric columns are expected to be:

   - Number of procedures: 56236
   - Number of supported procedures: 11939
   - Number of supported procedures using assertions: 532

   You can compute the sums using the following commands:

   ```
   (cat /tmp/crates/whitelist-report.csv | grep _ | cut -d, -f2 | tr '\n' '+'; echo 0) | bc
   (cat /tmp/crates/whitelist-report.csv | grep _ | cut -d, -f3 | tr '\n' '+'; echo 0) | bc
   (cat /tmp/crates/whitelist-report.csv | grep _ | cut -d, -f4 | tr '\n' '+'; echo 0) | bc
   ```

   As described in the paper, we manually blacklisted ten unusually large functions. This blacklist is available in the following file: `/tmp/prusti-dev/evaluation/crates/global_blacklist.csv`.

5. Generate and verify the core proof for each supported crate (Sec. 7.2.1).

   This step produces the file `/tmp/crates/coarse-grained-verification-report-supported-procedures.csv.csv`. The second column in the CSV should only contain `true` values. A `true` value indicates that the generation and verification of the core proof succeeded as expected without errors or crashes. To check if there is any `false` values use:

   ```
   cat /tmp/crates/coarse-grained-verification-report-supported-procedures.csv.csv | grep false
   ```

   The output of the previous command should be empty.

6. Check for absence of panics and overflows (Sec. 7.2.2).

   This step produces the file `/tmp/crates/fine-grained-verification-report-supported-procedures-with-assertions.csv.csv`. The second column in the CSV should contain exactly 52 `true` values and 480 `false` values. A `true` value indicates that the verification of the function succeeded without need of manual intervention. It is expected that such verification fails on most examples, as presented in the paper. To count the number of `true` and `false` values use:

   ```
   grep true -c fine-grained-verification-report-supported-procedures-with-assertions.csv.csv
   grep false -c fine-grained-verification-report-supported-procedures-with-assertions.csv.csv
   ```

   The output of the previous commands should be 52 and 480.

   The precise list of the 52 functions that have a `true` value is available at `/tmp/prusti-dev/evaluation/crates/successful-fine-grained.csv`.

At the end, in case you want you want to clean up the containers used by Docker and regain disk space, please refer to this guide: https://docs.docker.com/config/pruning/.

VII. Evaluation 3 using the Docker image (Sec. 7.2.3)

To run the evaluation 3 in a Docker container, start the Prusti Docker container with:

```
sudo docker run -it fpoli/prusti-artefact /bin/bash
```

Then follow the instructions in section IV from step 2.