TRUST IN SOFT

TrustInSoft Analyzer *Beta*

Getting Started — *Newcomers* edition

Version 1.0

Contents

1 Introduction

This tutorial is an introduction to the *Newcomers* edition of TrustInSoft Analyzer, which allows you to verify test drivers and guarantee that the code exercised by these test drivers does not rely on undefined behaviors. The document describes how to use TrustInSoft Analyzer through a couple of existing examples, and how to run it on your own projects.

2 Getting started

1. Go to `https://github.com/TrustInSoft/TrustInSoft` and fork the project.

   The `TrustInSoft` repository contains a collection of tests that show how to use TrustInSoft Analyzer to formally verify your test suites. Each test directory in the `examples` directory is organized as follows:

   - A `README.md` file that describes the purpose of the test.
   - The `tis.config` file necessary to run TrustInSoft Analyzer on the test.
   - C source files.

2. Go to `http://taas.trust-in-soft.com/` and click on *Sign in*:

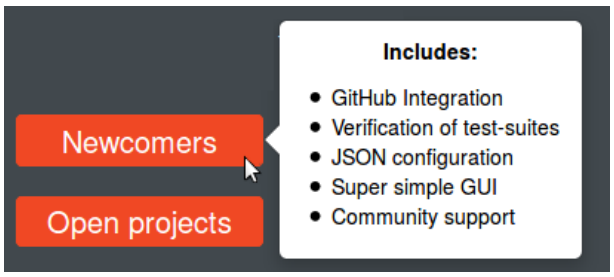
   

3. A new window appears, click on *Sign in with GitHub*:

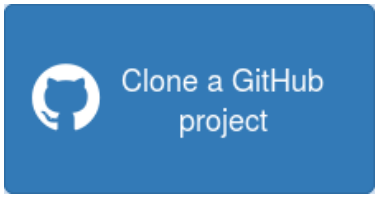
   Prerequisite You must have a public email address in your GitHub account to be able to use TrustInSoft Analyzer.

4. Select the *Newcomers* edition of TrustInSoft Analyzer:

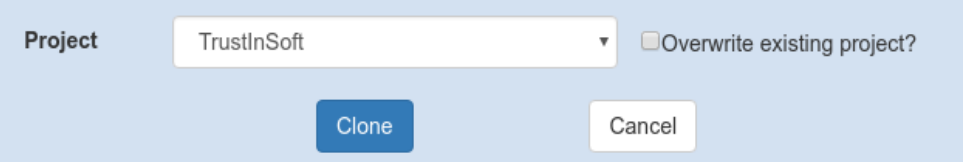
   A new window appears displaying: *"Starting your analyzer..."*. Starting the interface takes about 30 seconds.

5. From the TrustInSoft Analyzer interface, a pop-up should appear to ask you to clone a project into your workspace.

If you have already cloned any of your projects, the pop-up will not appear. If so, click on the top-right menu button ☰.

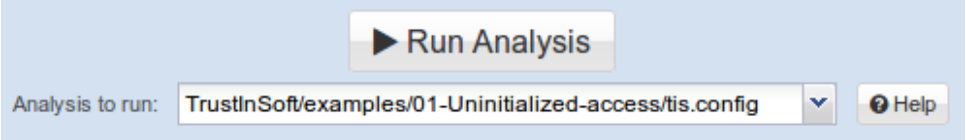
Clone the `TrustInSoft` project into your workspace:



3 Verifying existing tests

3.1 Uninitialized access

In the *Overview* panel, on your top-left, select `TrustInSoft/examples/01-Uninitialized-access/tis.config` in the drop-down menu.



TrustInSoft Analyzer automatically starts the verification of the test according to the given `tis.config`.

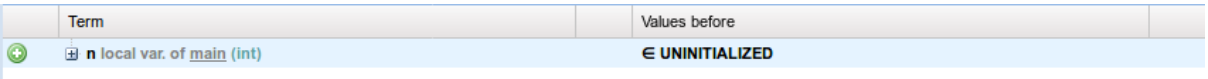
3.1.1 Understanding the alarm

The analysis finds an undefined behavior in the test.

Click on the link that appears in the interface to jump directly to the statement, in the source code, where the *Uninitialized access* undefined behavior is reported. The main panel in the interface shows the content of the `main` function. This alarm is shown as a special comment showing the assertion that failed to be proved:

```
/*@ assert Value: initialisation: \initialized(&n); */
```

Click on the variable `n` below the assert comment inside the loop. You can see in the bottom panel the values of this variable, before and after the statement. In the column *Values before*, you can see that the variable `n` has the value `UNINITIALIZED`. This means that the variable `n` has been declared at line 4, but it is never initialized before its usage in the loop.



3.1.2 Fixing the code

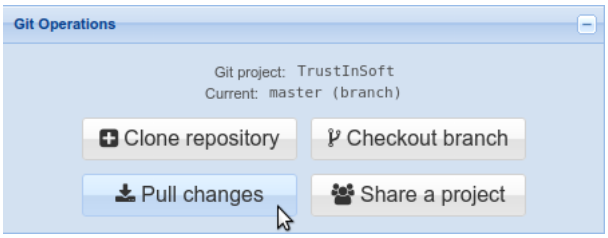
Edit `examples/01-Uninitialized-access/uninitialized.c` (for example, directly on GitHub) and fix the code by initializing `n`. For example, change the declaration at line 4

```
int i, n;
```

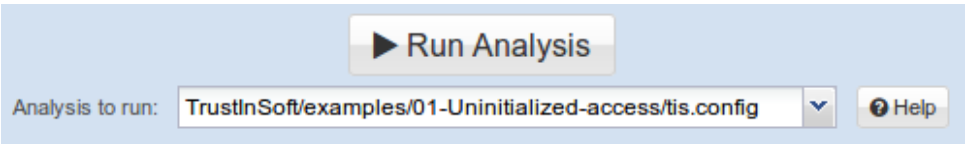
by initializing `n` to zero, as follows:

```
int i, n = 0;
```

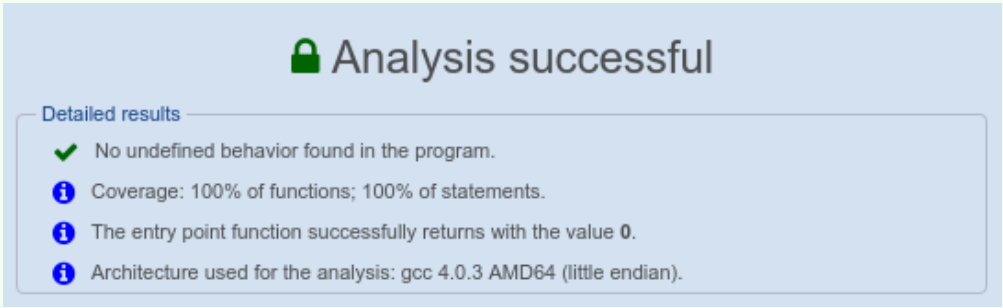
Commit and push your changes. Now, pull them into TrustInSoft Analyzer's workspace:



The verification starts automatically after the pull, selecting the `tis.config` for example 04. Select the appropriate `tis.config` file to restart the verification of the modified test:

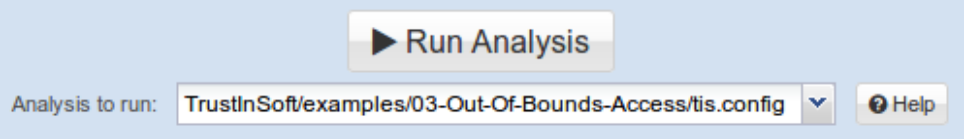


TrustInSoft Analyzer guarantees the absence of undefined behavior for the modified test:



3.2 Out-of-bounds access

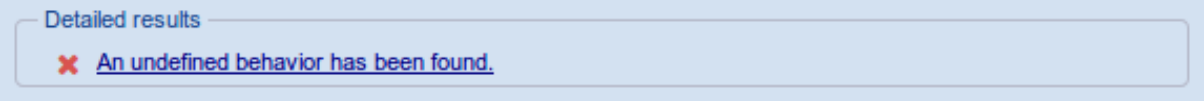
In the *Overview* panel, on your top-left, select `TrustInSoft/examples/03-Out-Of-Bounds-Access/tis.config` in the drop-down menu.



TrustInSoft Analyzer automatically starts the verification of the test according to the given `tis.config`.

3.2.1 Understanding the alarm

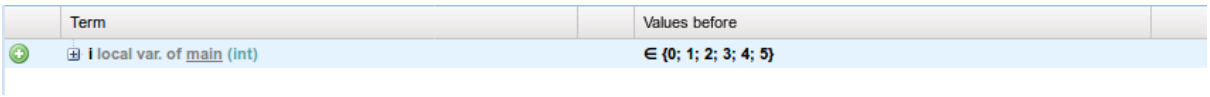
The analysis finds an undefined behavior in the test.



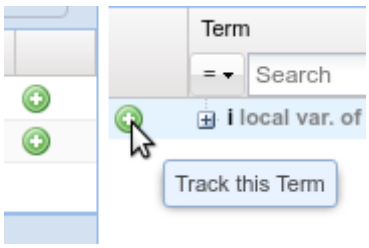
Click on the link that appears in the interface to jump directly to the statement, in the source code, where the *Out of bounds access* alarm is reported. The main panel in the interface shows the content of the `main` function. This alarm is shown as a special comment showing the assertion that failed to be proved:

```
/*@ assert Value: mem_access: \valid_read(array+i);*/
```

The code tries to access memory at position `array + i`, but that memory location is not always valid. Click on the variable `i` below the assert comment, on the right-hand side of the `+` operator, and look at the value displayed in the column *Values before*.



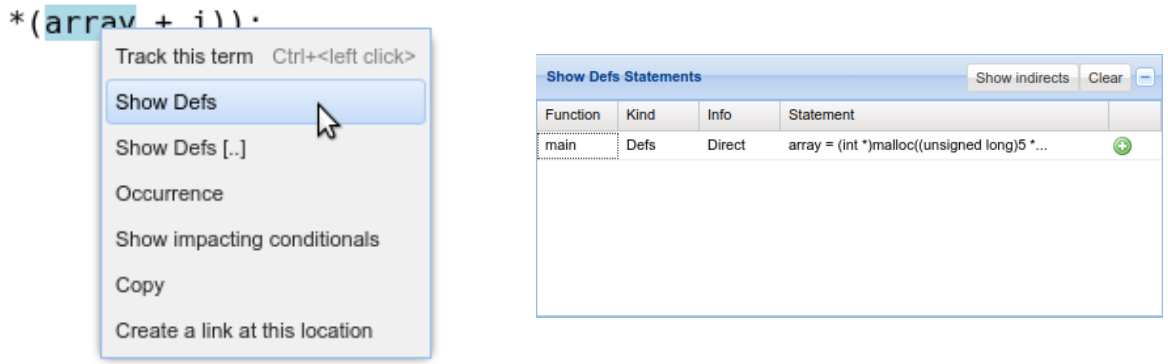
Here, `{0; 1; 2; 3; 4; 5}` is the set of all the values taken by `i` over all executions of the loop. Click on the  sign to track `i` (see below); tracking a term means that a new entry is permanently added in the bottom panel in order to show the value of that term at different points of the code. This can help understand the impact of one or multiple variables on the undefined behavior.



The second possible source of error is `array`. Click on the variable `array` inside the `printf` statement, and look at the value displayed in the column *Values before*. The displayed value `&__malloc_main_l10[0]` is the address of an internal variable, created by a call to `malloc`.

	Term		Values before
	array local var. of main (int *)		∈ {{ &__malloc_main_l10[0] }}
	i local var. of main (int)		∈ {0; 1; 2; 3; 4; 5}

You can find the corresponding `malloc` by right-clicking on this occurrence of `array`, and selecting *Show Defs*. The *Show Defs* actions collects all the operations that define `array` (see lower panel on the left).



Here, there is only one place where `array` is defined, highlighted in green:

```
array = (int *)malloc((unsigned long)5 * sizeof(int));
```

You can also follow the link shown inside the value of `array`:

	Term		Values before
	array local var. of main (int *)		∈ {{ &__malloc_main_l10[0] }}
	i local var. of main (int)		∈ {0; 1; 2; 3; 4; 5}

Clicking on `&__malloc_main_l10` displays the content of the internal variable along with its type, which is `int[5]` here (see the *Term* column):

	Term		Values before
	__malloc_main_l10 global var. (int [5])		[0] ∈ {13} [1] ∈ {7} [2] ∈ {42} [3] ∈ {0} [4] ∈ {1}
	i local var. of main (int)		∈ {0; 1; 2; 3; 4; 5}

The code allocates an array of size 5. Valid indices are thus from 0 to 4, but the upper bound of `i` is 5.

3.2.2 Fixing the code

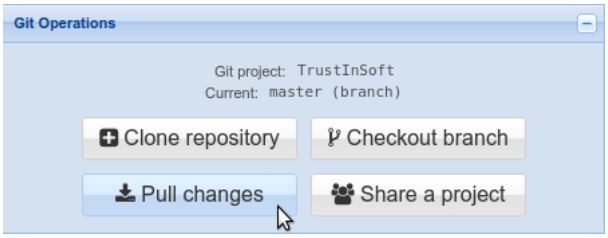
Edit `examples/03-Out-Of-Bounds-Access/array.c` (for example, directly on GitHub) and fix the code. For example, change the loop condition at line 19

```
    for (i = 0; i <= 5; i++) {
```

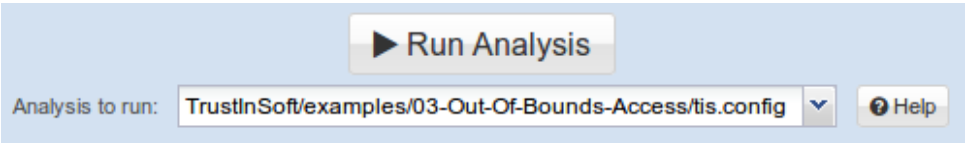
into a strict comparison instead:

```
    for (i = 0; i < 5; i++) {
```

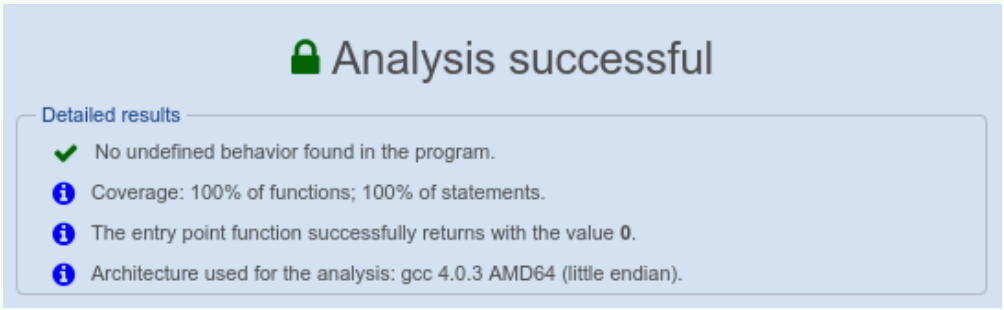
Commit and push your changes. Now, pull them into TrustInSoft Analyzer's workspace:



The verification starts automatically after the pull, selecting the `tis.config` for example 04. Select the appropriate `tis.config` file to restart the verification of the modified test:



TrustInSoft Analyzer guarantees the absence of undefined behavior for the modified test:



4 How to use TrustInSoft Analyzer on your own projects

To use TrustInSoft Analyzer on your projects, you need to provide, for each of them:

- A test driver.
- A `tis.config` file.

The test driver is a program that exercises your code, entirely or partly, and acts as the entry point for the analysis. The `tis.config` file describes how to configure TrustInSoft Analyzer to verify the test. In particular, it sets the parameters of the analysis, such as the compilation options or the target architecture.

4.1 `tis.config` format

A `tis.config` file corresponds to one analysis configuration, and should satisfy the following specification:

- It should have exactly the name `tis.config`.
- It should be part of your C project (committed to your GitHub project).
- It may specify the following analysis parameters:

  `name` : the name of the analysis.
  Default value: an uniquely generated name

  `files` : the list of files to analyze. Can also accept the value ”all” to select all files.
  Default value: `"all"`

  `main` : the entry point of your test driver.
  Default value: `"main"`

  `compilation_cmd` : the actual command used to invoke the compiler. From this information, TrustInSoft Analyzer extracts only some options, namely `-I`, `-D` and `-U`, and ignores the others.
  Default value: `null`

  `machdep` : the machine architecture used for the analysis.
  Only the following values are available: `gcc_x86_16`, `gcc_x86_32`, `gcc_x86_64`.
  Default value: `gcc_x86_64`

4.2 Example

The following example shows how you might organize your project to verify it with TrustInSoft Analyzer.

```
my_project
├── includes
│   └── ackermann.h
├── src
│   ├── ackermann.c
│   └── parser.c
└── tests
    ├── test1
    │   ├── main.c
    │   └── tis.config
    └── test2
        ├── main.c
        └── tis.config
```

Test 1. The first test is named `test-all`, and requires all the source files of the project, as well as a test driver `main.c`. The `tis.config` file explicitly lists the files needed for this test as well as the compilation command to compile the test. Note that the default value `all` for the `files` parameter cannot be used here because only the files relative to the `tis.config` file would be considered by the analysis.

```
{ "name":"test-all",
  "files":[ "../../src/ackermann.c", "../../src/parser.c", "main.c" ],
  "compilation_cmd":"gcc -Iincludes",
  "main":"main",
  "machdep":"gcc_x86_64" }
```

Test 2. The second test requires only the source file `parser.c` located in the `src` directory of the project, as well as a test driver `main.c`. The `tis.config` file explicitly lists the files needed for this test as well as the compilation command to compile the test. Note that the default value `all` for the `files` parameter cannot be used here because only the files relative to the `tis.config` file would be considered by the analysis.

```
{ "name":"test-parser",
  "files":[ "../../src/parser.c", "main.c" ],
  "compilation_cmd":"gcc",
  "main":"main",
  "machdep":"gcc_x86_64" }
```

Note that pathnames in the `files` parameter are relative to the `tis.config` file. The compilation command, however, is assumed to be executed from the root directory of the project. This applies in particular to the include directives in the `compilation_cmd` parameter.

Once you commit your configuration files and upload the modified project to GitHub, you can clone it from TrustInSoft Analyzer interface and start verifying your test suite.

That is all: you are ready to verify your tests with TrustInSoft Analyzer.

5 Conclusion

This tutorial is a quick introduction to TrustInSoft Analyzer *Newcomers* edition. It has access to your public GitHub projects and is easily configurable, with simple `tis.config` files. TrustInSoft Analyzer then verifies your tests by exhaustively checking for undefined behaviors.