TrustInSoft Analyzer *Beta*

Verification of the Caesar library

Version 1.0

Contents

1 Introduction . . . 3
2 Getting Started . . . 3
3 Verification of a Test Suite . . . 4
3.1 Parsing files . . . 4
3.2 Run an analysis . . . 4
3.3 Explore the first alarm . . . 4
3.4 Fixing the code and checking results . . . 6
3.5 Analysis review . . . 6
4 Exhaustive API verification . . . 7
4.1 Parse files . . . 7
4.2 Run an analysis . . . 7
4.3 Exploration mode . . . 8
4.4 Fixing the code . . . 8
5 Testing all buffers . . . 9
5.1 Code changes . . . 9
5.2 Analysis and review . . . 9
6 Architectural properties . . . 10
6.1 Setup . . . 10
6.2 Analysis . . . 10
6.3 Exploration mode . . . 10
7 Conclusion . . . 11

1 Introduction

This tutorial is a full tour guide to TrustInSoft Analyzer and its verification methodology.

The verification methodology is designed to give incremental mathematical guarantees from the first use. It starts by verifying a test suite to guarantee the absence of undefined behaviors. Then, it increases the verification coverage by generalizing a test suite, reaching exhaustive verification.

This tutorial also guides you through the GUI and introduces the recommended workflow.

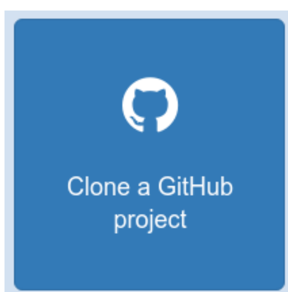
The tutorial requires the *Open Projects* edition of TrustInSoft Analyzer.

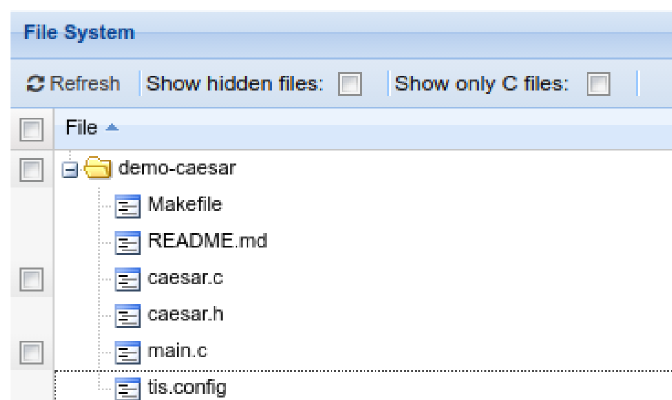
2 Getting Started

TrustInSoft Analyzer is available for public repositories on GitHub.

Prerequisites You must have a public email address in your GitHub account to be able to use TrustInSoft Analyzer. Fork the `demo-caesar` project so that it is visible from your profile. You may want to read the project's `README.md` first to learn more about the source code under verification.

From the TrustInSoft Analyzer interface, clone the project into your workspace:





To use TrustInSoft Analyzer on your projects, you first need to write a `tis.config` file that describes the files needed for the verification, compilation options.... The `demo-caesar` project already contains the following `tis.config`:

tis.config

```
{"files" : "all",
 "compilation_cmd" : "-I ."}
```

The configuration specifies, in the JSON format, the following options:

- Use all C files.
- Specify the necessary compilation options (here an include directive).
- Select a 32 bit little endian architecture (which is the default).

3 Verification of a Test Suite

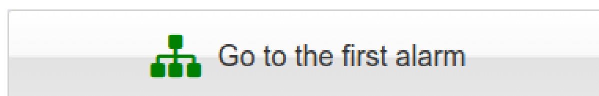
3.1 Parsing files

First, to select the first version of our code: click on *Checkout branch* on the left, and select the tag named `STEP_0`. Click on *Parse Files*, *Load configuration file & run parsing*, select *demo-caesar/tis.config*.

3.2 Run an analysis

Go to the **Value Analysis** mode and set the `Global slevel` parameter to 100. Run an analysis:

Then switch directly to the **Exploration** mode to inspect the first alarm:



3.3 Explore the first alarm

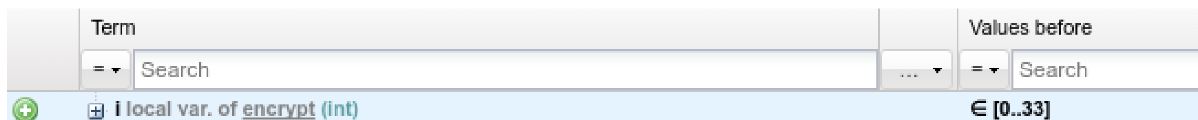
The **Exploration** mode is used to navigate the code and inspect the result of the analysis at every points of the program. It is the mode in which alarms are investigated.

TrustInSoft Analyzer provides distinct widget layouts named *templates*. When you first use the **Exploration** mode after an analysis, a popup appears asking you if you want to switch to the *Semantic template*. For this tutorial, you need to accept to ensure the expression evaluator is visible. If somehow you did not switch to the *Semantic template*, you can still do it from the *Workspace* menu.

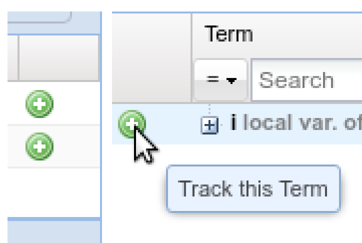
The first alarm is at this line:

```
/*@ assert Value: mem_access: \valid(buf+i); */
```

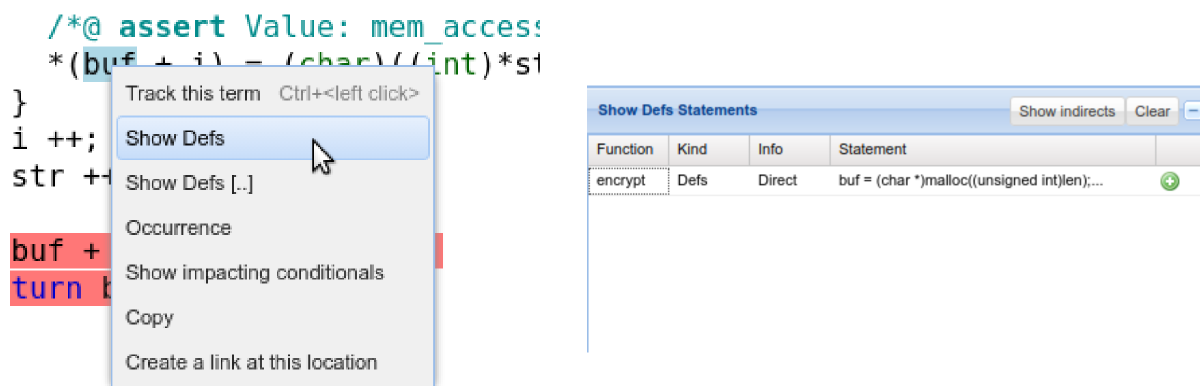
The code is trying to access an array at position `buf + i`, but that memory location is not always valid. When you click on `i`, the bottom tab named *Evaluate ACSL terms* displays the value of the variable at this point of the program.



Here, the value is the interval `[0..33]`, which are all the values assigned to `i` over all executions of the loop. You can click on the green "plus" sign to track `i` (see below); tracking a term means that a new entry is added in the *Evaluate ACSL terms* tab to permanently show the value of that term wherever you click in the code.



The second possible source of error is `buf`. If you right click on `buf`, you can see a *Show Defs* action, which collects all the operations that define `buf` (see lower panel on the left).



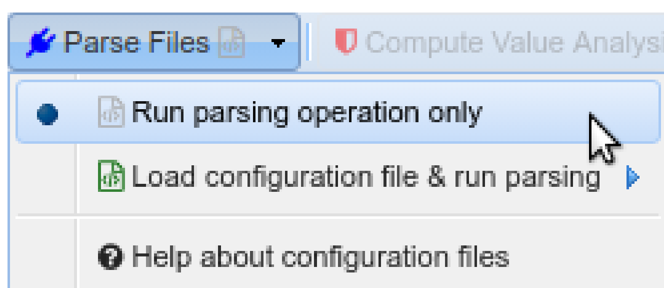
Here, there is only one place where `buf` is defined, highlighted in green.

```
buf = (char *)malloc((unsigned int)33);
```

The code allocates a buffer of size 33. The valid indices are thus from 0 to 32, but the upper bound of `i` is 33. This behavior is undefined (see the C standard ISO/IEC 9899:2011, §6.5.6.8) and was not detected when executing the actual program.

3.4 Fixing the code and checking results

There is a fix for this code in the next commit, tagged `STEP_1`. Go back to the **Pre-Processing** mode and checkout that version. In the *Parse Files* menu, select *Run parsing operations only*, to avoid resetting configuration variables.

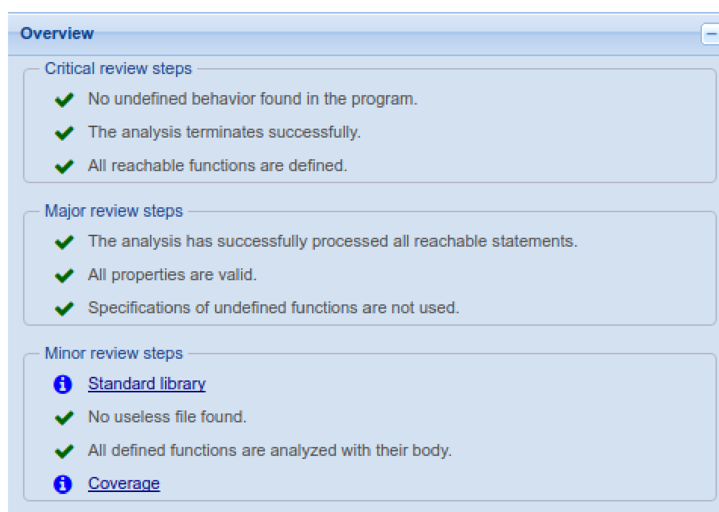


Then, go to **Value Analysis** and run an analysis; all checks pass:

TrustInSoft Analyzer guarantees the absence of undefined behavior for these two tests.

3.5 Analysis review

The **Analysis Review** mode shows a summary of the analysis and its perimeter. It shows the status of different kind of checks as well as coverage metrics.



4 Exhaustive API verification

Until now, the test only tried two different values for the *shift* parameter (the offset by which letters are displaced in the alphabet). However, this parameter could be equal to any of the 4 billions (e.g. 2^{32}) expressible integers.

In this new step of the tutorial, the perimeter of the analysis is extended so that the new verification exhaustively covers all possible values of this variable at once.

4.1 Parse files

For the second analysis, checkout `STEP_2`, and click *Parse Files*.

In the new version of `main.c`, there is an additional call to `gen_test` with an offset named `user_shift`. The latter is generalized through an auxiliary function named `tis_make_unknown`, provided by TrustInSoft Analyzer.

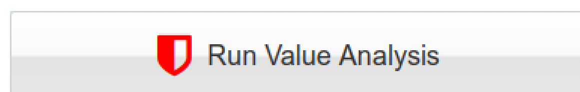
Generalized shift

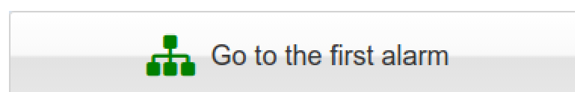
```
#include <tis_builtins.h>
...

int main(void)
{
    ...

    printf("\nTest 3: Shift with all possible input\n");
    int user_shift;
    tis_make_unknown((char *)&user_shift, sizeof(int));
    gen_test(user_shift, len, str);
}
```

`user_shift` is abstract and represents all possible signed integer values.

4.2 Run an analysis

Go to the **Value Analysis** mode, run a new analysis and go to the first alarm.





4.3 Exploration mode

32 bit signed integers can only express values from -2147483648 (`INT_MIN`) to 2147483647 (`INT_MAX`). The alarm indicates that the value of `-x` should be lower than or equal to `INT_MAX`:

```
/*@ assert Value: signed_overflow: - x <= 2147483647 ; */
```

But at the line the alarm is raised, the value of `x` is the interval of all values from `INT_MIN` to -1.

	Term		Values before
	= Search	...	= Search
	x formal var. of absolute_int (int)		∈ [-2147483648..-1]

The opposite value, `-x`, is consequently between 1 and `(- INT_MIN)`, but `INT_MIN` has not representable opposite value. In other words, TrustInSoft Analyzer determined that, when trying to take the opposite value of `-x`, it is possible that `x` equals to `INT_MIN`, in which case the operation overflows. That is an undefined behavior (see the C standard ISO/IEC 9899:2011, §6.3.1.4)

4.4 Fixing the code

The commit tagged `STEP_3` fixes the code. Check it out, parse all files and run an analysis. The analysis review shows that all checks are satisfied.

TrustInSoft Analyzer guarantees that there is no undefined behavior with the generalized test, which exhaustively verify all values for the shift parameter.

5 Testing all buffers

The test suite is extended even more to exhaustively cover all possible content of the input string.

5.1 Code changes

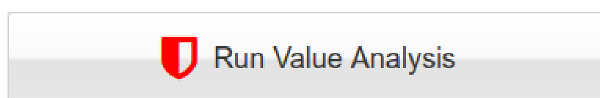
Commit `STEP_4` adds the following test:

Generalized input string

```
int main(void)
{
    ...

    printf("\nTest 4: All possible input\n");
    tis_make_unknown(str, len - 1);
    gen_test(user_shift, len, str);
}
```

In addition to the previous abstract shift parameter, the content of the input string is also made abstract by calling `tis_make_unknown` on variable `str`. This ensures that all bytes in `str` except the last one (which is always the null character) have arbitrary values. That multiplies the number of possible combinations, resulting in $256^{39} * 2^{32}$.

5.2 Analysis and review



The value analysis on this generalized test finds no alarm.

TrustInSoft Analyzer exhaustively verifies any combination of the input parameters and guarantees the absence of undefined behaviors.

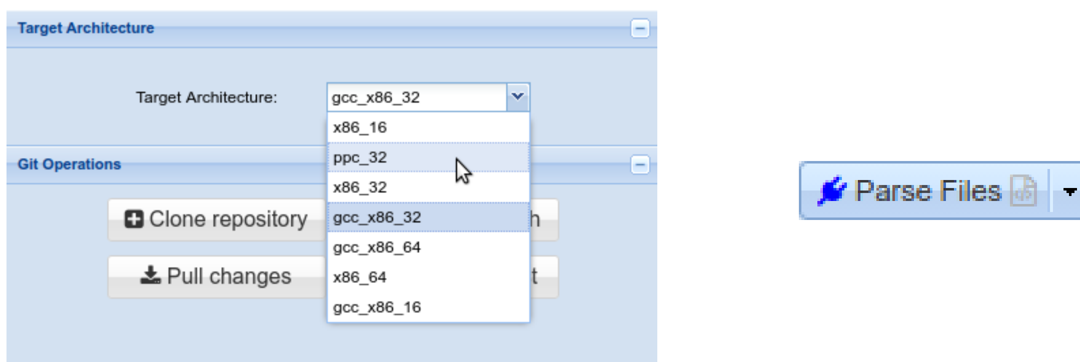
6 Architectural properties

TrustInSoft Analyzer faithfully models software architectural properties. This can help you find bugs specific to different platforms.

In this last section, a new analysis is performed on a big-endian architecture.

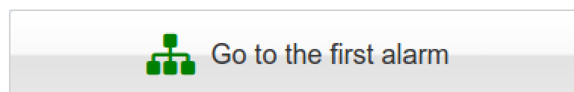
6.1 Setup

In **Pre-Processing** mode, select `ppc_32` in the *Target architecture* drop-down list. Then parse all files.



6.2 Analysis

Compute an analysis, and go to the first alarm.



6.3 Exploration mode

The following alarm is raised:

```
/*@ assert Value: shift: 0 ≤ (int)c < 32; */
```

The code performs a left shift with an offset `(int)c`, and in order to have a well-defined behavior, that value must be strictly lower than 32 (see the C standard ISO/IEC 9899:2011, §6.5.7.3). When you select `(int)c` and inspect its value, you can observe that it is 52, which is outside the valid range.

	Term		Values before
	= Search	...	= Search
	(int)c (int)		{52}

7 Conclusion

This tutorial is a brief tour of TrustInSoft Analyzer, demonstrating how the interface is organized around a workflow made of different modes: **Pre-Processing**, **Value Analysis**, **Exploration** and **Analysis Review**.

This tutorial also provides a recommended verification methodology, starting first with providing guarantees for an existing test suite, then generalizing inputs to eventually perform an exhaustive verification.