# Aardwolf: A Modular and Extensible Tool for Software Fault Localization

Petr Nevyhoštěný, Petr Máj    Department of Theoretical Computer Science, FIT CTU in Prague

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

## Motivation

The growth of software complexity raises the challenges for maintaining its quality and keeping the presence of faults at minimum. The determination of the nature and location of a fault is **tedious** and **time-consuming** activity. Software fault localization is a research field helping developers in this effort with automated techniques. The thesis introduces a new such tool with focus on **extensibility** and **user experience**, overcoming weaknesses of existing applications.

## Problem Definition

The task is to provide user with a helpful report of suspicious program elements to reduce the amount of time and effort committed to fault localization. We employ dynamic approach which augments the program with tracing calls to gather test-suite runtime data. Static analysis is performed as well. Generated data are used to estimate the suspiciousness of certain program elements (e.g., statements).

## Shortcomings of Existing Tools

- Mostly research prototypes with poor integration into preferred workflow and toolchain (e.g., editor or programming language) of the user.
- Often make impractical assumptions. For example, they do not provide a rationale for the suspiciousness calculation or a context.
- Not publicly available or abandoned. To our best knowledge, only one available, actively developed "competitor".

## Solution

We present a highly modular design that enables adding support for preferred programming language, user interface or state-of-the-art localization technique, merely by implementing respective component. Internal APIs encourage plugins developers to provide the user with rationale, source code context and additional details. Data structures are rich-enough to support majority of techniques published in the literature and commonly-used programming languages. Initially implemented techniques employ probabilistic reasoning or inference of program invariants.

## Architecture

The tool is composed of several components. *Frontends* process the source program. *Core* gathers the data and offers them via convenient API to *plugins*. The results are then delivered to the user through *interface* of choice.

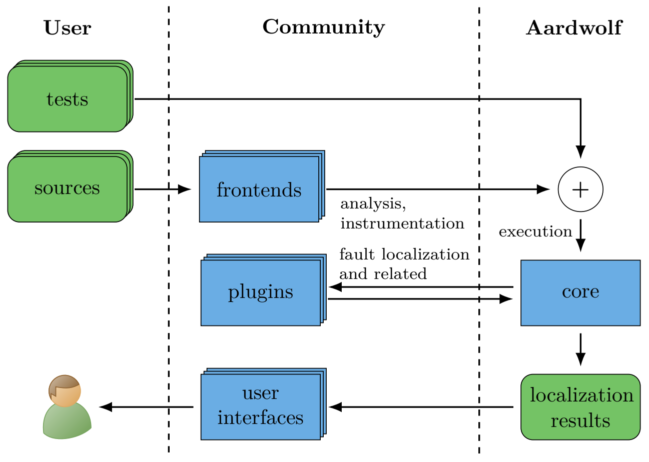


**Figure:** Architecture. Stacked blue nodes indicate extensibility.

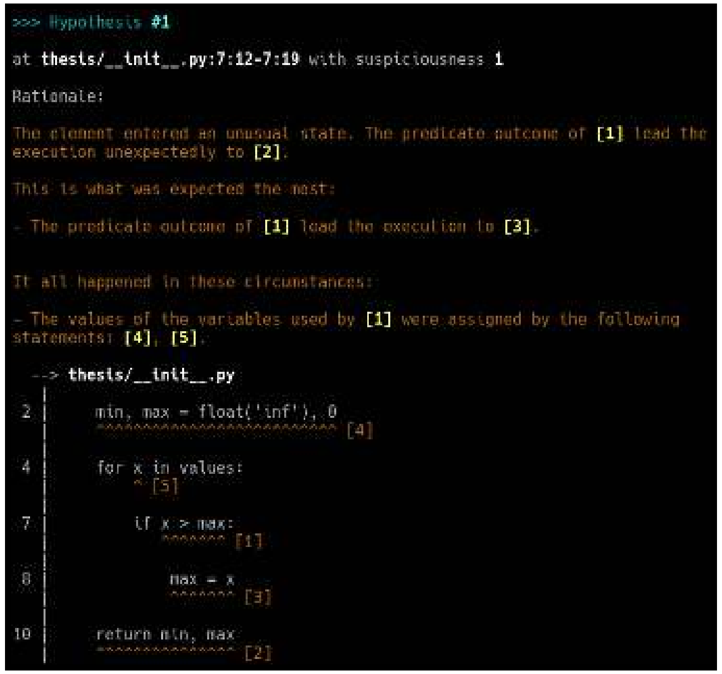
## Contributions

- Design of a new SFL tool that is extensible, language-agnostic and user-friendly. These goals are a **significant improvement** compared to existing solutions.
- Implementation of proposed framework, initially supporting three **diverse** localization techniques and two **diverse** programming languages (C and Python).
- The tool is openly available at

  `https://github.com/aardwolf-sfl/aardwolf`

## Results

Best techniques detected up to 55% of bugs in their top 10 list on a standard dataset – a result comparable with the literature. The tool was successfully integrated with two bigger real-world projects (namely `matplotlib` and `LibTIFF`), and the process required only around 40 lines of configuration for each and no change in existing workflow. A sample of the localization output is depicted in figure 1.



```
>>> Hypothesis #1

at thesis/__init__.py:7:12-7:19 with suspiciousness 1

Rationale:

The element entered an unusual state. The predicate outcome of [1] lead the
execution unexpectedly to [2].

This is what was expected the most:

- The predicate outcome of [1] lead the execution to [3].


It all happened in these circumstances:

- The values of the variables used by [1] were assigned by the following
statements: [4], [5].

  --> thesis/__init__.py
    |
  2 |     min, max = float('inf'), 0
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ [4]
    |
  4 |     for x in values:
    |         ^ [5]
    |
  7 |         if x > max:
    |            ^^^^^^^ [1]
    |
  8 |             max = x
    |             ^^^^^^^ [3]
    |
 10 |     return min, max
    |     ^^^^^^^^^^^^^^^ [2]
```

**Figure:** A sample of terminal output for **`get_range`** function that fails on negative-only arrays. The issue is wrong initialization of `max` on line 2. Aardwolf reports that a usual update on line 8 did not happen during the failing execution, and reveals data dependencies of the problematic conditional. From this rich information, the user should be able to deduce the root cause.

## References

[1] Petr Nevyhoštěný. A Modular and Extensible Tool for Software Fault Localization. Master's thesis, Faculty of Information Technology, CTU in Prague, 2020.