A Framework for Analysing a Subset of C

Alfredo Cruz & Anders Fischer-Nielsen

December 5th 2019

Contents

Todo list

1 Introduction

Analyzing a program can help detect errors in the program. Elaborate

Our analysis works on a a subset of the *C* language. Our analyzer supports a subset of the *TIP* language described by Schwartzbach in [1] as C. In the following, we give an overview of the supported language.

1.1 Expressions

The basic expressions all denote integer values:

$$
\begin{aligned}
E &\rightarrow \mathit{intconst} \\
&\rightarrow \mathit{id} \\
&\rightarrow E\texttt{+}E \mid E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \mid E\texttt{>}E \mid E\texttt{==}E \\
&\rightarrow \texttt{(}E\texttt{)}
\end{aligned}
$$

1.2 Statements

The simple statements are similar to C:

$$
\begin{aligned}
S &\rightarrow \mathit{id} = E \\
&\rightarrow \ \texttt{printf}\ E; \\
&\rightarrow S\ S \\
&\rightarrow \texttt{if(}E\texttt{)}\{S\} \\
&\rightarrow \texttt{if(}E\texttt{)}\{S\}\ \texttt{else}\ \{S\} \\
&\rightarrow \ \texttt{while}\ \texttt{(}E\texttt{)}\{S\} \\
&\rightarrow \ \texttt{int}\ \mathit{id}_1, \ldots, \mathit{id}_n
\end{aligned}
$$

1.3 An Example Program

A program implemented in the supported subset of C can be seen in Fig. 1.

```
int main()
{
    int x;
    x = 1;
    int y;
    y = 0;
    int z;
    z = 0 + 1;
    if (y == 0) {
        while(1) {
            x = 0;
        }
    }
    else {
        x = 1;
    }
    return x + y;
}
```

Figure 1: An example program in *TIP*-inspired subset of C.

2 Finding Double Unlock Bugs

A requirement of this project was that the user of the analyzer should be able to supply their own analyses. We decided to enable this by having the user define a lattice and transfer functions. These structures are then used by the analyzer in order to gain knowledge about input programs.

Due to the requirement of a user passing their own analyses to the framework and the fact that these analyses need to be loaded at runtime, Python was used for the implementation of our analyzer. Given the dynamic nature of the Python programming language, evaluating Python scripts dynamically at runtime is relatively easy and supported by the language. Using a compiled language, e.g. OCaml, turned out to be too unstable in practice given the static compiled nature of the language. Loading in arbitrary user code is to a great extent not supported in compiled languages and this dynamic loading of arbitrary modules is where a dynamic language shines.

The user supplies our framework with a list of file names containing Python class definitions. The framework then dynamically loads these analyses which the user is expected to formulate as individual Python scripts. These scripts must implement and expose an `Analysis` class containing the components our framework implementation expects to be present in order to analyze.

A disadvantage in using a dynamic language is the lack of type checks. Python supports type annotations, but these are generally completely ignored by the interpreter. The type annotations read more like comments in the source code than actual safety guarantees. This means that we have no way of enforcing that the analyses supplied by the user adhere to an interface or a class definition, and we can only provide examples for the user to follow when implementing their analysis. This is unfortunate, since the framework will simply throw an `Error` when attempting to evaluate the users' analyses. Using a compiled language would allow us to specify a signature for the analyses of the user to implement, guaranteeing that the analysis could be run without errors — provided that the users' own implementations does not raise errors.

The framework is invoked with a list of filenames of analysis files. These files are implemented as very basic Python modules, which are then imported at runtime. An invocation of the two analyses `AnalysisA` and `AnalysisB` would require the user to provide the list of analyses as to the framework as `"AnalysisA:AnalysisB"`. This list of names is split on commas, and the framework then attempts to import these modules. User analyses are expected to be located in the `analyzers` subfolder of the framework implementation in a corresponding subfolder. Due to the way Python expects modules to be structured, a file named `__init__.py` needs to be co-located with the analysis implementation file. This is merely a practicality in the way the Python module system works. An illustration of the module structure can be seen in Fig. 2.

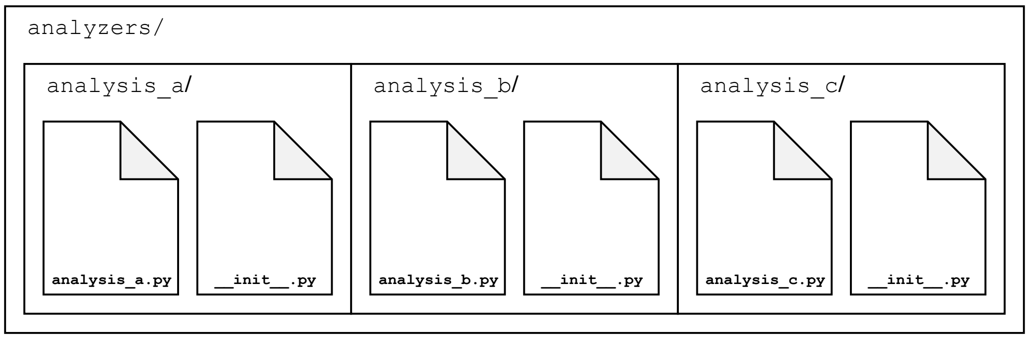

Figure 2: The folder structure for three analyses provided by the user.

An analysis provided by the user must expose an instance of an `Analysis` class. This class requires a list of transfer functions, variables and expressions to be provided for instantiation. The analyzer extracts this instance and applies the transfer functions for each provided `Analysis` to the input file. The result of applying an analysis results in a transformed input program, which is fed forward to the next provided `Analysis`. When no more analyses are present, the transformed input program is given as output to the user. An illustration of this can be seen in Fig. 3.

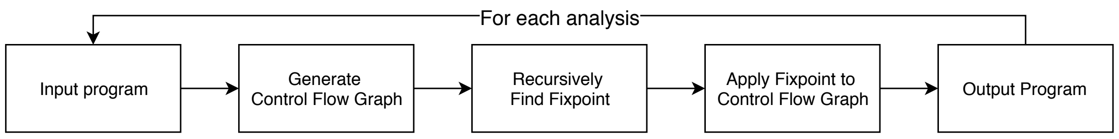
For each analysis
Input program
Generate Control Flow Graph
Recursively Find Fixpoint
Apply Fixpoint to Control Flow Graph
Output Program

Figure 3: The program flow for analyzing a program with multiple provided analyses.

`Analysis` implementation

Fixpoint finding

3 Results

3.1 Detectable Error Types

3.2 Undetectable Error Types

4 Future Work

The dynamic nature of the implementation should be made more type-safe when users provide analyses. Validating whether a given input analysis matches the expected structure of the framework should be implemented in order to reduce errors on the user's end. This could be accomplished either by dynamically verifying that members are present on the input analyses or attempting to implement the framwork in a strongly typed language, though this might not prove to be possible due to the requirement of dynamic loading.

5 Conclusion

References

[1] Anders Møller and Michael I. Schwartzbach. Lecture notes on static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

6 Appendix