《A System and Language for Building System-Specific, Static Analyses》Review

Paper Info

A System and Language for Building System-Specific, Static Analyses Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler

Major Contribution

In this paper, the authors propose a DSL to describe the system safe properties and design an engine of checker to verify the safe properties. The DSL, Metal is flexible and easy-to use, and the checker system, xgcc can perform a context-sensitive, interprocedural analysis. In order to make the analysis efficient, some ideas similar to dynamic programming and block caching are used in the system design.

Because of the unsoundness of the analysis, the authors propose some methods to suppress the false positives, such as a series of error ranking techniques, false path pruning and so on.

In my personal view, the shining points of this work include the design of the flexible DSL, caching techiques, and error ranking methods. Some ideas of false positive suppression are also worthy of further research.

Main Work

In this paper, the authors propose a framework for bug detection by defining checkers in a DSL. The checker engine process the property descriptions and report bugs. In order to suppress false positives, the authors design some mechanisms including error ranking, false path pruning and false positive memorization.

DSL: Metal

Metal is a DSL designed for describing the safe property of the project which needs to checked. The grammar of Metal is simple and pretty flexible. The following figure illustrates the instance of lock checker. For example, the second rule defines transitions in different code from the state of start. If the trylock() performs well, the value of l will be set to locked.

```
state decl { lock_t } l;

start:
    {trylock(l) != 0} ==> true=l.locked, false=l.stop
  | {trylock(l) == 0} ==> true=l.stop, false=l.locked
  | {lock(l);} ==> l.locked
  | {unlock(l);} ==>
   { err("%s is not locked", mc_identifier (l)); }
  ;

l.locked:
    {lock(l);} || {trylock(l)} ==>
       { err("dbl. lock of %s", mc_identifier (l)); }
  | {unlock(l);} ==> l.unlocked
  | $end_of_path$ ==>
       { err("path ends with lock held"); }
  ;
```

Intraprocedural and interprocedural analysis

The main framework of analysis is similar to the traditional dataflow analysis. In order to improve the efficiency, the following tricks are used in the system design.

- Caching techniques: In the intraprocedural analysis, caching is at the block level.
- Dynamic programming summaries: In the interprocedural analysis, additional summaries at the function level and at the suffix level are built based on block summaries.
- Top-down algorithm traverses the supergraph depth-first starting at all function roots. When a function call is encountered, the engine restart the traversal at the entry to the callee and return to the return-site node.

Ranking and other false positive suppression

Because of the unsoundness of the system, some methods are proposed in order to suppress false positives and list true positives over false positives in the bug report.

Error ranking is a simple technique, but from the error inspector's point of view, it makes an exhilarating difference. The false positives in the bug report is not distributed in a random pattern. However, in most of time, the false positives have a pattern which is named as a local explosion. Some criterias can be used to rank the error in a generic way, including the distance between the statement that rank contains the error and the statement where the extension stated checking the property that led to the error, number of conditionals, degree of indirection and local versus interprocedural. Some error ranking methods, including statistical ranking, are also efficient when handling the false positive suppression.

Besides error ranking technique, false path pruning and targeted suppression of false positives are also two efficient approaches to handle the false positive suppression. The xgcc's algorithm uses basic value tracking combined with a congruence closure algorithm to prune infeasible paths. This can skip some infeasible paths, and make the analysis more accurate. Meanwhile, if some false positives occur many times, they will be remembered by the system. The system match error reports across versions by comparing file name, function name, variable names involved in the analysis, and the actual error itself as stated by the checker. The history of analysis can be in a full use to decrease the number of false positives.

Future Work

In the last paper I read, the authors proposed a project ESP, which is most similar to the system in this paper. However, the analysis of ESP is sound, while the system in this paper is not. Hence, more efforts should be make to suppress false postives of the system, not only through some techniques of error ranking, but also by taking advantages of some other techniques.

In order to analyze large programs, it is necessary to create compact path summaries that only retain portions of the AST that are relevant to the analysis, which can assure the efficiency of the system when analyzing the large programs, such as the Linux kernel and so on.

The loop analysis in this work is not accurate enough, and the state after the loop is often set to unknown, which can not provide any useful information. Some techniques aimed at loop analysis, including loop invarints extraction, may be applied to this work to make the results more precise.

Caching technique is the biggest shinning point in this work. The objects in caching are mainly functions or code blocks. Some loops can also be analyzed in this pattern, if they occur many times in the project.