Using Historical Information to Improve Bug Finding Techniques

Chadd C. Williams
Department of Computer Science
University of Maryland
chadd@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
hollings@cs.umd.edu

Abstract

Tools used to identify bugs in source code often return large numbers of false positive warnings to the user. These false positive warnings can frustrate the user and require a good deal of effort to identify. Various attempts have been made to automatically identify false positive warnings. We take the position that historical data mined from the source code revision history is useful in refining the output of a bug detector by relating code flagged by the tool to code changed in the past.

1 Introduction

Tools used to identify bugs in source code often return large numbers of false positive warnings to the user. True positive warnings are often buried among a large number of distracting false positives. By making the true positives hard to find, a high false positive rate can frustrate users and discourage them from using an otherwise helpful tool.

Prior research has focused on inspecting the code surrounding the warning producing code with the assumption that a tool may produce a large number of false positive warnings very close together in the code [2].

More recent work has added user driven feedback to refine the ranking of warnings [1]. As the user inspects a warning and classifies it as either a bug or false positive, the remaining warnings are re-ranked. The intuition is that warnings that are part of some grouping are likely to all be either bugs or false positives. This approach has the advantage of giving the user a preliminary ranking of the warnings and then refining that ranking with as close to true fact as one can get: the opinion of the user.

2 Repository Mining as a Solution

We believe we can use data mined from the source code repository to help determine the likelihood of a warning being a true bug or a false positive by relating code flagged by warnings to code that was changed in the past. With the source code repository we have a record of each source code change. We can determine when a piece of code is added and, more importantly, when code is changed. The code changes may be used to highlight bug fixes through the life of the project.

Examining the code changes and the state of the code before and after the change may allow us to match previous code changes to warnings produced by a bug finding tool. Warnings could be matched to code changes in a number of ways. The functions invoked, the location in the code (module/API/function) or the control or data flow may be used to link the flagged code to the code from the repository. Warnings that flag code similar to code snippets that have been changed in the past may be more likely to be true positives.

In [3] we show how historical data can be used to rank warnings produced by a static analysis tool with a particularly high false positive rate. We mined the source code repository to determine which functions in a software project had a particular type of bug fix applied to their invocation. We produced a ranking of the warnings where warnings involving functions flagged with a bug fix were pushed to the top of the list. Our approach produced a ranking with a higher density of likely bugs near the top as compared to a more naïve ranking scheme.

We investigate function usage patterns mined from the software repository in [4]. Here we are trying to identify from the repository how functions should be invoked in the source code with respect to each other. We believe that discrepancies between how we expect functions to be called and how they are invoked in the current version of the software could be used to highlight code that may be incorrect. These discrepancies may indicate confusion on the part of the programmer. Warnings produced for these snippets of code may be more likely to be true bugs.

A ranking based on the past history is similar to the idea of ranking based on user feedback. However, when using past history the feedback is automatically generated (and could be augmented by interactive user feedback). The initial ranking the user is given will have the benefit of past code changes.

3 References

[1] Kremeneck, T., Ashcraft, K., Yang, J., Engler, D., Correlation Exploitation in Error Ranking, In *Proceedings of Twelfth ACM SIGSOFT Symposium on Foundations of Software Engineering* (SIGSOFT'04) Newport Beach, CA, USA, Nov. 2004.

[2] Kremeneck, T., Engler, D., Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations, In *Proceedings of 10th Annual International Static Analysis* Symposium, (SAS '03) San Diego, CA, USA, June 2003.

[3] Williams, C. C., Hollingsworth, J. K., Bug Driven Bug Finders, In *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[4] Williams, C. C., Hollingsworth, J. K., Recovering System Specific Rules from Software Repositories, In *Proceedings of International Workshop on Mining Software Repositories* (MSR '05), St. Louis, MO, USA, May 2005.