Review for *Tailoring Programs for Static Analysis via Program Transformation*

Hongchen Cao
Shanghaitech University
Shanghai, China
caohch1@shanghaitech.edu.cn

I. Paper Summary

Static analysis is an essential part of the modern software development process and is used in areas such as bug detection, program optimization, and system understanding. In practice, however, analysis tooling is approximate, which leads to false positives (i.e., mistakenly reported as a bug when it isn't). Existing work has shown that false positives are a headache and irritation for users of static analysis tools, who often modify their code to avoid false positives and undesirable warnings. For developers of static analysis tools, it often takes a lot of time to modify the analysis algorithm to avoid false positives. To address this problem, this paper introduces a new technique for automatic, generic, and ad hoc code modification that tailor to suppress false positives (i.e., spurious analysis errors).

The paper builds on the insight that an analysis user always has a significant ability to influence analysis behavior and output: modifying their programs. So they propose a rule-based approach where simple, declarative templates describe general syntactic changes for known problematic code patterns. Thus, by rewriting code fragments, undesired warnings and false positives are removed. This technique has the following three benefits:

- The analysis user can avoid all false positives of the same type in the program by writing templates only once, without having to modify each code instance;
- The rewriting of code fragments does not affect the source code written by the analysis user (i.e., the user does not have to change the code to avoid undesired warnings);
- The analysis developer does not have to spend a lot of time checking and modifying the source code of the analysis tool to solve the user's problem.

The authors evaluated their technique on a dataset containing 15 projects, three languages, and five static analyzers. The authors first manually analyzed several issues on GitHub related to false positives in three static analyzers, and thus manually write six templates for avoiding the six corresponding false positives. The experimental results show that they successfully suppress 111 false positives with an accuracy rate of 46.8%. The authors also experimented with the runtime of their technique. The results show that the added time is almost negligible compared to the runtime of the static analyzer itself (e.g., 2.5s and 0.3s, 17m29s and 1.6s, 2m47s and 1.0s).

Further, the authors also discuss whether their tool is likely to introduce new bugs, how to write templates efficiently, and how to apply the tool to other languages, which is very inspiring for researchers and developers in this field.

II. Strengths and Weaknesses

A. Strengths

+ Applying program transformation into suppressing false positives in static analysis is a creative and effective approach.

+ Writing templates for different false positive patterns, as the basis of the technique in the paper, is very easy to understand, comprehend and implement.

+ The approach in this paper is well-compatible with existing static analyzers and does not require modification of the source code of the static analyzer itself, which is advantageous for its application in practice.

+ The evaluation includes not only accuracy but also efficiency (i.e., runtime), and the results show that the time delay caused by their approach is not worth mentioning compared to the static analyzer and can effectively suppress false positives.

+ The authors list nine rewrite patterns and explain them in detail, which is very helpful for understanding and learning how to accomplish the only manual part(i.e., writing templates) of their method.

+ The discussion on pattern development and the limits of applicability and usability is illuminating and may facilitate the exploration of how to suppress false positives in static analysis.

B. Weaknesses

- Since the author uses declarative templates to describe general syntactic changes for code patterns, which makes part of the code instance not captured by the corresponding template, resulting in a loss of accuracy.

- Their Sec. IV-2-6 mentions that developing patterns (i.e., writing templates) may require several iterations to refine, the author should give detailed examples to explain it and give general iteration goals (i.e., what kind of templates or patterns are qualified)

- The author doesn't methion works related to false positives in static analysis such as [1]–[3] in their Sec. V Related Work.

- The authors do not compare with the SOTA method, or even any method in any related work in this field (e.g., [4]–[6]), and no reasonable explanation is given for this.

III. Overall Evaluation

A. Soundness

The approach of rewriting syntax declaratively is based on comby[1], a tool for searching and changing code structure by lightweight templates, and the authors clearly explain the rationale for using this tool and provide a detailed description. The author also provides nine examples (See their Sec. IV-2-5) of how to write templates and explains them one by one. Experiments on the dataset containing 15 projects, three languages, and five static analyzers, also experimentally verify the accuracy and efficiency of this approach.

However, the lack of comparison with existing related work is a fatal flaw for which the authors should provide an explanation or additional experiments.

B. Significance

False positives, or false alarms, are an unavoidable problem in modern static analyzers. Static parser users, often need to repeatedly use similar code modification techniques to suppress the same type of false positives, which can potentially introduce new bugs and is tedious and annoying. For static analyzer developers, fixing false positives often requires revising the analysis algorithm or modifying the implementation of the algorithm, which can take a lot of time for design, programming, and testing.

The technique proposed in this work is effective in suppressing false positives while introducing only a minimal runtime burden. Addressing false positives through program transformation can also inspire the community to actively consider and tackle this problem from a variety of other perspectives.

C. Novelty

In this paper, program transformation is used for the first time to suppress false positives in static analysis. The fact that analyzer users can modify their programs to influence analysis behavior allows authors to modify the code based on pattern and template to suppress false positives. Such ideas are very innovative and effective, but the most central part of the approach - retrieving the pattern and writing templates - relies on existing technologies (i.e., comby) while the latter relies on manual implementation by the user. This work would be even more innovative if the whole process could be further automated (e.g., more automatic generation of templates with user code changes).

D. Verifiability

The authors describe the datasets in detail used for the experiments, and for each experiment, the special preprocessing methods of the data are also given. Author-created artifacts relevant to this paper have been placed on a publically accessible archival repository. This is sufficient to allow all researchers to reproduce and validate the results of the paper.

[1]https://comby.dev/

IV. Discussion

A. Connection

As defined in CS224-Lecture1, static analysis analyzes a program P to reason about its behaviors and determines whether it satisfies some properties before running P. Unfortunately, by Rice's Theorem, there is no such approach to determine whether P satisfies such non-trivial properties (i.e., giving the exact answer Yes or No). Thus, any analysis tooling is approximate. False positives are one of the significant consequences of approximation, which causes a lot of trouble and annoyance to analyzer users.

The purpose of this paper is to suppress false positives by program transformation. This gives researchers in the field of program analysis a different way of thinking about the problem. In addition to designing more accurate analysis algorithms, some of the issues arising from approximations can be solved by designing tools independent of the analysis algorithms. I will consider the problems in program analysis through this perspective in my subsequent course learning as well.

B. Brainstorming

1. The template is written manually by the user, and this part of the work could probably be implemented automatically or semi-automatically. An algorithm could be designed to record the code modifications made by the user to suppress false positives and generate a template, and later use the template when encountering similar patterns and improve the template based on user feedback (i.e., code modifications).

2. Future work can incorporate richer static information. In particular,templates currently express purely syntactic properties. Thus, we can draw on information including type, context, etc. to inform code modifications.

3. Graphs such as DFG, CFG, and CG can represent program semantics and other information very well. Using such graphs as templates may catch more code instances of false positives that are hard to distinguish by purely syntactic properties.

References

[1] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of javascript web applications in the wild," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 61–70.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, M. Das and D. Grossman, Eds. ACM, 2007, pp. 1–8.

[3] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electron. Notes Theor. Comput. Sci.*, vol. 217, pp. 5–21, 2008.

[4] T. Muske and U. P. Khedker, "Efficient elimination of false positives using static analysis," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 270–280.

[5] B. Chimdyalwar, P. Darke, A. Chavda, S. Vaghani, and A. Chauhan, "Eliminating static analysis false positives using loop abstraction and bounded model checking," in *FM 2015: Formal Methods*, N. Bjørner and F. de Boer, Eds. Springer International Publishing, 2015, pp. 573–576.

[6] T. T. Nguyen, P. Maleehuan, T. Aoki, T. Tomita, and I. Yamada, "Reducing false positives of static analysis for sei cert c coding standard," in *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, 2019, pp. 41–48.