False Positives Over Time: A Problem in Deploying Static Analysis Tools

Andy Chou, Coverity Inc., andy@coverity.com

All source code analyzers generate *false positives*, or issues which are reported but are not really defects. False positives accumulate over time because developers fix real defects but tend to leave false positives in the source code. Several methods are available to mitigate this problem, some of which are shown in the following table (especially important advantages or disadvantages are labeled with (*)):

Technique	Advantages	Drawback(s)
Add annotations to the source code that indicate the location of false positives, which the tool can then use to suppress messages.	• Persistent across code changes and renamed files (*) • Seamlessly propagate between different code branches via version control systems • Also works for real bugs that users don't want to fix	• Some users are unwilling to add annotations to source code, even in comments (*) • Remain in the code even if analysis is changed to not find the false positives • Adding annotations to third party code is usually undesirable
Allow users to override analysis decisions that lead to false positives.	• Eliminates entire classes of false positives with same root cause (*) • Flexible: can be done using source annotations or tool configuration	• Difficult for users to understand (*) • Changing analysis decisions can have unexpected side-effects, such as missing other bugs
Change the source code to get the tool to "shut up."	• No changes to tool	• Not always obvious how to change the code to make the false positive go away (*) • Might not be possible to change the code in an acceptable way
Stop using the tool, or turn off specific types of checks causing the false positives.	• Simple • Minimize cost of dealing with false positive-prone analyses (*)	• Lost opportunity to discover real bugs
Rank errors using some criteria, or otherwise use statistical information to identify likely bugs vs. likely false positives.	• (mostly) Automatic • Adapts to application-specific coding conventions (*)	• Does not deal with all false positives • Larger development organizations want to distribute bugs to be inspected; each developer gets a small number of bugs to inspect, making ranking less useful (*) • Users don't like deciding when to stop; they fear missing bugs while simultaneously loathing false positives. • Users usually want "rank by severity" not "rank by false positive rate"
Annotate the output of the tool to mark false positives, then use this information in future runs to avoid re-reporting the same issues.	• No changes to code • Works for almost any type of static analysis (*)	• Need heuristics to determine when false positives are the "same" in the presence of code changes (*) • False positives may re-appear depending on the stability of the merging heuristic (*)

These are not the only ways of attacking this problem. Very little work has been done on classifying and evaluating these techniques, yet they are critical to the adoption of static analysis in industry. Some observations from Coverity customers include:

- Users are unwilling to add annotations unless the tool has already shown to be an efficient bug-finder.
- Users are usually unwilling to change source code to eliminate false positive warnings.
- Ranking errors by "likely to be a bug" can help pull real bugs to the forefront, but users have a hard time deciding where the cutoff should be. Users also dislike having no cutoff at all, because they will often query for bugs in a specific location of interest, where only a handful of results are found and ranking is not useful. Ranking by estimated severity of bug is more often useful.
- If users annotate tool output so the tool gains "memory," the heuristic used to determine that previous results are the "same" as new results must be robust. Users very much dislike false positives coming back.