Static Analyzers in Software Engineering

Dr. Paul E. Black
National Institute of Standards and Technology

Static analyzers can report possible problems in code and help reinforce the good practices of developers. This article contrasts the strengths of static analyzers with testing and discusses the current state-of-the-art.

A static analyzer is a program written to analyze other programs for flaws. Such analyzers typically check source code, but there are analyzers for byte code and binaries, too. Analyzers for requirements or design are possible, but most are focused on code and binaries. At a minimum, analyzers report the location and name of a possible problem. Some analyzers have far more capabilities. They may describe the problem and possible attacks or failure modes in-depth. They may detail the data or control flow leading from the source of values involved to the data or control flow leading from the source of values involved to the statement where the failure may have manifested or the value is passed to another component. They may also suggest mitigations.

A vulnerability is any property of system requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a failure. As described in [1]: “A vulnerability is the result of one or more weaknesses in requirements, design, implementation, or operation.” Because configuration, installation, operation, and other system components determine whether a certain code construct may lead to failure, I speak of weaknesses in the code, not vulnerabilities. To reiterate, static analyzers report weaknesses in software.

What Are Their Strengths and Limitations?

Every static analyzer has a built-in set of weaknesses to look for in code. Most have some means of adding custom rules. In contrast, testing requires test cases or input data. Testing also requires artifacts that are complete enough to be executable, possibly with supporting drivers, stubs, or simulated components.

Static analysis may be performed on modules or unfinished code, although the more complete the code, the more thorough and accurate the analysis can be.

Analyzers are limited by the sophistication of the reasoning in them. For instance, some static source code analyzers do not handle function pointers and few can deal with embedded assembler code. Even if the models of the programming language, compiler, hardware, and other pieces used in execution are perfect, analyzers have the same fundamental limitation as any other logical system. They cannot solve the halting problem or undecidable problems. In practice, this need not be a serious limitation. Important code “should be so clearly correct that it confuses neither human nor tools” [2]. Although running tests is straightforward, this same challenge of analysis arises in developing tests to exercise a particular property or module.

New tests must be developed when new attacks or failure modes are discovered. Static analyzers have some advantage in this case. The weakness check need only be added and validated once, then the analyzer is rerun on all code. Test generators can give a similar advantage.

Most importantly, static analyzers have the potential to find rare occurrences or hidden back doors. Since they consider the code independently of any particular execution, they can enumerate all possible interactions.

“Most importantly, static analyzers have the potential to find rare occurrences or hidden back doors. Since they consider the code independently of any particular execution, they can enumerate all possible interactions.”

Statistical correlation is an example of heuristic analysis. For instance, an open is usually followed by a close or
resources are typically locked within a critical section. Such rules may be derived automatically through machine learning of existing code. But heuristic analysis is susceptible to false alarms (false positives) or missing actual weaknesses (false negatives).

Analysis may be a combination of sound reasoning and heuristic techniques. Complete analysis of the termination conditions of every loop or possible states of all combinations of variables may be impractical, so most analyzers use algorithms that are not purely sound or purely heuristic. In addition, most analyzers are a system of analytic engines; examples are data flow, loop termination, value propagation, control flow, or property recognition.

Work from the June 2008 Static Analysis Tool Exposition [3, 4] shows that current analyzers vary widely. An analyzer may produce few false alarms for some weaknesses, but many false alarms for other weaknesses. Likewise, the rate of missed weaknesses differs greatly. Analyzers also only cover a subset of documented weaknesses [5]. Thus, the most comprehensive static analysis would result from a carefully used combination of analyzers. Other factors, such as cost and analyst support, must go into selecting the most appropriate static analyzer(s) for each situation. The Software Assurance Metrics and Tool Evaluation (SAMATE) Reference Dataset [6] has thousands of sample programs that may help such evaluation.

Static analyzers should be a key part of every software development process.

References