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 *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.

> *"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."*

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. The number of interactions tends to increase exponentially, defying comprehensive static analysis and test execution alike. Static analysis can focus on the interaction without testing's need to re-establish initial conditions or artificially constrain the system to produce the desired interaction. Worse, black-box testing cannot realistically be expected to discover, let's say, a backdoor accessible when the user ID is "JoshuaCaleb" since there are a nearly infinite number of arbitrary strings to test.

Testing and static analysis complement each other. Testing has the advantage of possibly revealing completely unexpected failures. Embedded systems can be tested, even when it is utterly impractical to analyze any software that may be tucked away in a component.

Static Analysis' Place in Software Engineering

Static analysis is no panacea. Complex and subtle vulnerabilities can always defeat the reasoning in a static analyzer. The utter lack of an important requirement, such as auditing or encryption, cannot reasonably be deduced from only the examination of post-production artifacts. Software with no resiliency or self-monitoring is open to errors in installation or operation, but static analysis can be one of the last lines of defense against vulnerabilities.

Static analysis can be understood in a continuum from sound to heuristic. A sound analysis is 100 percent correct in its judgments. If it reports a weakness, a weakness definitely exists. If it reports that a certain construct is okay, then one is assured that a weakness is not present. In some cases, a sound analysis may not have enough information to render a good/bad judgment.

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

1. "Source Code Security Analysis Tool Functional Specification Version 1.0." National Institute of Standards and Technology (NIST), Special Publication 500-268. May 2007 <http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf>.
2. Holzmann, Gerard J. "Conquering Complexity." Computer 40 (12): 111-113, Dec. 2007.
3. NIST. "Static Analysis Tool Exposition." 7 July 2008.
4. NIST. ACM SIGPLAN. Proc. of the Static Analysis Workshop. Tucson, AZ. 12 June 2008 <http://samate.nist.gov/index.php/SAW>.
5. MITRE. "Common Weakness Enumeration." 25 Nov. 2008 <http://cwe.mitre.org>
6. NIST. "NIST SAMATE Reference Dataset." Jan. 2006.

About the Author

Paul E. Black, Ph.D., has nearly 20 years of industrial experience in software for integrated circuit design and verification, assuring software quality and managing business data processing. He now works in the Software Quality Group, Information Technology Laboratory of the NIST and edits the online Dictionary of Algorithms and Data Structures. He has a doctorate in computer science from Brigham Young University and has published on topics including software testing, configuration control, networks and queuing analysis, formal methods, software verification, quantum computing, and computer forensics. Black is a member of the Association for Computing Machinery, IEEE, and the IEEE Computer Society.

NIST
100 Bureau DR Stop 8970
Gaithersburg, MD 20899-8970
Phone: (301) 975-4794
Fax: (301) 975-6097
E-mail: paul.black@nist.gov