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 good practices of developers. We contrast the strengths of static analyzers with testing and indicate the current state of the art.

What is a Static Analyzer?

A static analyzer is a program written to analyze other programs. 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 focus on code and binaries. Analyzers vary greatly in sophistication, purpose, and output. They may produce call graphs, compute complexity metrics, check for style conformance, track test coverage, produce an annotated listing, find data flow program slices, prove that a program has specific properties (or that it lacks them), or report bugs. This article concentrates on analyzers with the last function: those that look for bugs.

At a minimum, this kind of analyzer must 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 be manifest or where 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. “A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation.” [1] What’s the difference between a weakness and a vulnerability?

Because configuration, installation, operation, and other system components determine whether a certain code construct may lead to failure, we speak of weaknesses in the code, not vulnerabilities. As an example, nagios [2] reads a configuration file using huge string buffers, far larger than any regular user would need. However, a malicious user of the host system with sufficient privileges could write a configuration file to cause a buffer overflow. This is a weakness, but it is only considered a vulnerability if such a malicious user is deemed feasible and the buffer overwrite can lead to a security violation.

Restating the above, the class of static analyzers we discuss 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." [3] Although running tests is straight forward, this same analysis challenge 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 similar advantage.

Most importantly static analyzers have the potential to find rare occurrences or hidden backdoors. Since they consider the code independently of any particular execution, they can enumerate all possible interactions. The number of interactions tends to increase exponential, defying comprehensive static analysis and test execution alike. Static analysis can focus on the interaction, without testing's need to reestablish initial conditions or artificially constrain the system to produce the desired interaction. Worse, black box testing cannot realistically be expected to discover, 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 reasonable be deduced only from 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 lines of defense against vulnerabilities.

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

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 negative).

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”, for example data flow, loop termination, value propagation, control flow, or property recognition.

Recent work coming from the Static Analysis Tool Exposition [4] shows that current analyzers vary widely. An analyzer may produce few false alarms for some weaknesses, but relatively 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 SAMATE Reference Dataset (SRD) [6] has thousand of sample programs that may help such evaluation.

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

1. “Source Code Security Analysis Tool Functional Specification Version 1.0”, National Institute of Standards and Technology, Special Publication 500-268, May 2007. Available at http://samate.nist.gov/docs/source_code_security_analysis_spec_SP500-268.pdf (Accessed 21 April 2008)
2. “Nagios”, http://www.nagios.org/ (Accessed 2 June 2008)
3. Gerard J. Holzmann, “Conquering Complexity”, Computer, 40(12):111-113, December 2007.
4. “Static Analysis Tool Exposition (SATE) 2008”, Static Analysis Workshop, Tucson, AZ, June 2008. Available at http://samate.nist.gov/index.php/SATE and http://samate.nist.gov/index.php/SAW (Accessed 21 April 2008)
5. “Common Weakness Enumeration”, MITRE, http://cwe.mitre.org/ (Accessed 21 April 2008)
6. “SAMATE Reference Dataset”, National Institute of Standards and Technology, http://samate.nist.gov/SRD/ (Accessed 21 April 2008)