Tool Selection and Validation

Although rule checking can be performed manually, with increasing program size and complexity, it rapidly becomes infeasible. For this reason, the use of static analysis tools is recommended.

When choosing a compiler (which should be understood to include the linker), a C++-compliant compiler should be used whenever possible. A conforming implementation will produce at least one diagnostic message if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as *undefined* or *implementation-defined*. It is also likely that any analyzers you may use assume a C++-compliant compiler.

When choosing a source code analysis tool, it is clearly desirable that the tool be able to enforce as many of the guidelines as possible. Not all recommendations are enforceable; some are strictly meant to be informative.

Although CERT recommends the use of an ISO/IEC TS 17961–conforming analyzer, the Software Engineering Institute, as a federally funded research and development center (FFRDC), is not in a position to endorse any particular vendor or tool. Vendors are encouraged to develop conforming analyzers, and users of this coding standard are free to evaluate and select whichever analyzers best suit their purposes.

False Positives and False Negatives

In general, determining conformance to coding rules and recommendations is computationally undecidable. The precision of static analysis has practical limitations. For example, the halting theorem of computer science states that for every analysis tool that indicates whether a program terminates, there exists a program that the tool will not analyze correctly. Consequently, any property dependent on control flow—such as halting—may be indeterminate for some programs. A consequence of undecidability is that it may be impossible for any tool to determine statically whether a given guideline is satisfied in specific circumstances. The widespread presence of such code may also lead to unexpected results from an analysis tool.

Regardless of how checking is performed, the analysis may generate the following:

- **False positives**: The tool reports a flaw when one does not exist. False positives may occur because the code is too complex for the tool to perform a complete analysis. The use of features such as function pointers and libraries may make false positives more likely.
- **False negatives**: Failure to report a real flaw in the code is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security. Most tools err on the side of caution and consequently generate false positives. However, in some cases, it may be deemed better to report some high-risk flaws and miss others than to overwhelm the user with false positives.

To the greatest extent feasible, an analyzer should minimize both false positives and false negatives with respect to enforceable guidelines.

Compilers and source code analysis tools are *trusted* processes, meaning that a degree of reliance is placed on the output of the tools. Accordingly, developers must ensure that this trust is not misplaced. Ideally, trust should be achieved by the tool supplier running appropriate validation tests such as the **Secure Coding Validation Suite**.

False Positives

Although many guidelines list common exceptions, it is difficult if not impossible to develop a complete list of exceptions for each guideline. Consequently, it is important that source code complies with the *intent* of each guideline and that tools, to the greatest extent possible, minimize false positives that do not violate the intent of the guideline. The degree to which tools minimize false-positive diagnostics is a quality-of-implementation issue.