The Use and Limitations of Static-Analysis Tools to Improve Software Quality

Dr. Paul Anderson
GrammaTech, Inc.

Advanced static-analysis tools have been found to be effective at finding defects that jeopardize system safety and security. This article describes how these work and outlines their limitations. They are best used in combination with traditional dynamic testing techniques, and can even reduce the cost to create and manage test cases for stringent run-time coverage.

Static analysis has commonly been known as a technique for finding violations of superficial stylistic programming rules, and for alerting programmers to typing discrepancies in type-unsafe languages. The latest static-analysis tools go far beyond this, and are capable of finding serious errors in programs such as null-pointer de-references, buffer overruns, race conditions, resource leaks, and other errors. They can do so without requiring additional input from the users, and without requiring changes to development processes or practices. Actionable results are produced quickly with a low level of false positives. These static-analysis tools are not a silver bullet, however, because they can never prove that a program is completely free of flaws. The following is a description of how static-analysis tools work, followed by a discussion of how they can be used to complement traditional testing.

How Static Analysis Finds Flaws

The first thing a static analysis tool must do is identify the code to be analyzed. The source files that must be compiled to create a program may be scattered across many directories, and may be mixed in with other source code that is not used for that program. Static analysis tools operate much like compilers so they must be able to identify exactly which source files contribute and should ignore those that do not. The scripts or build system that builds the executable obviously know which files to use, so the best static analysis tools can extract this information by reading those scripts directly or by observing the build system in action. This way the tool gets to see not only the source files but also which compiler is being used and any command-line flags that were passed in. The parser that the static analysis tool uses must interpret the source code in the same way that the real compiler does. It does this by modeling how the real compile works as closely as possible. The command-line flags are an essential input to that.

As the build system progresses, each invocation of the compiler is used to create a whole program model of the program. This model consists of a set of abstract representations of the source, and is similar to what a compiler might generate as an intermediate representation. It includes the control-flow graph, the call graph, and information about symbols such as variables and type names.

> *"In order to understand the limitations of the techniques that these tools use, it is important to understand the metrics used to assess their performance."*

Once the model has been created, the analysis performs a symbolic execution on it. This can be thought of as a simulation of a real execution. Whereas a real execution would use concrete values in variables, the symbolic execution uses abstract values instead. This execution explores paths and, as it proceeds, if any anomalies are observed, they are reported as warnings. This approach is based on abstract interpretation [1] and model checking [2].

The analysis is *path-sensitive*, which means that it can compute properties of individual paths through the program. This is important because it means that when a warning is reported, the tool can tell the user the path along which execution must proceed in order for the flaw to be manifest. Tools also usually indicate the points along that path where relevant transformations occur and conditions on the data values that must hold. These help users understand the result and how to correct the problem should it be confirmed.

Once a set of warnings have been issued, these tools offer features to help the user manage the results, including allowing the user to manually label individual warnings. Warnings that correspond to real flaws can be labeled as true positives. Warnings that are false alarms can be labeled as false positives. Warnings that are technically true positives but which are benign can be labeled as *don't care*. Most tools offer features that allow the user to suppress reporting of such warnings in subsequent analyses.

Limitations of Static Analysis

In order to understand the limitations of the techniques that these tools use, it is important to understand the metrics used to assess their performance. The first metric, *recall*, is a measure of the ability of the tool to find real problems. Recall is measured as the number of flaws found divided by all flaws present. The second metric is *precision*, which measures the ability of the tool to exclude false positives. It is the ratio of true positives to all warnings reported. The third metric is *performance*. Although not formally defined, this is a measure of the computing resources needed to generate the results.

These three metrics usually operate in opposition to each other. It is easy to create a tool that has perfect precision and excellent performance – one that

reports no lines contain flaws will satisfy because it reports no false positives. Similarly, it is easy to create a tool with perfect recall and excellent performance – one that reports that all lines have errors will answer because it reports no false negatives. Clearly, however, neither tool is of any use whatsoever.

Finally, it is at least theoretically possible to write an analyzer that would have excellent precision and excellent recall given enough time and access to enough processing power. Whether such a tool would be as useless as the previous two example tools is debatable and would depend on just how much time it would take. What is clear is that no such tools currently exist and to create them would be very difficult.

As a result, all tools occupy a middle ground around a sweet spot that developers find most useful. Developers expect analyses to complete in time roughly proportional to the size of their code base and within hours rather than days. Tools that take longer simply do not get used because they take too long. Low precision means more false positives, which has an insidious effect on users. As precision goes down, even true positive warnings are more likely to be erroneously judged as false positives because the users lose trust in the tool.

For most classes of flaws, precision less than 80 percent is unacceptable. For more serious flaws, however, precision as low as five percent may be acceptable if the code is to be deployed in very risky environments. It is difficult to quantify acceptable values for recall as it is impossible to measure accurately in practice, but clearly users would not bother using these tools at all if they did not find serious flaws that escape detection by other means.

Each of these constraints introduces its own set of limitations, however they are all interrelated. The reasons that lead to low recall are explained in more detail in the following sections.

Path Limitations

As mentioned earlier, these analyses are path sensitive. This improves both recall and precision and is probably the key aspect of these products that makes them most useful. A full exploration of all paths through the program would be very expensive. If there are n branch points in a procedure, and there are no loops in that procedure, then the number of intraprocedural paths through that procedure can be as many as 2^n. In practice, this is fewer because some branches are correlated, but the asymptotic behavior remains. If procedure calls and returns are taken into account, the number of paths is *doubly* exponential, and if loops are taken into account then the number of paths is unbounded. Clearly it is not possible for a tool to explore all of these paths. The tools restrict their exploration in two ways. First, loops are handled by exploring a small fixed number of iterations: often, the first time around the loop is singled out as special, and all other iterations are considered en masse and represented by an approximation. Second, not all paths are explored. It is typical for an analysis to place an upper bound on the number of paths explored in a particular procedure or on the amount of time available, and a selection of those remaining paths are explored.

"If asynchronous paths can occur (such as those caused by interrupts or exceptions) or if the program uses concurrency, then the number of possible paths to consider increases further. Many tools simply ignore the possibilities."

If asynchronous paths can occur (such as those caused by interrupts or exceptions) or if the program uses concurrency, then the number of possible paths to consider increases further. Many tools simply ignore these possibilities. Finally, most tools also ignore recursive function calls, and function calls that are made through function pointers (or make very coarse approximations) as considering these also contributes to poor performance and poor precision.

Abstract Domain

As previously mentioned, these tools work by exploring paths and looking for anomalies in the abstract state of the program. The appeal of the symbolic execution is that each abstract state represents potentially many possible concrete states. For example, given an 8-bit variable x, there are 2^8 possible concrete values: 0, 1, …, 255. The symbolic execution, however, might represent the value as two abstract states: $x=0$, and $x>0$. So where a concrete execution has 256 states to explore, the symbolic execution has only two.

As such, the expressivity of this abstract domain is an important factor that determines the effectiveness of the analysis. Again, there is a trade-off here: better precision and recall can be achieved by more sophisticated abstract domains, but more resources will then be required to complete an analysis. Values in the abstract domain are equations that represent constraints on values, i.e., $x=0$, or $y>10$. As the analysis progresses, a constraint solver is used to combine and simplify these equations. A key characteristic of these abstract domains is that there is a special value, usually named *bottom*, which indicates that the analysis knows no useful information about the actual value. *Bottom* is the abstract value that corresponds to all possible concrete values. Reaching bottom is impossible to avoid for any non-trivial abstraction in general as this would require solving the halting problem. Once bottom is reached, the analysis has a choice of treating it as a potentially dangerous value, which would increase recall, or as a probably safe value, which would increase precision. Most tools opt for the latter as the former also has the effect of decreasing precision enormously.

If there are program constructs that step outside the bounds of what can be expressed in the abstract domain, this causes the analysis to lose track of variables and their relationships. For example, an abstract domain that allows the expression of affine relationships between no more than two variables admits expressions such as $x=2y$. However, something such as $x=y+z$ is out of bounds because it involves three variables and the analysis would be forced to conclude $x=bottom$ instead.

The consequence of this is the abstract domain that a tool uses determines a great deal about the kind of flaws that it is capable of detecting. For example, if the tool uses an abstract domain of affine relations between two variables, then it may fail to find flaws that depend on three variables.

```
if (a || b || c)
      x = 0;
```

Coverage	a	b	c
Statement	T	-	-
Decision	T	-	-
	F	F	F
MCDC	T	-	-
	F	T	-
	F	F	T
	F	F	F

Table 1: *Test Cases Needed for Statement, Decision, and MCDC Coverage*

Similarly, most tools choose a domain that allows them to reason about the values of integers and addresses but not floating-point values, so they will fail to find flaws in floating-point arithmetic (such as divide by zero).

Missing Source Code

If the source code to a part of a program is not available, as is almost always the case because of operating system and third-party libraries, or if the code is written in a language not recognized by the analysis tool, then the analysis must make some assumptions about how that missing code operates. Take, for example, a call to a function in a third-party library that takes a single pointer-typed parameter and returns an integer. In the absence of any other information, most analyses will assume that the function does nothing and returns an unknown value. This clearly is not realistic, but it is not practical to do better in general. The function may de-reference its pointer parameter, it may read or write any global variable that is in scope, it may return an integer from a particular range, or it may even abort execution. If the analysis knew this, it would have better precision and recall but it is forced to make the simple assumption unless told otherwise.

There are two approaches around this. First, if source is not available but object code is, then the analysis could be extended into the object code. This is a highly attractive solution but no products are available yet. The technological basis for such a tool exists, however [3], and it is expected that products capable of analyzing object code as well as C/C++ will appear.

A second approach to the problem is to specify stubs, or *models*, that summarize key aspects of the missing source code. The popular analysis tools provide models for commonly used libraries such as the C library. These models only have to approximate the behavior of the code. Users can, of course, write these themselves for their own libraries but it can be a tricky and time-consuming effort.

Out of Scope

There are, of course, entire classes of flaws that static analysis is unlikely ever to be able to detect. Static analysis excels at finding places where the fundamental rules of the language are being violated such as buffer overruns, or where commonly used libraries are being used incorrectly, or where there are inconsistencies in the code that indicate misunderstanding. If the code does the wrong thing for some other reason, but does not then terminate abnormally, then static analysis is unlikely to be able to help because it is unable to divine the intent of the author. For example, if a function is intended to sort in ascending order, but perfectly sorts in descending order instead, then static analysis will not help much. This kind of functionality testing is what traditional dynamic testing is good for.

Figure 1: *A Redundant Condition Warning*

```
                c:\CodeSonar\ex2.c
                Enter foo
 5                void foo (int rest, int length)
 6              {
 7                   if (rest <=1)
 8                        buf[pos-1] = '>';
 9                   else if (rest == 2)
10                        buf[pos++] = '>';
11                   else if (length > rest)
Always True:
rest > 1
12                        if (--rest > 1) {   /* Redundant Condition (ID: 1) */
13                             if (rest >= 2)
14                                  rest --;
15                        }
```

Figure 2: *A Second Redundant Condition Warning*

```
Never True:
($temp2 & 16) != 0
 8    if (!flags & MASK)     /*Redundant Condition */
 9    {
10        error("Cannot sign packet");
11        return;
12    }
```

Static Analysis and Testing

Static analysis should never be seriously considered as a replacement for traditional dynamic testing activities. Rather, it should be thought of as a way of amplifying the software assurance effort. The cheapest bug to find is the one that gets found earliest, and as static analysis can be used very early in the development cycle, its use can reduce the cost of development and liberate resources for use elsewhere. This is the traditional view of how static analysis can reduce testing costs. However, there is a second way in which the use of static analysis can reduce the cost of testing: it makes it easier to achieve full coverage.

One measure of the effectiveness of a test suite is how well it exercises or *covers* the code being tested. There are many different kinds of coverage. Statement coverage is the most common, but for riskier code more stringent forms are often required. Decision coverage is a superset of statement coverage, and requires that all branches in the control flow of the program are taken. In DO-178B, a development standard for flight software [4], the riskiest code is required to be tested with 100 percent modified condition/decision coverage (MCDC). This means that a test suite must be chosen such that all subexpressions in all conditionals are evaluated to both true and false. Table 1 illustrates how many different test cases are needed for each to achieve coverage. For the code sample on the left, the values required of the boolean variables a, b, and c to achieve each form of coverage is shown on the right.

Achieving full coverage, even for statement coverage, can be very time consuming. The engineer creating the test case must figure out what inputs must be given to drive the program to each statement. What can make it very frustrating is if it is fundamentally impossible to do so, but this may not be

apparent simply by looking at the code. If the program contains unreachable code, then statement coverage is impossible. If it contains redundant conditions (those that are either always true or always false), then MCDC is impossible. Developers can spend hours trying to refine a test case before it is evident that their efforts are pointless.

If the unreachable code or redundant conditions can be brought to the attention of the tester early, then they do not need to waste time in a futile attempt to achieve the impossible. This is what static analysis can do easily and efficiently. Figure 1 shows an example of a report from CodeSonar[1] illustrating a redundant condition in a sample of code taken from an open-source application. The variable rest, an unaliased integer, must be at least three by line 12. The decrement on that line means it is at least two, so the condition will always be true. The following line is also redundant and shown in a different report.

In this example, all the components of the code relevant to the redundancy are in close proximity so it is likely that a reviewer would have spotted this during a manual review. It would not have been so easy to spot if the code were more complex. If the code had spanned several pages, or if relevant parts had been embedded in function calls or macro invocations, then it would have been difficult to spot. Static analysis is not sensitive to superficial aspects of the code such as its layout, so it would not have been confused.

These kinds of redundancies correlate well with genuine flaws as well; for example, consider the example in Figure 2. This was distilled from a genuine flaw found in a widely used open-source program, and is a redundant condition warning where the tool has deduced that the true branch of the conditional will never be taken. The reason why it concluded so is shown to the left. The first operand to the bitwise AND (the `&` symbol) is either zero or one as this is the range of the negation operator (the `!` symbol). This is what is represented by `$temp2`. The constant `MASK` has the value 16. The result of the AND expressions `1&16` and `0&16` are both zero, so the conditional expression is guaranteed to be zero.

The programmer who wrote this code probably misunderstood the precedence of the operators in the conditional expression and assumed that the innermost operator had higher precedence. If so, then a correction would be to place parentheses around the inner expression. This is a potentially dangerous flaw as it means that the error condition would not be detected, which could result in unpredictable behavior.

When to Use Static Analysis Tools

The best time to use advanced static analysis tools is early in the development cycle. In Holzmann's 10 rules for safety-critical development [5], the most far-reaching rule states that these tools should be used throughout the development process. As well as reducing the cost of development by finding flaws earlier and reducing testing effort, early adoption exerts a force on programmers to write code that is more amenable to analysis, thereby increasing the probability that the tool will find errors. Care should be taken, however, to avoid a risk compensation phenomenon, where programmers use less care because they assume that the static analysis tool will find their mistakes.

If adopted late in the development cycle, static analysis may issue a large number of warnings. The best value is gained if these are all dealt with, either by fixing the code, marking them as false positives, or labeling them as *don't care* if they are believed to be benign. However, if scheduling time to sift through these is not feasible, then an alternative strategy is to operate in a *differential* mode, where programmers are only told about new warnings. This way they are alerted to flaws in code that they are working with while it remains fresh in their minds.

Conclusion

Advanced static analysis tools offer much to help improve the quality of software. The best tools are easy to integrate into the development cycle, and can yield high-quality results quickly without requiring additional engineering effort. They can be used not just for finding flaws, but also to guide testing activities. They use sophisticated symbolic execution techniques for which engineering trade-offs have been made so that they can generate useful results in a reasonable time. As such, they inevitably have both false positives and false negatives, and so should never be considered a replacement for traditional testing techniques.◆

References

1. Cousot, P., and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." ACM Symposium on Principles of Programming Languages. Los Angeles, CA., 1977.
2. Clarke, E.M., O. Grumberg, and D.A. Peled. Model Checking. MIT Press: Cambridge, MA: 1999.
3. Balakrishnan, G., R. Gruian, T. Reps, and T. Teitelbaum. "CodeSurfer/x86 – A Platform for Analyzing x86 Executables." International Conference on Compiler Construction. 2005.
4. RTCA/DO-178B. "Software Considerations in Airborne Systems and Equipment Certification." 1992.
5. Holzmann, G.J. "The Power of 10: Rules for Developing Safety-Critical Code." IEEE Computer 2006.

Note

1. GrammaTech's static analysis tool.

About the Author

Paul Anderson, Ph.D., is vice president of engineering at GrammaTech, a spin-off of Cornell University that specializes in static analysis, where he manages GrammaTech's engineering team and is the architect of the company's static analysis tools. He has worked in the software industry for 16 years, with most of his experience focused on developing static analysis, automated testing, and program transformation tools. A significant portion of Anderson's work has involved applying program analysis to improve security. His research on static analysis tools and techniques has been reported in numerous articles, journal publications, book chapters, and international conferences. Anderson has a B.Sc. from Kings College, University of London, and his doctorate in computer science from City University, London.

GrammaTech, Inc.
317 N Aurora ST
Ithaca, NY 14850
Phone: (607) 273-7340
Fax: (607) 273-8752
E-mail: paul@grammatech.com