Predicting Where Faults Can Hide from Testing

JEFFREY VOAS, NASA Langley Research Center
LARRY MORELL, Hampton University
KEITH MILLER, College of William and Mary

Testing seeks to reveal software faults by executing a program and comparing the output expected to the output produced. Exhaustive testing is the only testing scheme that can (in some sense) guarantee correctness. All other testing schemes are based on the assumption that successful execution on some inputs implies (but does not guarantee) successful execution on other inputs.

Because it is well known that some programming faults are very difficult to find with testing, our research focuses on program characteristics that make faults hard to find with random black-box testing. Given a piece of code, we try to predict if random black-box testing is likely to reveal any faults in that code. (By “fault,” we mean those that have been compiled into the code.)

A program’s testability is a prediction of its ability to hide faults when the program is black-box-tested with inputs selected randomly from a particular input distribution. You determine testability by the code’s structure and semantics and by an assumed input distribution. Thus, two programs can compute the same function but may have different testabilities. A program has high testability when it readily reveals faults through random black-box testing; a program with low testability is unlikely to reveal faults through random black-box testing. A program with low testability is dangerous because considerable testing may make it appear that the program has no faults when in reality it has many.

A fault can lie anywhere in a program, so any method of determining testability must take into consideration all places in the code where a fault can occur. Although you can use our proposed techniques at different granularities, this arti...
cle concentrates on locations that roughly correspond to single commands in an imperative, procedural language.

We expect that any method for determining testability will require extensive analysis, a large amount of computing resources, or both. However, the potential benefits for measuring testability are significant. If you can effectively estimate testability, you can gain considerable insight into four issues important to testing:

- Where to get the most benefit from limited testing resources. A module with low testability requires more testing than a module with high testability. Testing resources can thus be distributed more effectively.
- When to use some verification technique rather than testing. Extremely low testability suggests that an inordinate amount of testing may be required to gain confidence in the software's correctness. Alternative techniques like proofs of correctness or code review may be more appropriate for such modules.
- The degree to which testing must be performed to convince you that a location is probably correct. You can use testability to estimate how many tests are necessary to gain desired confidence in the software's correctness.
- Whether the software should be rewritten. You may use testability as a guide to whether critical software has been sufficiently verified. If a piece of critical software has low testability, you may reject it because too much testing will be required to sufficiently verify a sufficient level of reliability.

**SENSITIVITY**

We use the word “sensitivity” to mean a prediction of the probability that a fault will cause a failure in the software at a particular location under a specified input distribution. If a location has a sensitivity of 0.99 under a particular distribution, almost any input in the distribution that executes the location will cause a program failure. If a location has a sensitivity of 0.01, relatively few inputs from the distribution that execute would cause the program to fail, no matter what faults exist at that location.

Sensitivity is clearly related to testability, but the terms are not equivalent. Sensitivity focuses on a single location in a program and the effects a fault at that location can have on the program’s I/O behavior. Testability encompasses the whole program and its sensitivities under a given input distribution. Sensitivity analysis is the process of determining the sensitivity of a location in a program. From the collection of sensitivities over all locations, we determine the program’s testability.

One method of performing sensitivity analysis is successive analysis of program execution, infection, and propagation, which is dynamic in the sense that it requires execution of the code. You randomly select inputs from the input distribution and compare the code’s computational behavior on these inputs against the behavior of similar code. Although this analysis is dynamic, it is not software testing, since you check no outputs against a specification or oracle.

**FAULT/Failure MODEL**

If the presence of faults in programs guaranteed program failure, every program would be highly testable. But this is not true. To understand why, you must consider the sequence of location executions that a program performs. Each set of variable values after the execution of a location in a computation is called a data state. After executing a fault, the resulting data state might be corrupted; if there is corruption in a data state, infection has occurred and the data state contains an error, which we call a “data-state error.”

The program in Figure 1 displays an integral solution to the quadratic equation $ax^2+bx+c$ for integral values of $a$, $b$, and $c$. (We have fixed $a$, $b$, and $c$ so $a$ and $c$ fall between 0 and 10 and so $b$ falls between 1 and 1,000.) The program has a fault at line 3: The constant 5 should be the constant 4. Each computation of the program falls into one of four categories:

- The fault is not executed,
- the fault is executed but does not infect any data state,
- the fault is executed and some data states are infected, but the output is nonetheless correct, and
- the fault is executed, infection occurs, and the infection causes an incorrect output.

Only computations in the final category would make the fault visible to a tester. Here are examples of each type of computation:

- Table 1 shows the computation for the input $(a, b, c) = (0, 3, 6)$. The value of $a=0$ causes the selection of a path that does not include location 3. Clearly, any such execution will not fail.
Table 2 shows the computation for the input (3,2,0). The fault is reached, but the computation proceeds just as if there were no fault because c=0 prevents the fault from affecting the computation. No infection has occurred.

For the input (1,-1,-12), the fault infects the succeeding data state, producing d=61 instead of d=49 (see Table 3). This data-state error then propagates to location 6 where it is canceled by the integer square-root calculation, because 7 is computed in either case.

Executing the program with the input (10,0,10) executes the fault that then infects the succeeding data state so the data-state error propagates to the output (see Table 4).

The first computation type demonstrates that a program's execution can reveal only information about the part of the code that is executed. The second and third types provide a false sense of security to a tester because the fault is executed but no visible failure results. The fourth type shows three necessary and sufficient conditions for a fault to produce a failure:

- The fault must be executed.
- The succeeding data state must be infected.
- The data-state error must propagate to output.

These three phenomena comprise the fault/failure model. This model underlies our dynamic method to determine the sensitivity of a location in the code.

### Sensitivity Analysis

Sensitivity analysis requires that every location be analyzed for three properties: the probability of execution occurring, the probability of infection occurring, and the probability of propagation occurring. One type of analysis is required to handle each part of the fault/failure model.

#### Getting three estimates

You can make all three analyses at several different levels of abstraction — programs, modules, and statements are three such levels. The examples in this article show an analysis done on program locations where a location is a unit of code that changes a variable's value.
changes the control flow, or produces an output. A program location is similar to a single high-level statement, but some such statements contain multiple locations. (For example, `Read(a,b)` contains two locations.)

Execution, infection, and propagation analyses each involve significant execution time, since the required work is done location by location. However (unlike testing), none of this analysis requires an oracle to determine a “correct” output. Instead, you can detect changes from the original I/O behavior without determining correctness. This lets the entire sensitivity analysis be automated.

Sensitivity analysis, although computationally intensive, is not labor-intensive. This emphasis on computing power seems increasingly appropriate as machine executions become cheaper and programming errors become more expensive.

The expense of the combined analysis depends on two factors: the number of locations in the software and the amount of confidence you want for the estimates. The analysis is quadratic in the number of locations. The amount of confidence improves as you iterate through the analysis repeatedly. In the algorithms that follow, the variable `n` denotes the number of iterations through each algorithm. It is not required, but we assume that the three algorithms use the same value for `n`.

You can choose `n` either according to how much computation you can afford or according to how much confidence you require. In either case, `n` and a desired confidence are related according to confidence intervals. For example, if `p` is the number of times a location was infected, if `n` is the number of infection attempts, and if you want a 95-percent confidence, the confidence interval is

\[ \hat{p} \pm 2 \sqrt{\frac{p(1-p)}{n}} \]

**Execution analysis.** Execution analysis is the most straightforward of the three analyses. It requires a specified input distribution (as does any quantifiable testing method), executes the code with random inputs from that distribution, and records the locations executed by each input. This produces an estimate of the probability that a location will be executed by a randomly selected input according to this distribution. Thus, execution analysis is concerned with the likelihood that a particular location will have an opportunity to affect the output.

The algorithm for finding an execution estimate is:

- Set the `Count` array to zeroes, where the size of `Counts` is the number of locations in the program being analyzed.
- Instrument the program with `WriteLn` statements at each location that print the location number when the location is executed, making sure that, if a location is repeated more than once on some input, the `WriteLn` statement for that location is executed only once for that input.
- Execute `n` input points on the instrumented program, producing `n` strings of location numbers.
- For each location `l` in a string, increment the corresponding `Counts[l]`. If a location `k` is executed on every input, `Counts[k]` will equal `n`.
- Divide each element of `Counts[l]` by `n`, yielding an execution estimate for location `l`.

Each execution estimate is a function of the program and an input distribution.

**Infection analysis.** Our algorithm for infection analysis requires the creation of code mutants, an idea used extensively in mutation testing. Mutants are copies of the original code that have been syntactically altered. In the case of infection analysis, we use only semantically significant mutants. Infection analysis estimates the probability that a mutant at a location will infect the data state.

Infection analysis is similar to mutation testing; what is different is the information collected. For a given location in a program, you do not know whether a fault exists, and you don’t know what types of faults are possible at the location. So you create a set of mutants at each location. After creating a set of mutants, you obtain an estimate of the probability that the data state is affected by the presence of a mutant for each mutant in the set. You select a mutant from the set, mutate the code at the location, and execute each resulting mutant many times. You check the data states created by executing the mutants against the data states from the original location to determine if the mutants have infected the data states. The proportion of executions that infect for a particular mutant are the infection estimate for that mutant.

The algorithm for finding an infection estimate is:

- Set the variable `Count` to 0.
- Create a mutant, denoted as `b`, for location `l`.
- Present the original location `l` and the mutant `b` with a randomly selected data state from the set of data states that occur immediately before location `l` and execute both locations in parallel.
- Compare the resulting data states and increment `Count` when the function computed by `b` does not equal the function computed by `l` for this data state.
- Repeat the last two steps `n` times.
- Divide `Count` by `n`, yielding an infection estimate.

An infection estimate is a function of a location, the mutant created, and the set of data states that occur before the location. You generally perform this algorithm many times at a location to produce a set of infection estimates.

We are still researching the exact nature of the best code mutations for infection analysis, but we have gotten encouraging results from a small set of mutations based on semantic changes. These mutations are straightforward to describe and can be automated. Furthermore, the results of the analysis using these mutations...
have been encouraging. To illustrate this set of mutants, Table 5 shows the mutants generated for locations 3, 4, and 6 in our example program. For each mutant, the table shows the infection estimate obtained from executing 10,000 random inputs through each location.

Propagation analysis. Propagation analysis estimates the probability that an infected data state at a location will propagate to the output. To make this estimate, you repeatedly perturb the data state that occurs after some location, changing one value in the data state for each execution. Thus, one live variable receives an altered value. (We consider a variable to be live if the variable has any potential of affecting the output of the program. For example, a variable that is defined but never referenced is a variable that would not be live.)

By examining how often a forced change in a data state affects the output, you calculate a propagation estimate, which is an estimate of the effect that a live variable has on the program’s output at this location. You find a propagation estimate for a set of live variables at each location (assuming there is more than one live variable at a location), thus producing a set of propagation estimates with one propagation estimate per live variable.

If you were to find that at a particular location a particular live variable had a propagation estimate near 0.0, you would realize that this variable had very little effect on the program’s output at this location. Thus, propagation analysis is concerned with the likelihood that a particular live variable at a particular location will cause the output to differ after the live variable’s value is changed in the location’s data state.

Propagation analysis is based on changes to the data state. To get the data states that are then executed to completion, our method uses a mathematical function based on a random distribution that we call a perturbation function. A perturbation function inputs a variable’s value and produces a different value chosen according to the random distribution. The random distribution uses the original value as a parameter when producing the different value.

We are researching different perturbation functions; we now use a uniform distribution whose mean is the original value. The range of values that a variable had during the executions used to get the execution estimates determines the maximum and minimum different values that the perturbation function can produce.

An algorithm for finding a propagation estimate is:

1. Set the variable Count to 0.
2. Randomly select a data state from the distribution of data states that occur after location $l$.
3. Perturb the sampled value of the variable $a$ in this data state if $a$ is defined; otherwise, assign $a$ a random value. Execute the succeeding code on both the perturbed and original data states.
4. For each different outcome in the output between the perturbed data state and the original data state, increment Count. You would also increment Count if an infinite loop occurs (set a time limit for termination, and assume that an infinite loop occurred if execution has not finished in that time).
5. Repeat the last three steps $n$ times.
6. Divide Count by $n$, yielding a propagation estimate.

A propagation estimate is a function of a location, a live variable, the set of data states that occur after the location (that are a function of some input distribution), and the code that is possibly executed after the location.

Understanding the estimates. When all three analyses are complete, you have three sets of probability estimates for each location. For each location, there are several ways to manipulate these three estimates. This article describes only one; we have published others elsewhere.

To reveal a fault at a particular location with a particular test case, the location must be executed, an infection must occur, and the infection must propagate to the output. If any of these does not occur, the fault will be invisible to the tester. Therefore, you can derive a conservative estimate of a location’s sensitivity by using the minimum estimate from each of these three sets.

We choose the minimum estimate from each set because we believe it is better to overestimate the amount of random black-box testing needed to reveal a fault rather than to underestimate it. If you underestimate the amount of testing necessary, you may be fooled into thinking no fault exists when there really is one. If you overestimate the amount of testing, you
you have available is the program method for finding a location's sensitivity instead of an overestimated sensitivity, we modified the equation to account for this possibility and to produce the sensitivity of location $l$ (denoted as $\beta_l$):

$$\beta_l = \varepsilon_l \cdot \sigma((\min[f_x])_{\min} \cdot (\min[\omega_j])_{\min})$$

where $\sigma(a,b)$ equals $a-(1-b)$ if $a-(1-b)$ is greater than 0; otherwise, $\sigma(a,b)$ equals 0.

**Example of low testability.** To explore the relationship between sensitivity analysis and testability, we performed an experiment. We analyzed execution, infection, and propagation at locations 3, 4, and 6 of the program in Figure 1 with 10,000 random inputs and the following random input distributions: $a$ and $c$ were equally likely to be a number between 0 and 10 and $b$ was equally likely to be a number between 1 and 1,000. Thus, there are 121,000 different inputs to this program.

Table 6 shows the results of propagation analysis, Table 5 shows results of injection analysis, and Table 7 shows the results of execution analysis.

The sensitivities found using our modified equation for locations 3, 4, and 6 are all 0.0. Locations 3 and 6 produced very low propagation estimates, and location 4 produced a very low injection estimate, thus resulting in zero sensitivities.

At first glance, this seems to be less than informative, but this information is in fact both reasonable and useful, since location 8 is critical in reducing the propagation estimates found from the preceding seven locations. Locations 2 through 7 do virtually all the computing of $x$. At location 8, unless its condition is true, the computation of $x$ in locations 2 through 7 is not referenced in the output; location 10 prints out that there is no solution, and whatever computations occurred in locations 2 through 7 are ignored. This is a program that by its nature rarely has a solution with the particular input distribution we selected. (We purposely selected this input distribution to highlight what we mean by low sensitivities.)

Thus, data-state errors injected into locations 3, 5, 6, or 7 rarely are given the opportunity to affect the output, because location 9 has a low execution estimate. The low sensitivities for these locations reflect reality — with the input distribution given, data-state errors in locations 3 through 7 will have little if any effect on the output. Thus, under this distribution, the program has low testability.

**Example of high testability.** Now consider a new program:

In the real world, you do not have the luxury of knowing about the occurrence of a specific fault at a specific location. What you have available is the program and an input distribution.
Location 2 in this program has a higher sensitivity than the locations in our earlier program. Because location 2 is an output location, the propagation estimate is 1.0, since any change to the output data state changes the output.

Because the execution estimate at this location is 1.0, the sensitivity of location 2 depends almost entirely on the minimum infection estimate of location 2. Table 8 shows the infection estimates found for a small set of mutants tried at location 2. The minimal infection estimate mutant from this set is 0.8237, which is also the sensitivity (since \( r = 0.8237 - (1-1) \) equals 0.8237).

Thus, the minimum over the set of infection estimates becomes the sensitivity of location 2. For this location, sensitivity analysis predicts that this location will not require much black-box testing to reveal a fault in location 2 if one exists. This program has high testability.

**Blind experiment.** We ran a blind experiment with the program in Figure 1 to test the hypothesis that sensitivity analysis helps estimate testability. We hypothesized that, for an injected fault, the sensitivity for the location where the fault was injected was always less than or equal to the resulting failure probability estimate of any fault injected at that location.

We used sensitivities (which are found solely with a fault) to underestimate the failure probabilities that occur from a set of injected faults into an oracle version of the program. One of us produced the sensitivities previously shown, while another independently produced failure-probability estimates after placing faults at locations 3, 4, and 6 into a corrected version of the program in Figure 1. The failure-probability estimates were based on 10,000 inputs for the faults injected at locations 1 and 6 and on 100,000 inputs for the fault injected at location 4. Table 9 shows the resulting failure-probability estimates and faults. In both cases, the hypothesis was supported.

**Sensitivity as testing indicator.** The sensitivity of a location is an indicator of how much testing is necessary to reveal a fault at that location. For example, a sensitivity of 1.0 suggests that, on the first test of the location, failure will result if there is a fault there, and thus the existence of a fault will immediately be revealed. A sensitivity of 0.01 suggests that, on average, one in every 100 tests of a location will reveal a failure if a fault exists. Sensitivity gives you a rough estimate of how frequently a fault will be revealed if one exists.

You can use \( \beta \) from our earlier example as an estimate of the minimum failure probability for location \( I \) in the equation \( 1-(1-\beta)^{T} = c \), where \( c \) is the confidence that the actual failure probability of location \( I \) is less than \( \beta \). With this equation, you can obtain the number of tests \( T \) needed for a particular \( c \). To obtain a confidence \( c \) that the true failure probability of a location \( I \) is less than \( \beta \), given the location's sensitivity, you need to conduct \( T \) tests, where

\[
T = \frac{\ln(1-c)}{\ln(1-\beta)}
\]

When \( \beta \) is close to 0.0, you effectively have the confidence \( c \) after \( T \) tests that location \( I \) does not contain a fault. To obtain a confidence \( c \) that the true failure probability of a program is less than \( \beta \), given the sensitivities of its locations, you need to conduct \( T \) tests, where

\[
T = \frac{\ln(1-c)}{\ln(1-\min(\beta))}
\]

When \( \min(\beta) \) is close to 0.0, you effectively have the confidence \( c \) after \( T \) tests that the program does not contain a fault. For these equations, \( \beta \) cannot be 0 or 1.

We conservatively estimate the testability of an entire program to be the minimum sensitivity over all locations in the program: \( \min(\beta) \). Thus, the greater a program's testability, the fewer tests needed in the second equation to achieve a particular confidence that the program does not contain a fault.

**Sensitivity analysis can add another dimension to software quality assurance.** During initial code development, sensitivity analysis can identify code that will inherit a greater tendency to hide faults during testing; you can rewrite such code or subject it to other types of analysis to detect faults.

During random black-box testing, sensitivity analysis will help interpret testing results: You will have more confidence in code with high sensitivity that reveals no errors during random black-box testing than code with low sensitivity that reveals no errors during the same type of testing.

You can save testing resources by testing high-sensitivity locations less frequently than might otherwise be necessary to obtain confidence in the code. Conversely, low-sensitivity locations may require additional testing.

During maintenance, you can use sensitivity analysis to identify locations where subtle bugs could be hiding from conventional testing techniques; you could then

---

**Table 8**

<table>
<thead>
<tr>
<th>Location</th>
<th>Mutant</th>
<th>Infection estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>writeln (a<em>a + b</em>a + a)</td>
<td>0.9085</td>
</tr>
<tr>
<td>2</td>
<td>writeln (a<em>a + b</em>a - c)</td>
<td>0.9135</td>
</tr>
<tr>
<td>2</td>
<td>writeln (a<em>a - b</em>a + c)</td>
<td>0.909</td>
</tr>
<tr>
<td>2</td>
<td>writeln (a<em>c + b</em>a + c)</td>
<td>0.8237</td>
</tr>
</tbody>
</table>

**Table 9**

<table>
<thead>
<tr>
<th>Location</th>
<th>Injected fault</th>
<th>Failure-probability estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>d := sqrt(a) - 3<em>a</em>c</td>
<td>0.897</td>
</tr>
<tr>
<td>4</td>
<td>if (d&lt;0) then</td>
<td>0.00012</td>
</tr>
<tr>
<td>6</td>
<td>x := (-b + trunc(sqrt(d))) div 2*a</td>
<td>0.9001</td>
</tr>
</tbody>
</table>

---

1. read (a,b,c);
2. writeln (a*a + b*a + c);
3. d := sqrt(b) - 3*a*c
4. if (d<0) then
5. x := (-b + trunc(sqrt(d))) div 2*a

IEEE SOFTWARE 47
use alternative analyses at those locations. Sensitivity analysis includes some characteristics that are similar to mutation testing, but the differences are significant. Infection analysis mutates source code, and so is related to mutation testing. But the goals of the two techniques are different. Mutation testing seeks an improved set of test data; infection analysis seeks to identify locations where faults are unlikely to change the data state. Propagation analysis mutates the data state, not the code, and then examines whether the output is affected. This is similar to some dataflow research, but again the aim is different: Sensitivity analysis dynamically estimates relevant probabilities and uses these estimates to better understand test results. To our knowledge, this emphasis is unique.

EXECUTIVE POSITION
Carnegie Mellon University
Software Engineering Institute

Technology Division Director

The Software Engineering Institute (SEI) is a federally funded research and development center (FFRDC) sponsored by the Department of Defense under contract to Carnegie Mellon University. The SEI mission is to provide leadership in advancing the practice of software engineering. We are looking for a candidate with proven management skills to fill the newly created position of Director of the Technology Division.

Responsibilities
- Manage, motivate and build the division team (70-100 technical staff) in focus areas of real-time distributed systems, process and risk management, trusted fault tolerant systems, and software engineering methods/tools
- Improve the practice of software engineering by bringing promising approaches to the stage where they are ready for widespread use
- Participate with other division directors in developing and maintaining the SEI strategy and business plan

Qualifications
- A demonstrated record of technical and managerial accomplishment
- Extensive experience with software engineering advanced technology
- Demonstrated leadership at the program management level or above with emphasis on motivating excellence
- PhD in computer science, electrical engineering or a related field is highly desirable

Respond to:
Sally Miller (F-5)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

ACKNOWLEDGMENTS
This work was supported by a National Research Council resident research associateship and by NASA grants NAG-1-825 and NAG-1-884. The authors also thank the anonymous referees for their valuable suggestions.

REFERENCES

Jeffrey Voas is a research assistant professor at the College of William and Mary's Computer Science Dept. working as a National Research Council resident research associate at the National Aeronautics and Space Administration's Langley Research Center. His research interests include software testing, studies in data-state error propagation, and design techniques in improving software usability. Voas received a BS in computer engineering from Tulane University and an MS and PhD in computer science from the College of William and Mary.

Larry Morell is an associate professor of computer science at Hampton University. His research interests include include fault-based and error-based testing and the specification and implementation of software tools. Morell received a PhD in computer science from the University of Maryland.

Keith Miller is an associate professor of computer science at the College of William and Mary. Miller received a PhD in computer science from the University of Iowa. Address questions about this article to Voas at MS 478, NASA Langley Research Center, Hampton, VA 23665; Internet jmvoas@phoebus.lam.nasa.gov.