Uncovering Weaknesses in Code With Cyclomatic Path Analysis

Thomas J. McCabe Sr., McCabe Technologies
Thomas J. McCabe Jr., McCabe Software
Lance Fiondella, University of Connecticut

Abstract. Software flaws represent a serious threat to system integrity. Today, software plays an increasingly important role in the infrastructure of government agencies. These entities outsource and use open-source software within their critical infrastructure; however, the origins and security characteristics of this code are rarely certified. We compare the relative effectiveness of the statement, branch, and cyclomatic code coverage software testing methodologies for detecting flaws in software.

Foreign influence on DoD software is a major security concern [1]. A programmer can insert a flaw into code that looks like an honest mistake, but when triggered leads to unexpected behavior in the system on which the software resides. The consequences could be anything from system unavailability to outright hijacking of the system and all of its functionality. Given the potentially catastrophic consequences of allowing exploitable software flaws to reside in operational systems, software testing is now being acknowledged as a critical step to mitigate software supply chain risks [2].

Protecting against the "inside job" is not the only concern for those wishing to protect software systems from attack. Foreign adversaries persistently attempt to break into the networks of defense facilities and their contractors. A successful intruder would steal anything that could provide economic or strategic advantage. The speculated compromise of the Joint Strike Fighter [3] is a high profile example, with tens of thousands of hours of programming feared lost. Not only can code be copied, it can be studied intensively for weaknesses. By interfacing operational systems running the software and injecting attacks to trigger exploitable weaknesses, the range of consequences mentioned above could be realized. Any unprotected statements in code that could lead to failure become fair game. The only way to ensure compromised software can withstand external attacks is to subject it to rigorous testing and identify weaknesses for removal before they can ever be targeted for attack. A software testing methodology that can eliminate the majority of flaws, both intentional and unintentional, is essential for producing and preserving software dependability.

Software Weaknesses

The Common Weakness Enumeration (CWE) [4] has emerged as a knowledge base of software weaknesses and vulnerabilities. This repository categorizes software flaws across multiple dimensions, describing major properties. For each kind of weakness, the CWE enumerates when it is introduced, common consequences, how likely it is to be exploited, and some examples of code containing the weakness. The CWE was designed to serve as "a standard measuring stick for software security tools targeting these weaknesses" [4]. As such, the CWE may be likened to a medical compendium that focuses only on pathology, describing the conditions, processes, and results of a disease. Treatment methodologies and medications are beyond the scope of the CWE itself. Like medicine, diagnosis and prevention of software vulnerabilities will be critical to limit the harm that can be done by those wishing to do damage. Moreover, software testing will be a key tool for conducting this vulnerability analysis.

Specific software testing methodologies identify some weaknesses, but can fail to identify others. In the absence of a single panacea to the software vulnerability epidemic, a remedy against the majority of common software ailments will prove highly effective. The most widely studied set of software testing strategies are those that study various forms of code coverage [5]. Code coverage is a part of the DO-178B [6] software verification process, which provides guidelines for certifying software in airborne systems and equipment. Code coverage approaches characterize the static control flow paths of an application as a graph with vertex nodes representing code statements, and edges representing possible branches within the code like the if and else statements.

This article is the first to compare the relative effectiveness of the statement, branch, and cyclomatic path analysis software testing methodologies for targeting weaknesses. Statement coverage seeks to test all of the nodes, while the goal of branch coverage is to traverse every edge of the graph. Statement and branch testing have limitations because interactions between decision outcomes can mask errors during testing. As a result, neither statement nor branch testing is adequate to detect vulnerabilities and verify control flow integrity. Cyclomatic Path Analysis [7], on the other hand, detects more CWE vulnerabilities. The fundamental idea behind Cyclomatic Path Analysis, also known as Basis Path or Structured Testing, is that decision outcomes within a software function should be tested independently [8]. By identifying software vulnerabilities with standard testing, a majority of attack opportunities will be eliminated before they can ever be exploited.
Detecting Security Flaws With Cyclomatic Complexity-based Testing

A critical comparison of software testing methodologies is essential to illustrate how competing approaches can fail to identify particular weaknesses. The following three examples consider this additional aspect and demonstrate that cyclomatic complexity-based testing can successfully detect several common weaknesses.

Divide By Zero
CWE-369: Dividing by zero is a commonly occurring problem. In mathematics, dividing a number by zero is not permitted because the result is defined to be infinity. This poses a challenge for computers, which cannot work with such a large number. Attempting to divide by zero on a computer leads to a condition known as overflow. Though one may think this exception should be simple to eliminate, overflows happen quite frequently because many programming languages set a variable to zero before it is ever assigned a value. All too often, programmers neglect to initialize a variable before using it as the denominator of a statement that performs division. This frequent occurrence makes the divide by zero weakness a widespread problem. Dividing by zero can lead to a variety of unpredictable behavior in software. Potential outcomes include unintended branching to error handling routines, software crashes, and similar undesirable behaviors. A programmer who intentionally or unwittingly introduces a divide by zero flaw can induce system crashes, rendering a system unavailable to perform its appointed tasks.

Algorithm 1: Simple average routine.
1: void simpleAvg(int array[], int n)
2: int total = 0;
3: int count = 0;
4: for ( count = 0; count < n; count++ ) do
5: total += array[count];
6: end for
7: return total / count;

The SimpleAvg routine computes an arithmetic average by adding up the first \( n \) numbers in the array and then divides their total by \( n \).

Figure 1 shows the statement graph of the simple average routine.

The nodes in the graph correspond to the seven lines of code and the edges represent the possible transfer of control between these lines. This statement graph is used to measure the coverage with respect to each of the three testing methodologies under consideration. Passing an array with one or more elements and a positive value for the second parameter \( n \) will lead to successful loop entry and exit, exercising 100% of the statements and branches. For example, the following two lines of code achieve complete statement and branch coverage.

Algorithm 2 provides an instance of code containing an exploitable memory leak.

Memory Leaks
CWE-401 describes the failure to release memory before removing the last reference. This type of weakness is most commonly known as a “memory leak.” Memory leaks occur when an application does not properly track allocated memory so that it may be released after it is no longer needed. Leaking memory slowly eats away at this finite resource. If no scheduled restart of the system occurs, undesirable outcomes like an operating system freeze can result. Memory leaks contribute to the unreliability of software. A programmer who intentionally conceals a memory leak provides a digital beachhead from which an attacker can easily launch a denial of service attack that whittles down the memory, crashing the program and unleashing the unexpected consequences of system failure.

Algorithm 2: Fill arrays routine.
1: void fillArrays(void **s1, void **s2, int size1, int size2)
2: if ((s1=malloc(size1)) && (s2=malloc(size2))) then
3: memset(*s1, 0, size1);
4: memset(*s2, 0, size2);
5: else
6: *s1 = *s2 = NULL;
7: end if
The purpose of the fill arrays function is to allocate memory for two pointers and set the pointers to these newly allocated areas. The pointers are assigned if memory allocation succeeds, but are set to NULL otherwise. At first blush, the implementation appears to be a harmless decision with two possible outcomes. Figure 2 shows the statement graph corresponding to the fillArrays routine.

One may think that the two tests given in the following code fragment should be sufficient to achieve statement and branch coverage.

```c
1: void* ptr1 = 0;
2: void* ptr2 = 0;
3: fillArrays(&ptr1, &ptr2, 10, 100);
4: fillArrays(&ptr1, &ptr2, 0xFFFFFFFF, 2);
```

The first test, on line three, will cause the if statement to run, while the test on line four will cover the else statement because the attempt to allocate 0xFFFFFFFF memory will fail on machines with less than four gigabytes of available memory. These tests achieve statement coverage and appear to attain branch coverage. Note, however, that the if statement is actually composed of two conditions. When the first memory allocation (malloc) statement for string pointer s1 succeeds, but the second memory allocation statement fails, the code will still execute the else statement and set both pointers to NULL. The memory from the first successful allocation should be freed, but the reference to this memory is “leaked.”

Thorough coverage must also account for scenarios where only the first condition evaluates to true. Figure 3 shows this more detailed cyclomatic graph, where line two is divided into nodes 2a and 2b to represent the two malloc statements embedded in the if statement.

The following code fragment provides the additional test needed to exercise this basis path introduced by the compound logic in the if statement.

```c
1: void* ptr1 = 0;
2: void* ptr2 = 0;
3: fillArrays(&ptr1, &ptr2, 2, 0xFFFFFFFF);
```

The flaw lies on the edge between nodes 2b and 5 of Figure 3. This last test will trigger the memory leak because the first amount of memory requested is very small, but the second will fail. Repetitive execution of this last test could quickly chisel away at the memory resources. This is yet another instance where statement and code coverage can prove inadequate, but cyclomatic basis path testing detects the weakness.

**Out-of-bounds Read**

CWE-125 is an out-of-bounds read. This type of behavior occurs when software reads data before the beginning or past the end of the intended buffer. This can happen when a pointer or its index is increased or decreased to a position beyond the bounds of the buffer or by pointer arithmetic that results in a location outside of the appropriate memory location. Potential outcomes include: software crashing, unintended execution of code, and data corruption. A programmer who devises an out-of-bounds read can do potentially unlimited damage. In the worst case, they could hijack control of the system, turning it against its owners.

Algorithm 3 contains an exploitable out-of-bounds read.

**Algorithm 3**: Character copying routine.

```c
1: void copyChars(char** dest, char** src, int start, int end)
2:   int charsToCopy = 1;
3:   int lastPos = strlen(*src) - 1;
4:   if ( end > lastPos ) then
5:     end = lastPos;
6:   end if
7:   if ( start < 0 ) then
8:     start = 0;
9:   end if
10:  if ( start < 0 ) then
11:    copyToChars += (end - start);
12:  end if
13:  strncpy(*dest, (*src) + start, charsToCopy);
```

**What is Cyclomatic Complexity?**

Important facts about the cyclomatic complexity metric include:
- Cyclomatic complexity enables defensive coding procedures such as code flattening, which simplifies understanding the structural characteristics of software.
- Cyclomatic complexity models information flow control and can help discover sneak paths within source code.

Figure 2: Statement graph of fill arrays routine.

Figure 3: Cyclomatic graph of fill arrays routine.
The copyChars routine is intended to copy a range of characters from the source to destination array. There are three sequential checks that occur prior to this copying of characters. The first, on line four, validates that the end position is within the bounds of the source string. The second conducts a similar check to ensure the start position is within bounds, and the third ensures the end position is after the start.

The test case on line four initiates the opportunity for an out-of-bounds read. The first test on line four evaluates to true because the end variable equals 100, which is longer than the, “Hello My World!” string. As a result, line five of the character copying routine sets the end variable to the length of the source string. The second test on line seven, however, evaluates to false because the start variable equals 1,000, which is greater than zero. Thus, line eight is skipped. Finally, the test on line ten evaluates to false because 100 is not less than 1,000, so line 11 is not executed. Line 13 copies a byte from a location 1,000 positions beyond the start of the source string to the destination because charsToCopy=1. This type of out-of-bounds read can be used to feed an application the address of instructions to execute, introducing the potential to commit serious violations of system security. Cyclomatic path testing exposes this vulnerability, but statement and branch coverage do not.

Managing the Attack Map

Up until now, the article has focused on testing simple modules for vulnerabilities. In large-scale software testing, this search is not a mere hunt for vulnerable routines. Instead, it is a more comprehensive examination of relationships to explore control-flow graphs, routine reachability, and the attack map, attack surface, and attack target, which are defined in the following discussion. The attack surface of software is the code within a computer system that can be run by unauthenticated users. Recent research [10] proposed an I/O automata model of a system and its environment to formalize the notion of the attack surface. A concrete implementation of this formalism is cyclomatic path analysis.

The attack surface is the set of functions S that allow user inputs affecting system behavior. Examples include operations that read from configuration files, receive network data, and keyboard inputs. Library functions of potential interest might be input functions such as gets(), recv(), and scanf(). The attack target is the set of routines T that can cause critical impacts when exploits are attempted. Code that might trigger reformat of the hard drive or shutdown certain services are specific instances of attack targets. Calls that can perpetrate these abuses include system functions like exec(), which starts new processes, LoadLibrary(), which can load shared objects, and dynamically linked libraries are all potential threats. Finally, the attack map M is the application subgraph connecting the attack surface and attack target. This structural context promotes the joint analysis of routines that connect the surface and target, which will prove more revealing than study of the two in isolation. Identifying the control flow relationships between the surface and target provides the opportunity to apply a path-oriented approach to focus the review and testing on these connected components. This addresses a major challenge associated with vulnerability isolation, namely the overwhelming amount of source code that must be analyzed.
Many times flaws reside within millions of lines of code and are introduced somewhere along the software supply chain. By accounting for the connectedness of components, cyclomatic path analysis simplifies graph complexity to the routes by which an attacker can reach particular software vulnerabilities.

An additional advantage of structural security analysis is the ability to define lists of functions that must be considered in the performance of attack map analysis, modularizing the process. An example is the list of Microsoft Secure Development Lifecycle (SDL) [11] banned functions. Microsoft recommended processes on secure development specify a list of standard C functions. Microsoft discourages programmers from invoking these routines because they are prone to vulnerabilities like memory leaks and buffer overruns. This list of C functions is ideal for conducting security analysis on legacy applications to bring them into conformance with the Microsoft SDL. The scanf() and printf() functions are banned members of the attack surface and target respectively. Structural simplifications can effectively constrain analysis by aggregating the modules containing the surface and target into two “supercomponents”, simplifying the view of the potential paths from entry points to flaw exploitation. This grouping facilitates test specification, providing a global perspective on the analysis task at hand within the context of the application.

Summary

Software vulnerabilities are a consequence of multiple factors. Attackers can disrupt program operation by exercising a specific sequence of interdependent decisions that result in unforeseen behavior. To ensure program behavior is correct, these paths must be identified and exercised as part of secure software development. Software testing techniques that utilize complete line and branch coverage are insufficient and leave too many gaps. Cyclomatic complexity enables more comprehensive scrutiny of the structure and control flow of code, providing significantly higher vulnerability detection capabilities.

Static analysis for code review has been suggested as a valuable aid for critical software assurance [12]. The future of software engineering would benefit from tight integration of development with testing. Automatically warning developers of the security vulnerabilities present in their code will be a first step toward eradicating common weaknesses.

The Software Maintenance Group at Hill Air Force Base is recruiting civilian positions (U.S. Citizenship Required). Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance and time off for fitness activities. Become part of the best and brightest!

Hill Air Force Base is located close to the Wasatch and Uinta mountains with many recreational opportunities available.

Send resumes to: phil.coumans@hill.af.mil or call (801) 586-5325
Visit us at: http://www.309SMXG.hill.af.mil

Electrical Engineers and Computer Scientists
Be on the Cutting Edge of Software Development

The Software Maintenance Group at Hill Air Force Base is recruiting civilian positions (U.S. Citizenship Required). Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance and time off for fitness activities. Become part of the best and brightest!

Hill Air Force Base is located close to the Wasatch and Uinta mountains with many recreational opportunities available.

Send resumes to: phil.coumans@hill.af.mil or call (801) 586-5325
Visit us at: http://www.309SMXG.hill.af.mil
THE END OF THE PC

ABOUT THE AUTHORS

Thomas J. McCabe Sr. is a mathematician, author, and founder of McCabe Technologies <http://www.mccabe.com>, which he grew to 175 people. In 1998, he testified before Congress in the hearing entitled, “Year 2000: Biggest Problems and Proposed Solutions.” He has authored three books, is highly published in computer science, and lectures internationally on software development and entrepreneurship. In 2009, the ACM-SIGSOFT retrospectively selected his publication on Software Complexity [8] as one of the 23 highest impact papers in computer science.

McCabe Technologies
5163 Harpers Farm Road
Columbia, MD, 21044
E-mail: tom@mccabetech.com

Thomas J. McCabe Jr. has worked with clients of McCabe Technologies for more than 20 years to improve their quality assurance (QA) and test processes. He has delivered numerous presentations on software testing and QA to organizations and conferences including IEEE, OAI, Sticky Minds, DAU, NDU, and DHS. His recent Sticky Minds Webinar, “Uncovering Security Vulnerabilities Off the Beaten Path,” presented his experiences applying complexity metrics, verifying control flow integrity, and leveraging path analysis to uncover security vulnerabilities.

McCabe Technologies
3806 Spring Meadow Drive
Ellicott City, MD 21042
E-mail: skillssss@hotmail.com

Lance Fiondella is a Ph.D. student at the University of Connecticut (UConn). In 2007, he received a scholarship from the IEEE Reliability Society for his research on system and software reliability. He also conducts network vulnerability research for UConn’s Department of Homeland Security National Transportation Security Center of Excellence (NTSCOE). He was an invited speaker at the 2011 DHS Student Day, where he presented his research on the optimal deployment and protection of high-speed rail.

University of Connecticut
371 Fairfield Road
Storrs, CT, 06269-4155
Phone: 860-486-3665
E-mail: lfiondella@engr.uconn.edu

REFERENCES