Leveraging Cyclomatic Complexity and Path Analysis to Verify Control Flow Integrity of Secure Systems

Thomas McCabe Jr.
Structured Testing Specialist

"The future of digital systems is complexity, and complexity is the worst enemy of security." *Bruce Schneier, Founder and CTO, Counterpane Internet Security, Inc. 2000*

About this Paper

Many security exploits are about interactions. Being cognizant of paths and subtrees within source code is crucial for testing to verify control flow integrity and uncovering security flaws hiding along obscure paths or subtree structures within a codebase. Security vulnerabilities are often a consequence of multiple factors. Attackers can disrupt program operation by exercising a specific sequence of interdependent decisions that result in unforeseen behavior. As part of secure software development, these paths must be identified and exercised, to ensure that program behavior is correct and expected. Techniques for complete line and branch coverage leave too many gaps. Cyclomatic complexity and basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique.

In this paper we will discuss:

Why complexity is the worst enemy of security

How to measure control flow integrity

Ideas on using static and dynamic path analysis to secure applications

Why being cognizant of and measuring source code paths is crucial to software security

This paper will show you how using software complexity metrics, measuring control flow integrity, and performing sneak path analysis help you make your applications more secure that previously thought possible.

Complexity and Insecurity

Software systems are becoming less secure even as security technologies improve. There are many reasons for this seemingly paradoxical phenomenon, but they can all be traced back to the problem of complexity.

- Complex systems have more lines of code and therefore security bugs.
- Complex systems have more interactions and therefore more security bugs.
- Complex systems are harder to test and therefore are more likely to have untested portions.
- Complex systems are harder to design securely, implement securely, configure securely and use securely.
- Complex systems are harder for users to understand[i].

Security Debuggers vs. Security Testing

Tools that search for known exploits are analogous to debuggers and are employed using a reactive model rather than a proactive one. Many exploits deal with interactions: interactions between code statements, interactions between data and control flow, interactions between modules, interactions between your codebase and library routines, and interactions between your code and attack surface modules. Understanding code paths within a codebase can expose the true nature of these interactions. This is why cyclomatic complexity path and subtree analysis is an important complementary technique. Being cognizant of paths and subtrees within code is crucial for determining sneak paths, performing

impact analysis, and testing to verify control flow integrity. It is crucial that both security debuggers and security control flow integrity test tools are included in your arsenal.

Source Code Analysis vs. Binary Analysis

As is the case with static analysis and dynamic analysis, the two approaches of source and binary analysis are complementary. Source analysis is platform (architecture and operating system) independent, but language-specific; binary analysis is more language-independent but platform-specific. Source code analysis has access to high-level information, which can make it more powerful; dually, binary analysis has access to low-level information (such as the results of register allocation) that is required for some tasks.

Bottom line is: The binary approach effectively analyzes what the compiler produces, whereas the source approach effectively analyzes what the developer produces.It is true that binary (compiled) code represents the actual attack surface for a malicious hacker exploiting software from the outside. It is also true that source code analysis has differentiated itself in a complementary way by finding the enemy within software development shops. There have been studies indicating that exploits from within are far more costly than those from the outside. Source code analysis can be employed much earlier in the software development lifecycle (SDLC). Libraries and APIs can be tested early and independently of the rest of the system. Binary analysis requires that at least an entire executable, if not an entire subsystem or system is completed.

In binary analysis, it is true that white box analysis reporting can be generated. However, these reports are indirect, and do not always correlate exactly back to the source code logic; therefore, detailed analysis may be more difficult than humans analyzing source code analysis reporting. Furthermore, compilers and their options (such as optimization) can cause the correlation between binary analysis reporting and source code to be even more different.

Security Problems as Software Grows More Complex

As software grows more complex, it contains many more flaws for hackers to exploit. Powerful computer systems and increasingly complex code will be a growing cause of insecure networks. We are getting these great performance improvements, which leads to increases in complexity. Today, nobody has any clue what is running on their computer.

In the *Final Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence DOD Software - November 2007*, the following statements were made: The complexity of software itself can make corruption hard to detect. Software has been growing in the dimensions of size, complexity and interconnectedness, each of which exacerbates the difficulties of assurance. Software complexity is growing rapidly and offers increasing challenges to those who must understand it, so it comes to no surprise that software occasionally behaves in unexpected, sometimes undesirable ways. The vast complexity of much commercial software is such that it could take months or even years to understand. The Nation's defense is dependent upon software that is growing exponentially in size and complexity. The following findings were found in this report: The enormous functionality and complexity of IT makes it easy to exploit and hard to defend, resulting in a target that can be expected to be exploited by sophisticated nation-state adversaries. The growing complexity to the microelectronics and software within its critical systems and networks makes DoDs current test and evaluation capabilities unequal to the task of discovering unintentional vulnerabilities, let alone malicious constructs.

One of the key properties that works against strong security is complexity. Complex systems can have backdoors and Trojan code implanted that is more difficult to find because of complexity. Complex operations tend to have more failure modes. Complex operations may also have longer windows where race conditions can be exploited. Complex code also tends to be bigger than simple code, and that means more opportunity for accidents, omissions and manifestation of code errors.

A central enemy of reliability is complexity. Complex systems tend to not be entirely understood by anyone. If no one can understand more than a fraction of a complex system, then, no one can predict all the ways that system could be compromised by an attacker. Prevention of insecure operating modes in complex systems is difficult to do well and impossible to do cheaply. The defender has to counter all possible attacks; the attacker only has to find one unblocked means of attack. As complexity grows, it becomes ever more natural to simply assert that a system or product is secure as it becomes less and less possible to actually provide security in the face of complexity. Despite a wealth of testing tools that claim to catch bugs, the complexity of software makes security flaws and errors nearly unavoidable and increasingly common.

The complexity explosion in software is exponential. The challenges of rising system complexity for software developers cannot be overstated. There is a movement to more complex systems, and the operating system is forced to take on a larger role in managing that complexity. We have passed a critical juncture where a new paradigm is required. You get to a certain size of the software where your odds of getting a really serious error are too high. We have to change the whole rules of engagement. In the 1970s, the average car had 100,000 lines of source code. Today it's more than a million lines, and it will be 100 million lines of code by 2010. The difference between a million lines of code and 100 million lines of code definitely changes your life.[ii]

McCabe Complexity Metrics

Certain characteristics of computer software make it more- or less- vulnerable. Complexity drives insecurity. *Cyclomatic complexity* is the most widely used member of a class of static software metrics. *Cyclomatic complexity* may be considered a broad measure of soundness and confidence for a program. Introduced by Thomas McCabe in 1976, it measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. *Cyclomatic complexity* is often referred to simply as program complexity, or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format[iii].

The higher the complexity the more likely there are bugs. The more bugs the more security flaws. A certain number of "complexity bugs" can be found through programmer vigilance. Get to know your code. Get to know how the pieces work and how they talk to each other. The more broad a view you have of the system being programmed, the more likely you will catch those pieces of the puzzle that don't quite fit together, or spot the place a method on some object is being called for some purpose it might not be fully suited.

There is an absolute need to examine code for security flaws from this position of knowledge. When considering security-related bugs, we have to ensure the system is security proof against someone who knows how it works, and is deliberately trying to break it, again an order of magnitude harder to protect against than a user who might only occasionally stumble across the "wrong way to do things.

The McCabe Complexity Metrics can be used to quantify security exposure and impact and can be used to unravel previously incomprehensible logic, design and sneak paths.

Cyclomatic Complexity

Cyclomatic complexity v(G) is a measure of the logical complexity of a module and the minimum effort necessary to qualify a module. Cyclomatic complexity is the number of linearly independent paths and, consequently, the minimum number of paths that one should (theoretically) test. It

- Quantifies the logical complexity
- Measures the minimum effort for testing
- Guides the testing process

- Is useful for finding sneak paths within the logic
- Aids in verifying the integrity of control flow
- Can be used to test the interactions between code constructs

Module Design Complexity

Module design complexity iv(G) of a module is a measure of the decision structure which controls the invocation of the module's immediate subordinate modules. It is a quantification of the testing effort of a module as it calls its subordinates.

The module design complexity is calculated as the cyclomatic complexity of the reduced graph. Reduction is completed by removing decisions and nodes that do not impact the calling control of the module over its subordinates. Design complexities exist because:

- Modules do not exist in isolation
- Modules call child modules
- Modules depend on services provided by other modules

This metric quantifies the interaction of modules with subordinates under security review. How much security testing is required to integrate this module into the rest of the system?

Module Global Data Complexity

Global data complexity gdv(G) quantifies the complexity of a module's structure as it relates to global and parameter data. Global data is data that can be accessed by multiple modules. This metric can show how dependent a module is on external data and is a measure of the testing effort with respect to global data. Global data complexity also measures the contribution of each module to the system's data coupling, which can pinpoint potential maintenance problems. This metric:

- Isolates the modules with highest external data coupling.
- Combines control flow and data analysis to give a more comprehensive view of software than either measure would give individually.

Module Specified Data Complexity

Specified data complexity sdv(G) quantifies the complexity of a module's structure as it relates to user-specified data. It is a measure of the testing effort with respect to specific data. A data dictionary is used to select a single data element, all elements with a specific data type, or a variety of other selection criteria. The specified data complexity then quantifies the interaction of that data set with each module's control structure. It indicates the data complexity of a module with respect to a specified set of data elements, and equals the number of basis paths that you need to run to test all uses of that specified set of data in a module. It allows users to customize complexity measurement for data-driven analyses

For example*, specified data complexity* can be used to analyze the complexity, context and testing effort of the four Standard Windows Socket Routines: Which modules are using recv() (TCP), recvfrom() (UDP), WSARecv() (TCP) and WSARecvFrom() (UDP).

Actual Complexity

Actual complexity ac of a module is defined as the number of linearly independent paths that have been executed during testing, or more formally as the rank of the set of paths that have been executed during testing. The structured testing criterion requires that the actual complexity equal the cyclomatic complexity after testing. The actual complexity is a property of both the module and the testing. For example, each new independent test increases the actual complexity. Actual Complexity should be used to measure code coverage on attack surface modules. Code coverage metrics such as actual complexity can be used to augment fuzz testing which is often called "White Box Fuzz Testing"[iv]

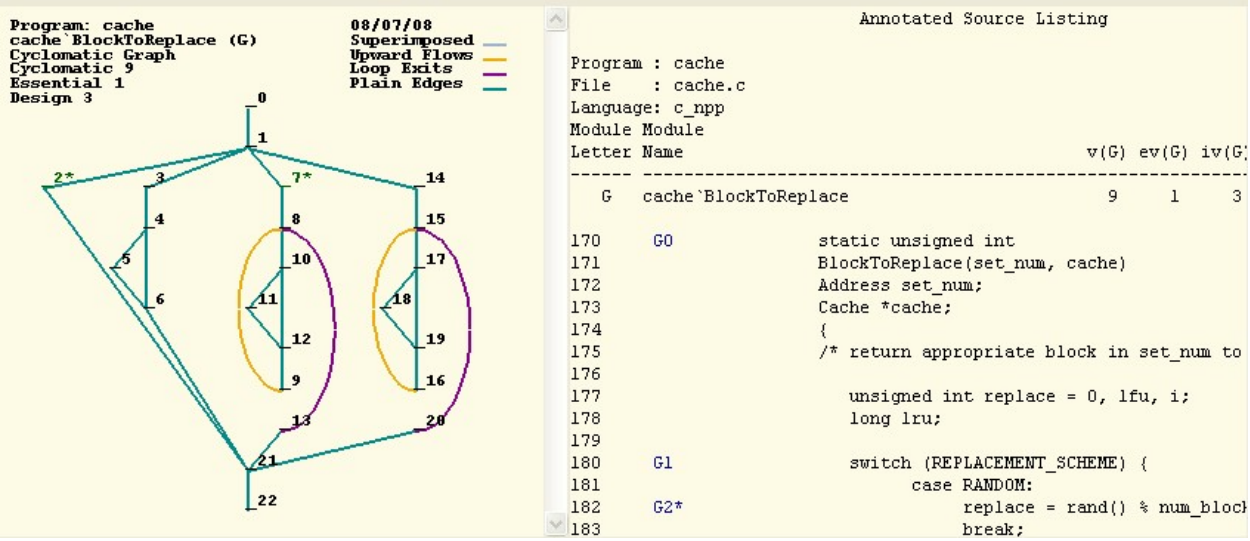
Design Complexity

Design complexity, S0, of a design G, is a measure of the decision structure which controls the invocation of modules within the design. It is a quantification of the testing effort of all calls in the design, starting with the top module, trickling down through subordinates and exiting through the top. It is a measure of the complexity of a design. It is used to identify the minimum yet effective integration tests. This metric can also help assessing sneak path subtrees and the impact on the rest of the system.

Measuring Control Flow Integrity

Given the ingenuity of would-be attackers and the wealth of current and undiscovered software vulnerabilities it is important that control flow integrity be verified to ensure strong guarantees against powerful adversaries. A Control-Flow Integrity security policy dictates that software execution must follow a path of a Control-Flow Graph determined ahead of time. The Control Flow Graph in question can be defined by source code analysis and/or execution profiling. A security policy is of limited value without an attack model[v].Security Analysis without a control and data flow diagram of logic and design is like doing security analysis of a house without schematics, such as a flooring plan or circuitry diagram. Only scanning for known exploits without verifying control flow integrity is comparable to a security expert explaining the obvious, such as windows are open and doors are unlocked, and being completely oblivious to the fact that there is a trap door in your basement. Those insecure doors and windows are only the low hanging fruit.

Module Flowgraphs can be used to understand the algorithms and their interactions. Visualizing logic using flowgraphs can be used for code comprehension, test derivation, identification of module interactions and for sneak path analysis.



Analysis of the module control flow diagram identifies ways that sources could combine with targets to cause problems. A *Cyclomatic Complexity* measurement of ten (10) means that 10 Minimum Tests will:

- Cover All the Code
- Test Decision Logic
- Test the Interaction between Code Constructs

Neither statement nor branch testing is adequate to detect security vulnerabilities and verify control flow integrity. Many exploits can hide in obscure paths and subtrees within a seemingly innocent appearing codebase. This paper shows how Cyclomatic Path Analysis, on the other hand, detects more security vulnerabilities and errors in your critical applications.

Structured Testing

Cyclomatic Path Analysis, also known as Basis Path Testing or as Structured Testing[vi], is the primary code-based testing strategy recommended by McCabe Software and Supported by McCabe IQ. The fundamental idea behind basis path testing is that decision outcomes within a software function should be tested independently.

Testing is Proportional to Complexity

A major benefit of basis path testing is that the number of tests required is equal to the cyclomatic complexity metric. Since complexity is correlated with errors, this means that testing effort is concentrated on error-prone software. Additionally, since the minimum required number of tests is known in advance, the software security testing process can be planned and monitored in greater detail than with most other testing strategies. Statement coverage, branch coverage, and even esoteric testing strategies such as variable definition/usage association coverage do not have this property-for only arbitrarily complex and error-prone code, it might be possible to satisfy those criteria with one or two tests, or it might take thousands.

Testing Detects Interaction Errors

Unlike other common testing strategies, basis path testing does not allow interactions between decision outcomes during testing to hide errors. The most common code based testing strategies are code coverage, statement coverage, and branch coverage. Code coverage, in which the number of executable

lines that were encountered during testing is compared to the total number of executable lines, can be dismissed immediately as a test strategy because it measures the code format rather than the code. In most programming languages, we could format any program as a single line and satisfy code coverage with one test. Statement and branch testing are stronger, but have the weakness that interactions between decision outcomes can mask errors during testing. By requiring each decision outcomes to be exercised independently during testing, basis path testing exposes the errors.

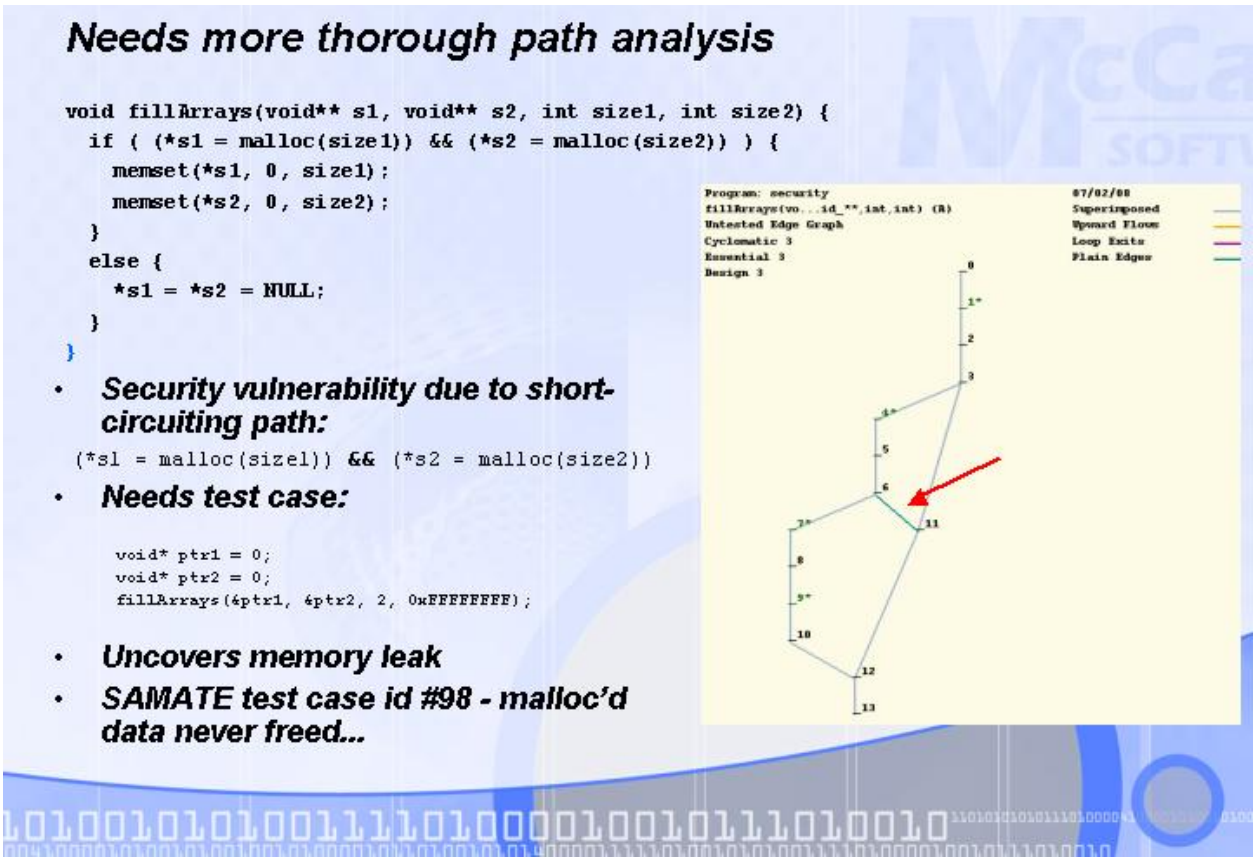
The following examples illustrate how basis-path analysis can facilitate detection of security vulnerabilities.

Example 1 – Short Circuiting Operations

One of the well-known vulnerable programming practices is writing conditional statements that incur side effects as part of the condition checking. In this example, the function is intended to allocate memory for two pointers and set the pointers to the newly allocated area. If memory allocation succeeds, the pointers are assigned; otherwise, they are set to NULL.

On cursory inspection, the implementation may appear valid. The code seems valid for a decision with two possible outcomes. However, the if statement is comprised of 2 conditions to be checked, and test coverage must account for scenarios where only one of the 2 conditions may evaluate to true. The test plans must be expanded to exercise the devious path introduced by the multi-condition if statement. A complete set of test cases will uncover a security vulnerability due to a memory leak.

Specifically, if the first allocation succeeds, but the second one fails, the code will execute the else side of the if statement and set both pointers to NULL. However, since the first allocation succeeded, the memory from the first allocation should be freed. In this example, no such clean-up is done, and the memory set aside for the first allocation is leaked. The nature of this vulnerability is described in SAMATE test case id #98 (malloc'd data never freed…) and also in CWE-401 – Failure to release memory before removing last reference.



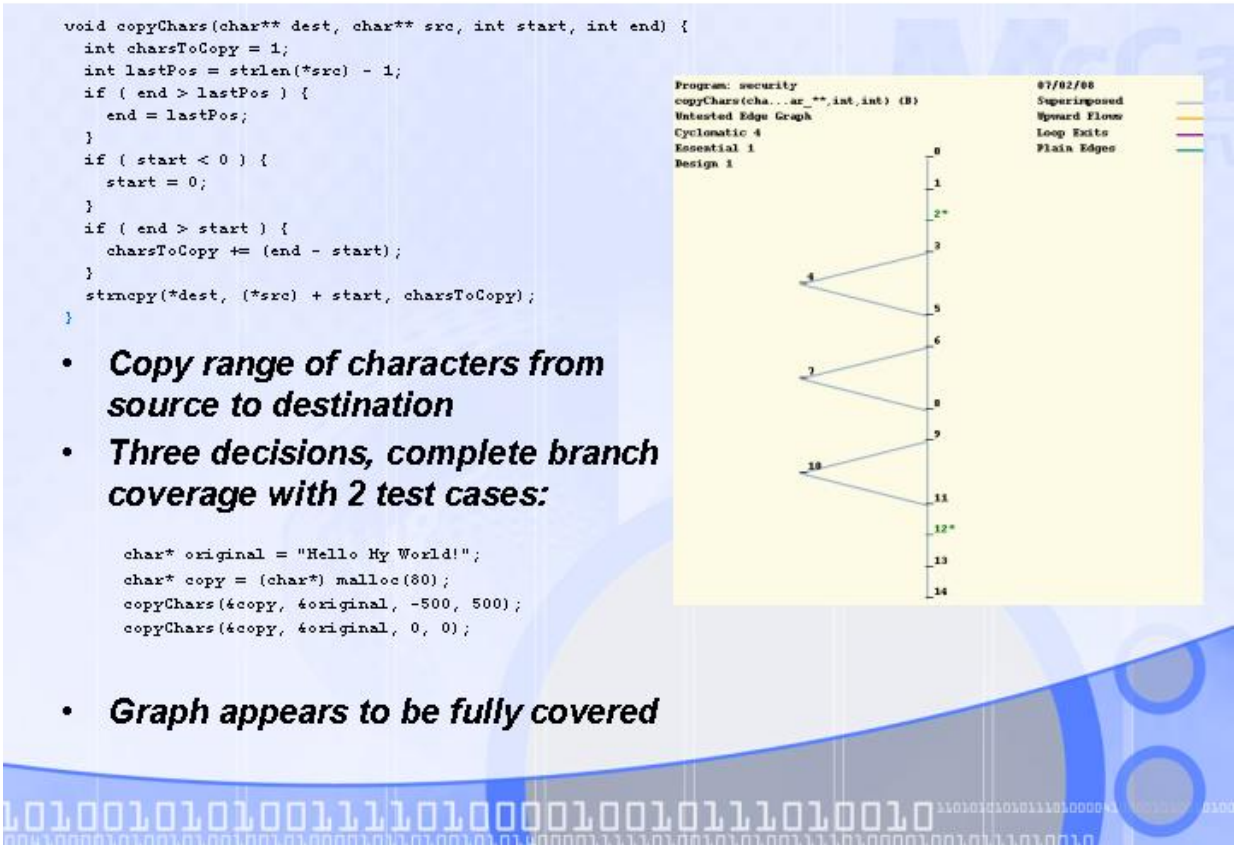
Example 2 – Sequential if Statements

This example uses a poorly coded array copying function to illustrate the shortcomings of branch coverage. To review, the branch coverage goal is to exercise all outcomes of a decision. The weakness of this approach is that it does not account for the effect that a given decision may have on subsequent decisions. This example shows a function that copies a range of characters from the source array to the destination. There are 3 sequential checks that occur prior to copying the range of characters: first to validate that the end position is within bounds, then to check that the start position is within bounds, and finally, to check that end position is after the start. With a coding structure like this, full branch coverage can be obtained with only 2 test cases. Since the conditions are executed sequentially, one run through the function can exercise a branch from all 3 decisions. Simply construct test case data that will exercise the true side of all 3, and another test case that will exercise the false branches of all 3, and branch coverage is 100%.

The problem with this approach is that 2 test cases are not adequate to detect potential vulnerabilities. This example contains a defect that is realizable from a specific sequence of decision outcomes. Since branch coverage does not take this into account, the defect may remain undetected. This is where basis path coverage proves superior.

The control flowgraph for this function has a cyclomatic complexity of 4, meaning that 4 basis paths must be exercised. The 2 test cases to obtain 100% branch coverage exercised only 2 linearly independent paths. There are 2 other paths that need to be covered. A software tool that supports basis path testing can indicate the sequence of decision outcomes that need to be exercised to test the remaining basis paths. The example shows the parameters needed to pass to the function in

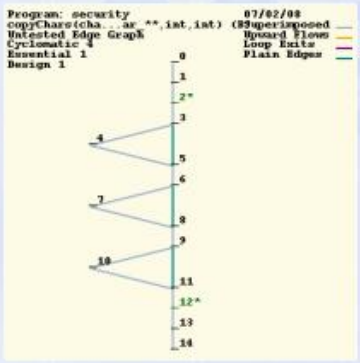
order to exercise these paths. The last test case of the example uncovers the security vulnerability described in SAMATE test case id #1492 (defective string manipulation), and also in CWE-125 – Out of bounds read/CWE-126 – Buffer overread.

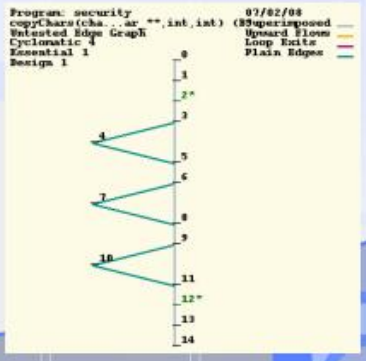


Test cases exercised only 2 paths

```
void copyChars(char** dest, char** src, int start, int end) {
  int charsToCopy = 1;
  int lastPos = strlen(*src) - 1;
  if ( end > lastPos ) {
    end = lastPos;
  }
  if ( start < 0 ) {
    start = 0;
  }
  if ( end > start ) {
    charsToCopy += (end - start);
  }
  strncpy(*dest, (*src) + start, charsToCopy);
}
```

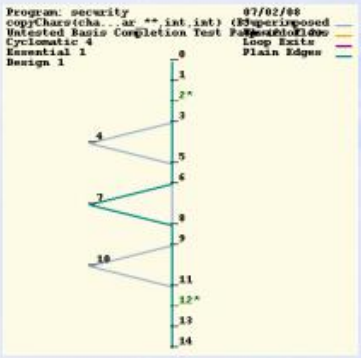
```
char* original = "Hello My World!";
char* copy = (char*) malloc(80);
copyChars(&copy, &original, -500, 500);
copyChars(&copy, &original, 0, 0);
```

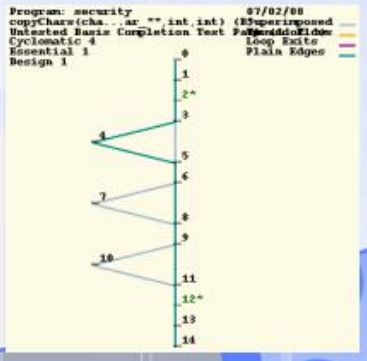




Need at least 2 more test cases

```
void copyChars(char** dest, char** src, int start, int end) {
  int charsToCopy = 1;
  int lastPos = strlen(*src) - 1;
  if ( end > lastPos ) {
    end = lastPos;
  }
  if ( start < 0 ) {
    start = 0;
  }
  if ( end > start ) {
    charsToCopy += (end - start);
  }
  strncpy(*dest, (*src) + start, charsToCopy);
}
```





- **Additional test cases to complete path coverage**

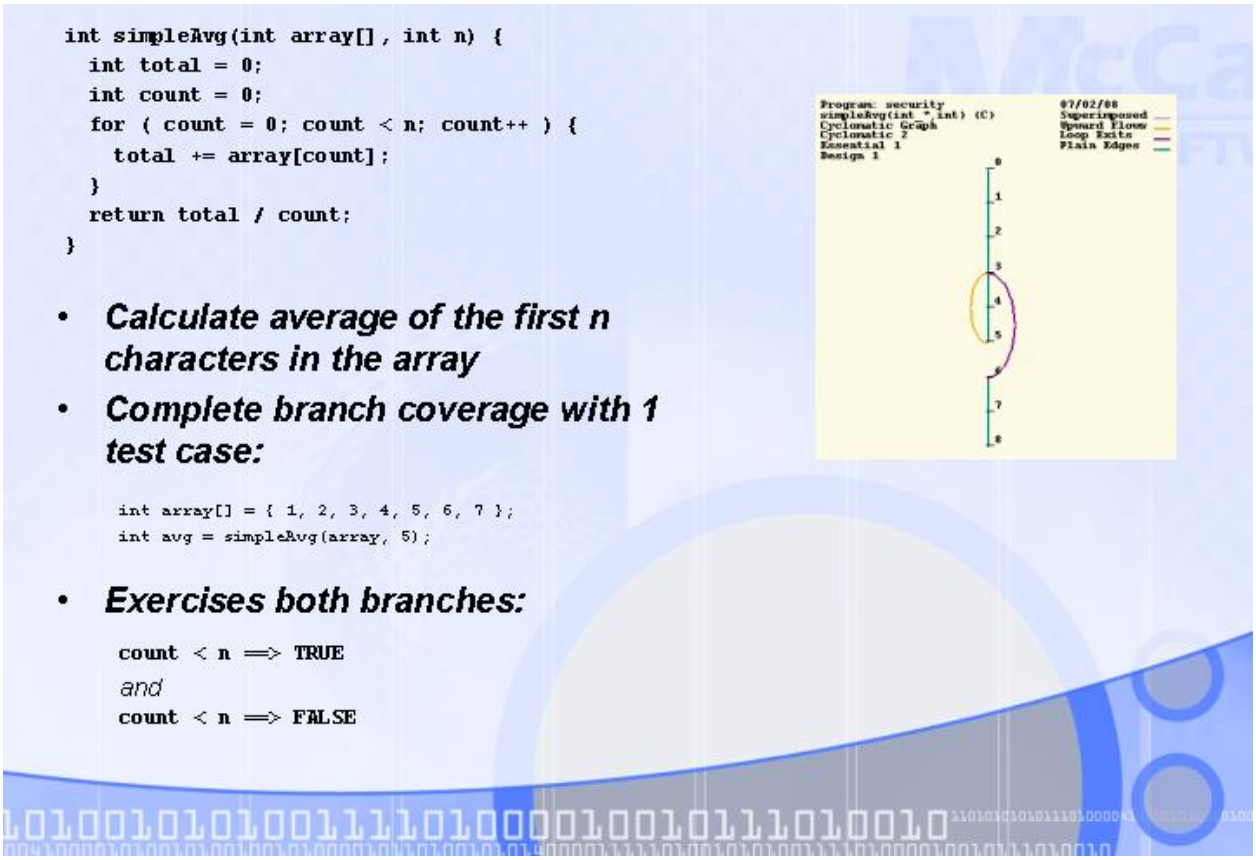
```
char* original = "Hello My World!";
char* copy = (char*) malloc(80);
copyChars(&copy, &original, -10, 0);
copyChars(&copy, &original, 1000, 100);
```

- **Last case exercises out-of-bounds access**
- **SAMATE test case id #1492 - defective string manipulation**

Example 3 – Looping Constructs

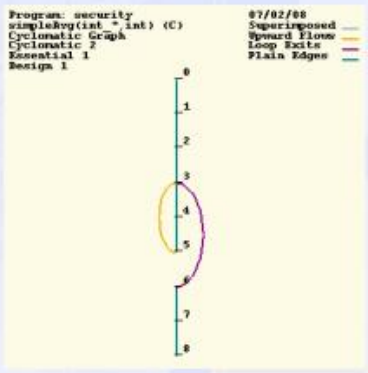
This is another example to illustrate where 100% branch coverage is not adequate to test for security vulnerabilities. The function is intended to calculate the average of the first n characters of an array, where n is passed in as a function argument. One of the weaknesses of using branch coverage for testing looping constructs is that a successful loop entry and exit exercises 100% of the branches.

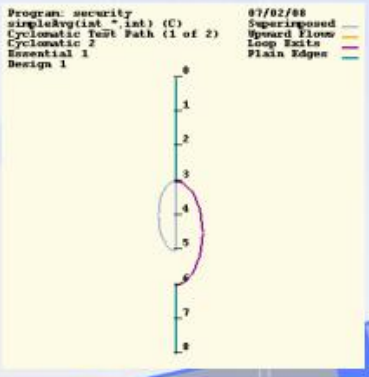
Consider this example. For simplicity, disregard any potential problems with array bounds, presuming the array will always be valid and the index will always be in range. If the function is invoked with an array of 10 values and asked to calculate the average of the first 5 values, it works properly. Furthermore, this case will also show 100% branch coverage. With this single test case, the condition count < n has evaluated to true 5 times, to repeat 5 times, and has also evaluated to false once, to exit the loop. Thus, all branch outcomes have been exercised. However, the discerning programmer will see that this function has a serious error in it; one that is also uncovered by basis path testing. This function has a cyclomatic complexity of 2, meaning that there are 2 linearly independent paths to be tested. Basis path testing requires one path that enters the loop and exits, and another path that does not enter the loop at all. If this second path is exercised, the code will incur a division by zero, SAMATE test case id #1525 (divide by zero), and also CWE-369 – Divide by zero.



Two cyclomatic paths

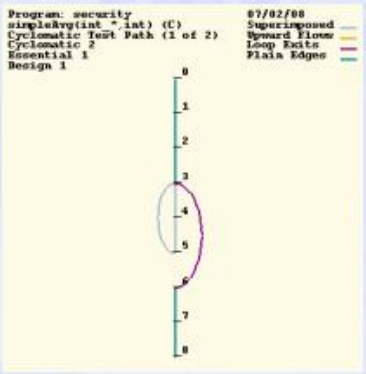
```
int simpleAvg(int array[], int n) {
  int total = 0;
  int count = 0;
  for ( count = 0; count < n; count++ ) {
    total += array[count];
  }
  return total / count;
}
```





Needs at least 1 more test case, one that doesn't go into the loop

```
int simpleAvg(int array[], int n) {
  int total = 0;
  int count = 0;
  for ( count = 0; count < n; count++ ) {
    total += array[count];
  }
  return total / count;
}
```



- *Additional test case to complete path coverage:*

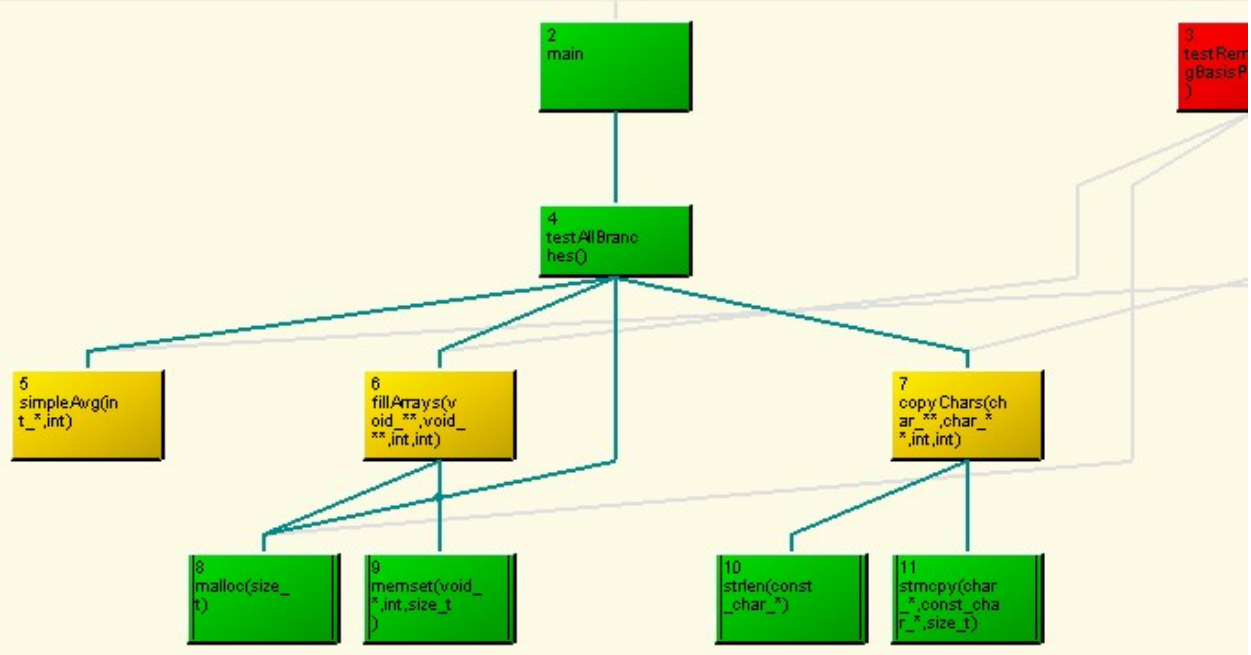
```
int array[] = { 1, 2, 3, 4, 5, 6, 7 };
int avg = simpleAvg(array, 0);
```

- *Uncovers security vulnerability - program crash*
- *SAMATE test case id #1525 - divide by zero*

Integration Path and Subtree Coverage Analysis

The integration-level Structured Testing strategy, based on design complexity and detailed within NIST Special Publication 500-235, requires independent testing of each decision outcome that affects the module-calling sequence and shares many of the benefits of basis testing. Call-pair coverage, a common integration testing measure based on exercising all caller/callee pairs, has the same weaknesses as branch coverage

Since all decision outcomes affect the calling sequence or subtree, integration-level Structured Testing is equivalent to basis path testing and therefore guaranteed to detect more errors.



Structural & Attack Surface Analysis

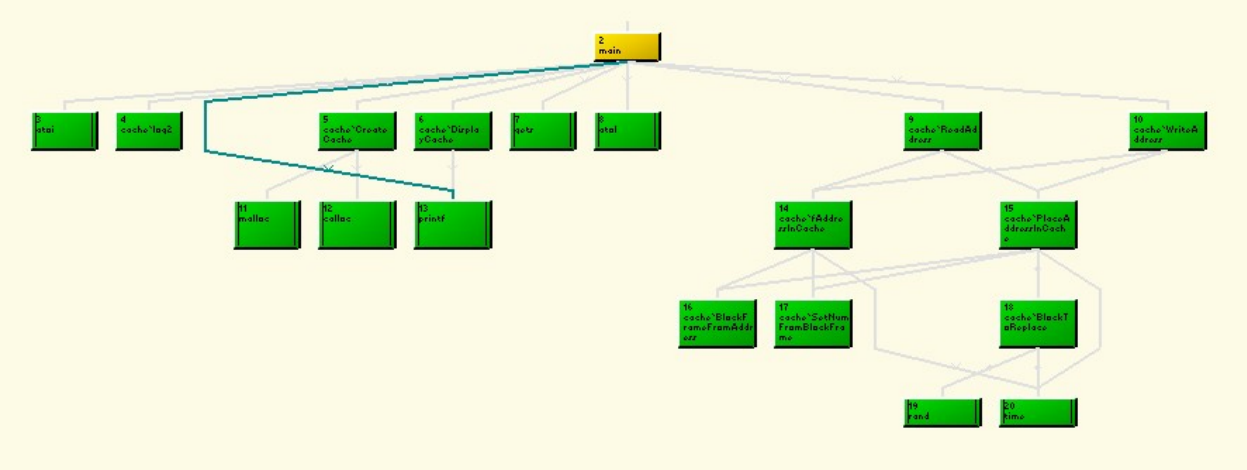
Analyzing a software system for security flaws can be a daunting task, and often begs the question: when is the analysis complete? Often a security analyst will answer this question by determining when they have run out of budget, time or have found bugs. These are not empirical pieces of evidence. One piece of evidence required is to understand how much of the software that is attackable was exercised.

Many experts point out that security requirements resemble those for any other computing task, with one seemingly insignificant difference ... whereas most requirements say "the system will do this," security requirements add the phrase "and nothing more." Not understanding code coverage limits the effectiveness of black-box software testing. If security teams don't exercise code in the application, they can't observe any bugs it might have. Code Coverage measurement should be done on attack surface modules to verify how much of the attackable surface was executed during testing.

Black Box Testing is positive testing. That is, the software security analyst is testing things in the specs or requirements that are already known. White Box Testing is negative testing in that you are testing things that may not be in your specs but in the implementation. Always remember code is what executes not requirements!

Code Coverage monitoring typical of a white box testing approach can verify how much of the source code that is attackable has been exercised using security testing. It can also illuminate how effective security tests actually are after completion. Code Coverage monitoring can give reassurance as to when the security analysis testing is complete and can also indicate what areas of source code penetration tests have hit. The McCabe actual complexity metric indicates how many of the cyclomatic paths were executed during security testing.

The McCabe Battlemap coverage diagram (below) indicates the code coverage of each module, by superimposing test execution information on the structure of the system. This helps you locate attack surface areas of your program that may require additional testing.

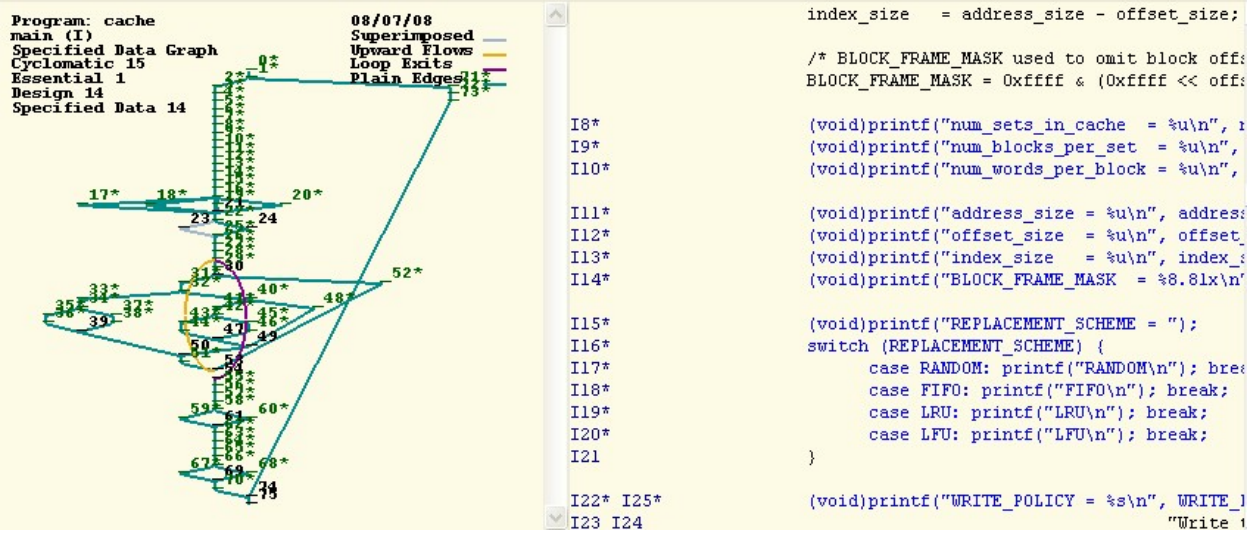


Use Basis Paths & Subtrees for Sneak Path Analysis

Using Cyclomatic basis path testing for software security analysis is analogous to using Sneak Path Analysis. The goal behind sneak path analysis, also called sneak circuit analysis (SCA), is to identify unexpected and unrecognized electrical paths or logic flows in electronics systems, called sneak circuits or paths, that under certain conditions produce undesired results or prevent systems from operating as intended. These paths come about in various ways.

Designers do not always have a complete view of the relationship between functions and components in complex systems, particularly those with many interfaces. System changes may falsely appear to be minor or of only local significance, and users may employ improper operating procedures. Historically, users discovered sneak paths when they observed an unintended effect during system operation. Sneak paths may lie in software, or user actions, or some combination thereof. They are latent conditions, inadvertently and unknowingly designed or engineered into the system, that do not cause system failure until triggered by unusual circumstances.

In Sneak Path Analysis, a topology diagram is built of the various components and interconnections. The system topology diagram is then analyzed to identify ways that sources could combine with targets to cause problems. This is accomplished by examining flow paths through the diagram. An example of the McCabe Cyclomatic Complexity Metric being used for Sneak Path Analysis can be found in *Space Product Assurance Guide: Sneak Analysis Methods and Procedures (ECSS-Q-40-04a) sponsored by the European Cooperation for Space Standardization (ECSS*)[vii].



A typical hacker may use features in software which are not obvious security holes, in order to probe the system for useful information to form an attack. Source code complexity can mask potential security weaknesses, or seemingly innocuous software configuration changes may open up security gaps. An example would be a change to the permissions on a system directory.

The implications of such changes may not be appreciated by a system administrator or software engineer, who may not be able to see the whole picture due to the complexity in any typical operating system. A reduction in this complexity may improve security.

Attack Surface Design Complexity

Attack Surface Design Complexity, is a measure of the decision structure which controls the invocation of modules within the design of an attack surface. It is a quantification of the testing effort of all calls in this design, starting with the top module, trickling down through subordinates and exiting through the top.

For example, your attack surface may be in software receiving network packets. What functions are responsible for receiving packets on the network, and how is the resulting data is passed along the internal routines of the software?[viii] A suggested process for analyzing control flow of attack surfaces is as follows:

Analyzing an Attack Surface's Control Flow Integrity

Step 1: Identify all modules with vulnerable attack surface

Step 2: Calculate McCabe Design Complexity, Integration Complexity

```
SUBTREES AND ASSOCIATED
INTEGRATION TEST CONDITIONS FOR PROGRAM cache
ROOT MODULE OF PROGRAM: main


SUBTREE #1:
   main > [printf] < main > [printf] < main > [printf] < main

END-TO-END TEST CONDITION LIST FOR SUBTREE #1:
 main 252(1): argc>5 ==> FALSE
```

Step 3: Analyze visual and textual design invocation subtrees

Step 4: Calculate and analyze all cyclomatic, module design, and global data complexity metrics and complexity algorithm graphs for impact analysis, risk, test execution and sneak paths

Step 5: Measure code coverage at point where the packet is received and is traversing memory into the program's logic

Error handling routines in software programs are typically sneak paths. Since error handling routines contribute to control flow, use flow graphs to decipher the programming logic and produce test conditions which will when executed test their logic. The most neglected code paths during the testing process are error handling routines. Error handling may include exception handling, error recovery, and fault tolerance routines.

Functionality tests are normally geared towards validating requirements, which generally do not describe negative (or error) scenarios. Validating the error handling behavior of the system is critical during security testing.

Measuring and Monitoring Code Slices

Measuring and monitoring code execution slices can help uncover your program's internal architecture. By compiling and running an instrumented version of your program, then importing the resulting trace file, you can visualize which parts of your program's code are associated with specific functionality. The slice concept is important to Software Security from several standpoints.

- Visualizing the software
- Tracing tainted and untainted data through the system
- Decomposing a complex system
- Finding Sneak Paths

To gain understanding of code coverage before a fuzzing run, it is important to first pass the application a piece of data that is correctly formed. By sending the right packet and measuring the execution slice, the common path that a normal packet takes through the application logic is determined.

Some Intrusion detection systems based on anomaly detection use a set of training data to create a database of valid and legitimate execution patterns that are constantly compared to real execution patterns on a deployed system. This approach assumes that the attack pattern substantially differs from the legitimate pattern.

Summary

Many exploits are about interactions: interactions between code statements, interactions between data and control flow, interactions between modules, interactions between a codebase and library routines. Being cognizant of paths and subtrees within source code is crucial for testing to verify control flow integrity and uncovering security flaws hiding along obscure paths or subtree structures within a codebase

Although rudimentary, the previous examples illustrate that security vulnerabilities are often a consequence of multiple factors. Attackers can disrupt program operation by exercising a specific sequence of interdependent decisions that result in unforeseen behavior. As part of secure software development, these paths must be identified and exercised, to ensure that program behavior is correct and expected. Techniques for complete line and branch coverage leave too many gaps. Cyclomatic complexity and basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique.

There are many benefits of basis path testing beyond the underlying "test all decisions independently" description. The key properties of basis path testing, which are not shared by other common testing strategies, are that testing is proportional to complexity, testing effort is concentrated on the most error-prone software, security testing progress can be monitored with precision, and errors based on interactions between decision outcomes are detected.

Use software complexity metrics, measure control flow integrity and do sneak path analysis for better security analysis. Basis cyclomatic test path and subtree analysis lends itself well in the area of software Sneak Path Analysis. White box security testing following the methodology as presented in NIST Special Pub. 500-235 Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric is a sound way to verify control flow integrity.

---

[i] Testimony of Bruce Schneier, Founder and CTO Counterpane Internet Security, Inc, Subcommittee on Cybersecurity, Science & Research and Development Committee of Homeland Security, U.S. House of Representatives , Jun 25, 2003

[ii] John Keller, "Developers of Real-Time Embedded Software Take Aim at Code Complexity" Military & Aerospace Electronics - April, 2007

[iii] Pete Lindstrom, "Software Security Labels: Should we throw in the towel?" Spire Security Viewpoint, October 24, 2005

[iv] Patrice Godefroid; Michael Y. Levin; David Molnar, "Automated Whitebox Fuzz Testing" Microsoft Research

[v] Martín Abadi; Mihai Budiu; Ulfar Erlingsson; Jay Ligatti, "Control-Flow Integrity" Microsoft Research, February 2005

[vi] Thomas McCabe & Arthur Watson, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric" NIST Special Publication 500-235, 1996

[vii] Space Product Assurance Guide: Sneak Analysis Methods and Procedures (ECSS-Q-40-04a) sponsored by the European Cooperation for Space Standardization (ECSS)

[viii] Justin Seitz, "Analyzing the Attack Surface Code Coverage" SANS Institute Information Security Reading Room, 2007