
# Using Cyclomatic Path Analysis to Detect Security Vulnerabilities

## About this Paper

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, 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.

### Methodology

**NIST Special Publication 500-235: *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, August 1996**

Arthur H. Watson
Thomas J. McCabe
Prepared under NIST Contract
43NANB517266

Dolores R. Wallace,
Editor Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

## 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.

## Examples

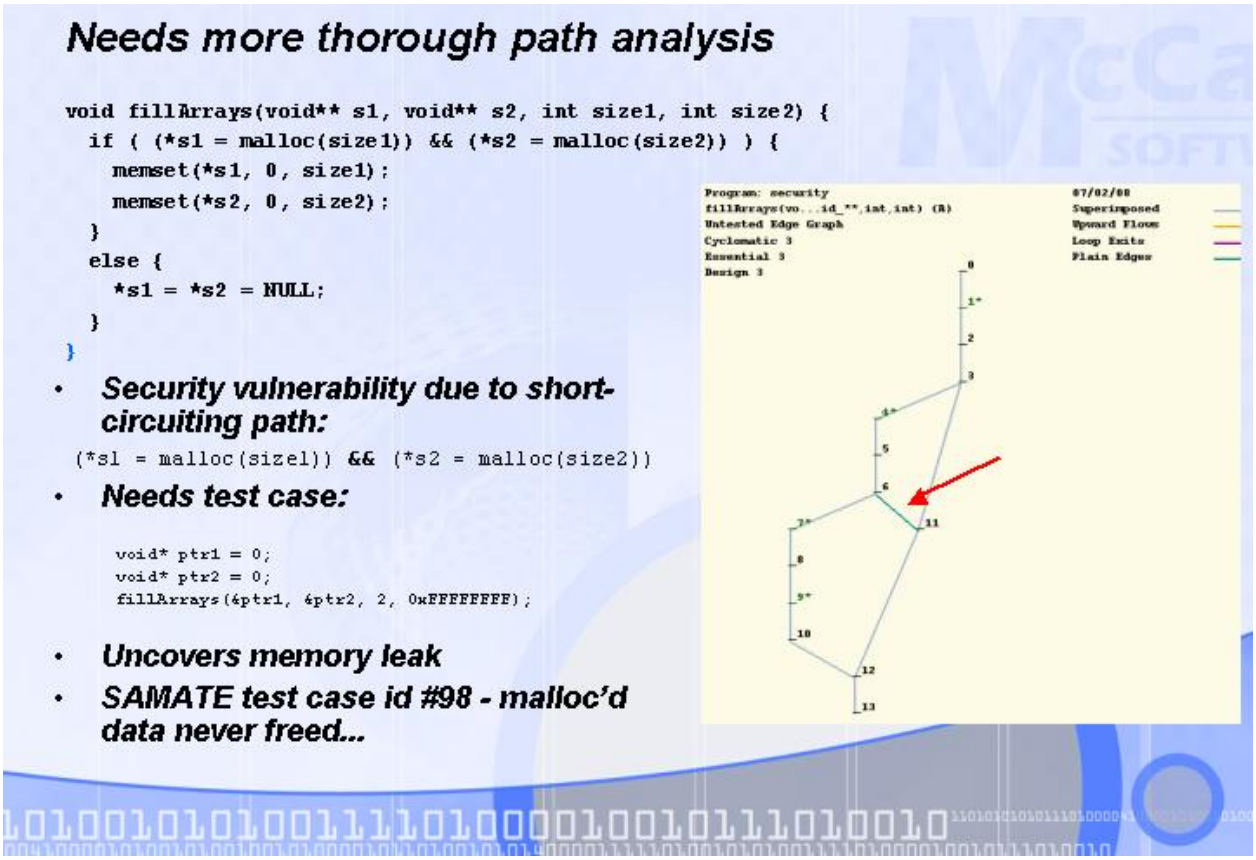
The following examples illustrate how basis-path analysis can facilitate detection of security vulnerabilities.

### 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.

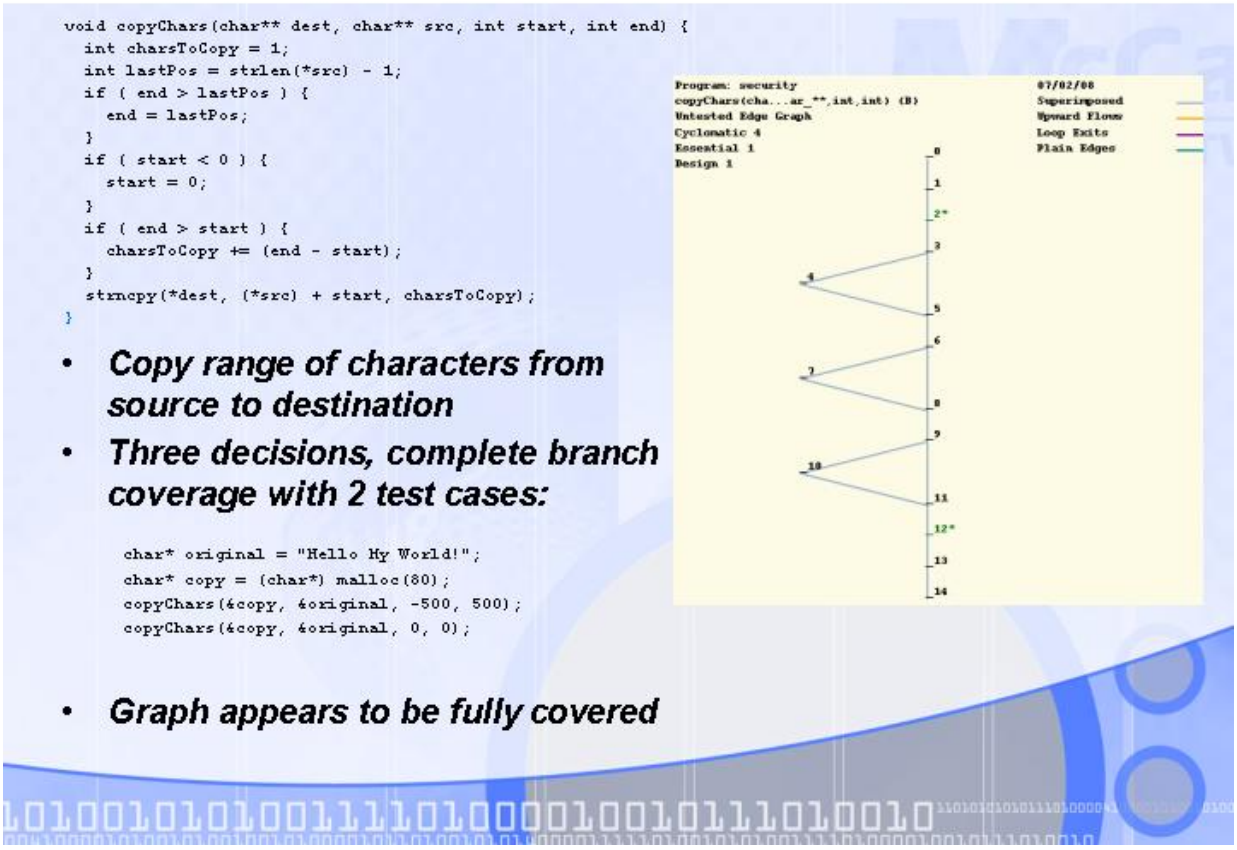


## Example 2 – Sequential if Statements

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.

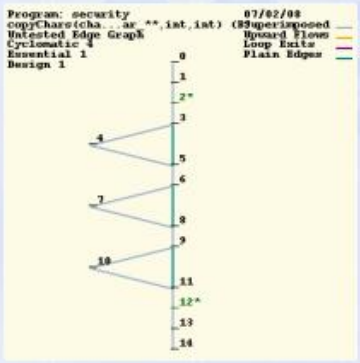
The control flowgraph for this function has a cyclomatic complexity of 4, meaning that 4 basis paths must be exercised. The 2 test cases to obtain 100% branch coverage exercised only 2 linearly independent paths. There are 2 other paths that need to be covered. A software tool that supports basis path 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.

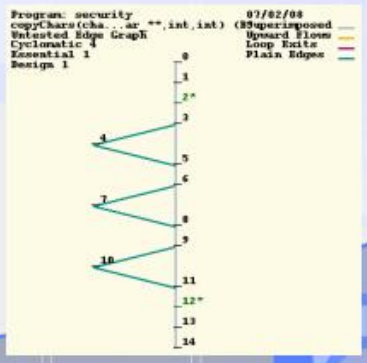


## Test cases exercised only 2 paths

```
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);
}
```

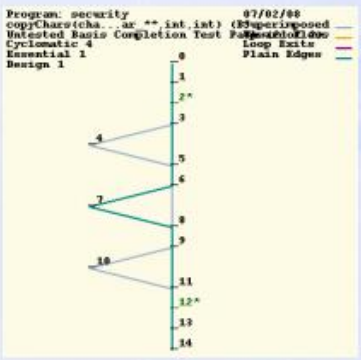
```
char* original = "Hello My World!";
char* copy = (char*) malloc(80);
copyChars(&copy, &original, -500, 500);
copyChars(&copy, &original, 0, 0);
```

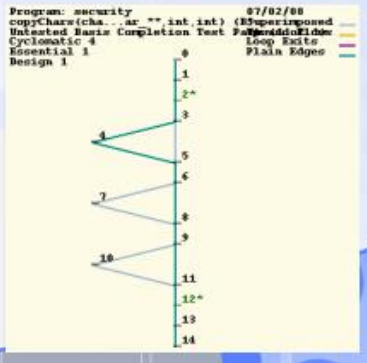



## Need at least 2 more test cases

```
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***

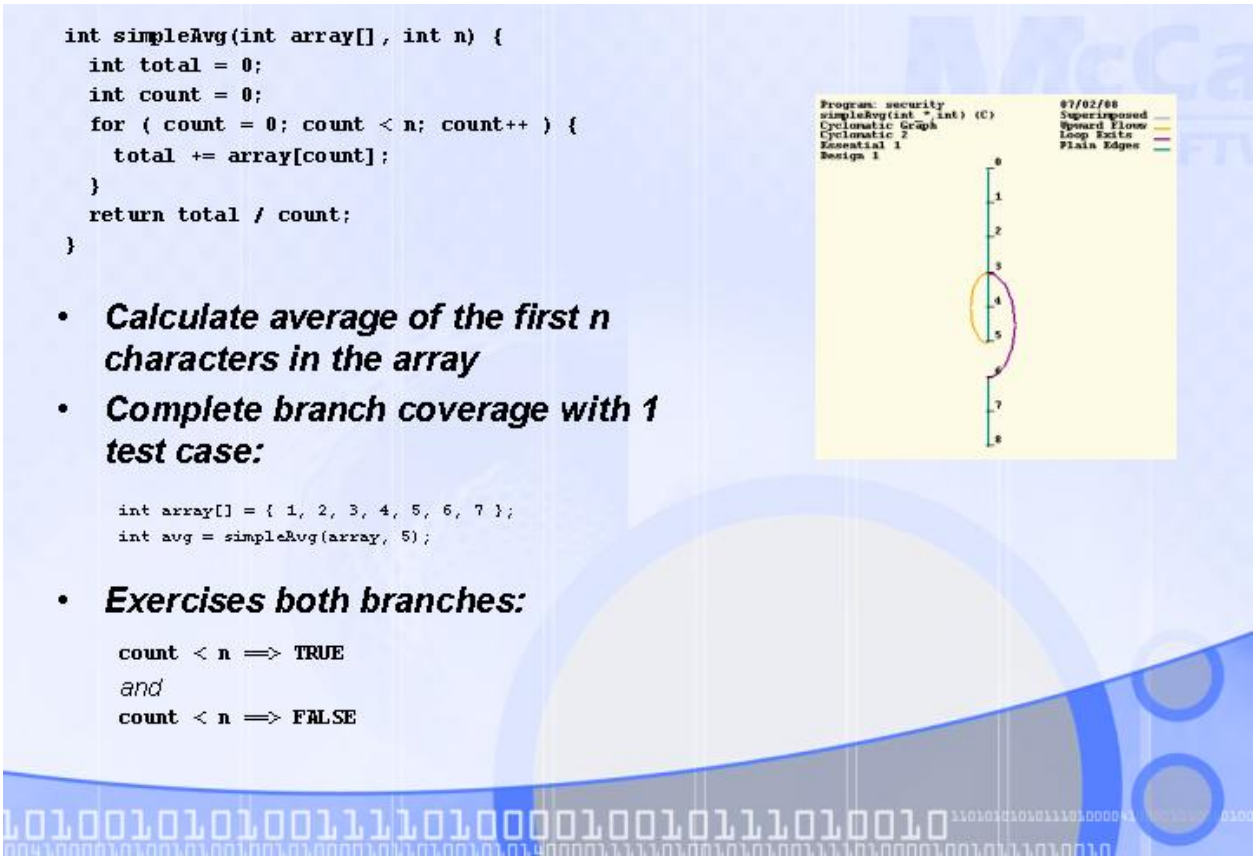
```
char* original = "Hello My World!";
char* copy = (char*) malloc(80);
copyChars(&copy, &original, -10, 0);
copyChars(&copy, &original, 1000, 100);
```

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

## Example 3 – Looping Constructs

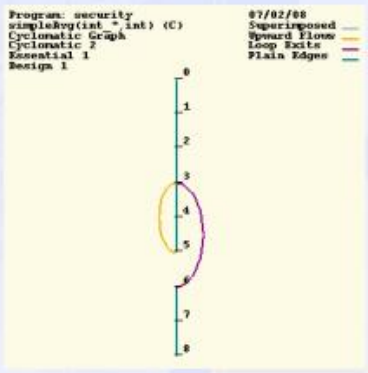
This is another example to illustrate where 100% branch coverage is not adequate to test for security vulnerabilities. The function is intended to calculate the average of the first n characters of an array, where n is passed in as a function argument. One of the weaknesses of using branch coverage for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

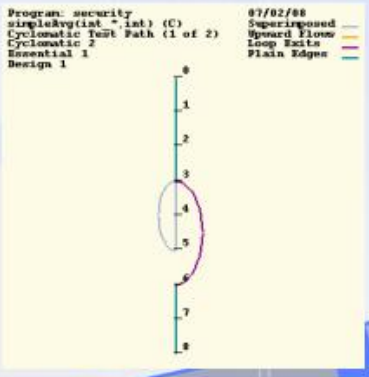
Consider this example. For simplicity, disregard any potential problems with array bounds, presuming the array will always be valid and the index will always be in range. If the function is invoked with an array of 10 values and asked to calculate the average of the first 5 values, it works properly. 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

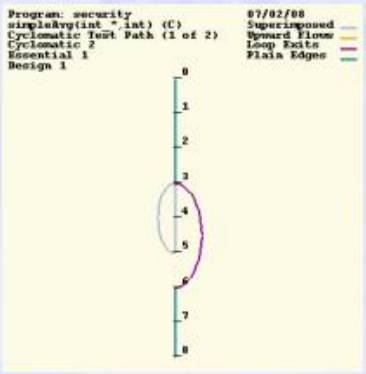
```
int simpleAvg(int array[], int n) {
  int total = 0;
  int count = 0;
  for ( count = 0; count < n; count++ ) {
    total += array[count];
  }
  return total / count;
}
```




## 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:*

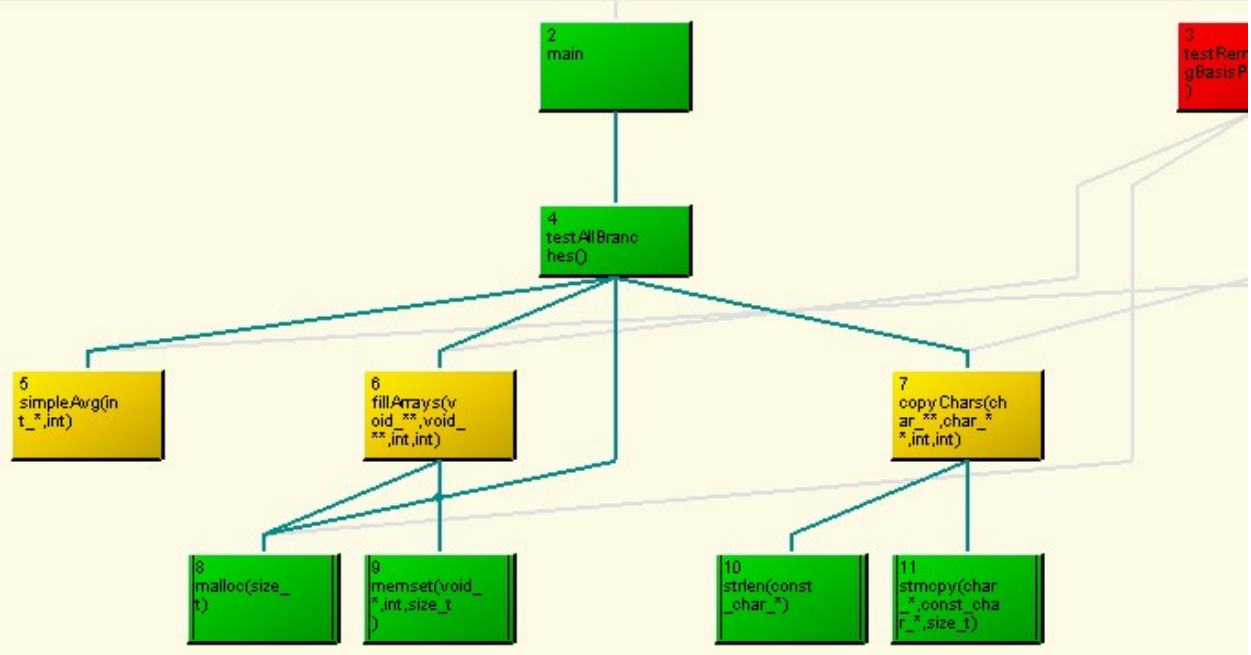
```
int array[] = { 1, 2, 3, 4, 5, 6, 7 };
int avg = simpleAvg(array, 0);
```

- *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.



## Conclusion

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.

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, and interactions between code and attack surface modules. Being cognizant of paths and subtrees within code is crucial for testing to verify control flow integrity and uncovering security flaws hiding along obscure paths or subtree structures within a codebase.