A Plethora of Paths

Eric Larson
Seattle University
elarson@seattleu.edu

Abstract

A common static software bug detection technique is to use path simulation. Each execution path is simulated using symbolic variables to determine if any software errors could occur. The scalability of this and other path-based approaches is dependent on the number of paths in the program. This paper explores the number of paths in 15 different programs. Often, there are one or two functions that contribute a large percentage of the paths within a program. A unique aspect in this study is that slicing was used in different ways to determine its effect on the path count. In particular, slicing was applied to each interesting statement individually to determine if that statement could be analyzed without suffering from path explosion. Results show that slicing is only effective if it can completely slices away a function that suffers from path explosion. While most programs had several statements that resulted in short path counts, slicing was not adequate in eliminating path explosion occurrences. Looking into the tasks that suffer from path explosion, we find that functions that process input, produce stylized output, or parse strings or code often have significantly more paths than other functions.

1. Introduction

Static analysis is an important tool in software verification and testing. It permits users to find bugs in software or prove that operations are safe without the need for extensive testing. Unfortunately, static analysis techniques are generally imprecise due to scalability and performance concerns. This can result, depending how the analyses are applied, in missed bugs or false bug reports. In general, there is a tradeoff in precision and performance. The more precise an analysis is, the longer it takes to run.

A common technique used to detect bugs is path simulation [3, 21]. Each execution path is simulated, with program variables being representing symbolically. At interesting points in the program, the symbolic variables are checked against constraints. Violations result in a bug report.

A major problem with path-based defect detection systems is path explosion - having too many paths to analyze in a reasonable amount of time. One technique to address path explosion is to use program slicing [11, 20] to remove code not relevant to the property being verified.

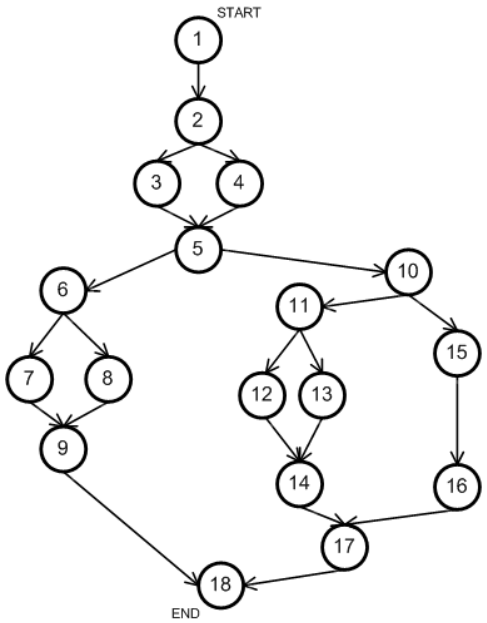

Figure 1. Sample Control Flow Graph.

Consider the sample control flow graph in Figure 1. The nodes in the graph refer to basic blocks that contain one or more statements. A directed edge is drawn from basic block *x* to basic block *y* if the code in basic block *y* can be immediately executed after basic block *x*. An execution path is a sequence of blocks from the beginning (block 1 in Figure 1) to the end (block 18).

In Figure 1, there are ten different paths. To compute this amount, note that all paths must go through node 5. There are two paths to get from the start to node 5: either traversing through node 3 or through node 4. There are five paths to get from node 5 to the end: two paths if traversing left from node 5 to node 6 and three paths if traversing right from node 5 to node 10. The total number of paths is obtained by multiplying the number of paths to get to node 5 by the number of paths to get to the end from node 5: 2 x 5 = 10.

This assumes the choice of paths to get from the start to node 5 is independent from the choice of paths to get from node 5 to the end. This is not always the case - the number of legal execution paths is often less than the number of theoretical paths derived from the control flow graph without further knowledge of the code.

To show how program slicing can reduce the number of paths, assume the statement of interest is in basic block 8 of Figure 1. It is not necessary to analyze the paths that traverse through basic blocks 10-17, nor is it necessary to traverse through block 7. This leaves only two execution paths: 1-2-3-5-6-8-9-18 and 1-2-4-5-6-8-9-18. The only difference between the two paths is the former traverses through basic block 3 while the latter traverses through basic block 4. Program slicing could determine that neither basic block 3 or 4 contains a statement that the statement of interest is dependent on. In this situation, the control flow graph can be further collapsed resulting in only one path.

This paper explores the number of execution paths in a variety of different programs written in the C programming language. In particular, we analyze where does path explosion come from? What types of tasks result in a large number of paths? What accounts for the difference in path counts between different programs?

In addition, this paper outlines results of path count experiments when program slicing is applied. For instance, to what extent does the path count decrease when program slicing is applied? Does program slicing eliminate or reduce path explosion? What effect does choosing different slicing criteria on the path count?

This paper makes the following contributions:

- A detailed analysis of paths in a program. We find that the most of the paths in a program are contained in a very small percentage of the functions (often just one).
- A quantitative analysis on the effects of program slicing on the number of paths in a program. We find the program slicing can greatly reduce the number of paths but path explosion still exists in the worst-case.
- A thorough qualitative analysis that looks at the types of tasks that lead to path explosion. We also dig deeper to better understand the effectiveness (or lack thereof) of program slicing.

The remainder of the paper is organized as follows. The next section describes the analysis framework used to carry out this study. Quantitative results are given in Section 3 while Section 4 examines our qualitative results. An overview of related work appears in Section 5. Section 6 concludes and presents directions for future work.

2. Analysis Framework

Our analysis uses a modified version of SUDS [13], a static analysis and dynamic instrumentation infrastructure that operates on the whole program. The dynamic instrumentation portion of SUDS was not used. The main flow for assessing the work necessary to verify an individual statement consists of the following steps:

1. Performing traditional compiler analyses such as data-flow and pointer analysis.
2. Slicing the program with respect to the slicing criterion and removing irrelevant code.
3. Counting paths using the code in the slice.

2.1 Traditional Compiler Analyses

This section describes some of the key implementation decisions for the initial phases:

Simplification: After parsing, each statement is converted into a simple statement in a similar fashion to CIL [16]. In the simple format, expression statements are converted such that each operator has at most two operands. Side effects from increment and short-circuited operators are removed.

Dynamic memory modeling: Dynamic memory is modeled by call-site. For each static call to `malloc`, a dummy variable is created with the return value pointing to the dummy variable. System functions, or any functions where source code does not exist, that return pointers are also modeled using a dummy variable for each call.

Control flow graph: The control flow graph is created intraprocedurally. Basic blocks that are not reachable from the starting point are discarded.

Call graph: A complete call graph is created including calls made by function pointers. The call graph and pointer analysis phase are performed together until both algorithms converge. Functions are that not reachable from main are removed from consideration and not analyzed in further stages. This omits signal handlers and functions passed into system functions from analysis.

Pointer analysis: The flow-sensitive interprocedural pointer analysis developed by Hind *et. al.* [10] is used to compute the set of variables that a pointer can point to. Structs, unions, arrays, and dummy variables are treated as single variables. Any updates to such variables are done in a weak fashion without killing prior pointer relationships.

Data-flow analysis: Data-flow analysis is used to determine the set of definitions generated, killed, and used by each statement. The analysis is done intraprocedurally using a standard data-flow algorithm that iterates until steady state. Basic but adequate interprocedural analysis is provided (see [13] for details).

2.2 Program Slicing

Program slicing [11, 20] removes instructions not relevant to the operation(s) being analyzed. A backwards slicing algorithm is employed starting with the slicing criteria.

The slicing criteria is often an individual statement that is of interest to the particular analysis task on hand. For this paper, we choose array accesses and pointer dereferences to serve as our slicing criteria. The choice of focusing on arrays access and pointer dereference errors was made because they are difficult to analyze thoroughly using a reasonably-sized finite state automaton (FSA). Array references and pointer arithmetic require integers to be modeled. For effective results, some level of constraint or symbolic analysis is necessary. For brevity, we consider a statement that contains an array access or pointer dereference to be dangerous.

In addition to having a single dangerous statement as a slicing criterion, we run experiments where the slicing criterion consists of all dangerous statements within a given function. For comparison purposes, we also compute the "worst-case" benefit to program slicing by using all dangerous statements in the program as the slicing criterion.

At the beginning of the slicing algorithm, only the uses that directly relate to memory checking are initially included in the slice. For instance, consider the statement: `x = a[i];` where `a` is a locally declared array of five elements. The only uses that matter to checking the array bounds are the uses that pertain to `i`; the contents of array `a` do not matter.

Since the slice is only used for analysis and the results of previous analyses are left in tact, no attempt is made to make the slice executable. The slice also does not include indirect uses from control statements since many static software bug detection systems ignore or abstract away the condition and simulate both directions of a control statement.

For function calls, the following algorithm is employed: If a function is added to the slice from reaching it by traversing upward from a statement in the slicing criteria, then slice information is propagated to all callers for this function. If a function cannot be reached from an upward traversal but is called from a function in the slice, then the function is added to the slice (provided at least one statement in the function is added to the slice). But other functions that call this particular function are not added to the slice. This prevents the slice from propagating to a completely irrelevant section of code. In either case, all information is propagated in a context-insensitive manner by combining the slicing information from multiple call sites instead of analyzing the callee function separately for each call site.

Once the slice has been computed, the control flow graph is reduced by removing all functions, basic blocks, and statements that are not in the slice.

2.3 Path Counting

Paths are counted using the reduced control flow graph after slicing. Each function is divided into a series of control constructs (if, while, for, do-while, and switch). For each control construct, both the total number of paths that reach the end of the construct and the number of paths that reach a return statement are computed. Starting with the beginning of the function the number of paths in each control construct is multiplied together. The number of early termination paths is kept separately and added to the final count.

There were three key design decisions when implementing the path counting algorithm. First, the path count is computed intraprocedurally. The total number of paths for a program is the sum of the paths for all functions. This approximation assumes a path simulation implementation similar to that of PREfix [3] where "leaf" functions on the call graph are analyzed first. Summaries capturing the behavior of the function are used by its caller functions during analysis. However, the approximation breaks down since function summaries may be more complex to analyze than other statements, possibly implicitly producing complex control flow.

The second design decision concerns loops since they can lead to an infinite number of paths. In our baseline configuration, we introduce two paths for each loop: one for not taking the loop and one for taking the loop once (the body of the loop can introduce additional paths). This approximation is reasonable because many static bug detection systems use some form of fixed-point analysis making it unnecessary to have an infinite number of paths. To improve the approximation, it is desirable to take the contents of the loop into account when making the approximation. This is left as future work. Examples of loop optimization techniques are presented in the related work section (Section 5).

Lastly, goto statements can introduce loops and unstructured control flow. To simplify the implementation but at a cost of imprecision, we consider goto statements to end a path just like return statements. In the programs used, there are very few gotos. In most cases, they are used to jump to finishing code at the end of a function or to some error processing code. These functions are often complex with high path counts anyway.

No attempt is made to remove illegal or invalid paths. This choice is intentional since work is needed in static bug detection systems to remove these illegal paths. In many systems, illegal paths are removed automatically but with significant performance loss. In other systems, the burden is shifted to the user who must filter out false bug reports - a time consuming task that takes away from the effectiveness of the system.

We also avoid specifying a specific threshold for path explosion since it depends on how the paths are used and the the computational requirements needed by the particular analysis. For the sake of giving a number, we loosely define functions or programs that have over 100,000 paths to suffer from path explosion.

3. Quantitative Analysis

Our analyses was carried out using 15 programs shown in Table 1. The number of functions only counts functions that reachable from main. The number of statements is a static count of the number of statements after simplification. Statements that are not in a reachable function or basic block are not counted.

3.1 Path Counts - No Slicing

Our initial experiment computes the number of paths without any slicing. This provides a worst-case number of paths for each of the programs. The results are shown in Table 2. The first column shows the total number of paths. The total path varies by program and ranges from 5,853 paths in *space* to 2.12 x 10^{17} paths in *indent.* The second column of Table 2 shows the number of paths in the function that has the most paths and the percent of paths this particular function contributes to the overall path count. All but two programs (*othello* and *space*) have one function that contains over 75% of the paths. The program *othello* has only 11 functions and none overwhelms one another. The program *space* consists of several small functions, each with a relatively small path count.

In 9 of the 15 programs, there is at least one function that contributes over 90% of the total path count. In the five programs with the largest path counts (*flex*, *gnuchess*, *gzip*, *indent*, and *thttpd*), the worst-case function suffers from massive path explosion and essentially contributes all of the paths for that program. Each of these programs contain additional (but not many) functions that have extremely high path counts but have orders of magnitude fewer paths than the worst-case function.

The last two columns in Table 2 show the number and percentage of functions that have 100 or fewer paths and the same for functions that have more than 100,000 paths. In all but two programs (*othello* and *yacr2*), at least 75% of the functions have only 100 paths or less. Nine of the programs had at least one function that contained over 100,000 paths with *flex*, *gnuchess*, *gzip*, and *indent* all having at least seven functions with at over 100,000 paths. The percentage of functions with over 100,000 paths was small - *diff3* had the highest percentage (9.4%).

3.2 Path Counts - Slicing on All Dangerous

This experiment enables slicing on all dangerous statements in the program. The number of paths are counted on the program that remains after slicing. The results are shown in Table 3. The first column shows the total path count after slicing is applied. The second column replicates the first column from Table 2 and displays the total number of paths without slicing while the third column shows the percent decrease after slicing is applied. The remaining columns are comparable to those found in Table 2 except that there is an additional column showing the number of functions that had zero paths - in other words, the function was completed sliced away.

Not surprisingly, the number of paths is greatly reduced for all programs. Eleven programs had a reduction of over 90%. For the most part, the reduction rate is independent on both the size of the program and the number of the paths without slicing. However, two programs (*bc* and *flex*) saw very little reduction in the number of paths. In both of these programs, the worst-case function was unaffected by the slicing due to a high number of dangerous operations in these functions.

On average, 15.8% of the functions had zero paths ranging from 5.3% (*indent*) to 37.5% (*diff3*). Each program saw moderate increases in the number of functions with 100 paths or fewer.

Despite the decreases in the number of the paths, programs that had numerous paths before slicing still have high path counts after slicing. While slicing reduced the path count by orders of magnitude for all programs except *bc* and *flex*, it was insufficient in reducing the total number of paths to a manageable number for the programs with high path counts. Of the nine programs that had at least one function with over 100,000 paths, eight of those programs (all but *diff3*) still had at least one functions over this arbitrary threshold. However, the number of functions containing over 100,000 paths was reduced in these programs especially *gzip* which went from having nine such functions without slicing to only one.

Table 1. Programs Used In Analysis.

Program	Description	Functions	Statements
bc	calculator	105	14,491
betaftpd	file transfer daemon	73	4,791
diff3	compares three files	32	4,016
find	file finder	398	31,098
flex	lexical analyzer	140	22,453
ft	spanning tree	33	1,879
ghttpd	web server	19	2,663
gnuchess	chess game	243	39,443
gzip	compression utility	106	11,380
indent	source code indenter	114	19,605
ks	graph partitioning	16	1,325
othello	othello game	11	1,055
space	specialized interpreter	127	11,652
thttpd	web server	130	12,500
yacr2	channel router	59	5,606

Table 2. Path statistics - no slicing.

Program	*Total paths*	*Paths in Worst Function*	*Functions with ≤100 paths*	*Functions with >100,000 paths*
bc	2,653,007	2,144,737 (80.8%)	87 (82.9%)	3 (2.9%)
betaftpd	68,365	55,297 (80.9%)	66 (90.4%)	0 (0.0%)
diff3	2,067,345	1,558,324 (75.4%)	23 (71.9%)	3 (9.4%)
find	22,453,011	21,748,720 (96.9%)	366 (92.0%)	3 (0.8%)
flex	7.33E+11	7.22E+11 (98.4%)	123 (87.9%)	7 (5.0%)
ft	10,498	10,082 (96.0%)	31 (93.9%)	0 (0.0%)
ghttpd	91,580	91,082 (99.5%)	16 (84.2%)	0 (0.0%)
gnuchess	2.35E+16	2.32E+16 (98.9%)	202 (83.1%)	12 (4.9%)
gzip	3.49E+11	3.44E+11 (98.8%)	80 (75.5%)	9 (8.5%)
indent	2.12E+17	2.12E+17 (100.0%)	94 (82.5%)	7 (6.1%)
ks	25,371	23,100 (91.0%)	14 (87.5%)	0 (0.0%)
othello	42,802	2,5057 (58.5%)	6 (54.5%)	0 (0.0%)
space	5,853	3,900 (66.6%)	123 (96.9%)	0 (0.0%)
thttpd	1.57E+14	1.57E+14 (100.0%)	108 (83.1%)	3 (2.3%)
yacr2	3,666,900	2,991,744 (81.6%)	40 (67.8%)	2 (3.4%)

Table 3. Path statistics - slicing criterion of all dangerous statements.

Program	*Total paths (no slicing)*	*Total paths (slicing)*	*% Paths Decrease*	*Paths in Worst Function*	*Functions with 0 paths*	*Functions with ≤100 paths*	*Functions with >100,000 paths*
bc	2,653,007	2,268,432	14.5%	2,144,736 (94.5%)	14 (13.3%)	91 (86.7%)	1 (1.0%)
betaftpd	68,365	5,212	92.4%	1,980 (38.0%)	7 (9.6%)	70 (95.9%)	0 (0.0%)
diff3	2,067,345	40,423	98.0%	20,412 (50.5%)	12 (37.5%)	26 (81.3%)	0 (0.0%)
find	22,453,011	4,146,604	81.5%	4,057,361 (97.8%)	64 (16.1%)	382 (96.0%)	1 (0.3%)
flex	7.33E+11	7.22E+11	1.6%	7.22E+11 (100.0%)	17 (12.1%)	128 (91.4%)	4 (2.9%)
ft	10,498	257	97.6%	194 (75.5%)	4 (12.1%)	32 (97.0%)	0 (0.0%)
ghttpd	91,580	2,701	97.1%	2,520 (93.3%)	5 (26.3%)	18 (94.7%)	0 (0.0%)
gnuchess	2.35E+16	3.41E+14	98.5%	2.66E+14 (77.9%)	54 (22.2%)	214 (88.1%)	11 (4.5%)
gzip	3.49E+11	8.26E+08	99.8%	8.26E+08 (100.0%)	15 (14.2%)	91 (85.8%)	1 (0.9%)
indent	2.12E+17	8.00E+13	100.0%	8.00E+13 (100.0%)	6 (5.3%)	96 (84.2%)	6 (5.3%)
ks	25,371	1,519	94.0%	1,400 (92.2%)	3 (18.8%)	15 (93.8%)	0 (0.0%)
othello	42,802	3,462	91.9%	3,249 (93.8%)	2 (18.2%)	10 (90.9%)	0 (0.0%)
space	5,853	1,892	67.7%	346 (18.3%)	8 (6.3%)	124 (97.6%)	0 (0.0%)
thttpd	1.57E+14	4.19E+12	97.3%	4.19E+12 (100.0%)	10 (7.7%)	111 (85.4%)	2 (1.5%)
yacr2	3,666,900	287,639	92.2%	259,328 (90.2%)	10 (16.9%)	46 (78.0%)	1 (1.7%)

3.3 Path Counts - Slicing on Statements

Now we explore the effect on the path count by slicing with respect to a single statement. In these experiments, the initial slicing criterion is a dangerous statement. All code not relevant to verifying that operation is discarded. Then the paths are counted using the remaining code. This is repeated for all dangerous statements in the program.

Figure 2 details the results of this experiment. We define a "run" to be the slicing and subsequent path counting with respect to one statement. If a program has *n* dangerous statement, it will have *n* runs - one for each of the dangerous statements. The graph plots the cumulative percentage of runs that have fewer than the given number of paths. For example, 78% of the individual operation runs for the program *ghttpd* have fewer than 100 paths.

In five of the programs, all runs had fewer than 1,000 paths. Four of these programs (*betaftpd*, *ft*, *ghttpd*, and *ks*) had relatively few paths when slicing was applied to all operations. However, the fifth program, *yacr2*, had 287,639 paths in the same experiment and over three million paths without slicing. This greatly reduces the burden to analyze *yacr2*. In four other programs (*diff3*, *find*, *othello*, and *space*), over 90% of the runs have fewer than 1,000 paths. The program *bc* is strange in that 13% of the runs have fewer than 100 paths but most of the remaining runs each contain around 225,000 paths.

The remaining five programs (*flex*, *gnuchess*, *gzip*, *indent*, and *thttpd*) all had a significant percentage of runs that contained over ten million paths. For these programs, slicing based on an individual statement criterion did not eliminate the path explosion. Except for *gzip*, these programs

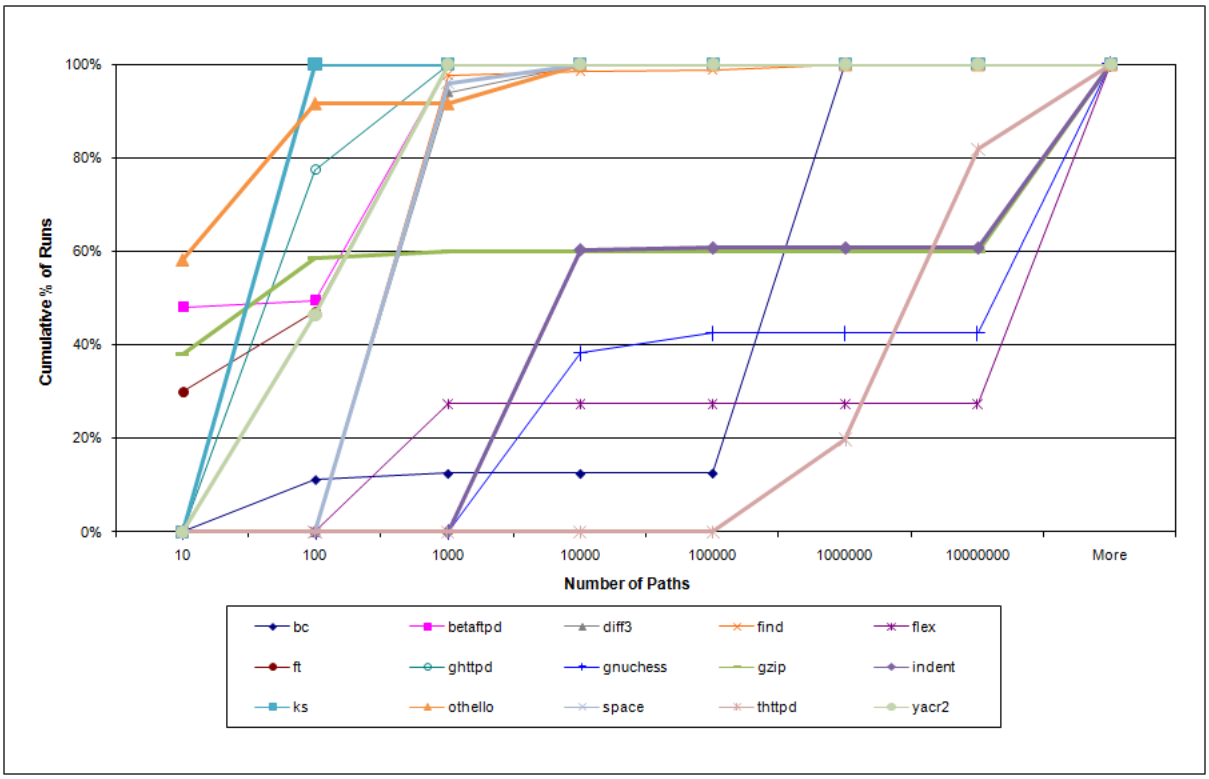

Figure 2. Individual statement runs.

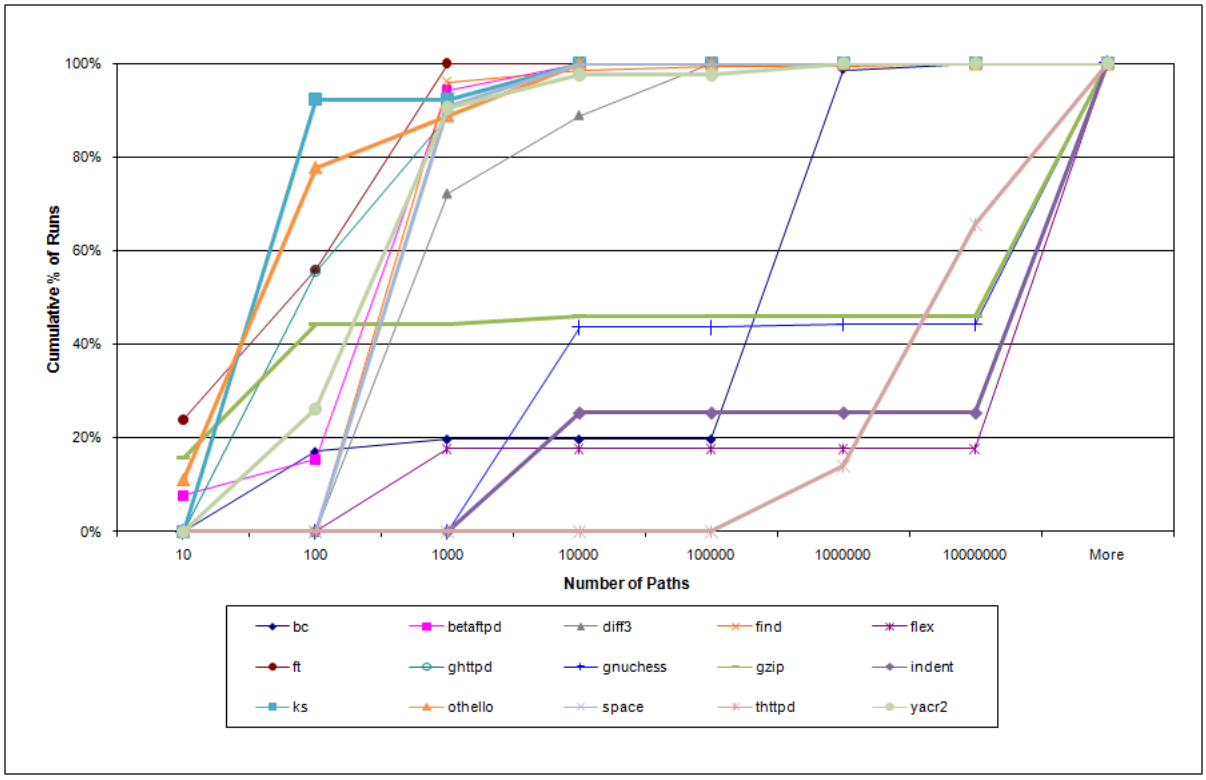

Figure 3. Individual function runs..

did not have very many runs that had few paths. The program *gzip* was bimodal like *bc* with 59% of the runs having fewer than 100 paths and most of the remaining runs having around 460 million paths.

The effectiveness of slicing is determined by the worst-case run - the run with the most paths. If the number of paths is similar to the number of paths when slicing on all the statements, then slicing by statement is not effective. In Table 4, the first column repeats the number of paths when the slicing criterion is all dangerous statements from Table 3. The second column displays the number of paths in the worst-case statement run. The third column displays the number of paths in the worst-case function run; this column is discussed in Section 3.4.

The results from Table 4 are a mixed bag. The program that benefits the most is *yacr2* where the worst-case run only contains 818 paths, compared to the 287,639 paths when all dangerous statements were used as the slicing criterion. The programs *betaftpd*, *ghttpd*, and *ks* all had fairly small path counts in the worst-case runs but they did not have many paths to begin with. In four of the programs (*ft*, *gzip*, *othello*, and *space*), the worst-case run contains almost the same number of paths as slicing with all statements. For these programs, slicing on individual statements is not fruitful. For four of the programs with high path counts (*flex*, *gnuchess*, *indent*, *thttpd*), the number of paths is greatly reduced but path explosion still exists.

3.4 Path Counts - Slicing on Functions

Figure 3 displays the results for a similar experiment but uses functions instead of statements. The slicing criterion is all dangerous statements within the given function. Here, a run refers to the analysis after slicing with respect to a function. The results are similar in nature to the individual statements. This is especially true for the programs that suffer from path explosion as the cumulative percentage lines are roughly the same. The programs with fewer paths still have a high percentage of runs with very few paths but the slope of the data lines are noticeably lower than the corresponding statement data lines (especially *diff3*). Of particular note is the percentage of runs with ten or less paths is significantly reduced when compared to the statement run and that only one program (*ft*) has 100% of runs with 1,000 paths or less, compared to five such programs in the statement runs.

To dig deeper, consult Table 4 where the last column shows the number of paths in the worst-case function run. First, there was little difference in the four programs (*ft*, *gzip*, *othello*, and *space*) that had a worst-case path count that was comparable to the paths when slicing on all statements. In the other 11 programs, the worst-case path count for a function run is significantly higher than the worst-case statement run and is often the same order of magnitude as the slicing on all statements run. For these programs, the function that triggered the worst-case run is the function with the most paths. Referring back to Table 3, this result is not unexpected as the function with the most paths contains over 75% of the paths. Though not shown, the function with the second-most paths often had a similar path count to the worst-case statement run. This suggests that using function runs for functions with few paths may be beneficial.

Table 4. Worst Case Path Count Comparison.

Program	*Total paths (slicing - all)*	*Worst Case Run*	
		Total paths (slicing - stmt)	*Total paths (slicing - func)*
bc	2,268,432	617,992	1,106,152
betaftpd	5,212	615	2,341
diff3	40,423	3,256	20,788
find	4,146,604	171,394	4,058,603
flex	7.22E+11	6.44E+10	6.44E+10
ft	257	244	244
ghttpd	2,701	132	1,614
gnuchess	3.41E+14	1.11E+13	2.66E+14
gzip	8.26E+08	6.19E+08	6.19E+08
indent	8.00E+13	7.36E+12	4.99E+13
ks	1,519	76	1,467
othello	3,462	3,286	3,290
space	1,892	1,231	1,231
thttpd	4.19E+12	9.04E+08	1.86E+11
yacr2	287,639	818	259,518

4. Qualitative Analysis

In this section, we qualitatively describe what went on behind the scenes. First, we give a brief analysis for each program and followed by a summary at the end.

4.1 Individual Program Analysis

bc: The two worst functions are `bc_divide` (performs division for the calculator) and `yyparse` (parsing the operation string from the user). The results of these functions are used by other arithmetic functions so they are almost always included in the slice. Subsequently, a large percentage of individual statement runs have around 200,000 - 600,000 paths.

betaftpd: The function `long_listing` has the most paths before slicing with 55,297 paths. It is a function that prints the contents of a directory like the "`ls -l`" command in UNIX. Most of the complexity comes from printing out the permission map - a series of if statements that conditionally print characters if a particular permission flag is set. These decisions all get eliminated during slicing. After slicing, only three functions have more than 1,000 paths - two parse input commands, and the third is a different function used to print the contents of a directory.

diff3: The function with the most paths is

process_diff_control which parses a diff control string (1,558,324 paths). It still has the most paths after slicing though many paths are removed (20,412 paths). The number of paths decreases further when slicing using individual statements. Outside of process_diff_control, only main and output_diff3_merge (a file processing function) have significant path counts.

find: The function quotearg_buffer_restyled has the most paths (over 21 million). This function modifies and buffers a string. There are several different options and special cases the function needs to address in addition to parsing the string for special characters. The complexity due to parsing is largely eliminated during slicing (resulting in just over 4 million paths). After slicing the function consider_visiting (a decision-making function based on the current state of the program) has 67,446 paths, five functions that have between 1,000-7,200 paths, and the rest having fewer than 1,000 paths. The statement and function runs were very effective for runs that did not involve quotearg_buffer_restyled or consider_visiting.

flex: The function flexend has the highest path count with over 700 billion paths without slicing and was one of six functions with over 100 million paths. The function flexend is run at the end and produces a myriad of output based on various settings, statistics, and error conditions. The function was unaffected by slicing internally with almost the same number of paths after slicing. The path count for individual runs was high for most runs. When flexend is not in the slice, functions flexinit (processing command line options), ntod (finite state automata conversion), and gentabs (internal processing function) were often in the slice and all had significant path counts after slicing.

ft: The worst function is DeleteMin - a function that finds and deletes a minimal node from a tree-like data structure. This function only has 10,082 paths and gets reduced to 194 paths after slicing. All other functions have at most 8 paths after slicing. With only 257 paths in the slicing on all dangerous statement runs, the individual statement and functions runs are rather pointless but do have very few paths.

ghttpd: The function serveconnection accounts for majority of the paths with and without slicing. This function sets up a connection and handles different types of HTTP requests. The individual statement and function runs were effective in bringing the path count down even more. Except for the function run using serveconnection as the slicing criterion, all individual statement and function runs had less than 200 paths.

gnuchess: There are 11 functions with over one million paths before slicing. After slicing, there are still seven such functions and five of those had over 26 billion paths. The two functions that have the most paths are Search (search for the best possible move) and GenMoves (generates all possible moves). Both of these functions are very specific to the game of chess. Most of the individual runs were unable to slice away the path explosion found in either function.

gzip: The function get_method has the most paths. It used to read the magic number of the zip file and set various options. The good news is that 66% of the individual runs did not include get_method in the slice. The bad news is that for the 33% of the runs, get_method was included and not enough paths could be removed after slicing to make the path count more reasonable.

indent: Seven functions are primarily responsible for a high path count. The two worst functions are output functions dump_line and print_comment. The functions make a number of decisions on how to print a line or comment based on the input file and the options supplied by the user. The third worst function is lexi, the lexer for the program that divides up the input source code file into tokens. The next four functions are various token processing functions. Slicing and the use of individual runs helps reduce the path counts but not enough to reduce the path explosion.

ks: The self-explanatory function PrintResults has the most paths with 23,100 before slicing and 1,400 when slicing on all dangerous statements. Fortunately, all of the individual and function runs completely slice away this function (except runs using slicing criteria within the function).

othello: The functions humanmove and validmove have the most paths before slicing (25,057 and 12,996 paths respectively). Most of humanmove is sliced away due to a lack of dangerous statements as the paths come from detecting different illegal inputs. This means that validmove has the most paths after slicing. An interesting phenomenon is noted with the statement and function runs. The worst-case statement and function runs occurs when using slicing criteria within the function testmove which itself has very few paths. However, it is called eight times by validmove causing it not to be sliced away. Runs with slicing criteria outside of testmove all have 100 paths or less.

space: Only one function (seqrotrg with 3,900 paths) had more than 350 paths before slicing. After slicing, all functions had less than 350 paths with a total of 1,892 paths. The individual statement and function runs improved the path count to some extent.

thttpd: The worst function is main with over 4 trillion paths in slicing on all dangerous statements run. Since main always remains in the slice, the effectiveness of the individual runs is limited. In the best case individual run, main still has 600,000 paths and often has more. The second worst function httpd_parse_request is also present in most of the individual runs and usually contributes over 200,000 paths.

yacr2: The function PrintChannel has the most number of paths with 2,991,744 paths before slicing and 259,328 paths after slicing on all dangerous statements. It is a function that displays results to the screen and has a large number of independent control decisions. Since it does not produce

any data, it is not included in the slices for statements in other functions, making the individual runs very effective for this program.

4.2 Summarizing Remarks

By and large, most functions contain a moderate amount of paths and do not suffer from path explosion. However, many of the programs used in this study have at least one function that has a significant number of paths. There were three types of functions that seemed to generate many paths: input processing functions, stylized output functions, and parsing functions. The stylized output functions are often sliced away since they are often run at the end of the program but the input processing and parsing functions were often included in the slices.

Not all functions that suffered from path explosion fit into these categories. These other functions were specific to the program under analysis. This includes, among others, functions like as divide in *bc*, finite state automata conversion in *flex*, and searching for the best move in *gnuchess*. These tasks are all inherently complex so it is not unexpected that the corresponding functions suffer from path explosion.

The effectiveness of individual runs (both statement and functions) was solely determined on whether a function suffered from path explosion was included in the slice. Since slicing did not sufficiently reduce the number of paths within these functions, the individual runs were only effective when the functions were removed from the slice altogether. Even though the worst-case performance of the individual runs was similar to that of a run using all dangerous operations as the slicing criteria, almost all of the programs had some statements that could be analyzed without suffering path explosion.

5. Related Work

Several groups have explored path-based analyses of different sorts. Hampapuram et. al. [9] constructed a general-purpose path simulator. Dillig et. al. [7] developed a scalable path-sensitive analysis that is sound and complete. Unfortunately, the generated constraints are not solvable. They are able to pose and solve queries based on conditions extracted from the generated constraints.

Numerous research groups have explored software bug detection. This study was inspired by path-based static verification systems including PREfix [3], ARCHER [21], and Marple [14]. The PREfix paper [3] describes interprocedural analysis using function summaries. Coverity Prevent [5] and CodeSonar [8] are commercial tools that use path-based analysis.

One deficiency in our analysis is that it does not account for different types of loops. Loop squashing by Hu et. al. [12] applies a series of transforming rules with the hope of converting a loop into a conditional statement. In the dynamic execution tool created by Stewart [19], loops that do not have any data dependencies between iterations are detected and executed in a single pass. ARCHER [21] tries to match loops to commonly occurring patterns (such as an incrementing for loop with a constant upper bound) and optimizes the analysis based on the particular pattern.

Binkley and Harman [2] compare different program slicing approaches. They show that slices, on average, have a size that is roughly 30% of the original program. Using a context-insensitive approach increases the slice size by 50%. They also investigate the effect of expanding structure fields, something not done in this work. This decreased the slice size by 2% on average. Utilizing a full context-sensitive algorithm would likely improve the precision of our results.

Another related area of research is the use of metrics to measure the complexity of the program. The cyclomatic complexity [15] of a program measures the number of closed loops in a flow graph. The Understand for C++ tool [18] uses a variety of metrics such as number of lines, number of inputs, number of outputs as a gauge of complexity. This idea was expanded by Pan *et. al.* [17] to include program slicing metrics. The metrics are used to classify bugs allowing them to predict bugs in the future. Ball and Larus [1] use profiling to determine which paths are executed most frequently.

6. Conclusion

This paper explores the number of paths present in 15 different programs. Most functions with a program had relatively few paths. Often one function contributes most of the paths. Over half of the programs have at least one function that suffers from path explosion.

We also explored the effect of slicing has on path counts by using three different types of slicing criteria: all dangerous statements in a program, all dangerous statements in a function, and a single dangerous statement. In general, slicing reduces the number of paths in a program but it is not sufficient to address path explosion.

The runs that slice on individual statements or functions depend on whether a function that suffers from path explosion appears in the slice. If such a function does not appear, the resulting number of paths is often manageable - most programs had several runs in this situation. If a function with path explosion is in the slice, the path count is similar to the run using all dangerous statements.

Looking further at the programs, we found that functions that process input, produce stylized output, or parse have high path counts often resulting in path explosion. The results can also guide static analysis and bug detection research. Future analyses could specifically look at these areas and determine if there are ways to further reduce the

number of paths. Unfortunately, this does not work in all cases as some programs also had inherently complex tasks specific to that program that also lead to path explosion.

In the future, we plan to explore the effects of some of our design decisions have on the path count and whether that changes the findings. In particular, we will analyze different techniques for counting loops and accounting for interprocedural analysis. We could also expand the analysis to include programs written in other languages such as C++. Programs written in an object-oriented style tend to have smaller functions but have more complex interprocedural relationships.

Another avenue of future work is to explore how programs with large path counts can be broken down into more manageable pieces of program analysis tools. One possibility is to have some level of user guidance to work around functions that have high path counts. For instance, in the *bc* calculator program the multiply and parse functions could be analyzed independently even though the multiply function depends on the parse function. A more ideal solution would be to have more automated analyses possibly based on heuristics that analyze how a function is dependent on another function.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] T. Ball and J. Larus. Efficient Path Profiling. Proceedings of the 29th Annual International Symposium on Microarchitecture. Dec. 1996.

[2] D. Binkley and M. Harman. A Large-Scale Empirical Study of Forward and Backwards Static Slice Size and Context Sensitivity. Proc. of the 2003 International Conference on Software Maintenance. Sept. 2003.

[3] W. Bush, J. Pincus, D. Sielaff. A static analyzer for finding dynamic programming errors. Software Practice and Experience, July 2000.

[4] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. Proceedings of the Conference on Computer and Communications Security, November 2002.

[5] Coverity, Inc. Converity Prevent <http://www.coverity.com/html/coverity-prevent-static-analysis.html> 2008.

[6] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Verification in Polynomial Time. Proceedings of the Conference on Programming Language Design and Implementation, June 2002.

[7] I. Dillig, T. Dillig, And A. Aiken. Sound, Complete, and Scalable Path-Sensitive Analysis. Proceedings of the Conference on Programming Language Design and Implementation, June 2008.

[8] GrammaTech, Inc. CodeSonar <http://www.grammatech.com/products/codesonar/overview.html> 2008.

[9] H. Hampapuram, Y. Yang, and M. Das. Symbolic Path Simulation in Path-Sensitive Dataflow Analysis. Proceedings of Workshop on Program Analysis for Software Tools and Engineering, Sep. 2005.

[10] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural Pointer Alias Analysis. ACM Transactions on Programming Languages and Systems. July 1999.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems. Jan. 1990.

[12] L. Hu, M. Harman, R. Hierons, and D. Binkley. Loop Squashing Transformations for Amorphous Slicing. Proceedings of the 6th International Workshop on Source Code Analysis and Manipulation, Sep. 2006.

[13] E. Larson. SUDS: An Infrastructure for Creating Bug Detection Tools. Proc. of the Conf. on Source Code and Manipulation, Sept. 2007.

[14] W. Le and M. Soffa. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. Proceeding of the Conference on Foundations of Software Engineering, Nov. 2008.

[15] T. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, Dec. 1976.

[16] G. Necula, S. McPeak, S. P. Rahul, W.Weimer. Cil: Intermediate Language and Tools for Analysis and Transformation of C Programs. International Conference on Compiler Construction, Apr. 2002

[17] K. Pan, S. Kim, and J. Whitehead. Bug Classification Using Program Slicing Metrics. Proceedings of the 6th International Workshop on Source Code Analysis and Manipulation, Sep. 2006.

[18] Scientific Toolworks Inc. Understand for C. http://www.scitools.com/products/understand/cpp/product.php

[19] M. Stewart. Towards a Tool for Rigorous, Automated Code Comprehension Using Symbolic Execution and Semantic Analysis. Proc. of the 29th IEEE/NASA Software Engineering Workshop. Mar. 2005.

[20] M. Weiser. Program slicing. Proceedings of the 5th International Conference on Software Engineering, Mar. 1981.

[21] Y. Xie, A. Chou, and D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. Proc. of the Symposium on the Foundations of Software Engineering, Sep. 2003.