Program Analysis Too Loopy? Set the Loops Aside

Eric Larson
Seattle University

This paper is a postprint of a paper submitted to and accepted for publication in IET Software (Volume 7, Issue 3, June 2013) and is subject to Institution of Engineering and Technology Copyright. The copy of record is available at IET Digital Library.

Abstract
Among the many obstacles to efficient and sound program analysis, loops may be the most prevalent. In program analyses that traverse paths, loops introduce a variable, possibly infinite, number of paths. This paper assesses the potential of a program analysis technique that analyzes loops separately and replaces the loop with a summary, similar to how many analyses use summaries for interprocedural analysis. This study is conducted by comparing the path counts when loops are analyzed separately to a baseline path count where loops are traversed at most once. While the number of paths is decreased in many cases, the magnitude of the decrease is typically not sufficient for long, complex functions. In addition, loops are classified by the task they perform, analyzed using the number of paths as an estimate of their complexity and further inspected for programming elements that may make loop analysis more difficult. Of the 2,869 loops used in this study, 84% of the loops have fewer than ten paths and only 1.3% have more than 10,000 paths. Nearly 60% of the loops traverse arrays or strings and roughly half of the loops contain a function call.

Keywords
loops; program analysis; paths

1. Introduction
Dealing with loops is a necessary evil in program analysis. Every program contains loops but they can be difficult to analyze. Loops are used to do a variety of different things such as traversing a data structure, obtaining data from an input source, scanning a string for a particular character, and much more. Some uses of loops are common to different programs while other uses are specific to a particular program.

There are several challenges that must be addressed when analyzing loops. The number of iterations of a loop is often variable and possibly infinite. Dependencies can be hard to track when the results of one iteration depend on the result of previous iterations. In addition, loops are commonly associated with large, possibly complex, data structures. These data structures can be hard to model during program analysis.

Symbolic execution [1, 2, 3] is a program analysis technique that attempts to simulate every path through a function. Obviously, loops complicate symbolic execution by introducing a large, possibly infinite, number of paths. A common strategy to address this issue is to limit the number of iterations a loop can take. Another complication in symbolic execution is function calls. In PREFix [2], functions are analyzed bottom-up and a summary is created that describes the function's behavior. When analyzing a function that calls another function, the function call is replaced with the summary. Can a similar analysis strategy be used with loops? Can loops be analyzed separately first and then replaced with summaries? If so, this would decrease the number of paths to analyze and nearly eliminate the need to analyze infinite loops.

Since the number of paths in a function with loops can be unbounded, summarizing the impact of all the paths through each loop in a function would greatly simplify a path-based analysis (such as symbolic execution) of the function. This paper analyzes the potential benefit that such a summarization could achieve. The number of paths

that need to be analyzed after loops are summarized is compared to conservative baseline path count. This conservative baseline counts the number of paths through a function by assuming that each loop executes at most one time, providing a lower bound on the reduction in the number of paths achieved if loops can be replaced by summaries.

This empirical study uses a collection of 25 moderately-sized C programs. The key question considered is the magnitude of the decrease in path count when loops are summarized. In addition to the path count reduction, this research study also analyzes the individual loops present in the programs. Each loop is analyzed manually and classified by the task it performs. The complexity of a loop is estimated using the number of paths through the loop. To better assess the challenges of analyzing loops, we determine how many loops contain programming elements that are difficult to analyze such as function calls, alternate exits, and nested loops.

This paper investigates the following research questions:

- *What is the reduction in path count attained from simulating the replacement of loops with summaries?* While replacing loops with summaries reduces the number of paths within most functions with loops, the technique is not sufficient in reducing the number of paths to a reasonable number in the most complex functions.
- *Using the number of paths in the loop body as a complexity measure, how complex are loops?* Most, but not all, loops have simple bodies with few paths.
- *How often do loops contain certain programming constructs or properties that may make analysis difficult?* Of the key findings: 59.5% of loops traverse arrays or strings, 51.3% contain a function call, 28.4% contain an alternate exit beyond the normal stopping condition, and 21.6% contain a nested loop. Complex loops with many paths are more likely to contain these constructs making analysis even more difficult.

The rest of the paper is organized as follows. Section 2 describes the terminology using an example. The implementation of our loop analyses is described in Section 3. Section 4 presents the results of these analyses. Threats to validity are described in Section 5. Related work is outlined in Section 6 and Section 7 concludes.

2. Definitions

This study explores the number of paths using the control flow graph. In a *control flow graph*, the nodes correspond to basic blocks which contain one or more statements. The directed edges refer to different control decisions. For instance, a basic block ending with an if statement will have two edges: one when the condition is true and one when the condition is false. An intraprocedural control flow graph is used for this study – each function has its own separate control flow graph. The node with the first statement of the function is the *starting node*. The control flow graph also has one or more ending nodes. An *ending node* either contains a return statement, the last statement of the function, or a known function that unconditionally exits the program (such as `exit`).

A *path* within a function is defined as a path through the control flow graph from the starting node to an ending node. The *theoretical path count* is the number of unique paths within the function. For functions that contain loops, the theoretical path count is infinite since the number of iterations in a loop could be increased without limit. As defined, the theoretical path count does not consider whether a path is possible to execute or not. In practice, some loops have a fixed number of iterations resulting in a finite number of realizable paths.

Path-based analyses must cope with functions that have an infinite number of paths. One common way of addressing this issue is to limit the number of iterations in the loop. While this restriction makes the number of paths in a function finite, it may introduce imprecision in the analysis. For this study, the baseline path count is computed such that each loop is traversed at most one time. Despite this lenient restriction, the number of paths is extremely high for several functions.

More formally the *baseline path count* for a function is the number of unique paths from the starting node to an ending node of the function's control flow graph. Loops (represented using cycles in the control flow graph) are traversed at most once. The path counts are computed intraprocedurally. A baseline path count for the entire program is computed by summing the path counts of its functions. Since the path counts are computed only using the control flow graph – no attempt is made to remove infeasible paths that are impossible to execute. We deem this to be appropriate since any path-based analysis technique must be able to detect and handle infeasible paths.

Consider the example in Figure 1. The baseline path count is 114. One way of computing this path count is to divide the control flow graphs into sections: the control flow graph before node 6 (contains nodes 1-5), the control flow graph representing the loop at node 6 (contains nodes 6-20), and the control flow graph representing the loop after node 6 (contains nodes 21-25). There are 2 paths from node 1 to 6. There are a total of 18 paths through the loop at node 6 plus a 19th path that skips the loop (zero iterations). There are 3 paths from node 21 to the end. Multiplying the path counts from these sections gives $2 \times 19 \times 3 = 114$.

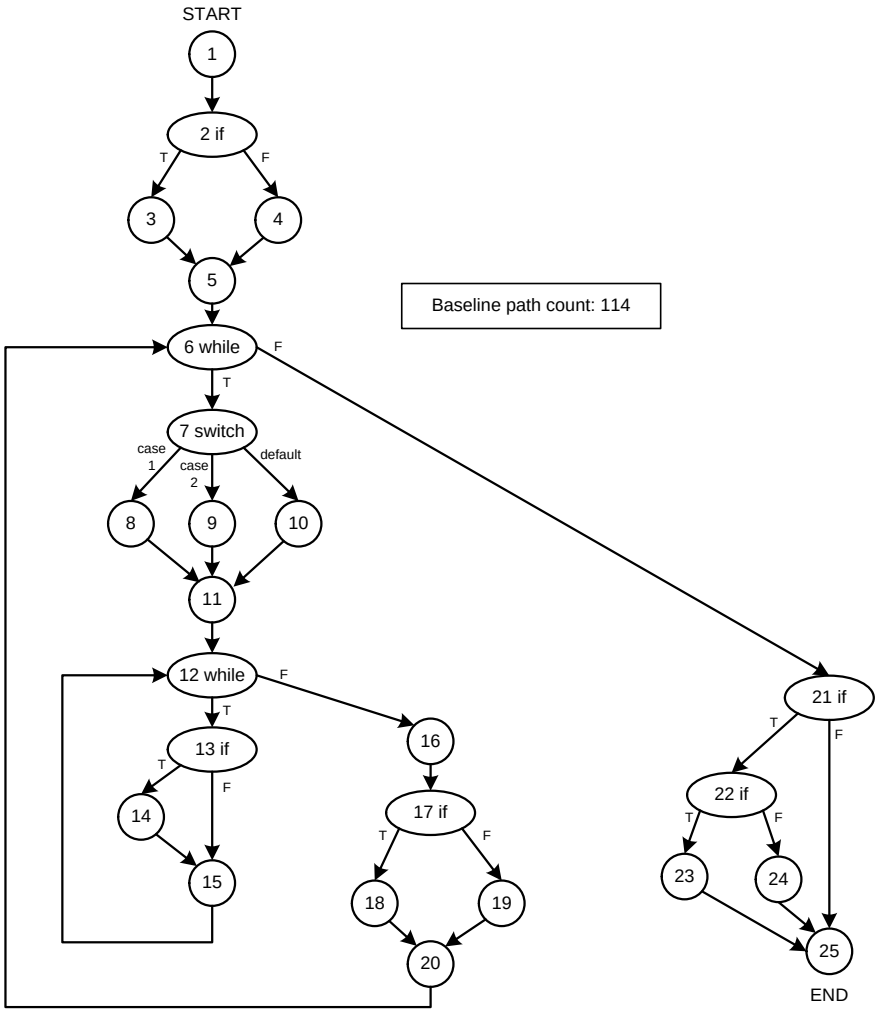

Figure 1. Example Control Flow Graph

```
int a[SIZE];
int *p = a;
int i;

for (i = 0; i < SIZE; i++) {
  p++;
}

for (i = 0; i < SIZE; i++) {
  if (a[i] == 0) {
    p++;
  }
}
```

Figure 2. Example Loops

Our research assess the potential of replacing a loop with a loop summary. A *loop summary* is a summary that captures state changes during the loop. To illustrate the concept of a loop summary, consider the two loops in Figure 2. Assume the analysis that is being used determines whether a pointer points to valid memory or not. The first loop of Figure 2 can be summarized by stating that p points to the memory location just outside of the array. The second loop is more difficult to summarize and depends on the precision of the analysis. One possible summary simply notes that it is possible for p to point to the location after the array. A more precise summary will indicate that it is only possible for p to point to the location after the array if the array consists of all zeros. More precise still is a summary that notes the location of p after the loop based on the number of zeros in the array. The format and contents of a loop summary will differ based on the specific analysis being used and the precision of that analysis. The broader question of creating loop summaries [4, 5, 6] is beyond the scope of this paper.

To assess the benefit of separating loops, we envision a *loop-separate analysis* that is carried out as follows:

1. Analyze each loop in the program creating a loop summary. In the case of nested loops, the inner loops are analyzed and summarized first.
2. Replace the loops with their loop summaries.
3. Use a path-based analysis technique such as symbolic execution to exhaustively analyze all paths.

The feasibility of a path-based analysis approach is dependent on the number of paths it needs to traverse. This study explores the effect on the path count when using the loop-separate analysis as described above. To start, the number of paths without loops is computed for each function. This is accomplished by augmenting the control flow graph such that nodes that immediately precede the loop (*loop predecessors*) are redirected to point to the node that immediately follows the loop (*loop successor*). The same path counting algorithm is used on this augmented control flow graph. The resulting number of paths is the *outside path count* – the number of paths outside any loop in the function.

The augmented control flow graph for Figure 1 is shown in Figure 3. To better illustrate how loop-separate analysis works, the augmented control flow graph is shown with a loop summary. The outside path count computed from the augmented control flow graph in Figure 3 is six, representing the complexity of the function that resides outside the loops of the program.

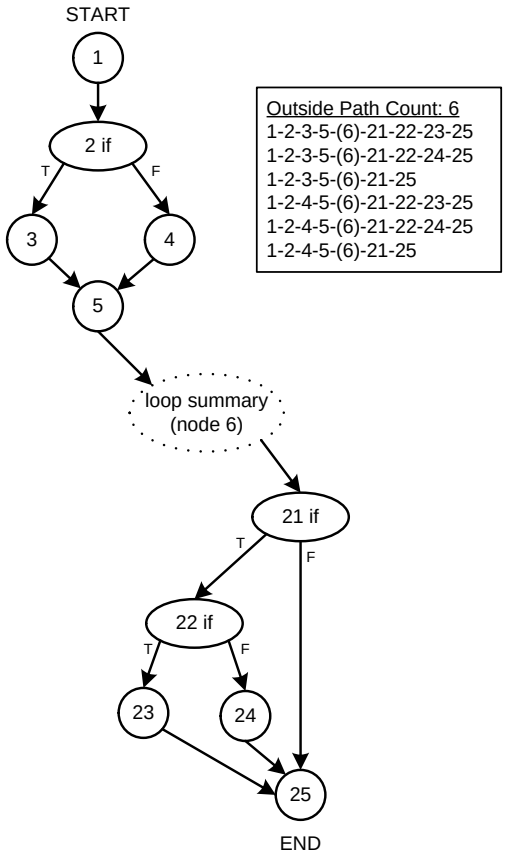

Figure 3. Augmented CFG for Computing Outside Path Count

Then, the number of paths is computed for each loop. To count the number of paths within a loop, the control flow graph for a function is augmented to only include nodes in the loop body. In the augmented control flow graph, the *loop starting node* is the basic block that contains the first statement in the loop body. A *loop ending node* is a node that corresponds to the end of loop body or a node that contains a statement that exits the loop (such as break or return). In Figure 1, consider the while loop at node 6. The loop starting node for this loop is node 7 and the lone loop ending node is node 20. Similarly, the while loop at node 12 has node 13 as its loop starting node and has node 15 as its one loop ending node.

A *loop path* is a path in the control flow graph from the loop starting node to an loop ending node. Since the loop may contain an inner loop, two path count measures are used:

- *Loop path count with loops*: The number of unique paths in a loop where each inner loop is traversed at most one time.
- *Loop path count without loops*: The number of unique paths in a loop when inner loops are replaced with loop summaries.

Consider the innermost while loop in Figure 1 – the loop at node 12. The resulting augmented control flow graph is shown in Figure 4. There are two paths in this control flow graph. Since the loop contains no inner loops, the loop path count with loops and the loop path count without loops are both equal to two. For the outermost while loop at node 6 in Figure 1, the two loop path count metrics differ due to the presence of the inner loop. The augmented control flow graph in Figure 5 is used to compute the loop path count with loops – 18 in this case. To

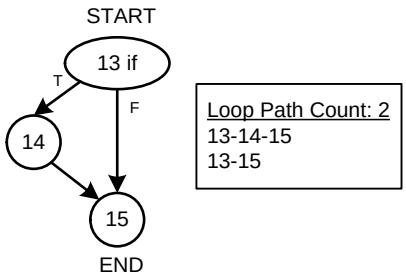

Figure 4. Augmented CFG for Computing Loop Path Count (with or without loops) for Loop at Node 12

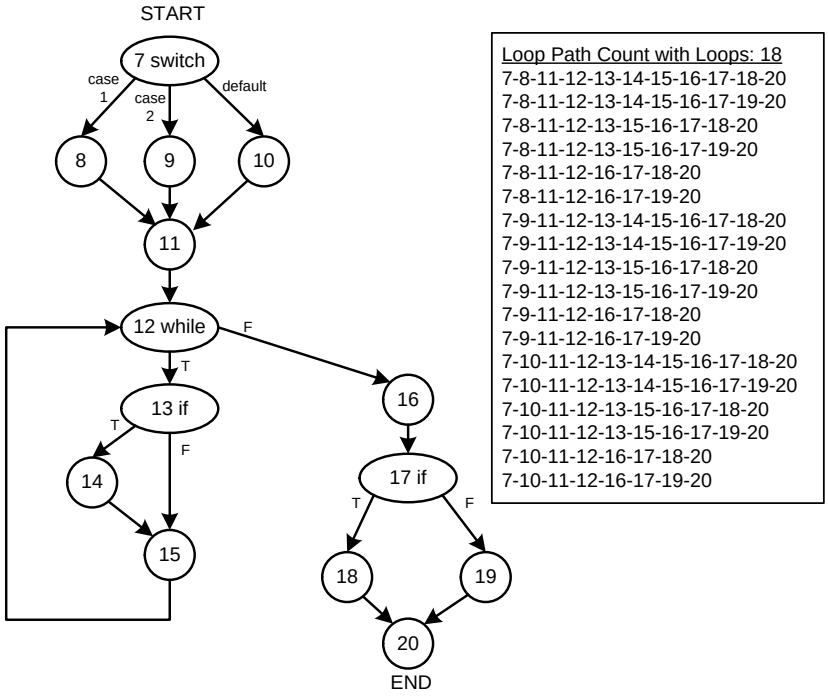

Figure 5. Augmented CFG for Computing Loop Path Count with Loops for the Loop at Node 6

compute the loop path count without loops, the inner loop is replaced with a loop summary, resulting in the augmented control flow graph shown in Figure 6. The loop path count without loops is six.

The *inside path count* is the number of paths inside the loops of a function. It is the sum of the loop path count without loops over all the loops in the function. The *loop-separate path count* for a function is the sum of outside path count and the inside path count. The loop-separate path count is compared to the baseline path count to assess the possible benefit of loop-separate analysis. For functions without loops, the loop-separate path count and the baseline path counts are identical. For a vast majority of the functions containing a loop, the number of paths decreases when using loop-separate analysis. This research explores the magnitude of that decrease.

Using the example in Figure 1, the inside path count is eight: two paths for the loop at node 12 and six paths for the loop at node 6. The loop-separate path count is 14: six outside paths plus eight inside paths. Note that the loop-separate path count represents the maximal benefit of using loop-separate analysis: loops are skipped whenever possible. Loops are skipped when computing the outside path count and inner loops are skipped when

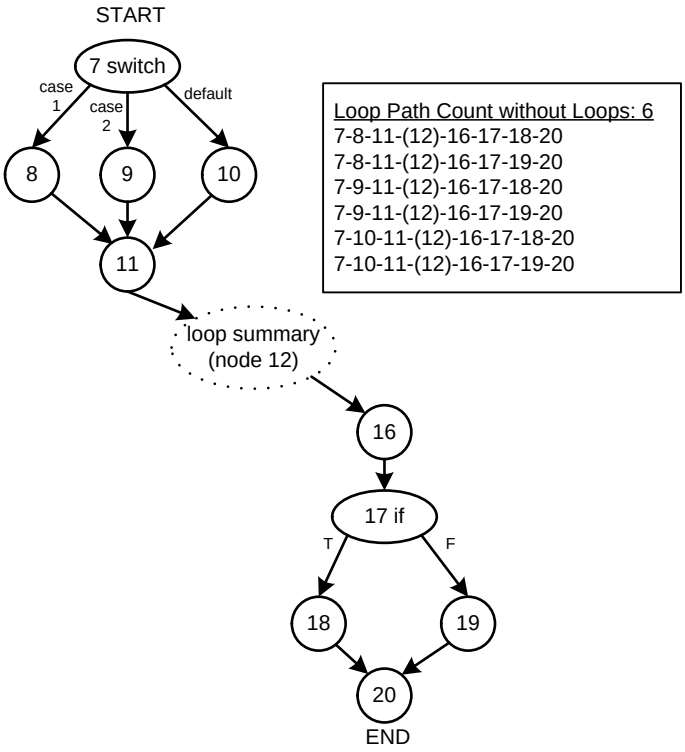

Figure 6. Augmented CFG for Computing Loop Path Count without Loops for the Loop at Node 6

computing the inside path count. For this example, the effect of using loop-separate analysis is that the control flow graph in Figure 1 is replaced with the three less complex control flow graphs shown in Figures 3, 4, and 6.

3. Loop Analyses

This section describes the different loop analyses employed in this study.

3.1 Path Counting Algorithm

In this section, we describe the path counting algorithms used in this study. Paths are counted using a depth-first traversal of a control flow graph. The control flow graph was generated using CodeSurfer [7]. CodeSurfer parses preprocessed C source code created using '`gcc –E`'. Using preprocessed source code exposes any loops that are embedded in macros. CodeSurfer was run with the options "`-basic-blocks yes -handle-exception no -handle-abnormal-exit no gcc`". The consequence of these options directs CodeSurfer to use the gcc compiler model and to suppress the addition of edges due to abnormal exits and exceptions The CodeSurfer internal representation converts short-circuited operators into the appropriate if-else constructs, increasing the number of paths for functions containing these operators.

In the baseline path counting algorithm, loops are traversed at most once. This rule is enforced by modifying the control flow graph such that loop back edges are removed from the graph and replaced with an edge that connects the end of the loop body to the loop successor (the basic block that immediately follows the loop).

Loops constructed using backward goto statements are ignored. A backwards goto statement is considered to end the path. This is done by removing the back edge after a backwards goto statement from the control flow graph and marking the node as a function ending node. There are only 32 backwards goto statements in the 25 programs analyzed in this study. Of these 32 statements, 19 occur in four scanning and parsing functions that were created

using scanner and parser generation tools. No modifications are made to the control flow graph for forward goto statements.

The baseline path count is the number of unique paths from the starting node to a function ending node. A recursive depth-first search is used to count the unique paths. The following nodes are considered function ending nodes:

- The node containing the last statement of the function.
- Any node containing a return statement.
- Any node containing a backwards goto statement.
- Any node that contains a function call that unconditionally exits the program.

To compute the number of outside paths, the control flow graph is further augmented such that loops are skipped. For each loop, nodes that immediately precede the loop (*loop predecessors*) are redirected to point to the node that immediately follows the loop (*loop successor*). The same recursive depth-first search algorithm is used to count the unique paths.

The loop path count is computed using a control flow graph that represents the body of the loop. Starting with the acyclic control flow graph used to compute the baseline path count, here is the process of creating a loop control flow graph:

1. Identify all nodes that are part of the loop.
2. Create a dummy exit node *e*.
3. For each edge that is from a node *u* in the loop to a node *v* outside the loop, replace that edge with an edge from *u* to the dummy exit node *e*. Examples of such edges include the following:
    - The back edge from the end of the loop to the loop predicate.
    - Edges due to break and continue statements.
    - Edges of forward goto statements that jump out of the loop.

4. Denote the node containing the first statement of the loop body as a loop starting node. Other potential entry points due to goto targets and other unstructured control flow are not considered.
5. Denote the following nodes as a loop exit node:
    - The dummy exit node.
    - Any node containing a return statement.
    - Any node containing a backwards goto statement.
    - Any node that contains a function call that unconditionally exits the program.

The resulting graph is used to compute the loop path count with loops. To compute the loop path count without loops, each loop control flow graph is augmented such that inner loops are skipped. The procedure is identical to that of skipping loops in the function's control flow graph: loop predecessors are redirected to point to the loop successor. In both loop path counting algorithms, the same recursive depth-first search algorithm is used.

3.2 Loop Classifications

To better understand the behavior of different loops, we manually classified loops based on what they are trying to accomplish. The following seven different classifications emerged:

Array traversals: The loop consists of traversing an array that is not a string. Typically, the loop will stop when either the end of the array is encountered or, in the case of searches, when a desired element is found. The loops must traverse in sequential fashion (either forward or backward) but there are no further restrictions on the increment or decrement – it does not need to be a constant increment or decrement by one. For instance, a

manually unrolled loop that processes multiple elements per iteration is still classified as an array traversal. Loops can use index variables with array references or pointers with dereferences to traverse the loop.

String traversals: The loop traverses a string. Strings are treated separately from arrays because their loops are often written differently than other arrays. In particular, the size of the string and consequently the stopping condition of the loop is determined by the location of the null termination character, not the size of the array holding the string. In addition, loops traversing strings often take advantage of an extensive library of string-related functions.

Linked list traversals: The loop traverses a linked list of nodes. At the end of each iteration, a pointer moves to the next element in the list until reaching the end of the list. The loop may end early in loops that search for a desired element.

Other data structure traversals: The loop traverses a common data structure that is not an array, string, or linked list. In the programs in our study, this includes traversals of stacks, trees, heaps, and sets. Note that if a data structure was implemented using an array or linked list, loops traversing the data structure would still be classified as an array or linked list traversal.

Input loops: The loop is used to get data from input (either from the console, file, or network) and ends when there is no more input (such as the end of the file) or when a special "end-of-input" marker is reached. Loops that terminate early due to input errors are still classified as input loops. Loops that initialize arrays from input are classified as input loops even though they could also be classified as array traversals.

Not over loops: The loop stops when a certain algorithmic condition is met. Typically, but not necessarily, the condition is represented by a flag that is set within the body and checked in the loop predicate. A common type of a loop in this classification is a loop that continues until steady state is reached.

Other: The loop does not fit one of the above classifications or we were unable to determine whether it meets one of the above classifications.

Each loop is classified in exactly one of these categories. If a loop could be classified into multiple classifications, the following rules were used to disambiguate:

- If a loop processes input until there is no input left, it is an *input* loop even if the loop is placing the data in a data structure.
- If a loop is searching a data structure and the stopping condition is a flag that indicates the element is found, the loop is considered to be a data structure traversal rather than a *not over* loop.
- If the contents of a data structure are copied into a different type of data structure, it will be a data structure traversal of the data structure that is being read. For example, copying a linked list into an array would be considered a *linked list traversal*. There are very few loops of this type.
- The very few loops not covered by one of the situations above were handled on a case by case basis. Many of these were large loops that performed several different tasks; such loops were often classified as *other*.

For the data structure traversals, we also tracked the type of data structure traversal:

- ***Copy:*** A copy from one data structure to another.
- ***Initialize:*** The data structure is being initialized (but is not being copied from another data structure).
- ***Modify:*** The contents of the data structure are modified.
- ***Read:*** The contents of the data structure are read (used).

- ***Search:*** The data structure is being searched for a particular element or for an element that exhibits a certain property.
- ***Other:*** The data structure is being used for some other purpose than those listed above.

As with the initial classification, each loop is assigned exactly one traversal type. If a loop could fit into multiple categories, we made a decision based on what we felt the primary purpose of the loop was. For instance, a loop that traversed an array using the elements in a calculation but also searched for errors into the loop would be classified as a *read* traversal as the primary operation is to use the data, the search for errors was secondary. In the rare case where a loop clearly has multiple different primary purposes, it is classified as *other*.

The classification was done manually by inspecting the source code of each loop. Functions that are called within the loop were further inspected if it was necessary in classifying the loop. Fortunately, the vast majority of the loops did not require this type of inspection.

3.3 Loop Characteristics

The goal of this analysis is to determine how often loops contained "hard-to-analyze" features. In particular, we looked for the following:

Function calls: A loop that contains a function call (besides those that are classified separately: function calls that unconditionally exit and input/output system calls). In order to fully analyze a loop that calls another function, some level of interprocedural analysis is necessary. Path-based analysis tools often struggle with interprocedural analysis. Whether the function call occurs in the stopping condition, in the loop body, or both, is also tracked.

Alternate exits: A loop that can exit in another manner besides the specified stopping condition. This make loop analysis more difficult for two reasons. First, the stopping condition is spread out over the loop making it necessary to collect the conditions necessary for the loop to exit. Second, an alternate exit in the middle of the loop may only see some of the state changes present in the loop, missing state changes that occur after the alternate exit. This could make the creation and resulting complexity of a loop summary more complicated than a loop that only exits via the normal stopping condition. The programming construct used for an alternate exit is also noted: break statement, return statement, goto statement, or a function call that unconditionally exits the program.

Input: A loop that has a system input function call. These loops can be difficult to analyze due to the variability of the input data. Proper analysis often requires representing the input data symbolically.

Output: A loop that has a system output function call. Since these loops produce output, it can be difficult for an analysis tool to fully capture the full behavior of the loop.

Nested loops: A loop contains another loop. Furthermore, the depth of the nesting is recorded. Our analyses also notes loops that are nested within another loop.

This analysis was primarily carried out using an extension to CodeSurfer. The extension generates a list of basic blocks associated with each loop and scans the individual instructions within those basic blocks to determine which of the hard-to-analyze elements it contains. During this analysis, function calls were not inspected further to see if those functions contained one or more of these elements. However, some programs contained functions that were simply wrapper functions for exiting the program, gathering input, or sending output. For example, some programs have their own exit routines that consist of printing an error message in a consistent manner followed by exiting the program. For each of the three properties (exit, input, and output), a list of functions was created via inspection for each program. Our extension would read each of the lists. If a loop called a function on

the list, the loop is considered to have that property. For instance, if a loop calls a function that is on the input list, the loop is considered to have an input statement.

The only hard-to-analyze element not detected automatically is determining whether a function call is present in the stopping condition. Isolating the stopping condition in the CodeSurfer intermediate representation was too difficult in cases where the condition was converted into multiple if-else statements based on short-circuited operators and it was straightforward to inspect the loops manually using the source code.

4. Results

The 25 programs used in this study, written using the C programming language, are shown in Table 1. Programs range in size from 1,725 lines (*othello*) to 125,360 lines (*espresso*). In total, 2,869 loops were analyzed across the 25 programs. Some programmers use a dummy do-while loop with a constant predicate zero as a wrapper in macros; these (non)loops were discarded.

Table 1 also breaks down the numbers of loops by type (*do*, *for*, *while*, and *while(1)*). A *while(1)* loop is any loop that does not have a normal stopping condition such as `while(1)` or `for(;;)`. Since these loops do not use a stopping condition, we felt it was appropriate to consider these separately. 18 of the 25 programs have at least one *while(1)* loop. The most prevalent type of loop is the *for* loop (62.6% of the 2,869 loops).

Table 1. Programs Used

Name	*Description*	*Funcs*	*Lines*	*Loops*	*do*		*for*		*while*		*while (1)*	
barcode	barcode generator	61	15,634	71	0	(0.0%)	58	(81.7%)	13	(18.3%)	0	(0.0%)
bc	calculator	101	19,423	103	1	(1.0%)	33	(32.0%)	67	(65.0%)	2	(1.9%)
betaftpd	file transfer daemon	66	8,599	17	2	(11.8%)	6	(35.3%)	8	(47.1%)	1	(5.9%)
diff3	compares three files	29	7,471	53	10	(18.9%)	24	(45.3%)	19	(35.8%)	0	(0.0%)
ed	text editor	125	11,738	72	5	(6.9%)	21	(29.2%)	37	(51.4%)	9	(12.5%)
espresso	logic minimizer	401	125,360	738	114	(15.4%)	597	(80.9%)	21	(2.8%)	6	(0.8%)
find	file finder	295	25,565	50	1	(2.0%)	33	(66.0%)	16	(32.0%)	0	(0.0%)
flex	lexical analyzer	144	8,794	153	1	(0.7%)	116	(75.8%)	33	(21.6%)	3	(2.0%)
ft	spanning tree	37	4,706	23	7	(30.4%)	8	(34.8%)	8	(34.8%)	0	(0.0%)
ghttpd	web server	16	8,037	20	0	(0.0%)	8	(40.0%)	10	(50.0%)	2	(10.0%)
grep	text search tool	131	9,681	276	10	(3.6%)	143	(51.8%)	109	(39.5%)	14	(5.1%)
gzip	compression utility	92	15,174	181	31	(17.1%)	62	(34.3%)	81	(44.8%)	7	(3.9%)
http_get	basic HTTP client	8	2,785	9	0	(0.0%)	4	(44.4%)	3	(33.3%)	2	(22.2%)
indent	source code indenter	103	18,433	109	10	(9.2%)	33	(30.3%)	59	(54.1%)	7	(6.4%)
ks	graph partitioning	13	2,302	35	1	(2.9%)	33	(94.3%)	1	(2.9%)	0	(0.0%)
make	makefile processor	179	13,756	349	11	(3.2%)	200	(57.3%)	133	(38.1%)	5	(1.4%)
othello	othello game	11	1,725	26	2	(7.7%)	19	(73.1%)	4	(15.4%)	1	(3.8%)
sed	text filter tool	246	10,834	220	13	(5.9%)	152	(69.1%)	35	(15.9%)	20	(9.1%)
space	specialized interpreter	136	7,094	53	1	(1.9%)	15	(28.3%)	35	(66.0%)	2	(3.8%)
spell	spell checker	13	2,538	10	0	(0.0%)	3	(30.0%)	3	(30.0%)	4	(40.0%)
sudoku	sudoku solver	47	4,371	59	0	(0.0%)	54	(91.5%)	4	(6.8%)	1	(1.7%)
thttpd	web server	126	16,877	83	1	(1.2%)	50	(60.2%)	22	(26.5%)	10	(12.0%)
time	time measuring tool	12	5,090	7	0	(0.0%)	2	(28.6%)	5	(71.4%)	0	(0.0%)
which	program finder	25	3,845	29	3	(10.3%)	10	(34.5%)	15	(51.7%)	1	(3.4%)
yacr2	channel router	58	8,061	123	6	(4.9%)	112	(91.1%)	5	(4.1%)	0	(0.0%)
	TOTAL	2,475	357,893	2,869	230	(8.0%)	1,796	(62.6%)	746	(26.0%)	97	(3.4%)

Table 2. Number of Loops by Classification

Name	Loops	array		string		linked list		ds: other		input		not over		other	
barcode	71	15	(21.1%)	52	(73.2%)	0	(0.0%)	0	(0.0%)	2	(2.8%)	0	(0.0%)	2	(2.8%)
bc	103	52	(50.5%)	9	(8.7%)	14	(13.6%)	2	(1.9%)	3	(2.9%)	2	(1.9%)	21	(20.4%)
betaftpd	17	7	(41.2%)	2	(11.8%)	1	(5.9%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	7	(41.2%)
diff3	53	20	(37.7%)	17	(32.1%)	6	(11.3%)	0	(0.0%)	6	(11.3%)	1	(1.9%)	3	(5.7%)
ed	72	16	(22.2%)	23	(31.9%)	15	(20.8%)	0	(0.0%)	4	(5.6%)	1	(1.4%)	13	(18.1%)
espresso	738	390	(52.8%)	0	(0.0%)	101	(13.7%)	146	(19.8%)	18	(2.4%)	1	(0.1%)	82	(11.1%)
find	50	19	(38.0%)	6	(12.0%)	9	(18.0%)	5	(10.0%)	2	(4.0%)	1	(2.0%)	8	(16.0%)
flex	153	101	(66.0%)	16	(10.5%)	2	(1.3%)	0	(0.0%)	1	(0.7%)	0	(0.0%)	33	(21.6%)
ft	23	4	(17.4%)	0	(0.0%)	8	(34.8%)	5	(21.7%)	0	(0.0%)	1	(4.3%)	5	(21.7%)
ghttpd	20	3	(15.0%)	7	(35.0%)	0	(0.0%)	0	(0.0%)	5	(25.0%)	0	(0.0%)	5	(25.0%)
grep	276	130	(47.1%)	27	(9.8%)	4	(1.4%)	9	(3.3%)	3	(1.1%)	2	(0.7%)	101	(36.6%)
gzip	181	118	(65.2%)	12	(6.6%)	0	(0.0%)	0	(0.0%)	3	(1.7%)	4	(2.2%)	44	(24.3%)
http_get	9	2	(22.2%)	4	(44.4%)	1	(11.1%)	0	(0.0%)	2	(22.2%)	0	(0.0%)	0	(0.0%)
indent	109	16	(14.7%)	62	(56.9%)	5	(4.6%)	0	(0.0%)	8	(7.3%)	5	(4.6%)	13	(11.9%)
ks	35	8	(22.9%)	1	(2.9%)	25	(71.4%)	0	(0.0%)	0	(0.0%)	1	(2.9%)	0	(0.0%)
make	349	72	(20.6%)	94	(26.9%)	121	(34.7%)	0	(0.0%)	7	(2.0%)	0	(0.0%)	55	(15.8%)
othello	26	17	(65.4%)	1	(3.8%)	0	(0.0%)	0	(0.0%)	1	(3.8%)	1	(3.8%)	6	(23.1%)
sed	220	102	(46.4%)	36	(16.4%)	19	(8.6%)	7	(3.2%)	16	(7.3%)	0	(0.0%)	40	(18.2%)
space	53	11	(20.8%)	0	(0.0%)	28	(52.8%)	0	(0.0%)	3	(5.7%)	1	(1.9%)	10	(18.9%)
spell	10	1	(10.0%)	3	(30.0%)	0	(0.0%)	0	(0.0%)	3	(30.0%)	0	(0.0%)	3	(30.0%)
sudoku	59	46	(78.0%)	0	(0.0%)	2	(3.4%)	0	(0.0%)	0	(0.0%)	1	(1.7%)	10	(16.9%)
thttpd	83	23	(27.7%)	22	(26.5%)	11	(13.3%)	0	(0.0%)	8	(9.6%)	0	(0.0%)	19	(22.9%)
time	7	3	(42.9%)	2	(28.6%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	2	(28.6%)
w hich	29	9	(31.0%)	13	(44.8%)	0	(0.0%)	0	(0.0%)	2	(6.9%)	0	(0.0%)	5	(17.2%)
yacr2	123	112	(91.1%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	2	(1.6%)	3	(2.4%)	6	(4.9%)
TOTAL	2,869	1,297	(45.2%)	409	(14.3%)	372	(13.0%)	174	(6.1%)	99	(3.5%)	25	(0.9%)	493	(17.2%)

4.1 Loop Classifications

Looking at the loop classifications presented in Table 2, nearly 60% of the loops traverse arrays in some fashion: 45.2% traverse non-string arrays while 14.3% traverse string arrays. Linked lists account for 13.0% of the loops. Only six of the programs have loops that traverse other data structures. The program *espresso* accounted for most of these loops as it contains a set data structure represented as a bit vector that is used extensively in the program. The program uses macros for many of these operations so the loops are replicated throughout the code. Other programs relied on functions for common data structure operations so most of the loops are constrained to the functions that implemented those data structure operations. 17.2% of the loops could not be classified using one of our six categories and are classified as *other*.

Exploring the four types of data structure traversal loops further, Table 3 shows the number of loops based on the type of traversal. Over half of the traversals are *read* operations – the contents of the data structure are used in a computation or task. The breakdown among the different types of data structures is intuitive. Arrays are unique in that they are often initialized using a loop where the other data structures do not need to use a loop to be initialized. For instance, a string can be initialized using a constant string literal. Searches are more common for strings than other data structures; frequently loops look for delimiter characters. The lack of search loops for other data structures (*ds: other*) can be attributed to that most of these loops operate on sets in *espresso*. Most of the common set operations such as union or intersection do not involve searches.

Table 3. Number of Loops by Type of Data Structure Traversal

Type	*Loops*	*copy*	*initialize*	*modify*	*read*	*search*	*other*
array	1,297	69 (5.3%)	173 (13.3%)	200 (15.4%)	654 (50.4%)	190 (14.6%)	11 (0.8%)
string	409	38 (9.3%)	10 (2.4%)	15 (3.7%)	147 (35.9%)	170 (41.6%)	29 (7.1%)
linked list	372	0 (0.0%)	4 (1.1%)	44 (11.8%)	252 (67.7%)	39 (10.5%)	33 (8.9%)
ds: other	174	0 (0.0%)	1 (0.6%)	38 (21.8%)	105 (60.3%)	5 (2.9%)	25 (14.4%)
TOTAL	2,252	107 (4.8%)	188 (8.3%)	297 (13.2%)	1,158 (51.4%)	404 (17.9%)	98 (4.4%)

Table 4. Path Counts by Program

Program	*Baseline Path Count*	*Loop-Separate Path Count*		
		Total	*Outside Loops*	*Inside Loops*
barcode	8.07E+09	32,927,520	32,926,918	602
bc	949,346	34,101	33,623	478
betaftpd	45,692	42,311	42,205	106
diff3	572,718	40,095	38,864	1,231
ed	210,545	36,531	35,871	660
espresso	2.71E+11	4.36E+09	1,205,009	4.36E+09
find	1,966,770	1,791,895	1,790,964	931
flex	7.40E+11	7.22E+11	7.22E+11	1,396
ft	10,594	526	481	45
ghttpd	9,487	1,154	1,075	79
grep	6.97E+20	29,917,218	262,245	29,654,973
gzip	3.05E+10	2.37E+09	2.37E+09	873
http_get	4,501	347	160	187
indent	9.82E+17	5.59E+11	5.59E+11	30,420,548
ks	24,452	153	47	106
make	1.09E+20	6.30E+12	6.29E+12	1.48E+09
othello	13,382	13,196	13,029	167
sed	4.37E+12	34,872,378	28,863,158	6,009,220
space	6,227	2,007	1,672	335
spell	5,225	3,245	2,923	322
sudoku	1.94E+09	21,216	10,099	11,117
thttpd	2.84E+12	41,622,428	41,612,355	10,073
time	1,406	243	152	91
which	1.98E+10	5,945	2,031	3,914
yacr2	2,249,048	3,104	1,575	1,529

4.2 Loop-Separate Path Count Comparison

Table 4 compares the baseline path count to the loop-separate path count for each program. The path count for a program is obtained by summing the path counts for each of the functions contained within the program. The first column of Table 4 shows the baseline path count. The second column displays the loop-separate path count. This number is broken down into the number of paths outside any loops and the number of paths inside the loops.

The results of this experiment varied widely. Some programs such as *ks*, *grep*, *sudoku*, *which*, *yacr2* saw a significant reduction in the number of paths. In particular, *which* went from almost 2 billion baseline paths to 5,945 loop-separate paths. All but 2,194 of the baseline paths are contained in two functions: `main` (1.97 billion baseline paths and 8 loops) and `process_alias` (88 million baseline paths and 10 loops). With a high number of loops, these programs saw a large benefit from loop-separate analysis. The function `main` has 3,705

loop-separate paths and the function `process_alias` has 1,327 loop-separate paths. The function `process_alias`, shown in Figure 7, serves as a good example of a function that benefits from loop-separate analysis. It contains many simple string search loops that increment a pointer until a condition is met. Even limiting the number of iterations to at most one, the sheer number of these loops generates many combinations that are encapsulated in over 88 million paths. Using loop-separate analysis reduces the path count to 1,327 paths – most of them in the large for loop that increments `argv`. The program *ks* experiences a similar phenomenon. 23,100 of its 24,452 baseline paths resided in the function `PrintResults`. The function with its 11 relatively simple loops, saw a large benefit from loop-separate analysis. It only contains 22 loop-separate paths.

Eleven of the programs have over one billion baseline paths. When using loop-separate analysis: five of the eleven programs (*espresso*, *flex*, *gzip*, *indent*, *make*) still have over 1 billion paths, four programs (*barcode*, *grep*, *sed*, *thttpd*) coincidentally have around 30-40 million paths, and two programs (*sudoku*, *which*) have fewer than 25,000 paths.

In most programs, the number of paths is driven by the number of paths outside the loops. Of the nine programs with over 10 million paths using loop-separate analysis, seven of those programs have significantly more paths outside the loops than inside loops. Even the two programs that have more paths inside loops still have a significant number of paths outside loops: 1.2 million (*espresso*) and 262,245 paths (*grep*). The implication of these results is that many programs have significant complexity outside of loops. If loops were to be analyzed separately, a path-based analysis tool would still be required to analyze large numbers of paths.

In the 11 programs with over one billion baseline paths, the large majority of the paths are contained in just a few, sometimes one, function. The effectiveness of loop-separate analysis on these few functions determines the size of the decrease for the whole program. To that end, we explore the effect on individual functions instead of whole programs. The results are tabulated in Table 5. In this table, the rows are organized by the baseline path count divided into exponentially-size ranges. The second column shows the number of functions that have a path count with loops in the specified range. The third column shows the number of functions that have no loops. For instance, there are 205 functions with a baseline path count from 101 – 1,000. Of those 205 functions, 32 have no loops. Functions with no loops receive no benefit from using loop-separate analysis. Out of the 2,475 functions under analysis, 1,315 (53%) have no loops. Most of these are simple functions with very few paths. For functions that have over 100 paths, only 13% of the functions have no loops. Out of the 57 functions that contain over one million paths, only three (5%) have no loops.

Table 5. Effect of Loop-Separate Analysis

Baseline Path Count	Functions		Functions with Loop-Separate Path Counts of:									
	Total	No Loops	1	2 - 10	11 - 100	101 - 1k	1k - 10k	10k- 100k	100k - 1M	1M - 10M	10M - 100M	> 100M
1	477	461	*461*	16	0	0	0	0	0	0	0	0
2 - 10	1,151	665	0	*1,150*	1	0	0	0	0	0	0	0
11 - 100	435	137	0	121	*314*	0	0	0	0	0	0	0
101 - 1k	205	32	0	1	140	*64*	0	0	0	0	0	0
1k - 10k	76	11	0	0	25	30	*21*	0	0	0	0	0
10k- 100k	48	5	0	0	9	19	8	*12*	0	0	0	0
100k - 1M	26	1	0	0	6	6	6	6	*2*	0	0	0
1M - 10M	28	1	0	0	0	5	2	4	5	*2*	0	0
10M - 100M	11	0	0	0	0	0	3	2	1	3	*2*	0
> 100M	28	2	0	0	0	0	3	4	4	2	5	*10*
TOTAL	2,485	1315 52.9%	461 18.6%	1288 51.8%	495 19.9%	124 5.0%	43 1.7%	28 1.1%	12 0.5%	7 0.3%	7 0.3%	10 0.4%

```
void process_alias(const char *str, int argc, char *argv[], const char *path_list, int
function_start_type)
{
  const char *p = str;
  int len = 0;

  while(*p == ' ' || *p == '\t')
    ++p;
  if (!strncmp("alias", p, 5))
    p += 5;
  while(*p == ' ' || *p == '\t')
    ++p;
  while(*p && *p != ' ' && *p != '\t' && *p != '=')
    ++p, ++len;

  for (; argc > 0; --argc, ++argv) {
    char q = 0;
    char *cmd;

    if (!*argv || len != strlen(*argv) || strncmp(*argv, &p[-len], len))
      continue;

    fputs(str, stdout);

    if (!show_all)
      *argv = NULL;

    while(*p == ' ' || *p == '\t')
      ++p;
    if (*p == '=')
      ++p;
    while(*p == ' ' || *p == '\t')
      ++p;
    if (*p == '"' || *p == '\'')
      q = *p, ++p;

    for(;;){
      int found = 0;

      while(*p == ' ' || *p == '\t')
        ++p;
      len = 0;
      while(*p && *p != ' ' && *p != '\t' && *p != q && *p != '|' && *p != '&')
        ++p, ++len;

      cmd = (char *)xmalloc(len + 1);
      strncpy(cmd, &p[-len], len);
      cmd[len] = 0;
      if (*argv && !strcmp(cmd, *argv))
        *argv = NULL;
      if (read_functions && !strchr(cmd, '/'))
        found = func_search(1, cmd, functions, function_start_type);
      if (show_all || !found)
        path_search(1, cmd, path_list);
      free(cmd);

      while(*p && (*p != '|' || p[1] == '|') && (*p != '&' || p[1] == '&'))
        ++p;

      if (!*p)
        break;

      ++p;
    }

    break;
  }
}
```

Figure 7. Example of a Function that benefits from Loop-Separate Analysis (from the program *which*)

The right side of Table 5 shows the number of functions that have a loop-separate path count in the specified range (the same exponentially-sized ranges are used here). For instance, of the 76 functions that have a baseline path count of 1k – 10k paths: none have fewer than 11 paths when loop-separate analysis is used, 25 have between 11 and 100 paths, 30 have between 101 and 1k paths, and 21 still have between 1k and 10k paths. This shows that the number of paths has gone down at least one order of magnitude for 55 of the 76 functions represented in this row. However, the reduction was less significant or non-existent for the remaining 21 functions. Of these 21 functions, 11 contain no loops and saw no difference in the number of paths when loop-separate analysis is applied.

The main diagonal in Table 5 is highlighted in gray. This diagonal shows the number of functions where the use of the loop-separate analysis was minimal or non-existent. Values to the left of this diagonal show the number of functions that benefit from using loop-separate analysis. Functions that are furthest most to the left show the largest benefit. For functions with smaller path counts, there are several functions that have minimal or no benefit when using path-separate analysis (many because they do not have any loops) but there are also several functions that see a decrease of one to two orders of magnitude. For functions with larger path counts, more functions see a benefit to using loop-separate analysis and the extent of that benefit is larger (more orders of magnitude). However, there is a limit to the benefit obtained using loop-separate analysis as evidenced by the zeros in the lower left portion of the table. As an example, all of the functions that have at least 1 million paths with loops all have at least 101 paths when using loop-separate analysis. Similarly, all of the functions that have at least 10 million paths with loops all have at least 1,000 paths when using loop-separate analysis. The reason for this result is that virtually all of the functions with large path counts have significant complexity outside the loops serving as a limit on how low the loop-separate path count can be.

The area to right of the main diagonal in Table 5 shows cases where loop-separate analysis increases the path count such that the resulting count is in a higher range. The loop-separate path count can actually be higher in functions where do-while loops are used. The path-counting algorithm iterates through a do-while loop exactly once since do-while cannot execute zero times. (Unlike other infeasible paths that are dependent on data analysis, this restriction is encoded in the control flow graph.) As an example, a function that contains a single do-while loop and no other control statements contains exactly one baseline path but has two paths when loop-separate analysis is employed: one outside path and one inside path for analyzing the do-while loop separately. Similar scenarios can occur with functions with higher path counts and multiple do-while loops. In total, 73 functions saw an increase in the number of paths when using loop-separate analysis. Of these 73 functions, 54 functions saw an increase of only one path and the maximum increase was seven paths.

The results from Table 5 show that loop-separate analysis can help in some functions, but the effectiveness is limited due to the complexity outside the loops. Looking further, Table 6 breaks down the 28 functions that have a baseline path count of over 100 million paths, presented in decreasing order. Functions 12 and 26 have no loops – the number of paths remains the same when using loop-separate analysis. Both of these functions are in the program *flex* and both make several decisions based on different global flags that correspond to a particular command line setting. Many of the other functions in the list also have a large number of decisions based on command line settings, especially the five `main` functions in the list.

Apart from the two functions without loops, all of the functions saw a lower loop-separate path count. In the worst case, the number of paths decreased by one or two orders of magnitude. This occurred in eight functions (13, 15, 16, 18, 19, 21, 22, 28). In five of these functions (15, 16, 19, 21, 22), there is a significant number of paths outside the loops and relatively few paths within the loops. The reverse was true for functions 13, 18, and 28. All contain a loop that has over one million paths. The functions contain very little complexity outside these giant loops – all have four or fewer paths outside loops.

Table 6. Paths Breakdown for Functions with over 100 Million Paths

	Program	Function	Loops	Baseline Path Count	Loop-Separate Path Count		
					Total	Outside Loops	Inside Loops
1.	grep	regex_compile	101	6.97E+20	28,641,328	21	28,641,307
2.	make	main	15	1.09E+20	6.29E+12	6.29E+12	75
3.	indent	dump_line	18	9.82E+17	5.59E+11	5.59E+11	82
4.	grep	re_match_2_internal	85	1.00E+16	236,452	7,609	228,843
5.	make	update_file_1	32	8.18E+14	109,518	109,204	314
6.	grep	dfastate	22	5.40E+14	73,832	10	73,822
7.	make	pattern_search	17	1.93E+14	3,092	180	2,912
8.	make	read_makefile	15	1.59E+14	1.35E+09	4,896	1.35E+09
9.	indent	print_comment	8	8.22E+12	961,670	959,088	2,582
10.	sed	re_search_internal	9	4.37E+12	28,802,514	28,798,740	3,774
11.	thttpd	main	4	2.84E+12	41,058,147	41,057,280	867
12.	flex	flexend	0	7.22E+11	7.22E+11	7.22E+11	0
13.	espresso	massive_count	5	2.70E+11	4.36E+09	1	4.36E+09
14.	make	record_files	6	1.89E+11	6,002,863	4	6,002,859
15.	indent	lexi	14	4.46E+10	1.66E+08	1.66E+08	86
16.	gzip	get_method	4	3.05E+10	2.36E+09	2.36E+09	12
17.	which	main	8	1.97E+10	3,705	1,872	1,833
18.	make	reap_children	3	7.90E+09	1.23E+08	3	1.23E+08
19.	flex	flexinit	5	7.67E+09	40,824,046	40,824,000	46
20.	flex	gentabs	10	5.29E+09	19,485	19,440	45
21.	barcode	main	2	5.14E+09	32,514,228	32,514,048	180
22.	flex	make_tables	3	4.53E+09	4.53E+08	4.53E+08	5
23.	barcode	Barcode_ps_print	6	2.89E+09	31,180	31,106	74
24.	sudoku	main	4	1.94E+09	12,122	1,728	10,394
25.	flex	ntod	14	9.64E+08	1,017	864	153
26.	flex	readin	0	2.14E+08	2.14E+08	2.14E+08	0
27.	grep	state_index	5	1.27E+08	589,493	8	589,485
28.	sed	compile_program	7	1.11E+08	6,001,369	4	6,001,365

The eight functions with the most paths at the top of the list all have at least 15 loops and over 100 trillion baseline paths. As a result of having many loops, using loop-separate analysis was very beneficial for these functions. In four of these cases, the resulting path count was less than 250,000 paths – a much more manageable number than 100 trillion paths. The other functions saw significant decreases in the number of paths but the resulting path counts are still extremely high. For instance, function 2 (`main` in *make*) still has 6 trillion loop-separate paths.

Though data is not presented, results are similarly varied for functions with less than 100 million paths: some functions see significant path count decreases, some functions have loops with large path counts, in some functions a majority of the paths are outside the loops, and some functions have no loops at all. Functions with fewer paths tend to have fewer loops. Since using loop-separate analysis works best with functions with many loops, the decreases are less significant in functions with fewer paths (as shown in Table 5).

To conclude this part of the study, using loop-separate analysis can significantly reduce the number of paths, especially in functions with many loops. In several functions, the number of paths can be reduced from an unmanageable number to a much more manageable number. However, there are three types of loops where the benefit is mitigated or non-existent:

- *The function contains no loops.* Fortunately, the functions with a large baseline path counts typically contain multiple loops. However, there are two functions in this study that contain over 100 million paths but no loops.
- *The function contains a giant loop with many paths.* Fortunately, as noted in the next section, loops tend to have small paths counts. However, there are two loops in this study that contain over 100 million paths even when skipping inner loops.
- *The function contains an extremely large number of paths.* Even with a significant decrease in the number of paths when employing loop-separate analysis, the number of paths that remain might be too large.

4.3 Loop Path Count

In this section, we explore the complexity of the loops themselves using the number of paths through the loop body as the metric. The results are summarized in Table 7. The table is divided into two sections based on the two variants for counting paths in a loop – the left section displays results for the loop path count with loops (inner loops are traversed at most once) and the right section displays results for the loop path count without loops (inner loops are skipped). Within each section, the loops are partitioned into exponentially-sized buckets based on the number of paths. Also, the number of paths in the loop with the highest number of paths is shown for each program.

Most loops are simple. Using the loop path count with loops, 38.1% of the loops are very basic and have a single path. Based on our observations, most of these loops accomplish simple things such as initializing an array, printing the contents of a data structure, or moving a pointer to a particular point in a string. Looking further, 46.3% of the loops have two to ten paths. Combining these first two columns, 84% of the loops have ten or fewer paths. In five of the programs (*betaftpd*, *ft*, *ghttpd*, *ks*, and *time*), all the loops have fewer than 100 paths with the largest loop in *ft* having a mere 20 paths. This suggests that it is likely possible to analyze many of the loops present in these programs.

There are very few loops with a large number of paths. Only 37 (1.3%) loops analyzed have over 10,000 paths with loops. 23 of these 37 loops are in the programs *grep* and *make*. These two programs each have two loops with an astronomical number of paths. The program *grep* contains one loop with 4.84×10^{18} paths and another that contains 9.80×10^{12} paths. The program *make* contains loops with 4.73×10^{10} paths and 1.62×10^{10} paths. Only 7 of the other 23 programs contain loops with more than 10,000 paths.

When inner loops are skipped, the number of loops with ten or fewer paths increases to 90% and there are even fewer loops with large path counts. Only 17 loops across 6 different programs have more than 10,000 paths. The impact on analyzing inner loops separately differs from loop to loop and depends on the number of inner loops and the number of paths within these inner loops. For instance, the two loops in *grep* that contain 4.84×10^{18} paths and 9.80×10^{12} paths have several inner loops. Their path counts decreased to 287,777 and 7,833 paths respectively when inner loops are analyzed separately – large improvements. However, *grep* contains a third loop that has over 28 million paths and no inner loops.

The loop with the largest number of paths that contains no inner loops is in the program *espresso* with 4,362,470,402 paths. The contents of the loop are largely uninteresting – it primarily consists of checking each bit of an array element and incrementing an appropriate counter. While this checking and updating could be done with another loop, the code was written as an unrolled loop consisting of 32 checks like this:

```
if (val & 0x00080000)
 cnt[19]++;
```

Table 7. Loop Path Counts by Program

Program	Loop Path Count with Loops							Loop Path Count without Loops						
	Most Paths in Loop	Percent of Loops with Paths Counts of ..						Most Paths in Loop	Percent of Loops with Paths Counts of ..					
		1	2 - 10	11 - 100	101 - 1k	1k - 10k	> 10k		1	2 - 10	11 - 100	101 - 1k	1k - 10k	> 10k
barcode	160	25.4%	57.7%	15.5%	1.4%	0.0%	0.0%	160	25.4%	60.6%	12.7%	1.4%	0.0%	0.0%
bc	672	49.5%	41.7%	5.8%	2.9%	0.0%	0.0%	119	52.4%	41.7%	4.9%	1.0%	0.0%	0.0%
betaftpd	37	29.4%	52.9%	17.6%	0.0%	0.0%	0.0%	37	29.4%	58.8%	11.8%	0.0%	0.0%	0.0%
diff3	9,608	49.1%	34.0%	7.5%	5.7%	3.8%	0.0%	581	54.7%	32.1%	7.5%	5.7%	0.0%	0.0%
ed	164	34.7%	43.1%	19.4%	2.8%	0.0%	0.0%	120	37.5%	43.1%	18.1%	1.4%	0.0%	0.0%
espresso	4.36E+09	43.4%	49.2%	6.1%	0.9%	0.1%	0.3%	4.36E+09	51.6%	45.7%	2.4%	0.1%	0.0%	0.1%
find	396	40.0%	44.0%	10.0%	6.0%	0.0%	0.0%	396	40.0%	48.0%	6.0%	6.0%	0.0%	0.0%
flex	8,448	41.2%	47.7%	7.8%	2.0%	1.3%	0.0%	464	45.8%	48.4%	3.9%	2.0%	0.0%	0.0%
ft	20	43.5%	47.8%	8.7%	0.0%	0.0%	0.0%	4	52.2%	47.8%	0.0%	0.0%	0.0%	0.0%
ghttpd	23	45.0%	45.0%	10.0%	0.0%	0.0%	0.0%	23	45.0%	45.0%	10.0%	0.0%	0.0%	0.0%
grep	4.84E+18	36.2%	39.1%	17.0%	3.3%	1.1%	3.3%	28,344,976	48.2%	40.6%	5.8%	2.2%	0.7%	2.5%
gzip	20,883	42.5%	45.9%	7.7%	2.8%	0.0%	1.1%	198	48.1%	45.3%	6.1%	0.6%	0.0%	0.0%
http_get	102	22.2%	44.4%	22.2%	11.1%	0.0%	0.0%	100	22.2%	55.6%	22.2%	0.0%	0.0%	0.0%
indent	43,830,540	52.3%	33.9%	8.3%	0.9%	1.8%	2.8%	30,352,140	53.2%	33.9%	9.2%	0.9%	0.9%	1.8%
ks	72	40.0%	42.9%	17.1%	0.0%	0.0%	0.0%	36	54.3%	40.0%	5.7%	0.0%	0.0%	0.0%
make	4.73E+10	35.5%	43.6%	8.9%	4.9%	3.2%	4.0%	1,353,452,844	44.1%	41.0%	9.2%	3.2%	1.1%	1.4%
othello	121	42.3%	50.0%	3.8%	3.8%	0.0%	0.0%	121	69.2%	26.9%	0.0%	3.8%	0.0%	0.0%
sed	27,860,547	24.1%	54.5%	15.9%	3.2%	1.4%	0.9%	6,001,347	27.3%	54.5%	14.5%	2.3%	0.9%	0.5%
space	1,252	30.2%	60.4%	7.5%	0.0%	1.9%	0.0%	164	43.4%	47.2%	7.5%	1.9%	0.0%	0.0%
spell	273	30.0%	20.0%	20.0%	30.0%	0.0%	0.0%	165	30.0%	20.0%	30.0%	20.0%	0.0%	0.0%
sudoku	41,480	22.0%	47.5%	13.6%	11.9%	1.7%	3.4%	10,368	32.2%	52.5%	11.9%	1.7%	0.0%	1.7%
thttpd	17,267	28.9%	53.0%	9.6%	4.8%	2.4%	1.2%	5,768	33.7%	53.0%	8.4%	2.4%	2.4%	0.0%
time	77	57.1%	28.6%	14.3%	0.0%	0.0%	0.0%	77	57.1%	28.6%	14.3%	0.0%	0.0%	0.0%
which	979,887	44.8%	24.1%	13.8%	6.9%	3.4%	6.9%	1,767	44.8%	27.6%	13.8%	6.9%	6.9%	0.0%
yacr2	3,537	27.6%	49.6%	14.6%	6.5%	1.6%	0.0%	392	37.4%	52.8%	6.5%	3.3%	0.0%	0.0%
OVERALL	4.84E+18	38.1%	46.3%	10.2%	3.0%	1.1%	1.3%	4.36E+09	45.0%	45.2%	7.0%	1.7%	0.5%	0.6%

Table 8 breaks down the number of paths by different kinds of loop. Out of the four loop types, *while(1)* loops tend to have more paths and *do* loops tend to be smaller. When comparing the different classifications, traversals of data structures tend to have lower path counts than those that did not. Based on manual observations, most of the traversal functions are simple – accomplishing a single straight-forward task. Many of the loops with high paths counts classified as *other* are associated with parsing.

There is a wider disparity in results when looking at the different types of data structure traversals (the bottom section of Table 6). *Copy* and *initialize* operations have very few paths. In both cases, over 85% of the loops have only one path. Many of these loop bodies consist of a single statement that copies or initializes the array element. All *copy* operations and all but two *initialize* operations have ten paths or less. *Modify* operations also typically have relatively few paths. Several have more than one path. A relatively common scenario is to selectively modify an element based on a condition. These loops often have two paths in their bodies – one where the loop element is modified and one where the loop element is not. *Read* operations are more mixed. The complexity depends on what was done with the value after it is read from the data structures – this varies from loop to loop and program to program. *Search* operations also have very few paths. Most of the loops have from two to ten paths. Typically there is at least one path corresponding to finding the element being searched for that iteration and at least one

Table 8. Loop Path Counts by Type

Loop Type	Loop Path Count with Loops							Loop Path Count without Loops						
	Most Paths in Loop	Percent of Loops with Paths Counts of ..						Most Paths in Loop	Percent of Loops with Paths Counts of ..					
		1	2 - 10	11 - 100	101 - 1k	1k - 10k	> 10k		1	2 - 10	11 - 100	101 - 1k	1k - 10k	> 10k
do	10,056	52.2%	34.8%	9.6%	2.2%	0.9%	0.4%	735	53.9%	37.0%	7.4%	1.7%	0.0%	0.0%
for	4.73E+10	35.4%	51.9%	8.3%	2.3%	1.0%	1.1%	4.36E+09	43.5%	49.2%	5.2%	1.3%	0.2%	0.4%
while	4.84E+18	45.2%	35.8%	12.9%	3.9%	0.8%	1.5%	1.35E+09	51.5%	36.2%	8.8%	2.1%	0.7%	0.7%
while(1)	9.80E+12	0.0%	49.5%	27.8%	11.3%	5.2%	6.2%	3.04E+07	2.1%	58.8%	24.7%	6.2%	4.1%	4.1%
array	4.36E+09	41.0%	48.5%	6.9%	1.9%	0.8%	0.8%	4.36E+09	48.0%	45.6%	4.9%	0.9%	0.3%	0.3%
string	555	48.4%	42.3%	7.6%	1.7%	0.0%	0.0%	405	48.7%	43.5%	7.1%	0.7%	0.0%	0.0%
linked list	4.73E+10	36.6%	48.1%	7.3%	3.5%	2.2%	2.4%	6,002,667	47.3%	42.7%	6.2%	2.4%	0.5%	0.8%
ds: other	228	35.1%	53.4%	10.9%	0.6%	0.0%	0.0%	228	43.7%	54.0%	1.7%	0.6%	0.0%	0.0%
input	1.62E+10	19.2%	54.5%	14.1%	7.1%	1.0%	4.0%	1.35E+09	22.2%	56.6%	14.1%	4.0%	1.0%	2.0%
not over	43,830,540	4.0%	28.0%	16.0%	28.0%	12.0%	12.0%	30,352,140	8.0%	40.0%	28.0%	12.0%	4.0%	8.0%
other	4.84E+18	29.4%	39.1%	22.1%	5.5%	1.6%	2.2%	123,458,340	39.6%	42.0%	12.6%	3.7%	1.0%	1.2%
copy	7	89.7%	10.3%	0.0%	0.0%	0.0%	0.0%	6	89.7%	10.3%	0.0%	0.0%	0.0%	0.0%
initialize	28,344,976	86.7%	12.2%	0.5%	0.0%	0.0%	0.5%	28,344,976	91.0%	8.0%	0.5%	0.0%	0.0%	0.5%
modify	169	62.0%	35.7%	2.0%	0.3%	0.0%	0.0%	18	64.6%	34.7%	0.7%	0.0%	0.0%	0.0%
read	4.73E+10	26.1%	56.3%	11.6%	3.1%	1.5%	1.5%	4.36E+09	36.9%	52.0%	8.5%	1.8%	0.4%	0.4%
search	9,306	36.9%	60.1%	1.7%	1.0%	0.2%	0.0%	308	37.6%	60.4%	1.2%	0.7%	0.0%	0.0%
other	10,368	33.7%	39.8%	19.4%	5.1%	1.0%	1.0%	10,368	35.7%	49.0%	12.2%	1.0%	1.0%	1.0%

path corresponding to not finding the element that iteration. Data structure traversals classified as *other* tend to have more paths, largely because these loops tend to use the data structures in complex and multiple ways.

These results suggest that loops that perform *copy*, *initialize*, *modify*, and *search* data structure operations are loops that can be effectively analyzed and summarized due to their low complexity. The complexity of the other types of loops are more varied and more dependent on the functionality of the specific program.

4.4 Loop Characteristics

Table 9 provides a count of how many loops meet the "hard-to-analyze" characteristics described in Section 3.3. Table 10 shows the same breakdown but divides the loops by the number of paths while Table 11 displays the characteristics by type of loop. The three tables are divided into five sections: function calls, alternate exits, input, output, and nesting.

4.4.1 Function Calls

From Table 9, over half of the loops (51.3%) contain a function call. Looking at the data in Table 10, loops with more paths are more likely to contain a function call. Only 28.6% of the loops with one path contain a function call. Of the loops with 2-10 paths, 58.1% contain a function call. But when considering loops with 11 or more paths, 85% of the loops contain a function call. These results strongly suggest that interprocedural analysis is necessary in order to fully analyze loops.

With a few exceptions, the type of loop does not seem to make much of a difference in whether a loop contains a function call. From Table 11, *while(1)*, *input*, and *not over* loops have a higher percentage of loops with a function call. This makes sense for *input* loops since virtually all of them have a function call that gathers the input. Referring back to Table 8, *while(1)* and *not over* loops tend to be more complex with more paths. The fact that these type of loops have more paths is consistent with the results earlier that found that loops with more paths are

Table 9. Loop Characteristics by Program

Program	Num Loops	Function Call			Alternate Exit					Input	Output	Nesting	
		Yes	Location		Yes	Contains				Yes	Yes	Has Nested	Is Nested
			Body	Cond		Break	Return	Exit	Goto				
barcode	71	74.6%	69.0%	19.7%	47.9%	14.1%	33.8%	1.4%	0.0%	2.8%	15.5%	8.5%	8.5%
bc	103	49.5%	48.5%	5.8%	11.7%	3.9%	4.9%	4.9%	2.9%	2.9%	5.8%	15.5%	27.2%
betaftpd	17	82.4%	76.5%	5.9%	17.6%	5.9%	5.9%	5.9%	0.0%	5.9%	5.9%	5.9%	11.8%
diff3	53	43.4%	41.5%	7.5%	34.0%	1.9%	18.9%	18.9%	3.8%	13.2%	32.1%	20.8%	35.8%
ed	72	59.7%	59.7%	4.2%	55.6%	18.1%	48.6%	0.0%	1.4%	4.2%	6.9%	16.7%	20.8%
espresso	738	38.9%	38.5%	0.9%	19.6%	5.3%	7.5%	3.1%	5.3%	2.4%	12.9%	24.5%	32.4%
find	50	66.0%	58.0%	18.0%	34.0%	18.0%	14.0%	2.0%	0.0%	4.0%	18.0%	8.0%	14.0%
flex	153	47.1%	45.1%	2.0%	20.3%	9.2%	7.2%	5.2%	5.2%	0.0%	13.7%	18.3%	27.5%
ft	23	60.9%	60.9%	0.0%	13.0%	4.3%	8.7%	0.0%	0.0%	0.0%	13.0%	17.4%	17.4%
ghttpd	20	90.0%	80.0%	60.0%	45.0%	20.0%	20.0%	5.0%	0.0%	10.0%	15.0%	10.0%	10.0%
grep	276	47.8%	46.4%	3.6%	40.6%	13.4%	25.7%	2.5%	10.5%	1.4%	0.4%	29.3%	56.2%
gzip	181	46.4%	45.9%	1.7%	17.7%	7.7%	9.4%	2.8%	1.1%	1.7%	2.8%	17.1%	29.8%
http_get	9	22.2%	22.2%	0.0%	33.3%	22.2%	0.0%	0.0%	22.2%	22.2%	33.3%	11.1%	11.1%
indent	109	51.4%	49.5%	3.7%	26.6%	17.4%	9.2%	0.0%	5.5%	7.3%	13.8%	14.7%	21.1%
ks	35	37.1%	37.1%	2.9%	17.1%	2.9%	0.0%	14.3%	0.0%	2.9%	40.0%	31.4%	37.1%
make	349	59.0%	53.0%	12.6%	29.5%	17.5%	8.0%	2.9%	6.0%	1.7%	8.0%	25.8%	34.7%
othello	26	38.5%	38.5%	3.8%	30.8%	11.5%	15.4%	3.8%	0.0%	19.2%	30.8%	34.6%	46.2%
sed	220	74.1%	72.3%	7.3%	43.2%	15.5%	28.6%	0.5%	9.5%	1.8%	3.2%	15.0%	32.7%
space	53	58.5%	56.6%	11.3%	18.9%	1.9%	13.2%	5.7%	0.0%	5.7%	35.8%	17.0%	22.6%
spell	10	80.0%	70.0%	20.0%	40.0%	20.0%	20.0%	10.0%	0.0%	0.0%	50.0%	20.0%	30.0%
sudoku	59	74.6%	74.6%	3.4%	39.0%	10.2%	32.2%	1.7%	0.0%	1.7%	16.9%	33.9%	40.7%
thttpd	83	62.7%	61.4%	12.0%	43.4%	15.7%	24.1%	7.2%	2.4%	9.6%	8.4%	19.3%	24.1%
time	7	57.1%	42.9%	28.6%	42.9%	14.3%	28.6%	14.3%	14.3%	0.0%	42.9%	14.3%	14.3%
which	29	44.8%	41.4%	13.8%	41.4%	27.6%	17.2%	0.0%	0.0%	6.9%	13.8%	17.2%	41.4%
yacr2	123	36.6%	36.6%	0.0%	21.1%	13.0%	6.5%	1.6%	0.0%	1.6%	18.7%	24.4%	35.8%
TOTAL	2,869	51.3%	49.3%	5.7%	28.4%	10.9%	14.3%	3.2%	4.8%	3.0%	11.3%	21.6%	32.5%

Table 10. Loop Characteristics by Number of Paths.

Number of Paths	Num Loops	Function Call			Alternate Exit					Input	Output	Nesting	
		Yes	Location		Yes	Contains				Yes	Yes	Has Nested	Is Nested
			Body	Cond		Break	Return	Exit	Goto				
1	1092	28.9%	25.0%	6.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.5%	8.4%	2.3%	33.2%
2-10	1328	58.1%	57.2%	4.7%	40.3%	16.3%	19.1%	3.5%	4.7%	3.7%	10.2%	22.0%	33.7%
11-100	294	83.0%	83.0%	8.8%	60.9%	21.1%	37.4%	7.5%	11.2%	3.7%	18.7%	61.6%	28.9%
101-1,000	87	87.4%	87.4%	8.0%	67.8%	19.5%	31.0%	20.7%	21.8%	6.9%	27.6%	74.7%	20.7%
1,001-10,000	31	93.5%	93.5%	3.2%	58.1%	22.6%	32.3%	6.5%	32.3%	6.5%	32.3%	80.6%	25.8%
> 10,000	37	91.9%	91.9%	5.4%	62.2%	32.4%	24.3%	13.5%	35.1%	8.1%	18.9%	86.5%	27.0%

Table 11. Loop Characteristics by Type

Loop Type	Num Loops	Function Call			Alternate Exit					Input	Output	Nesting	
		Yes	Location		Yes	Contains				Yes	Yes	Has Nested	Is Nested
			Body	Cond		Break	Return	Exit	Goto				
do	230	30.9%	30.9%	4.8%	25.7%	6.1%	15.2%	2.6%	3.0%	6.1%	6.5%	15.2%	51.7%
for	1796	49.1%	48.4%	1.5%	25.1%	9.4%	10.6%	2.6%	4.6%	0.9%	12.2%	21.9%	29.5%
while	746	58.7%	52.8%	16.9%	27.9%	9.0%	16.9%	4.2%	3.8%	6.2%	10.1%	19.3%	35.7%
while(1)	97	83.5%	83.5%	0.0%	99.0%	66.0%	59.8%	9.3%	19.6%	10.3%	13.4%	48.5%	16.5%
array	1297	39.4%	39.1%	0.4%	24.2%	9.2%	11.2%	1.7%	4.6%	0.2%	9.6%	19.1%	34.1%
string	409	49.6%	41.3%	12.2%	26.4%	11.7%	13.2%	2.2%	2.4%	0.0%	6.8%	8.6%	35.7%
linked list	372	57.0%	56.7%	1.3%	22.3%	8.1%	9.9%	3.8%	1.6%	0.5%	14.8%	21.8%	29.0%
ds: other	174	58.6%	58.6%	0.0%	13.8%	2.9%	7.5%	1.7%	2.9%	0.0%	10.9%	30.5%	18.4%
input	99	81.8%	72.7%	46.5%	63.6%	25.3%	29.3%	18.2%	7.1%	75.8%	26.3%	24.2%	20.2%
not over	25	96.0%	96.0%	8.0%	48.0%	16.0%	36.0%	8.0%	12.0%	0.0%	32.0%	72.0%	12.0%
other	493	68.6%	66.9%	11.4%	42.6%	16.8%	24.9%	5.1%	9.3%	1.4%	12.8%	32.7%	36.5%
copy	107	15.9%	15.9%	1.9%	2.8%	0.9%	1.9%	0.0%	0.0%	0.0%	0.0%	1.9%	32.7%
initialize	188	17.0%	17.0%	0.0%	3.7%	0.5%	1.1%	2.1%	0.5%	0.5%	2.1%	5.3%	30.3%
modify	297	36.4%	36.4%	0.3%	4.0%	1.3%	1.3%	1.3%	0.0%	0.0%	3.0%	8.8%	34.0%
read	1158	58.8%	58.5%	1.2%	22.5%	6.6%	11.5%	2.5%	5.2%	0.3%	17.1%	28.7%	30.0%
search	404	35.9%	27.2%	10.1%	53.0%	26.7%	22.8%	1.7%	3.7%	0.2%	1.2%	4.0%	41.8%
other	98	45.9%	44.9%	2.0%	33.7%	11.2%	16.3%	4.1%	5.1%	0.0%	10.2%	31.6%	19.4%

more likely to contain function calls. Similarly, *copy* and *initialize* loops tend to be simpler with few paths. Not surprisingly, these types of loops are less likely to contain function calls.

The presence of function calls in the stopping condition varies widely across the different programs and can be attributed more to the programming style rather than the functionality of the program. One common pattern that was noticed was the use of the functions in the loop stopping condition when gathering file input. Examples include using `read` / `fread` until it returns zero and using `feof` to determine if the end of the file has been reached. As a result, 46.5% of input loops contained a function call in the stopping condition, a significantly higher percentage than other loops.

4.4.2 Alternate Exits

Looking at Table 9, just over a quarter of all loops (28.4%) contain an alternate exit with the percentages in programs varying from 11.7% (*bc*) to 55.6% (*ed*). By definition, none of the 1,092 loops with one path have an alternate exit. From Table 10, of the loops with 2-10 paths, 40.3% contain an alternate exit. But when considering loops with 11 or more paths, 62% of the loops contain an alternate exit. As with function calls, loop analysis will need to account for alternate exits.

The most popular construct for an alternate exit is the early return, appearing in 14.3% of all loops. However, the prevalence of early returns varied widely across different programs. Two programs (*http_get* and *ks*) have no loops with an early return while *barcode* and *ed* respectively have 33.8% and 48.6% of their loops contain early returns. The break statement is the second most popular alternate exit construct; all programs have at least one loop with a break statement. The use of exits (functions that unconditionally exit the program) and gotos are less prevalent. Their usage is more dependent on programming style. For instance, 12 of the 25 of the programs adhere to good programming practices by not having any loops with a goto.

From Table 11, all but one *while(1)* loop contains an alternate exit. This is not surprising since *while(1)* loops will run forever without an alternate exit. The one loop that does not have an alternate exit is in a daemon program that

runs indefinitely until the program is aborted. Many *input* loops employ alternate exits (63.6%). The primary purpose for the alternate exits is to handle various errors that may occur when handling input. Search loops also often contained alternate exits (53.0%). This is not surprising as many loops end early when the desired data is found using either a break or return. Loops that *copy*, *initialize*, or *modify* data structures rarely contain alternate exits. These operations commonly do not have the need to use alternate exits. For instance, it is not typical to copy only part of an array and then exit.

One factor in the use of alternate exits is the role of sanity checking / error checking. Some programs have extensive checking manifesting in more loops with alternate exits in loops. Others had little to no checking within the primary functionality of the program. When analyzing loops and programs in general, a decision must be made in how to handle sanity checks.

4.4.3 Input

Using the data from Table 9, only 3.0% of all loops contain an input system call . Not surprisingly, most of these loops are classified as *input* loops in Table 11. Only 12 of the 2,770 loops (0.4%) not classified as *input* contained an input system call. One conclusion is that being able to handle input system calls is not important except in loops that are explicitly used to get input.

4.4.4 Output

Output system calls are found in 11.3% of the loops as shown in Table 9. The percentage of loops with output varies across programs largely based on how much output a program is capable of producing and how frequently a program prints error and/or debugging messages. Loops with more paths are more likely to contain an output statement. Some of these loops have the specific goal of producing formatted output with many paths based on options specified by the user. Others are simply complex loops that happen to print an error or debug message somewhere in the body. *Input* and *not over* loops are more likely to contain output statements than other types of loops while loops that *copy*, *initialize*, *modify*, or *search* data structures are less likely to contain output statements.

4.4.5 Nesting

From Table 9, roughly one of every five loops (21.6%) contains a nested loop. Nearly a third of all loops (32.5%) are nested within another loop. The maximum loop depth is four (quadruple nested loops). There are 16 quadruple nested loops, 93 triple nested loops, and 511 doubly nested loops. Using the data from Table 10, loops with more paths are more likely to contain a nested loop while loops with fewer paths are more likely to be nested within another loop.

When analyzing Table 11, not many correlations can be based on the type of loop with respect to nesting. *String* traversal loops and loops that *copy*, *initialize*, *modify*, or *search* data structures in general contain nested loops less often than other loop types while *not over* loops contain nested loops more often. Loops that *search* data structures and *do* loops are more likely to be nested in other loops than other loop types.

It is difficult to draw conclusions from this data. More analysis is needed to determine how closely coupled the nested loops are with respect to one another. For example, the two loops that traverse a two-dimensional array are closely coupled. This coupling is an important factor in how nested loops are analyzed; further analysis in this area is left as future work.

5. Threats to Validity

This section describes some shortcomings that are not addressed in this study. First, the use of paths in the loop body is not necessarily the most accurate measure of its loop complexity. The path count only captures complexity associated with the control within the loop (or more generally within the function). It does not capture complexity associated with data operations and dependencies, especially dependencies across different loop iterations. Data dependencies are often measured separately such as the Halstead effort measures [8] that measure complexity based on the number of operands and operators. Developing a complexity metric that captures both control and data dependencies is challenging due to the variety of program analysis techniques that can be employed. The most accurate measure of complexity would be tied to the program analysis technique being used such that it captures the strengths and weaknesses associated with that technique.

The goal of this study is to show the maximum benefit of using loop-separate analysis. In the ideal case, the loop summary does not introduce any paths. In practice, this is not necessarily the case. Consider a pointer analysis that is checking for NULL pointer dereferences. The resulting loop summary may be of the form:

```
if (condition that causes p to be NULL)
  p is NULL
else
  p is not NULL
```

The path-based analysis may wish to process the loop summary by separating the two cases into two paths: one where `p` is NULL and one where `p` is not NULL. Furthermore, the condition that causes `p` to be NULL may be complex to the point where it is desirable to create even more paths. There is a trade-off between analyzing fewer but more complex paths and analyzing more paths that are less complex. Exploring this trade-off is left as future work – it is highly specific to the particular analysis and its goals. To further improve performance, it is possible to create approximate loop summaries that are less complex but create imprecision in the overall analysis.

6. Related Work

Several research groups have worked on analyzing loops in programs using various forms of static analysis. Kovács and Voronkov [9] describe how to find loop invariants, expressions that are true throughout the loop, for loops involving arrays. Martel [10] unrolls loops using partitioning in order to increase the precision of invariants found. This unrolling technique is most applicable to numerical programs where arithmetic errors can accumulate over several iterations of a loop. Flanagan and Qadeer [11] developed a predicate abstraction technique to determine loop invariants. Lokuciejewski et al. [12] present a static analysis that computes loop iteration counts. The analysis is interprocedural but uses slicing to eliminate code not relevant to the loop. Kirner [13] implements a technique for automatically determining the lower bound and upper bound for the number of loop iterations. While it effectively handles difficult programming constructs such as alternate exits, it does not handle nested loops. Burnim et al. [14] use an on-the-fly symbolic execution engine to determine if a program is currently executing an infinite loop.

Another direction for loop analysis is to recognize common loop patterns. White and Wiszniewski's SILOP tool [15] identifies simple loop paths – paths that iterate through a single loop a variable number of times. By altering this parameter, a simple loop pattern is created and used to assist in testing. The array checker ARCHER [16] identifies loops that iterate a constant number of times and iterator loops. Hu et al. [17] describe a technique that squashes loops of a particular canonical form, replacing them with a single non-looping statement. This technique was used to reduce the size of slices but also could be used in other program analyses. Ngo and Tan [18] detect illegal paths by recognizing patterns. The "looping-by-flag" pattern matches `while` loops that stop when a flag variable is set. Paths that meet certain conditions will be deemed infeasible. Binkley et al. [19] propose transformations for looping-by-flag loops with the goal of improving search-based testing techniques.

One way to analyze loops separately is to generate a summary that captures the behavior of the loop. Godefroid and Luchaup [4] developed a technique that automatically creates partial loop summaries that consist of a loop precondition and postcondition. The summaries are used in automatic test generation. Abd-El-Hafiz and Basili [5] created a knowledge-based approach to creating loop summaries. First, loops are divided into fragments based on data flow. Then, annotations are created for each fragment. Finally, an annotation that summarizes the whole loop is synthesized from the individual fragment annotations. A novel aspect to their approach is that the analysis used differs based on a classification of a loop. Kroening et al. [6] developed LoopFrog, a tool that uses abstract symbolic transformers to infer loop invariants in creating loop summaries permitting loop-free program analysis. In addition, many research groups [2, 21, 22, 23, 24] have developed techniques for creating summaries for procedures.

Symbolic execution [1] is a common analysis technique employed in bug detection tools. Such tools must have a solution for loops. Popular tools like PREFix [2] and Symbolic Java PathFinder [3] both place limits on the number of iterations. PREFix places the limit by stopping execution when a user-defined limit of paths has been simulated in the function. Symbolic Java PathFinder restricts the underlying model checker's search depth and also restricts the number of constraints associated with a particular path. In loop-extended symbolic execution [24], symbolic variables that represent the number of loop iterations are introduced. Using these variables, they can find other variables that are linearly dependent on the iteration count. Symbolic execution is also used in automated test generation including CUTE [25] and Pex [26].

Our previous research [27] analyzes the paths in programs and examines the effect program slicing has on the path count. Most functions have few paths but slicing does not sufficiently reduce path counts on the functions that do have many paths. Other program analysis studies include the work by Gabel and Su [28], which explores the uniqueness of source code by analyzing over 6,000 software projects consisting of 420 million lines of code. At granularities of one to seven line chunks, they found software to generally be similar. Harman et al. [29] developed a theory surrounding the merging of nodes in the control flow graph and how it applies to program slicing. A similar analysis could be applied to our technique as we are essentially merging nodes from a loop into one node that captures the loop behavior. Das et al. [30] and Dillig et al. [31] describe sound and efficient path-sensitive analyses. Both use their analyses to find temporal safety properties such as null dereference checks and file errors. Ball and Larus [32] developed a path profiling algorithm that computes how frequently acyclic paths are executed.

To estimate the complexity of loops, we used the path count. Cyclomatic complexity [33] is a popular path-based complexity metric. Henry et al. [34] found that cyclomatic complexity is highly correlated with Halstead's effort metrics [8] based on the number of operands and operators in a function. CodeSurfer [7] and Understand [35] compute several control-based and data-based complexity measures.

7. Conclusions and Future Work

This paper presents results from a study that explores the potential of analyzing loops separately. When using loop-separate analysis, many functions saw a significant reduction in the number of paths. This suggests that using loop-separate analysis could improve the performance of program analysis since fewer paths would need to be explored. However, more work is needed using an actual analysis that employs loop-separate analysis to fully determine to what extent performance improvements are possible. For the most complex functions, using loop-separate analysis was not sufficient to eliminate the sheer number of paths that occur in these functions. These functions have significant complexity outside loops from other control statements including a few complex functions with no loops at all.

To assess the potential challenge of analyzing loops, each loop was inspected and analyzed individually. On the plus side, most loops are short and are traversing common data structures. Many of these short loops contain few programming constructs that make analysis difficult. However, loops that have more paths are more likely to

contain hard-to-analyze programming constructs especially function calls, alternate exits, and nested loops. Of the hard-to-analyze features, over half of the loops contain a function call, meaning that some level of interprocedural analysis is necessary for loops to be properly analyzed.

There are several directions for future work. One direction is to apply these analyses to programs written in other languages such as Java or C++. These languages have standard libraries for data structures. Consequently, programmers may be more likely to use a provided function as opposed to writing a loop. On the other hand, these languages also implement exception handling – another feature that makes program analysis challenging.

Another direction is to actually implement the main idea in the paper of analyzing loops separately within symbolic execution to determine how well loops can be summarized. A study could explore the trade-off of analyzing fewer paths that are more complex versus more paths that are less complex.

A broader direction for this work is exploring the appropriate "unit" of program analysis. Many tools already break the program into functions and analyze them separately using rudimentary interprocedural analysis. This division is necessary in that the analysis does not scale interprocedurally. Similarly, some large complex functions cannot be analyzed because they contain too many paths. Can large complex functions be broken into different "units" of analysis to make the analysis feasible while maintaining the soundness of the analysis? This paper presents just one possible way of doing that – analyzing loops separately.

Acknowledgments

The author would like to thank GrammaTech for providing CodeSurfer and the anonymous referees for their valuable comments.

References

[1] King, J.: 'Symbolic execution and program testing', Communications of the ACM, 1976, 19, 7, pp. 385–394
[2] Bush, W., Pincus, J., and Sielaff, D.: 'A Static Analyzer for Finding Dynamic Programming Errors', Software—Practice & Experience, 2000, 30, 7, pp. 775–802
[3] Păsăreanu, C., Mehlitz, P., Bushnell, D., Burlet, K., Lowry, M., Person, S., and Pape, M.: 'Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software'. Proceedings of the International Symposium on Software Testing and Analysis, Seattle, Washington, 2008, pp. 15–26
[4] Godefroid, P., and Luchaup, D.: 'Automatic Partial Loop Summarization in Dynamic Test Generation'. Proceedings of the 2011 International Symposium on Software Testing and Analysis, Toronto, Canada, 2011, pp. 23–33
[5] Abd-El-Hafiz, S., and Basili, V.: 'A Knowledge-Based Approach to the Analysis of Loops', IEEE Transactions on Software Engineering, 1996, 22, 5, pp. 339–360
[6] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C.: 'Loop Summarization Using Abstract Transformers'. Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis, Seoul, Korea, 2008, pp. 111–125
[7] GrammaTech, Inc.: 'CodeSurfer' http://www.grammatech.com/products/codesurfer/overview.html, accessed March 2012
[8] Halstead, M., 'Elements of Software Science' (Elsevier Science Inc., 1977)
[9] Kovács, L., and Voronkov, A.: 'Finding Loop Invariants for Programs over Arrays Using a Theorem Prover'. Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, York, United Kingdom, 2009, pp. 470–485
[10] Martel, M.: 'Improving the Static Analysis of Loops by Dynamic Partitioning Techniques'. Proceedings of the Third International Workshop on Source Code Analysis and Manipulation, Amsterdam, The Netherlands, 2003, pp. 13–21
[11] Flanagan, C., and Qadeer, S.: 'Predicate Abstraction for Software Verification'. Proceedings of the Symposium on Principles of Programming Languages, Portland, Oregon, 2002, pp. 191–202
[12] Lokuciejewski, P., Cordes, D., Falk, H., and Marwedel, P.: 'A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models'. Proceedings of the International Symposium on Code Generation and Optimization, Seattle, Washington, 2009, pp. 136–146

[13] Kirner, M.: ‘Automatic Loop Bound Analysis of Programs written in C’. Master’s Thesis, Technischen Universitat at Wien, 2006
[14] Burnim, J., Jalbert, N., Stergiou, C., and Sen, K.: ‘Looper: Lightweight Detection of Infinite Loops at Runtime’. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, 2009, pp. 161–169
[15] White, L., and Wiszniewski, B.: ‘Path Testing of Computer Programs with Loops using a Tool for Simple Loop Patterns’, Software – Practice and Experience, 1991, 21, 10, pp. 1075–1102
[16] Xie, Y., Chou, A., and Engler, D.: ‘ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors’. Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, Finland, 2003, pp. 327–336
[17] Hu, L., Harman, M., Hierons, R., and Binkley, D.: ‘Loop Squashing Transformations for Amorphous Slicing’. Proceedings of the 11th Working Conference on Reverse Engineering, Delft, The Netherlands, 2004, pp. 152–160
[18] Ngo, M., and Tan, H.: ‘Detecting Large Number of Infeasible Paths through Recognizing their Patterns’. Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 2007, pp. 215–224
[19] Binkley, D., Harman, M., and Lakhotia, K.: ‘FlagRemover: A Testability Transformation for Transforming Loop-Assigned Flags’, ACM Transactions on Software Engineering and Methodology, 2011, 20, 3, pp. 1–33
[20] Gulwani, S., and Tiwari, A.: ‘Computing Procedure Summaries for Interprocedural Analysis’. Proceedings of the European Symposium on Programming, Braga, Portugal, 2007, pp. 253 – 267
[21] Zhao, Y., Gong, Y., Liu, L., Xiao, Q., and Yang, Z.: ‘Context-Sensitive Interprocedural Defect Detection Based on a Unified Symbolic Procedure Summary Model’. Proceedings of the 11th International Conference on Quality Software, Madrid, Spain, 2011, pp. 51–60
[22] Yorsh, G., Yahav, E., and Chandra, S.: ‘Generating Precise and Concise Procedure Summaries’. Proceedings of the 35th Annual Symposium on Principles of Programming Languages, San Francisco, California, 2008, pp. 221–234
[23] Ancourt, C., Coelho, F., and Irigoin, F.: ‘A Modular Static Analysis Approach to Affine Loop Invariants Detection’, Electronic Notes in Theoretical Computer Science, 2010, 267, 1, pp. 3–16
[24] Saxena, P., Poosankam, P., McCamant, S., and Song, D.: ‘Loop-extended Symbolic Execution on Binary Programs’. Proceedings of the International Symposium on Software Testing and Analysis, Chicago, Illinois, 2009, pp. 225–236
[25] Sen, K., Marinov, D., and Agha, G.: ‘CUTE: A Concolic Unit Testing Engine for C’. Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, 2005, pp. 263–272
[26] Tillmann, N., and de Halleux, J.: ‘Pex–White Box Test Generation for .NET’, Tests and Proofs – Lecture Notes in Computer Science, 2008, 4966, pp. 134–153
[27] Larson, E.: ‘A Plethora of Paths’. Proceedings of the IEEE 17th International Conference on Program Comprehension, Vancouver, Canada, 2009, pp. 40–49
[28] Gabel, M., and Su, Z.: ‘A Study of the Uniqueness of Source Code’. Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, New Mexico, 2010, pp. 147–156
[29] Harman, M., Hierons, R., Danicic, S., Howroyd, J., Laurence, M., and Fox, C.: ‘Node Coarsening Calculi for Program Slicing’. Proceedings of the Eighth Working Conference on Reverse Engineering, Stuttgart, Germany, 2001, pp. 25–34
[30] Das, M., Lerner, S., and Seigle, M.: ‘ESP: Path-Sensitive Program Verification in Polynomial Time’. Proceedings of the Conference on Programming Language Design and Implementation, Berlin, Germany, 2002, pp. 57–68
[31] Dillig, I., Dillig, T., and Aiken, A.: ‘Sound, Complete and Scalable Path-Sensitive Analysis’. Proceedings of the Conference on Programming Language Design and Implementation, Tucson, Arizona, 2008, pp. 270–280
[32] Ball, T., and Larus, J.: ‘Efficient Path Profiling’. Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Paris, France, 1996, pp. 46–57
[33] McCabe, T.: ‘A Complexity Measure’, IEEE Transactions on Software Engineering, 1976, SE-2, 4, pp. 308–320
[34] Henry, S., Kafura, D., and Harris, K.: ‘On the Relationships Among Three Software Metrics’, ACM SIGMETRICS Performance Evaluation Review, 1981, 10, 1, pp. 81–88
[35] Scientific Toolworks, Inc: ‘Understand Source Code Analysis & Metrics’. http://scitools.com/index.php, accessed March 2012