Program Analysis Too Loopy? Set the Loops Aside

Eric Larson
Computer Science and Software Engineering
Seattle University
Seattle, WA USA
elarson@seattleu.edu

Abstract—**Among the many obstacles in 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 looks at whether loops could be analyzed separately and replaced with a summary. First, the complexity of a loop is estimated by counting the paths through the body of the loop. 85% of the loops have fewer than ten paths and less than 1% have more than 10,000 paths. Second, the number of paths is computed by analyzing loops separately to assess the feasibility of such an analysis approach. While the number of paths is decreased in many cases, it is typically not sufficient for long, complex functions. Finally, loops are classified based on their stopping condition and further analyzed for programming elements that may make loop analysis more difficult. Nearly half of the loops are array traversals and over half of the loops contain a function call.**

Keywords-loops; program analysis; paths

I. INTRODUCTION

Loops are 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 until the end is reached, scanning an array for the first character that is not a space, and many more. Some uses of loops are common to different programs while other uses are specific to a particular algorithm or program.

There are several challenges that must be addressed when analyzing loops. The number of iterations of a loop can be 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 [2, 9, 17, 19, 20] 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 and replaced with summaries? If so, this would decrease the number of paths to analyze and nearly eliminate the need to analyze infinite loops.[1]

This paper analyzes loops from 15 different moderately-sized C programs providing data that can be used to determine to what extent loops can be analyzed and subsequently summarized. In particular, we count the number of paths through the loop, providing an estimate of its complexity.

If loops were to be analyzed separately, would it make any difference? To answer this question, we performed a feasibility study that counts the number of paths that remain in a function if loops are analyzed separately. This number of paths provides an ideal lower bound on the number of paths necessary if loops are analyzed separately.

Not all loops are the same. To this end, we also analyze how many loops contain programming elements that are difficult to analyze such as function calls, alternate exits, and nested loops. Each loop was analyzed manually and classified based on their stopping condition. Informally, similarities from different loops across programs were noted.

This paper makes the following contributions:

- An analysis of how many paths are within in a loop, providing an estimate of the complexity necessary to analyze the loop. Most, but not all, loops have simple bodies with few paths.
- Results of a feasibility study that counts the number of paths if loops are analyzed separately. Most functions with many paths had significant complexity outside of loops such that skipping loops was not sufficient in reducing the number of paths to a reasonable number.
- An analysis of how often loops contained certain programming constructs or properties. Of the key findings: 48.7% of loops are array traversals, 24.2% contained an alternate exit beyond the normal stopping condition, and 57.6% contained a function call.

The rest of the paper is organized as follows. Section II describes the implementation of our loop analyses and Section III presents the results of these analyses. Related work is outlined in Section IV and Section V concludes.

[1] This paper does not address loops constructed via backward goto statements or recursion.

II. LOOP ANALYSES

This section describes the loop analyses employed in this study. The loop analyses were applied to preprocessed source code written in the C programming language, exposing any loops that were embedded in macros.[2] All automated analyses were performed using an extension to CodeSurfer [6]. Initially, CodeSurfer parses the source code and create a control flow graph. Our extension does the following:

1. Identify which basic blocks are associated with each loop.
2. Traverse the basic blocks within each loop, looking for specific programming elements.
3. Using the control flow graph created by CodeSurfer, count the number of the paths in the loop.

As described later in this section, some of the analyses were carried out by manually inspecting each loop.

A. Paths In Loops

Our first experiment consists of counting the number of paths in each loop. The path count serves as an estimate of how much work is needed to analyze the body of the loop.

The path counting algorithm uses the beginning of the body of the loop as the starting point. Within the body of the loop, paths are counted using a depth first search of the control flow graph. If-else statements introduce two paths and switch statements introduce a path for each case. The CodeSurfer internal representation converts short-circuited operators into the appropriate if-else constructs which are subsequently accounted for in the path counting algorithm.

While there is a single starting point, there are several possible ending points for a loop:

- Reaching the end of the loop body.
- *A break statement corresponding to the loop (opposed to a break statement associated with a switch or a nested loop).
- A continue statement corresponding to the loop.
- *A goto statement that leads outside the loop. (In the programs under consideration, all backwards goto statements exited the loop).
- *A return statement.
- *A function call that unconditionally aborts the program.

A loop has an alternate exit if it is possible to exit the loop without using the provided stopping condition. The items in the list above marked with '*' are alternate exits.

Two different algorithms are employed for nested loops. In one variant, the path counting algorithm will traverse the loop zero times and one time. This variant was chosen as it is common for path-based analyses to place limits on the number of iterations in the loop. In the second variant, the nested loop is assumed to be replaced with a summary and does not introduce any paths. This variant assumes that each loop is analyzed separately starting with the inner most loop. When outer loops are analyzed, the summary, which is assumed to not introduce any paths, is used in place of the loop.

The path counting algorithm only returns a number of possible paths through the loop. No attempt is made to eliminate illegal paths. We deem this to be appropriate since any analysis technique to summarize loop behavior would have to be able to detect and handle illegal paths. In addition, all paths are considered equal even though paths differ in length and complexity. Accounting for the complexity of different paths is left as future work.

B. Paths Counting by Separating Loops

This experiment explores the effect on path count if loops were analyzed separately. Here is how this analysis could be carried out:

1. Analyze each loop in the program summarizing its behavior. In the case of nested loops, the inner loops would be analyzed and summarized first.
2. Replace the loops with their summaries.
3. Use symbolic execution (or some other analysis technique that relies on traversing paths) and exhaustively analyze all paths.

Paths in this experiment are counted intraprocedurally. For a base case, the path count is computed with loops present. Loops are traversed at most one time. The number of paths in a program is simply the sum of the paths from each function in the program.

To count the paths when the loops are analyzed separately, loops from the control flow graph are removed and the loop predecessor is connected directly to the loop successor. The total number of paths in each loop is also counted using the second variant described in the previous section where nested loops are assumed to be analyzed separately and replaced with a summary. The number of paths needed to analyze a function is the sum of the number of paths outside the loops (when loops are removed) and the number of paths inside the loops.

This goal of this experiment is to obtain an ideal lower bound on the number of paths that can be achieved if symbolic execution is structured in this fashion. This algorithm assumes that each loop can be summarized and that the summary does not introduce any additional paths. In practice, the resulting summary might be very complex. This in essence defers the complexity from the path traversal engine to the underlying constraint solver or theorem prover. 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. To further improve performance, it is possible to create approximate summaries that are less complex but create inaccuracies that may result in missed bugs in bug detection systems. The creation of summaries is beyond the scope of this paper as it highly specific to the system and its goals.

C. Loop Characteristics

To address the complexity in creating a summary that captures the behavior of a loop, we explored an assortment of characteristics of loops. The purpose of this study was two-fold. First, we wanted to see how similar loops are both

[2] Some programmers use a dummy `do-while` loop with a constant predicate 0 as a wrapper in macros. These (non) loops were discarded.

within the same program and across different programs. Our second goal was to determine how often loops contained "hard-to-analyze" elements such as function calls, early returns, and input / output commands.

This experiment consisted of both automated analysis using CodeSurfer and manual inspection of the loops. Automated analysis was used for simple tasks such as determining if a loop contains a function call. Manual analysis was used to judge the intent of the loop and to note similarities between different loops.

Towards the first goal of identifying common loop patterns, loops are manually classified based on their stopping condition into the following five categories:

Array traversals: The loop consists of an index variable that is used in an array reference operator within the loop. The array reference can be applied to either an array or to a pointer to an array. The variable is incremented with a constant increment or decrement (typically one) at the end of each iteration. The loop stops when the index variable reaches a particular value that remains constant (invariant) during the loop. For forward traversals, this is often the size of the array, but this is not a requirement.

Loops that use pointers to traverse an array may also be considered to be array traversals provided that the stopping condition is based on a fixed number of elements. In other words, the loop will be considered to be an *array traversal* if the loop could easily be rewritten in a form that meets the above requirements.

Data structure traversals: The loop consists of a traversal of a common data structure such as a string or a linked list (but not an array). The stopping condition is when the end of the data structure is reached.

For strings, a loop is considered a *data structure traversal* if the loop stops when the null terminating character is encountered. If the loop stops based on the size of the array, it is considered an *array traversal* (described above). If the loop stops when a different character other than the null terminating character is encountered, the loop is classified as a *sentinel loop* (described below).

Sentinel loop: The loop consists of a traversal of a data structure that ends when a particular character is encountered. In most cases, the traversed data structure is an array, possibly a string. There are two common uses of sentinel loops: i) as an "end-of-data" marker and ii) finding a delimiter in parsing a string.

Input sentinel loop: The loop is used to get data from input (either from the console, file, or network). The stopping condition for the loop is either when there is no more input (such as the end of the file) or when a special "end-of-input" marker is reached.

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

Each loop is classified in exactly one of these categories. In the event a loop satisfies multiple classifications, we prioritize in this order: *input sentinel*, *array traversal*, *data structure traversal*, *sentinel*, and *other*. For instance, if a loop that traverses an array can stop either by encountering some value or reaching the end of the array, we classify the loop as an *array traversal* based on the priority above.

The presence of alternate exits other than the normal stopping condition does not automatically cause the loop to be classified as *other*. For instance, an *input sentinel* loop is permitted to have a check for valid input that causes the program to abort. Separately, we classify whether or not the loop has an alternate exit other than the normal stopping condition.

One shortcoming to our manual inspection is that we do not further inspect functions that are called within the loop. This can cause loops to be classified as *other* when it could be classified differently. For instance, the `for` loop in this trivial example would be classified as *other* but it could be classified as *array traversal* if `foo` was inspected during analysis.

```
…
for (j = 0; j < n; j++)
  foo(a, j);
…

void foo(int *a, int j) {
  a[j] = 0;
}
```

The second part of collecting loop characteristics was to determine if a loop contained "hard-to-analyze" features. In particular, we looked for the following:

- Function call in the stopping condition
- Function call in the loop body (excludes function calls that unconditionally exit the program, which are tracked separately)
- Break statement (corresponding to the loop)
- Continue statement (corresponding to the loop)
- Goto statement (may or may not jump out of the loop)
- Return statement
- Function call that unconditionally exits the program
- Function call that gets input
- Function call that produces output

This analysis was done automatically using our CodeSurfer extension, except for detecting function calls in the stopping condition, input function calls, and output function calls. 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. In the case of input and output functions, we were concerned with having an incomplete list of I/O functions and felt more comfortable analyzing the loops manually.

As noted earlier, function calls were not inspected further to see if the functions called within the loop could exit the program, gather input, and/or produce output. However, we did note if a function was simply a wrapper function for exiting, input, and 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

TABLE I. PROGRAMS USED

Name	Description	Funcs	Lines	Loops	do	for	while	while(1)
bc	calculator	101	18,876	103	1	33	67	2
betaftpd	file transfer daemon	66	8,861	17	2	6	8	1
diff3	compares three files	29	7,310	53	10	23	19	1
find	file finder	295	26,135	50	1	33	16	0
flex	lexical analyzer	144	8,809	153	1	116	33	3
ft	spanning tree	37	4,910	23	7	8	8	0
ghttpd	web server	16	7,664	22	0	10	10	2
gzip	compression utility	92	15,308	181	31	62	81	7
indent	source code indenter	103	18,635	109	10	33	59	7
ks	graph partitioning	13	2,327	35	1	33	1	0
othello	othello game	11	1,629	26	2	19	4	1
space	specialized interpreter	136	6,988	52	1	14	35	2
sudoku	sudoku solver	47	4,339	59	0	54	4	1
thttpd	web server	126	16,454	85	1	49	22	13
yacr2	channel router	58	8,402	123	6	112	5	0

program. If a loop called one of these wrapper functions, it would be marked appropriately as if the contents of the wrapper functions were inlined in the loop.

Another complication for analyzing loops is when they contained other loops. We mark loops that have at least one inner nested loop. We also mark loops that are nested within another loop. Furthermore, the depth of the nesting is recorded. As with previous analyses, called functions are not analyzed to determine if they contain loops.

III. RESULTS

The 15 programs used in this study are shown in Table I. All programs used in this study are written in the C programming language.

Table I also shows how many loops are in each program further subdivided by the type of loop. The `while(1)` loop is for any loop that does not have a 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. 12 of the 15 programs have at least one `while(1)` loop. `for` loops are the most common type of loop and heavily used in the programs *flex*, *ks*, *othello*, *sudoku*, and *yacr2*.

A. Paths in Loops

The results of our experiment that measures the number of paths in each loop is summarized in Table II. The table is divided into two sections - one for each of the variants that differ in how inner loops are counted. Within a variant, 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.

When inner loops are traversed at most once, 39.8% 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 character in a string. Looking further, 45.4% of the loops have two to ten paths. Combining these first two columns, 85% of the loops have ten or fewer paths. This suggests that it may be possible to analyze many of the loops present in these programs.

There are very few loops with a large number of paths. Less than 1% of all the loops analyzed had over 10,000 paths. In four of the programs (*betaftpd*, *ft*, *ghttpd*, and *ks*), all the loops had fewer than 100 paths. Only four programs (*gzip*, *indent*, *sudoku*, and *thttpd*) had loops that contained over 10,000 paths. The program *indent* was the only program that had loops with over 100,000 paths. It had two such loops with 43,830,540 and 3,275,384 paths.

When inner loops are analyzed separately and not traversed, the number of loops with ten or fewer paths increases to 91% and there are very few loops with large path counts. Only six loops have more than 1,000 paths and only two loops (the two big loops from *indent*) have more than 10,368 paths.

Table III breaks down the number of paths by type of loop. For brevity, data is only shown for the variant where inner loops are traversed at most once. Out of the four loop types, `while(1)` loops tended to have more paths. `for` and `do-while` loops tend to be smaller, the largest number of paths in each type of loop was just over 10,000 paths. Of the six loops that had the most paths, four were `while` loops and two were `while(1)` loops.

B. Path Counting by Separating Loops

A comparison of counting paths when loops are included with the rest of the function versus when loops are analyzed separately is presented in Table IV. The first column shows the number of paths present in the program that includes paths through loops (at most one iteration). The second column shows the number of paths in each program if loops are analyzed separately. This number is then broken down into the number of paths not in loops (third column) and the total number of paths within loops (fourth column).

The results of this experiment varied widely. Some programs saw a significant reduction in the number of paths. In particular, *ks* had 24,452 paths when loops are included and only 153 when loops are analyzed separately, *yacr2* went from over 2 million paths to 3,104, and *sudoku* saw a decrease from almost 2 billion paths to 21,216. In the case of *sudoku* and *ks*, there were more paths inside loops than paths outside of the loops. In *yacr2*, the number of paths inside loops (1,529) was very close to the number of paths outside loops (1,575). In all other programs, the number of paths outside loops was significantly higher than the number of paths inside loops. As a result, some programs (*betaftpd*, *find*, *flex*, and *othello*) saw very little decrease in the number of paths when loops are analyzed separately. Programs *gzip*, *indent*, and *thttpd* saw a significant decrease in the number of paths but still had a large number of paths even when loops are removed. Of the five programs that had over a billion paths with loops, four of them still had a billion paths.

TABLE II. LOOP PATH COUNTS BY PROGRAM

Name	Inner Loops Traversed at Most Once							Inner Loops Separated and Not Traversed						
	Most paths in loop	1	2 - 10	11 - 100	101 - 1k	1k - 10k	> 10k	Most paths in loop	1	2 - 10	11 - 100	101 - 1k	1k - 10k	> 10k
bc	672	51	43	6	3	0	0	119	54	43	5	1	0	0
betaftpd	37	5	9	3	0	0	0	37	5	10	2	0	0	0
diff3	9,608	26	18	4	3	2	0	581	29	17	4	3	0	0
find	396	20	22	5	3	0	0	396	20	24	3	3	0	0
flex	8,448	63	73	12	3	2	0	464	70	74	6	3	0	0
ft	20	10	11	2	0	0	0	4	12	11	0	0	0	0
ghttpd	23	11	7	4	0	0	0	23	11	9	2	0	0	0
gzip	20,883	77	83	14	5	0	2	198	87	82	11	1	0	0
indent	43,830,540	57	37	9	1	2	3	30,352,140	58	37	9	2	1	2
ks	72	14	15	6	0	0	0	36	19	14	2	0	0	0
othello	121	11	13	1	1	0	0	121	18	7	0	1	0	0
space	1,252	16	31	4	0	1	0	164	23	24	4	1	0	0
sudoku	41,480	13	28	8	7	1	2	10,368	19	31	7	1	0	1
thttpd	17,267	26	44	8	4	2	1	5,768	30	44	5	4	2	0
yacr2	3,537	34	61	18	8	2	0	392	46	65	8	4	0	0
TOTAL	43,830,540	434 39.8%	495 45.4%	104 9.5%	38 3.5%	12 1.1%	8 0.7%	30,352,140	501 45.9%	492 45.1%	68 6.2%	24 2.2%	3 0.3%	3 0.3%

TABLE III. LOOP PATH COUNTS BY TYPE

Number of Paths	do	for	while	while(1)
1	21	240	173	0
2 - 10	37	286	152	20
11 - 100	13	53	25	13
101 - 1k	1	20	14	3
1k - 10k	1	5	4	2
10k- 100k	1	1	3	1
> 100k	0	0	1	1
Number of loops	74	605	372	40
Most paths in loop	10,056	10,368	3,275,384	43,830,540

TABLE IV. PATH COUNTS BY PROGRAM

Program	Paths (loops traversed at most once)	Paths (analyzing loops separately)		
		Total	Outside Loops	Inside Loops
bc	949,346	56,965	56,487	478
betaftpd	45,692	42,315	42,209	106
diff3	572,718	40,966	39,735	1,231
find	1,966,770	1,804,370	1,803,439	931
flex	7.40E+11	7.22E+11	7.22E+11	1,398
ft	10,594	526	481	45
ghttpd	9,679	1,156	1,075	81
gzip	3.05E+10	2.37E+09	2.37E+09	873
indent	9.82E+17	8.38E+11	8.38E+11	30,421,708
ks	24,452	153	47	106
othello	13,382	13,201	13,034	167
space	6,227	2,011	1,676	335
sudoku	1.94E+09	21,216	10,099	11,117
thttpd	2.84E+12	3.48E+10	3.48E+10	10,345
yacr2	2,249,048	3,104	1,575	1,529

To further explain the results of this experiment, we performed the same comparison for each function. The results are presented in Table V. The first column shows the number of functions that have the corresponding number of paths when loops are included. The next two columns respectively show the number of functions that have no loops and the number of functions that consist of a single path outside of the loop (in other words, the function consists of a single loop and all the paths are contained within that loop). The last column shows the number of functions that contain the corresponding number of paths when loops are analyzed separately. The number of paths is the sum of the number of paths outside loops in the function and the number of paths in all of the loops within that function.

Not surprisingly, we see a shift downward when loops are excluded with more functions having fewer paths and fewer functions having more paths. However, there are still many functions that have high path counts even when loops are excluded. In particular, there are 12 functions that have over 100 million paths when loops are included and there are still seven when loops are analyzed separately. Of those seven functions, two functions (both in the program *flex*) do not contain any loops so analyzing loops separately has no benefit.

Out of the 1,274 functions under analysis, 807 (63.3%) have no loops. Most of these are simple functions with very few paths. For functions that have over 100 paths, 21% of the functions have no loops. For functions that over one million paths, 11% of the functions have no loops.

At the other end of the spectrum, 172 functions (13.5%) only have one path when loops are analyzed separately. This means that the function consists of a single loop and once that loop is analyzed, there are no additional paths to

TABLE V. COMPARING PATH COUNTS (PER FUNCTION)

Number of Paths	Loops Included			Loops Separate
	Total Funcs	No Loops	1 Path Out of Loop	Total Funcs
1	270	269	1	269
2 - 10	634	428	124	687
11 - 100	195	73	30	205
101 - 1k	83	20	10	58
1k - 10k	34	8	3	21
10k- 100k	23	5	2	14
100k - 1M	11	1	0	6
1M - 10M	8	1	1	4
10M - 100M	4	0	1	3
> 100M	12	2	0	7

traverse. Unless the loop contains nested loops that can be analyzed separately, there is no benefit to analyzing loops separately in such function. Most of the functions that have only one outside path have relatively few paths. However, there are functions with many paths that only have one outside path. This includes a function in *indent* that consists of a single complex loop that has over 30 million paths itself.

Probing further, most of the complex functions with high path counts contain multiple (possibly nested) loops and significant functionality outside of these loops. While the number of paths decreased for most of the functions, the decreases were moderate in most cases, limiting the effectiveness of analyzing loops separately.

C. Loop Characteristics

Table VI provides a count of how many loops met the characteristics described in Section II. The table is broken down into four groups. The first group classifies the loops based on the stopping condition: *array traversal*, *data structure traversal*, *input sentinel*, *sentinel*, and *other*. The second group separates loops into those that can only exit normally via the stopping condition specified within the loop and those that also contain at least one alternate exit. The third group provides a count of how many loops contain different "hard-to-analyze" features. The final group is concerned with nesting and indicates how many loops contain a nested loop and how many loops are nested within another loop.

With respect to the stopping condition, 48.7% are array traversals. The percentage within individual programs can vary widely. The programs *flex*, *othello*, *sudoku*, and *yacr2* use arrays heavily while the programs *find*, *ks*, and *space* tend to use other data structures. 29.3% of the loops were labeled as *other* and did not fit cleanly into one of the other stopping conditions.

Many of the loops (75.8%) did not have alternate exits. By definition, none of the 434 loops with one path had an alternate exit. Of the 657 loops that had at least two paths, 264 (40.2%) had an alternate exit.

Over half of the loops (57.6%) overall contained a function call. In each program, at least 47% of the loops contained a function call. This suggests that interprocedural analysis will be necessary in order to fully analyze loops. In other "hard-to-analyze" characteristics: 16.8% of the loops contained an output statement, 11.5% contained a return statement, and 9.8% contained a break statement.

Roughly one of every five loops (19.1%) contained a nested loop. 28.4% of the loops were nested within another loop. The maximum loop depth was four (quadruple nested loops). There were 5 quadruple nested loops, 31 triple nested loops, and 176 double nested loops.

Tables VII and VIII show the same characteristics by type of loop (Table VII) and by number of paths (Table VIII). A vast majority of the *array traversals* are coded using `for` loops. Most of the *sentinel* loops were implemented with `while` loops. The `while(1)` loops appear to be difficult to analyze: all but three of the `while(1)` loops were classified as *other*, all but one[3] had alternate exits, and contained a higher percentage of "hard-to-analyze" elements with respect to the other loops.

Other noteworthy results from Table VII: function calls within stopping conditions occur most frequently in `while` loops, very few `for` loops contain input commands, and there is not a lot of difference among `do-while`, `while`, and `for` loops for the other "hard-to-analyze" elements.

From Table VIII, loops with large path counts tended to be classified as *other*, have alternate exits, and contain function calls. This result is intuitive since loops with large path counts tend to be longer and more complex.

All of the *sentinel* loops and all but four of the *input sentinel* loops contain ten or fewer paths. Based on our observations, the goal of many *sentinel* loops is to just to get to the sentinel character with very little, if any, processing in the loop. Loops that were nested within other loops also tended to have very few paths.

Additional statistics were gathered for *array traversals* since they are more common than other types of loops and are summarized in Table IX. First, we looked at whether the array size was known at compile time. This heavily varies by the program. Almost all of the array traversals in *othello*, *space*, and *sudoku* had sizes known at compile time. Roughly half of the arrays in *gzip* were known at compile time. For all of the other programs, there were significantly more arrays where the sizes were not known until run-time. Four programs (*find*, *ghttpd*, *indent*, and *yacr2*) did not have any array traversals with sizes known at compile time and four other programs (*betaftpd*, *flex*, *ft*, and *ks*) only had one. Many of these programs used dynamically allocated arrays with sizes dependent on the input to the program.

During our classifications of loops as *array traversals*, a noticeable number of loops had additional stopping conditions beyond an index or size comparison. The last column of Table IX shows how many array traversal loops stop only based on a particular index or the size of the loop.

[3]The one loop that does not have an alternate exit is in a daemon program that runs indefinitely until the program is aborted.

TABLE VI. LOOP CHARACTERISTICS BY PROGRAM

		Stopping Condition					Exit		Loop Contains									Nesting	
Program	Num Loops	Array Trav	Data Struct	Input Sent	Sent	Other	Norm Exit	Alt Exit	Func Stop	Func Call	Brk	Cont	Goto	Retn	Exit	In put	Out put	Has Nested	Is Nested
bc	103	41	17	4	5	36	91	12	8	50	4	0	3	5	5	8	20	16	28
betaftpd	17	7	4	0	0	6	14	3	0	13	1	4	0	1	1	0	1	1	2
diff3	53	17	6	4	13	13	34	19	4	30	1	5	2	11	10	7	17	11	19
find	50	14	22	1	2	11	33	17	8	37	9	4	0	7	1	1	10	4	7
flex	153	101	14	0	3	35	122	31	3	75	14	0	8	11	8	1	22	28	42
ft	23	5	9	0	0	9	20	3	1	16	1	0	0	2	0	0	3	4	4
ghttpd	22	9	0	4	1	8	13	9	9	17	4	3	0	4	1	4	0	2	4
gzip	181	82	9	1	1	88	149	32	2	86	14	8	2	17	5	4	9	31	54
indent	109	43	18	4	15	29	80	29	2	71	19	4	6	10	0	8	22	16	23
ks	35	9	22	0	1	3	29	6	0	18	1	0	0	0	5	1	9	11	13
othello	26	19	0	2	0	5	18	8	0	17	3	1	0	4	1	5	8	9	12
space	52	10	24	2	0	16	43	9	4	37	1	1	0	6	3	3	18	9	12
sudoku	59	43	2	0	0	14	36	23	2	46	6	4	0	19	1	1	10	20	24
thttpd	85	28	15	5	6	31	49	36	10	52	13	14	2	20	6	7	11	16	22
yacr2	123	103	0	2	2	16	96	27	0	63	16	2	0	9	2	2	23	30	44
TOTAL	1091	531	162	29	49	320	827	264	53	628	107	50	23	126	49	52	183	208	310
		48.7%	14.8%	2.7%	4.5%	29.3%	75.8%	24.2%	4.9%	57.6%	9.8%	4.6%	2.1%	11.5%	4.5%	4.8%	16.8%	19.1%	28.4%

TABLE VII. LOOP CHARACTERISTICS BY LOOP TYPE

		Stopping Condition					Exit		Loop Contains									Nesting	
Loop Type	Num Loops	Array Trav	Data Struct	Input Sent	Sent	Other	Norm Exit	Alt Exit	Func Stop	Func Call	Brk	Cont	Goto	Retn	Exit	In put	Out put	Has Nested	Is Nested
do	74	6	8	7	6	47	56	18	1	48	4	1	1	9	5	11	17	25	24
for	605	462	77	0	6	60	474	131	6	309	55	26	9	58	16	4	91	112	176
while	372	63	77	20	36	176	296	76	46	238	26	20	8	36	23	32	64	52	104
while(1)	40	0	0	2	1	37	1	39	0	33	22	3	5	23	5	5	11	19	6

TABLE VIII. LOOP CHARACTERISTICS BY NUMBER OF PATHS

		Stopping Condition					Exit		Loop Contains									Nesting	
Number of Paths	Num Loops	Array Trav	Data Struct	Input Sent	Sent	Other	Norm Exit	Alt Exit	Func Stop	Func Call	Brk	Cont	Goto	Retn	Exit	In put	Out put	Has Nested	Is Nested
1	434	239	62	10	39	84	434	0	21	178	0	7	0	0	0	11	56	2	121
2-10	12	4	1	0	0	7	7	5	0	11	0	3	3	5	2	1	7	10	1
11-100	38	14	5	2	0	17	16	22	3	35	1	8	4	13	9	5	14	29	9
101-1,000	104	37	14	2	0	51	49	55	7	87	22	16	6	26	14	4	25	65	28
1,001-10,000	495	237	79	15	10	154	319	176	22	309	82	14	7	78	22	29	75	95	150
>10,000	8	0	1	0	0	7	2	6	0	8	2	2	3	4	2	2	6	7	1

The "Yes" column provides the number of loops in which the only way to exit the loop is to reach the particular index. In these loops, the number of iterations is fixed, making it more straightforward to analyze the loop. If a loop contains an extra condition in the loop predicate and/or contains any form of alternate exit, it is counted in the "No" column.

Overall, 77.0% of the array traversal loops only exit when the index reaches a particular value. Across the different programs, the percentage of such loops range from 55.8% (*indent*) to 95.1% (*gzip*). Many programs lie near the average in the 75-80% range.

We also looked at whether other types of loops beyond those classified as *array traversals* to determine if the number of iterations could be determined at compile time. Clearly, this is impossible for any *input sentinel* loop since they depend on input. Unless the data structure itself is constant (not likely), the number of iterations of *data structure traversals* and *sentinel* loops are also not known at

TABLE IX. ARRAY TRAVERSAL STATISTICS

Name	Array Loops	Size known at compile time				Stop on size only?			
		Yes		No		Yes		No	
bc	41	8	19.5%	33	80.5%	33	80.5%	8	19.5%
betaftpd	7	1	14.3%	6	85.7%	6	85.7%	1	14.3%
diff3	17	6	35.3%	11	64.7%	13	76.5%	4	23.5%
find	14	0	0.0%	14	100%	8	57.1%	6	42.9%
flex	101	1	1.0%	100	99.0%	81	80.2%	20	19.8%
ft	5	1	20.0%	4	80.0%	4	80.0%	1	20.0%
ghttpd	9	0	0.0%	9	100%	6	66.7%	3	33.3%
gzip	82	40	48.8%	42	51.2%	78	95.1%	4	4.9%
indent	43	0	0.0%	43	100%	24	55.8%	19	44.2%
ks	9	1	11.1%	8	88.9%	7	77.8%	2	22.2%
othello	19	19	100%	0	0.0%	15	78.9%	4	21.1%
space	10	10	100%	0	0.0%	6	60.0%	4	40.0%
sudoku	43	38	88.4%	5	11.6%	26	60.5%	17	39.5%
thttpd	28	5	17.9%	23	82.1%	16	57.1%	12	42.9%
yacr2	103	0	0.0%	103	100%	86	83.5%	17	16.5%
TOTAL	531	130	24.5%	401	75.5%	409	77.0%	122	23.0%

compile time. Of the 320 loops classified as *other*, only 5 had iteration counts known at compile time.

D. Qualitative Analysis

In this section, we go into more detail about the loops in each of the 15 programs. These notes are derived from both observations made when manually analyzing the loops and from the raw data obtained from the different loop analyses.

bc: Some loops used a data structure that was similar to a microprocessor simulator - traversals were made by updating a program counter. Some loops involved dividing a number into its digits; many of these were classified as *other* but probably could be analyzed.

betaftpd: Employed some strange loops that removed bytes from a string (using `memmove`) until a sentinel character was reached. These loops were classified as *other* because they were not technically traversals. The function with the most paths, `long_listing`, has no loops.

diff3: Some loops traversed arrays with three elements for the three files being compared. Many of the loops with many paths were in functions that produced output.

find: Some functions traversed a tree data structure. Many functions did not contain any loops, including one that contained over a million paths.

flex: Contains two functions that have over 100 million paths but no loops. Both functions, along with several others, have many control decisions based on the large number of user options this program supports.

ft: The function with the most paths, `DeleteMin`, has several loops. While none are particularly complex, separate loop analysis brings the path count of this function from 10,082 to 27. Some functions traverse a heap data structure.

ghttpd: This program forks multiple processes. Some loops used the system call `waitpid` as a stopping condition, only exiting when all of the child processes have completed.

gzip: Five of the functions have over one million paths when loops are included. All five of these functions contain both loops and control statements outside the loop resulting in path counts that are a couple orders-of-magnitude lower when loops are analyzed separately. Noteworthy loops includes string processing functions that are specific to *gzip*, bit manipulation loops, and loops within tree data structure routines that use fixed-sized arrays.

indent: Contains the two loops with the highest path counts. The loop with the most paths is in the function `indent_main_loop` and is the main loop that governs the entire operation of the program. The loop with the second most paths is in the function `print_comment`. Neither loop is in the function `dump_line` which has 9.82×10^{17} paths when loops are included. The function has 18 loops. By analyzing loops separately when counting paths, the number of paths drops to 8.38×10^{11} paths, which is still unreasonable. The primary reason for the large number of paths in *indent* is that the program provides a myriad of options that allows the user to configure how the input file is indented. The functionality of many functions and loops is controlled by these options.

ks: The function with the most paths, `PrintResults`, primarily consists of a single loop which contains many nested loops. The number of paths shrinks from 23,100 to 22 when loops are analyzed separately (only two paths exist outside the loop). Most of the loops are `for` loops used to traverse linked lists. This program employs `malloc` sanity checks leading to some loops having alternate exits that would otherwise not have them.

othello: The board size is a fixed eight by eight grid. Hence many loops are array traversals with a fixed size of eight. The function with the most paths, `validmove`, has no loops.

space: Some loops use the function `TapeGet` to get the next piece of data. These loops are classified as *other* because they are not simple traversals. Many loops print error messages (possibly exiting the loop early) or have sanity checks (such as `malloc` checking).

sudoku: The board size is a fixed nine by nine grid but is represented by a single one-dimensional array of 81 squares. Many loops are fixed size: either 9 squares of a row or column or all 81 squares. The two loops with the most paths are in the function `main`. One of the loops contains many nested loops while the other does not contain any. When analyzing loops separately, the number of paths decreases from almost 2 billion to 12,122 paths. However, 10,368 paths are within a single loop in `main`.

thttpd: The number of paths in the program is dominated by the function `main`. This function contains a mix of loops and other control decisions. When loops are separated, the number of paths in `main` drops a couple orders-of-magnitude from 2.84×10^{12} to 3.48×10^{10}.

yacr2: The function with the most paths, `PrintChannel`, has over 2 million paths when loops are included. Since the function consists of a single complex loop, there is only one path outside of the loop. The complex loop contains 11 nested loops at various levels. Together,

there are 407 paths total in the loops. Many of the loops in the program are array traversals. While the array sizes are not known at compile time, they are fixed throughout the program once they are initialized by user input.

When comparing the different loops across the programs, there were definitely common loop patterns such as traversing an array or linked list. The structures of the loops are similar, mirroring common operations such as initialization, updating, and searching. Loops often contained content that was specific to the program such as a calculation used in setting in an array value. Sometimes these calculations were complex. However, many programs have loops that were both similar in structure and content to other loops within that same program.

There was a lot of variation in the loops based on programming practice. Some programs employed more rigorous error checking that led to more complexity compared to loops in other programs.

Sentinel loops also seemed to be implemented differently among different programs, especially for strings. Some used string functions to help find the stopping data while others carried out sentinel loops by manually comparing individual elements. Another distinction for sentinel loops was whether or not the sentinel marked the end of the data to be processed or whether it marked the end of the data to be skipped. Loops in the latter case tended to be simple and were similar across multiple programs.

Complex loops with many paths were unique, even within the same program. All of these loops were specific not only to the program, but to the specific task within the program.

A more formal analysis is left as future work. One possible improvement is to normalize the loops in some manner (such as converting all applicable `while` loops to `for` loops) to reduce the effect on different programming styles. Another approach would involve breaking the code into tokens and comparing them, like the study by Gabel and Su [5]. This would allow for a more realistic comparison of the loop structure.

IV. Related Work

Several research groups have worked on analyzing loops in programs using various forms of static analysis. Kovács and Voronkov [11] describe how to find loop invariants, expressions that are true throughout the loop, for loops involving arrays. Martel [14] unrolls loops using partitioning in order to increase the precision of invariants found. This is most applicable to numerical programs where arithmetic errors can accumulate over several iterations of a loop. Gulwani and Tiwari [7] outline a fixed point analysis of loops when creating interprocedural summaries. This type of analysis could be used to analyze the loops separately. Lokuciejewski et al. [13] 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 [10] 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.

Another direction for loop analysis is to recognize common loop patterns. White and Wiszniewski's SILOP tool [22] 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 which is used to assist in testing. The array checker ARCHER [23] identifies loops that iterate a constant number of times and iterator loops. Hu et al. [8] 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 [16] 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.

Symbolic execution [9] 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 [17] 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 [18], 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 [19] and Pex [20].

Our previous research [12] analyzes the paths in program 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 [5], 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. Das et al. [3] and Dillig et al. [4] 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 [1] 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 [15] is a popular path-based complexity metric. Understand [21] is a tool that computes several complexity measures including cyclomatic complexity, path count, lines of code, and many others.

V. Conclusions and Future Work

This paper presents results from three loop analyses that explore the feasibility of analyzing loops separately. Our first study explores the number of paths in a loop to estimate the complexity necessary to analyze the loop. 85% of the loops

have ten paths or fewer; less than 1% of the loops have over 10,000 paths. Our second experiment calculates paths counts for functions when loops are analyzed separately. Many functions saw a significant reduction in the number of paths. However, the reductions were not sufficient in functions that have large path counts as these functions had significant complexity outside loops from other control statements. In our last experiment, loops are classified based on the stopping condition and whether they contained specific programming elements that made analysis difficult. 48.7% of the loops are classified as array traversals. Of the hard-to-analyze features, function calls are the most prevalent, appearing within 57.6% of the loops.

Is summarizing loops within symbolic execution worthwhile? On the plus side, most loops are short and fall into one of four common classifications based on the stopping condition. However, since many loops contain function calls, loop analysis will need to be interprocedural to fully summarize their behavior. The practicality of summarizing loops is limited in that many functions have no loops and many complex functions have significant complexity outside any loop. The approach works best for functions with a large number of (possibly nested) loops.

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, especially strings. Consequently, programmers may be likely to more use a provided function as opposed to writing a loop.

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 analysis for software bug detection tools. 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. Some large complex functions also do not scale sufficiently because they contain too many paths. Can large complex functions broken into different "units" of analysis such that the effect on finding bugs is minimized? This paper presents just one possible way of doing that – analyzing loops separately.

ACKNOWLEDGMENTS

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

REFERENCES

[1] T. Ball and J. Larus, "Efficient Path Profiling," Proc. of the 29th Symposium on Microarchitecture (MICRO), 1996.

[2] W. Bush, J. Pincus, and D. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," Software—Practice & Experience, 2000.

[3] M. Das, S. Lerner, and M. Seigle, "ESP: Path-Sensitive Program Verification in Polynomial Time," Proc. of the Conference on Programming Language Design and Implementation (PLDI), 2002.

[4] I. Dillig, T. Dillig, and A. Aiken, "Sound, Complete and Scalable Path-Sensitive Analysis," Proc. of the Conference on Programming Language Design and Implementation (PLDI), 2008.

[5] M. Gabel and Z. Su, "A Study of the Uniqueness of Source Code," Proc. of the Symposium on Foundations of Software Engineering (FSE), 2010.

[6] GrammaTech, http://www.grammatech.com.

[7] S. Gulwani and A. Tiwari, "Computing Procedure Summaries for Interprocedural Analysis," Proc. of the European Symposium on Programming (ESOP), 2007.

[8] L. Hu, M. Harman, R.M. Hierons, and D. Binkley, "Loop Squashing Transformations for Amorphous Slicing," Proc. of the 11th Working Conference on Reverse Engineering, 2004.

[9] J. King, "Symbolic execution and program testing," Communications of the ACM, 1976.

[10] M. Kirner, "Automatic Loop Bound Analysis of Programs written in C," Master's Thesis, Technischen Universitat Wien, 2006.

[11] L. Kovács and A. Voronkov, "Finding Loop Invariants for Programs over Arrays Using a Theorem Prover," Proc. of the Conference on Fundamental Approaches to Software Engineering (FASE), 2009.

[12] E. Larson, "A Plethora of Paths," Proc. of the IEEE 17th International Conference on Program Comprehension (ICPC), 2009.

[13] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models," Proc. of the Symposium on Code Generation and Optimization (CGO), 2009.

[14] M. Martel, "Improving the Static Analysis of Loops by Dynamic Partitioning Techniques," Proc. of the Workshop on Source Code Analysis and Manipulation (SCAM), 2003.

[15] T.J. McCabe, "A Complexity Measure," Proc. of the International Conference on Software Engineering (ICSE), 1976.

[16] M.N. Ngo and H.B.K. Tan, "Detecting Large Number of Infeasible Paths through Recognizing their Patterns," Proc. of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE), 2007.

[17] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Burlet, M. Lowry, S. Person, and M. Pape, "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software," Proc. of the International Symposium on Software Testing and Analysis (ISSTA), 2008.

[18] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended Symbolic Execution on Binary Programs," Proc. of the International Symposium on Software Testing and Analysis (ISSTA), 2009.

[19] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," Proc. of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE), 2005.

[20] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for.NET," Tests and Proofs - Lecture Notes in Computer Science, 2008.

[21] Scientific Toolworks, "Understand Source Code Analysis & Metrics," http://scitools.com/index.php.

[22] L.J. White and B. Wiszniewski, "Path Testing of Computer Programs with Loops using a Tool for Simple Loop Patterns," Software - Practice and Experience, 1991.

[23] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," Proc. of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE), 2003.