Appendix to Detecting and Escaping Infinite Loops with Jolt

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard

Massachusetts Institute of Technology, Cambridge, MA, USA
{mcarbin, misailo, mkling, rinard}@csail.mit.edu

Abstract. Infinite loops make applications unresponsive. Potential problems include lost work or output, denied access to application functionality, and a lack of responses to urgent events. We present *Jolt*, a novel system for dynamically detecting and escaping infinite loops. At the user's request, Jolt attaches to an application to monitor its progress. Specifically, Jolt records the program state at the start of each loop iteration. If two consecutive loop iterations produce the same state, Jolt reports to the user that the application is in an infinite loop. At the user's option, Jolt can then transfer control to a statement following the loop, thereby allowing the application to escape the infinite loop and ideally continue its productive execution. The goal is to enable the application to execute long enough to save any pending work, finish any in-progress computations, or respond to any urgent events.

We evaluated Jolt by applying it to detect and escape eight infinite loops in five benchmark applications. Jolt was able to detect seven of the eight infinite loops (the eighth changes the state on every iteration). We also evaluated the quality of escaping an infinite loop as an alternative to terminating the application. In all of our benchmark applications, escaping an infinite loop produced a more useful output than terminating the application. Finally, we evaluated how well escaping from an infinite loop approximated the correction that the developers later made to the application. For two out of our eight loops, escaping the infinite loop produced the same output as the fixed version of the application.

1 Case Studies

In [1] we have presented Jolt, a system to detect, and if desired, escape the infinite loops. In addition to the case studies of grep and indent that we presented in the paper ([1], Section 5), in this appendix we present the remaining detailed case studies of the infinite loops that we have used in our evaluation:

- ctags-fortran (Section 2)
- ctags-python (Section 3)
- ping (Section 4)
- look (Section 5)

2 Ctags Fortran

Ctags scans program source files to produce an index that maps program entities (e.g., modules, functions, and variables) to their line numbers within the source files [2]. Ctags version 5.5 contains an infinite loop in the Fortran module (Ctags Bugzilla tracker #734933). The loop is located in the method `parseProgramUnit` in the file `fortran.c`.

Figure 1 presents the source code that contains the loop that recursively parses the input source file, starting from the Fortran program unit. Each iteration of the loop is designed to read a single lexical token from the input file and then identify and handle the corresponding language keyword as appropriate. The called functions may consume additional input tokens until they close the scope of the program construct they are parsing. The result of the computation is a set of index entries that identify the locations of the program entities within the file.

The loop does not have an explicit exit condition. Instead, the computation uses an exception mechanism implemented using the `setjmp`/`longjmp` library calls to exit from the loop when all characters from the input stream have been consumed.

```
1  do {
2    if (isType (token, TOKEN_STATEMENT_END))
3      readToken (token);
4    else
5      switch (token->keyword) {
6      case KEYWORD_block:      parseBlockData (token);            break;
7      case KEYWORD_end:        skipToNextStatement (token);       break;
8      case KEYWORD_function:   parseFunctionSubprogram (token);   break;
9      case KEYWORD_module:     parseModule (token);               break;
10     case KEYWORD_program:    parseMainProgram (token);          break;
11     case KEYWORD_subroutine: parseSubroutineSubprogram (token); break;
12     default:
13       if (isSubprogramPrefix (token)) {
14         readToken (token);
15       } else {
16         parseSpecificationPart (token);
17         parseExecutionPart (token);
18       }
19       break;
20   }
21 } while (TRUE);
```

Fig. 1. Source Code for Ctags Fortran Infinite Loop

Infinite Loop: The infinite loop happens when an input token which the loop body does not recognize appears at the outer-most scope in the input source

code. Typically, the developers intended to handle such tokens within some inner scope (handled by the specific parsing functions).

We have identified two scenarios which lead to infinite looping. In one scenario, ctags infinitely loops 1) on a syntactically valid file, as a consequence of not recognizing certain token and failed recovery, and 2) a syntactically invalid input source file, with unmatched scope start and end statements. Finally, the value of variable `token` is a token that the loop body cannot handle, but it does not read new tokens from the input file — all future loop iterations only compare this token against a set of expected keywords.

For the infinite loop to occur on a syntactically valid Fortran File, ctags must first encounter a semicolon as a delimiter between multiple statements on the same line. This version of ctags does not recognize the semicolon as a delimiter. It attempts to recover by discarding input tokens until the next end statement. As part of this recovery process it can lose track of the nesting structure of the program components — it exits from the first end statement it encounters, even it was an end statement of a nested component, instead of the end statement of the component where semicolon appeared.

Ctags then proceeds on to process the following tokens as part of the outermost scope (i.e., part of no component at all) rather than within the scope of the outer component. When it encounters a keyword that is specific to the outer component (and that it is therefore not expecting to see in the outermost scope), the body of the loop does not contain a handler for the keyword, ctags does not read the next token, and the computation makes no progress. An example of such a keyword is `CONTAINS`, which separates the specification part of the module from the subprocedure definitions.

Syntactically invalid Fortran programs can also illicit the infinite loop with improperly nested module components (whether the Fortran source code uses the semicolon separator or not). When encountering the statements that end the scope, the parser does not check whether it actually closes the opened scope. Instead, the parser just continues on to close the opened scope. For example, an additional, unmatched `END TYPE` statement within the definition of the module may be matched with the module opening statement, causing the module scope to close, returning control to the main loop. Main loop can then encounter the tokens that the developers expected only within the module definition, causing it to infinitely loop.

To reproduce the infinite loop, we used the input that was submitted with the bug report (`y.f90`). The input consists of one Fortran module with multiple fields and type definitions followed by subprocedure definition section. It contains semicolon between the definition of a field and the definition of the compound type. Based on this input we created additional inputs that also cause the infinite loop in the code. Some of the inputs contain semicolons located at different places in the code, while some of the inputs that we created do not have semicolons, but instead have unmatched scoping keywords (causing the inputs to be syntactically invalid).

Effects of Escaping Infinite Loop: When Jolt identifies and escapes the infinite loop, the application exits and produces an output file. Executing the program with Valgrind shows escaping the infinite loop does not introduce memory errors.

Our inspection of the output shows that the net effect of using Jolt to escape the infinite loop is that the user loses some of the indexing information in the regions of the program surrounding the semicolon that causes the infinite loop, but otherwise obtains a complete index. The index does not contain the definitions between the semicolon character and the next end statement following the semicolon (ctags discards all of these tokens when it attempts to recover from the unrecognized semicolon). Subsequent definitions that appear after the end token may not have proper scoping information (i.e., they may be indexed as appearing in the surrounding module). This lack of proper scoping information does not affect the operation of at least some of the tools that use the ctags information. Specifically, the vim and emacs ctags modules are able to use the generated index to navigate to the corresponding entities even if the module information is not correct for all entries.

Comparison with Termination: As a comparison, terminating the application when it infinite loops causes it to generate no output at all for the program that elicited the infinite loop. And if the input consists of multiple files, ctags produces no output at all for any file after the file that elicits the infinite loop.

Comparison with Manual Fix: The output of the application is an index of modules, functions and variables defined in the source file. The index contains names, locations, and scoping information for each element. We compare the output that the version of the application with Jolt produces with the output produced by Version 5.5.1, which has been updated to correctly recognize the semicolon separator, which enables the parser to handle the following keywords correctly. Specifically, ctags does not lose track of the component nesting structure of the application and handles each keyword appropriately within its correct component.

We note, however, that the fix in Version 5.5.1 works only for syntactically correct programs. If the program has incorrectly nested component entry and exit keywords, Version 5.5.1 can still infinitely loop. This cause of infinite looping was fixed by Version 5.7.

3 Ctags Python

Ctags version 5.7beta (svn commit 646) contains an infinite loop in Python module (Ctags BugZilla tracker #1988027). The loop is located in the function `find_triple_end` in `python.c`.

Listing 2 presents the loop which recognizes the termination of multi-line string literals. Multi-line string literals begin and end with triple-quote literals (`"""` or `'''`) and can span more than one line. Multi-line strings are by convention used as documentation comments in Python.

```
1  static void find_triple_end(char const *string, char const **which) {
2    char const *s = string;
3    while (1) {
4      s = strstr (string, *which);
5      if (!s) break;
6      s += 3;
7      *which = NULL;
8      s = find_triple_start(s, which);
9      if (!s) break;
10     s += 3;
11   }
12 }
```

Fig. 2. Source Code for Ctags

Infinite Loop: Figure 2 presents `find_triple_end()`, the function from Ctags's Python module, which serves to identify if `string`, which points to a character buffer containing a single line of text from a parsed file, closes an already open multi-line string. The parameter `which` contains the delimiter that began the multi-line string (either `'''` or `"""`).

At the beginning of each iteration of the loop, `s` points to some position in `string` and `which` contains the triple-quote that began the last multi-line string. Within the loop, if `s` does not contain a matching triple-quote, then the loop exits (Line 5). If `s` does contain a matching triple-quote, then the computation 1) records that the currently opened multi-line string has been closed, by setting `which` to `NULL` on Line 7, and 2) checks if `s` contains any additional triple-quotes.

If `s` does not contain an additional triple-quote, then the computation exits the loop (Line 9). Otherwise, the computation 1) records that a new multi-line string has been opened (by updating `which` in `find_triple_string()`), and 2) updates `s` to point to the character after the newly found triple-quote. The computation then returns to the beginning of the loop to look for a triple-quote that closes the newly opened multi-line string.

The programmer wrote this loop with the intention that each iteration of the loop would start at some position in `string` (given by `s`) and either exit, or continue with another iteration that starts at a later position in `string`. To establish this, the value of `s` is incremented by the functions `strstr()` (Figure 2, Line 4) and `find_triple_string()` (Figure 2, Line 8).

However, in the call to `strstr()` the developer mistakenly passed `string`, instead of `s`, as the starting position for each iteration. As a consequence, every iteration of the loop starts over at the beginning of `string`, which can cause an infinite loop. For example, if the triple-quotes of the first and the second multi-line string are of the same type, then at the beginning of every loop iteration (except the first), the values of `s` and `which` are always the same: `s` equals to the starting position of the second multi-line string and `which` contains the triple-quote that starts the second multi-line string.

We used the input python program that was submitted as a part of the bug report (`triple_string_loop.py`). The input consists of one python source

file with three functions defined. Each function contains multi-line strings. Note that the input from the bug report elicits and additional bug (by mistake ctags recognizes the character `'#'` as a beginning of a comment within the multi-line string), which eventually causes the application to encounter the infinite loop.

Based on our understanding of the bug and the code, we created more inputs that elicit the infinite loop. Inputs that we have created isolate the infinite loop, and check for different number of functions, and different literal configurations. We also identified four input files from numpy package (used in performance tests) that cause ctags to loop infinitely.

Effects of Escaping Infinite Loop: After Jolt escapes the infinite loop, the application eventually terminates, producing a well-formed output file. Running the application under Valgrind shows that escaping from the loop does not introduce memory errors.

Escaping the loop only affects the file that elicits the infinite loop — the other files are fully processed. Escaping the loop may have different effects depending on the position of the closing triple of the second multi-line string. If the second multi-line string is closed on a subsequent line, ctags recognizes the closing of the string and successfully parses the rest of the file. The resulting index file in this case is identical to the index file produced by the reference application.

If, on the other hand, the string is closed on the same line as it is opened, ctags does not update the variable `which`, which indicates that the application is in the multi-line string literal. In this case, the triple quotes become unmatched, causing the computation to consider the content of the source file until the next triple quote symbol (which actually opens another string literal) as a string literal.

The result is that ctags parses the portion of code between the quotes not as python source code, but as a long string literal. Moreover, when it reaches the next string literal, it treats the string literal as python source code, then treats the following python source code as a string literal, and so on until the end of the file. In effect, ctags gets out of phase with respect to the program. In this case ctags produces a well-formed output file that omits the indexing information for all of the entities that it (incorrectly) treated as part of string literals.

Comparison with Termination: Terminating ctags when it enters the infinite loop causes ctags to lose some or all of the indexing information for the file (depending on when the output is flushed to the output file). Moreover, terminating ctags leaves it unable to process any subsequent files.

Comparison with Manual Fix: We compare the output after escaping the infinite loop with the output produced by ctags 5.7beta (svn commit 668). In this version of the program, the developers passed `s` instead of `string` as the first parameter of the function `strstr()`, which allows the computation to continue the search for the additional comment delimiters.

4 Ping

Ping [3] is a computer network utility which checks for the reachability of a remote computer. Ping client from the package iputils, version 20100214 (Bug id CVE-2010-2529) has an infinite loop in the function `pr_options`, line 984.

Ping uses Internet control message protocol (ICMP), a part of IP protocol suite, to communicate between the local and the remote computer. In particular, ping client sends ICMP echo requests, and the remote ICMP server responds with ICMP echo replies. Each request and reply messages have mandatory and optional headers.

Listing 3 presents the relevant parts of the loop that parses an optional part of ICMP reply messages. ICMP supports multiple optional headers, including the timestamp (`IPOPT_TS`) and the recorded route between the client and the server (`IPOPT_RR`).

```
1  while (totlen > 0) {
2     cp = optptr;
3     switch (*cp) {

4        case IPOPT_TS:
5           j = *++cp;               /* get length */
6           i = *++cp;              /* and pointer */
7           if (i > j) i = j;
8           i -= 5;
9           if (i <= 0) continue;

10          /* handle IPOPT_TS */
11          break;

12       case IPOPT_RR:
13          j = *++cp;               /* get length */
14          i = *++cp;              /* and pointer */
15          if (i > j)  i = j;
16          i -= IPOPT_MINOFF;
17          if (i <= 0) continue;

18          /* handle IPOPT_RR reply */
19          break;

20       /* handle other replies */
21    }

22    totlen -= j;
23    optptr += j;

24 }
```

Fig. 3. Source Code for Ping Infinite Loop

Infinite Loop: The infinite loop may happen while handling timestamp optional header or route record optional header.

The handler for the timestamp message is in the branch `IPOPT_TS`. The handler initially reads the length and the offset of the data. The handler processes the header only if the value of the offset (which is bounded by the length of the header) is greater than 5. The handler for the route record header is in the branch `IPOPT_RR`. The handler also reads the length and the offset of the data. The handler processes the header only if the value of the offset is greater than the constant `IPOPT_MINOFF`.

If the condition on the offset value is not met, the `continue` statement is executed, skipping the rest of the current header and returning to the start of the loop. Note that the value of the pointer to the optional header (`optptr`) does not change within the loop. Thus, the following loop iteration reads the same optional header again, and, being unable to process the header, keeps spinning in the infinite loop.

To produce ICMP replies that elicit the infinite loop, we use the server based on [4]. We ran the client and the server on physically separate machines, as connecting to the local host does not execute this loop. We used the following relevant parameters for the messages: for timestamp optional headers, the header length was 8 (four bytes for the optional header header and four bytes for the timestamp data) and offset value was 5; for recorded route headers we set the header length to 8 and the length of the data to 4.

To produce ICMP replies that do not elicit infinite loop, we use the same server, but with different header parameters. For the timestamp header, we set length value 8 and offset value 6. For the recorded route header, we use value 8 for the lenght of the data.

Effects of Escaping Infinite Loop: After escaping the loop using Jolt the program eventually terminates, returning the correct number of received replies, and a correct list of times for the replies. The program does not print the content of the optional reply header that caused the infinite loop. Running the application under Valgrind confirms that escaping the loop does not introduce memory errors.

Comparison with Termination: Terminating the program does not handle the ICMP reply, and does not report the number of sent messages and the number and the status of the received messages. On the other hand, when escaping from the infinite loop using Jolt, the program is able to send and receive additional echo messages and report on the number of successfully received replies.

Comparison with Manual Fix: The version of the iputils 20101006 fixes the infinite loop by replacing the `continue` statements in both `IPOPT_TS` and `IPOPT_RR` branches with the `break` statements, which exit the loop immediately after encountering the unsupported optional header.

The effect of the manual fix is essentially identical to the effect of applying Jolt. In both cases, the program can send and receive additional ping messages. Note that if ping sends multiple requests to the same server, Jolt needs to run in vigilant mode, as each reply will elicit the same infinite loop bug.

5 Look

Look searches for a word in lexicographically sorted dictionary. The computation performs binary search, returning the lines which begins with a word. Look version 1.1 from System V Revision 4 has an infinite loop in the function `getword`, line 172.

Listing 4 presents the source code for the function `getword` in which look may enter an infinite loop. The function copies a single line of the text from the input file and stores it into the buffer pointed to by the input parameter `w`.

```
1  int getword(register char *w) {
2      register c;
3      register avail = WORDSIZE - 1;

4      while(avail--) {
5          c = getc(dfile);
6          if(c==EOF) return 0;
7          if(c=='\n') break;
8          *w++ = c;
9      }

10     while (c != '\n')
11         c = getc(dfile);

12     *w = 0;
13     return 1;
14 }
```

Fig. 4. Source code for Look Infinite Loop

Infinite Loop: Because the buffer pointed to by the variable `w` has finite size `WORDSIZE`, the application reads the file in two loops: 1) the first loop on Line 4 reads `WORDSIZE - 1` characters from the input and stores it into `w`, 2) the second loop on Line 10 drains the rest of the line so that the next call to read from the file will start at the next line.

The second loop (Line 10) does not account for the case when the file is not terminated by `'\n'`. In this case, the function `getc()` will always return End-Of-File (`EOF`). Because the loop does not check for this return value, it iterates forever.

Look uses length-based binary search strategy to find matching lines. Since the input is lexicographically sorted, at every step the application calculates the next search range for comparison by calculating the midpoint between the two previously selected index entries. Look infinitely loops any time its midpoint calculation causes it to examine the last line of the file. The loop may start looping infinitely disregarding whether the searched word is or is not in the dictionary.

We used multiple queries of words that exist and words that do not exist in a single dictionary with 50 entries. We also created an additional dictionary where the length of the last entry exceeds the `WORDSIZE` length bound set by the application, also eliciting the infinite loop.

Effects of Escaping Infinite Loop: Applying Jolt to look when it has entered this infinite loop allows look to gracefully escape the loop and return a well-formed output in which `w` is null-terminated and contains the correct content from the last line of the file.

Comparison with Termination: In comparison to the alternative of terminating the application, using Jolt allows a user to gain any additional output that may be computed after processing the contents of the last line of the file.

Comparison with Manual Fix: The output of the application is the list of lines in input file that begin with the searched word. We did not have a reference version of this application. Instead, we manually fixed the infinite loop bug by adding a check for an end of file in the second loop. Escaping the infinite loop produces the same output as this corrected version of look.

References

1. M. Carbin, S Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*, 2011.
2. Exuberant ctags. `http://ctags.sourceforge.net`.
3. iputils. `http://www.skbuff.net/iputils/`.
4. Ping infinite loop analysis. `http://blog.stalkr.net/2010/07/cve-2010-2529-ping-infinite-loop.html` .