Keeping It Simple: Agile Analysis

Gerard J. Holzmann

Writing reliable software is hard. No matter how determined we all are to avoid it, we do make mistakes, and at a fairly predictable rate at that too. What makes a software system reliable is not just determined by the care we take in planning and structuring a design, but also by the methods we use in catching those pesky bugs that come along for the ride.

For safety critical code, the use of strong static source code analyzers has become an essential part of the development process. The best analyzers are based on solid theory and maintained by large teams of highly skilled developers. The tools can patiently trace through complex execution paths to reveal a range of subtle bugs, though of course only the types of bugs that the tools have been trained on. The tools can be slow and costly, though, and the warnings they generate are sometimes difficult to evaluate accurately by humans.

The relatively slowness of the best analyzers means that they tend to be used only infrequently and can't provide the type of rapid feedback that is most useful in code review. The difficulty in fully understanding the tools' output can further result in valid warnings being ignored, because a user is often more easily persuaded that a warning is inaccurate than that their understanding of the report is limited.

There is a need for more agile forms of code analysis that can fill the gap between informally eyeballing code and full-blown formal analysis: efficient methods that can help us scan code quickly to find suspicious fragments.

Currently, to resolve simple queries fast, the most readily available methods include the old Unix pattern matching tools *grep* or *awk*. They are fast and robust, but they are fundamentally text processing tools, not program analysis tools, and that means it's much harder to use them to identify troublesome coding patterns. I've been working on the design of a new type of pattern matching tool (http://spinroot.com/cobra/) that could fill the existing gap, by targeting a level somewhere in between text processing and static analysis. Performance is important in the design of a tool of this type. I want to be able to perform code analyses in real-time, with quick responses to most queries, even for large code archives.

I'll give a few examples of the types of issues that the tool I've been building can catch in otherwise well-vetted code. As a target, I'll use the source code distribution of Linux version 5.0.9. There are two main reasons for selecting this target. First, it is big. There are 26,634 individual files with the extension .c in the archive, with a total of 18.4 Million lines of code. Second, the code is of good quality, and routinely scanned by the best commercial source code analyzers, resulting in an unusually low number of residual warnings from these analyzers. Can we still find new types of problems in this code archive by probing it interactively with a more agile tool?

Out-of-Bound Array Indexing

It's clear that in a C program one should never try to access an array with a negative index value. Accessing an array out of bounds can cause many types of problems, not to mention create security holes that can be exploited by hackers. Can we quickly check those 18.4 Million lines of C code in the Linux distribution for unintentional negative array indexing errors? A completely thorough answer does of course require deep flow analysis, tracking potentially negative values through expressions and chains of function calls, which is time consuming, and also a little error-prone. The existing commercial analyzers have done their best at finding all those cases, so now it's our turn. We start simple and look at functions that can return negative values that are used to index an array, possibly far from the location of the function definition.

We can concisely express what we are looking for as a code pattern over lexical tokens, with don't cares for the parts that we are not interested in. In the pattern we refer to arbitrary type-names with the shorthand @type, and to arbitrary identifier names with @ident. The familiar combination '.*' from standard regular expressions is useful for indicating don't care fragments, or holes, with the dot matching any token type and the Kleene star indicating a sequence of zero or more lexical tokens. Explicit names like 'return' match tokens with that exact name. Similarly, single round and curly braces, and square brackets match tokens of the same type in the source code.

The pattern matcher itself is an extension of Ken Thompson's algorithm for converting regular expressions into efficient non-deterministic finite state automata [1], adapted to work over lexical tokens instead of plain text, and with some extensions to support name binding, brace pairing, and the use of embedded regular expressions over the lower-level token text attributes. The need for these extensions will become clearly shortly.

Name Binding – By preceding a token descriptor with a variable name and a colon we bind the text of the matched token to the variable. For instance, in 'x:@ident' the variable 'x' gets assigned the text attribute of the token (an identifier) that is matched. We can refer to the bound value later in the pattern by typing a colon followed by the variable name, as in ':x'.

A code pattern to find function definitions in C that contain at least one return statement followed by a minus sign can now be expressed as follows, where we use name binding to remember the name of any function matched.

```
@type  x:@ident  (  .*  )  {  .*  return  -  .*  }
```

[GH note for layout: keep at least one space between the symbols where indicated.]

There are three places in this pattern where we have used a don't care, to indicate that it's not important what the details are in these places. This is clearly only a partial check that ignores cases where a more complex expression is returned that may evaluate to a negative value. But, this is the type of check that can be executed very quickly, without waiting for a detailed analysis that may or may not succeed in revealing the more complex cases.

The final part of the code pattern is to find places in the code where the returned value of the matched function is used in an array index. We can, for instance, extend the above pattern with the following part, using don't cares in two more places.

```
.*  [  :x  (  .*  )  ]
```

The pattern tries to locate any use of the function name that was bound to variable 'x' with arbitrary function arguments, enclosed in the square brackets of an array index.

Brace Pairing – The closing brace of any pair of braces (round, curly, or square) that explicitly appear in the pattern is guaranteed by the search tool to match the corresponding opening brace in the input, at the right level of nesting. So, even though there could be additional function calls used as arguments in the call of the tracked function, the closing ')' token in the pattern is guaranteed to match the closing brace of the argument list of the tracked function. This seemingly minor feature turns out to be quite useful in code searches.

If I run the check on the roughly twelve thousand lines of Cobra sources, the result is as expected:

```
$ time cobra -c 'pe @type x:@ident ( .* ) { .* return - .* } .* [ :x ( .* ) ]' *.c
0 patterns matched
real    0m0.111s
user   0m0.093s
sys     0m0.015s
```

The check is executed quickly and there are of course no matches to the pattern. We passed a list of source files on the command line for the tool to process. If all input files are in the same directory this is the easiest way to use the tool, but for the Linux distribution it gets a little more involved, since the files are spread over many directories. To do the check, we can first create a list of C files to process, and then expand this file on the command-line when calling the checker. In the following example I also use six cpu-cores to read in and process the set of 26,634 files in parallel, to speed up the startup phase. How fast this executes will depend on the speed of the disk you use and the efficiency of disk caching. On my system, it executes as follows, using the same code pattern as before.

```
$ time cobra -N6 -c '…'  `cat c_files`
3 patterns matched
real       0m26.203s
user      1m46.312s
sys        0m5.701s
```

This time the command executed in about 26 seconds real-time, with the majority of the time (19 seconds) spent on reading in the files, and a smaller part (7 seconds) to run the pattern matching algorithm itself. This means that for a large code base it is generally better to start up the tool by reading in all files, once, and then quickly probe the code with different types of pattern searches in an interactive code exploration session.

There are no less than three matches of the pattern in the Linux 5.0.9 code base. Understandably, the matched patterns can span many lines of code. The first of the three matches is reported in file ./drivers/isdn/i4l/isdn_common.c, where the function definition appears on line 219, and the call within an array index appears on line 1971. The following snippet of the code in question shows the first and the last two lines of that match.

```
bound var 'isdn_dc2minor'
  219  int
  220  isdn_dc2minor(int di, int ch)
  …
  1970         if (ret > 0)
> 1971                 dev->obytes[isdn_dc2minor(drvidx, chan)] += ret;
```

There is no check on the return value of function ‘isdn_dc2minor()’, which can indeed be negative, with potentially disastrous consequences. At this point it would make sense to experiment further with the pattern to check for cases where the call precedes the function definition in the code archive, but you get the idea. These pattern searches are relatively simple to formulate, resolved in seconds, and can reveal real bugs.

It is easy to do a quick check for even more obvious cases of out-of-bound array indexing, like locating indices that explicitly start with a unary minus operator. Minimally this means just looking for the token pair ‘[-‘, or if we want to see a bit more context: ‘[- .*]’. For the Linux distribution this check takes under 2 seconds to complete, and is surprisingly revealing. Here is how the pattern would be typed in an interactive session, after the relevant source files have been read in:

```
: pe [ - .* ]
467 patterns matched
: dp 1              # dp: display pattern 1
./tools/testing/selftests/proc/proc-loadavg-001.c:54..54
  54              if (!(p[-3] == ' ' && p[-2] == '1' && p[-1] == '\n'))
```

There are 467 instances where an array index is explicitly negative. I’ve shown the result of the first match alone, to give an idea of the type of code that is found. A little further digging shows that in 350 of the cases (75%) the minus sign is followed by a constant, and in the remaining cases by a variable. Very likely, each of these uses is deliberate, though that makes them no less concerning.

Divide by Zero

Could we, in the same code archive, also find cases where a function can return zero, but a direct call to that function can appear as a denominator in a fraction? It would be a certain crash on a divide by zero error. Here’s the query with the result.

```
: pe @type  x:@ident  (  .*  )  {  .*  return  0   .*  }  .*  /  :x ( .* )
1 patterns matched
:   dp 1
./drivers/s390/block/dasd_eckd.c:176..2603
bound var 'recs_per_track', line 2603
  176  static unsigned int
  177  recs_per_track(struct dasd_eckd_characteristics * rdc,
 ...
 2602              format_step = DASD_CQR_MAX_CCW /
> 2603                     recs_per_track(&private->rdc_data, 0, fdata->blksize);
```

The function recs_per_track() can return 0 (if you’re curious, it appears on line 206 in file ./drivers/s390/block/dasd_eckd.c) which may be used to divide the constant DASD_CQR_MAX_CCW (which is defined elsewhere in a macro). Tracing a full execution path through the code that could reveal the same problem could be challenging for any tool.

Security Vulnerabilities

As one final example of quick checks that can reveal issues that can escape standard tools: the danger of using the return value of a call to library function snprintf() as part of one of its arguments in later calls. I've mentioned this issue also in an earlier column [2], after it had been identified as a security vulnerability in the rsyslog/librelp code by researchers from Semmle (documented as CVE-2018-19475). At the time, I didn't check in other code archives to see how common this coding pattern is, and it looks like nobody else has either. Using variable binding we can express the pattern with a simple one-liner, here again in an interactive Cobra session with the Linux 5.0.9 code:

```
: pe x:@ident  +=  snprintf  (  .*  :x  .*  )
987 patterns matched
: dp 1
./sound/soc/soc-pcm.c:3302..3304
bound var 'offset', line 3302
> 3302        offset += snprintf(buf + offset, size - offset,
  3303                    "[%s - %s]\n", fe->dai_link->name,
  3304                    stream ? "Capture" : "Playback");
```

There turn out to be 987 matches in the archive. I've shown the first one, which spans three lines. The pattern matcher doesn't care about the newlines, since it just worries about lexical tokens, and not the white-space that separates them. The problem with these uses is that snprintf does not return the number of characters written into the target buffer, but the number of characters that its unsafe sibling sprintf would have written (don't ask), and that creates a security hole that hackers can exploit.

Embedded Regular Expressions – The most dangerous are calls to snprintf where an unbounded format specifier '%s' is also used, as in the match of the pattern I showed. We can restrict the code pattern to these cases by replacing the final '.*' fragment in the code with the sequence '.* /%s .*'. Where the '/%s' is an embedded regular expression that matches only tokens with text attributes (e.g., format strings) that contain the character sequence '%s'. If we do so, the number of matches reduces to 130.

Out of curiosity, I went back through earlier Linux releases to see when the example code patterns I discussed first started appearing. The counts are summarized in Table I. The warning counts themselves are not the most relevant part though. What is more interesting is the ease with which issues like these can be found with simple one-line queries that are resolved in real-time as you type them. Collecting the data for Table I, for example, took just a few minutes of my time. What Table I illustrates is that greater agility in code analysis can be extremely useful.

Table I

Number of Matches of Four Suspicious Code Fragments in Seven Linux Distributions

Linux Version	.c Files	SLOC	Explicit Negative Array Index	Negative Function Return Value Used as Array Index	Zero Function Return Value used in Division	Incorrect Use of Return Value of snprintf
1.0	282	141,361	10	0	0	0
2.0	780	564,360	36	1	0	0
2.4.18	8,301	3,707,652	277	1	0	1
3.0.1	16,193	10,877,261	291	2	0	779
4.0.1	20,978	14,200,899	311	3	0	907
4.3	21,988	14,900,380	314	3	1	901
5.0.9	26,634	18,359,393	467	3	1	987

References

1. K. Thompson, “Regular expression search algorithm,” Comm. ACM, Vol. 11, No. 6, 1968, pp. 419-422.
2. G.J. Holzmann, “Code Overload,” IEEE Software, Vol. 36, No. 6, Nov/Dec 2019, pp 73-75.