CHAOS GOLUBITSKY

simple software flow analysis using GNU cflow

Chaos Golubitsky is a software security analyst. She has a BA from Swarthmore College, a background in UNIX system administration, and an MS in information security.

chaos@glassonion.org

A CALL GRAPH IS A TEXT-BASED OR graphical diagram showing which functions inside a code base invoke which other functions. Accurate call graphs aid many debugging and software analysis tasks. For example, when viewing a code base for the first time, an examiner can tell from a call graph whether the code structure is flat or modular, and which functions are the busiest. Later in analysis, a call graph can be used to answer specific questions, such as which other functions within the code invoke a specific function of interest.

GNU cflow is a new tool which can be used to quickly and easily generate flexible and accurate text-based call graphs of C programs. In this article I will introduce cflow, with an eye towards describing how it can be used to easily create accurate call graphs.

History and Motivation

The cflow tool was initially developed in the 1990s, and the older version is referred to as POSIX cflow. I first encountered POSIX cflow while performing a vulnerability analysis of open source software [1], for which I needed a simple source of data about reachable functions within a code base. The POSIX specification for the cflow tool [2] requires that the tool be capable of generating forward and reverse flow graphs up to a specified depth, and that the user be able to specify classes of symbols, such as static functions or typedefs, which should be printed or omitted. The POSIX tool provides this relatively limited functionality, and is no longer being actively maintained.

The cflow project was restarted last year due to interest in a simple tool which could generate call graphs, and the first alpha release of GNU cflow [3] occurred in April 2005. The GNU version of the tool is significantly more flexible than the POSIX specification requires, and is being actively maintained and improved.

Basic Functionality

In its simplest use, cflow is called with the name of one or more C source files as arguments. Cflow uses a custom C lexical analyzer to interpret the source code, and prints a call graph of the code, starting with the main() function.

Cflow's basic functionality can be demonstrated using a classic example:

```
#include <stdio.h>

void howdy();

int main() {
  howdy();
  exit(0);
}
void howdy() {
  printf("hi, world!\n");
}
```

If this example is stored as hello.c, then running cflow hello.c will produce:

```
main() <int main () at hello.c:5>:
    howdy() <void howdy () at hello.c:9>:
        printf()
    exit()
```

GNU cflow's main strength is that it can easily be configured to present call data in useful ways. Cflow's behavior can be modified using two major approaches. First, cflow can be invoked with options which present the call graph data in various ways. These options can be used to quickly find whatever data is needed to answer a particular question, or to format the call data for processing via script or some other external program. Second, cflow can be called with options which modify how it processes the source code and, therefore, what information will be contained in its results. I will discuss each of these in turn.

Customizing Cflow's Output Format

Cflow has two major output modes. In tree mode, which is the default, cflow prints functions one per line, using indentation to indicate call relationships. In cross-reference mode, cflow prints a two-column list containing one line for each caller/callee pair within the code base.

CROSS-REFERENCE MODE

Cross-reference mode, which is invoked using the -x flag, is the simpler of the modes, and is not very customizable. In addition to cross-references, it includes a special line for the beginning of each function definition in a file. Therefore, this mode can also be used to quickly obtain a list of the locations of all function definitions within a given file:

```
cflow -x filename.c | awk '$2=="*" {print $1 "\t\t" $3}'
```

TREE MODE

Tree mode is the default and is much more flexible. When invoked without arguments, cflow looks for a function called main(), and produces an indented call graph of that function and all functions it calls. The -m flag tells cflow to begin the tree using a different function. If the specified function is not found, cflow will print a tree for every function in the examined file or files. Reverse mode (-r) prints a reverse call tree, and always prints information about every function in the file.

Several flags are available which affect how much information, beyond function names, is included in cflow's output. These include -n (number each line of output), -l (label each line of output with the call depth of the function listed on that line), --omit-arguments and --omit-symbol-names (shorten the information printed about each function declaration). The --level-indent flag can be used to gain fine-grained control over the spacing and layout of the functions, but -T provides a good set of defaults which give reasonable visual call tree output. Further, the argument --format=posix can be used to obtain output similar (though not identical) to that produced by the older POSIX cflow program.

In tree mode (either standard or reverse), the -d *N* argument tells cflow to report only *N* levels of output. This option can be used to quickly print a list of all functions which are called by any function within a file of interest. (Note that this is most easily done in reverse tree mode, since forward tree mode examines only the main() function by default):

```
cflow -r -d 1 filename.c
```

I typically format cflow output for automated processing by custom scripts. However, cflow output can also be used as input for other graphing or processing software. A couple of examples are worth mentioning here. Cflow can be used in combination with the tool cflow2vcg to produce visual call graphs under the VCG graphing package [4]. Additionally, Emacs users may be interested in the emacs cflow-mode module which is packaged with cflow [5].

Customizing Cflow's Source Code Analysis

Cflow implements its own lexical analyzer for the C language, and there are several ways to control its behavior. In this section I will discuss some options which affect how cflow finds functions and definitions within C source code.

At the simplest level, the -i flag can be used to define subsets of symbols which should or should not be reported, including static symbols, typedefs, symbols whose names begin with underscores, and external symbols.

PREPROCESSOR OPTIONS

GNU cflow does not use a preprocessor by default. When invoked with the argument --cpp, cflow preprocesses the code using the cpp executable or a user-specified preprocessor. Using --cpp increases the accuracy of cflow's output, but has some visible effects. Most notably, functions which are implemented as #define statements are silently unrolled. This can occasionally cause confusing output: for instance, getc() is often implemented by operating systems as a wrapper for another function. It may be confusing to find __srget() in cflow's output with no indication of what invoked it. The older POSIX cflow always used a preprocessor, and preprocessor mode is likely to be desirable for most analysis, but it can sometimes be helpful to produce GNU cflow output without a preprocessor.

When invoked with --cpp, GNU cflow searches for function definitions in system header files. It is possible to tweak the set of directories which cflow should search for function definitions using the -I (include dir) and -U (undefine) flags. (These flags imply --cpp.) These flags are needed if we wish to use cflow to parse complex source code accurately.

EMBEDDING CFLOW IN MAKEFILES

For very small code bases, or to answer simple or file-specific questions, it can be sufficient to manually run cflow on a small number of C source files. However, in order for cflow to provide accurate results for complex code bases, it must process the code the same way the makefile processes it, to ensure that the function relations cflow finds are the same as those compiled into the software. Some more complex source analysis tools (e.g., the OCaml-based C representation language Cil [6]) compile the code as a side effect of analyzing it, and can therefore be trivially embedded in makefiles as compiler replacements. Since cflow does not do this, it is necessary to manually insert cflow-specific rules into the makefile. Makefile editing requires some effort, but it is often worthwhile due to the increased accuracy.

The general idea is to create a separate make target named, for instance, program.cflow, and configure this target to run cflow using:

- The compiler definitions used for this code base
- The include directives used for this code base
- The preprocessor flags used for this code base
- The file names compiled by this code base

It should be possible to use makefile variables to obtain the correct values for each of these items. In addition, the cflow -o flag is used to save the output to a file, and any desired cflow-specific flags are also set. Here is an example of this configuration which is appropriate for inclusion in a GNU-style Makefile.in file [7]:

```
program_CFLOW_INPUT = $(program_OBJECTS:.@OBJEXT@=.c)
CFLOW_FLAGS = -i^s --brief

program.cflow: $(program_CFLOW_INPUT) Makefile
  cflow -o$@ $(CFLOW_FLAGS) $(DEFS) $(DEFAULT_INCLUDES) \
            $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) \
            $(program_CFLOW_INPUT)
```

With this configuration, the invocation make program.cflow should suffice to run cflow on the code base as it will be compiled. The CFLOW_FLAGS variable can be changed in order to run cflow with a different set of options.

Summary

In software analysis, it is often useful to be able to identify caller/callee relationships within a code base, and to display such relationships in usable formats. GNU cflow is a simple tool which performs this function accurately. Cflow builds on the decade-old tool of the same name by providing flexible options which significantly increase cflow's utility and ease of use. GNU cflow is recommended as a first-line tool for answering questions about software call flow.

REFERENCES

[1] http://www.usenix.org/events/lisa05/tech/golubitsky.html

[2] http://www.opengroup.org/onlinepubs/009695399/utilities/cflow.html

[3] http://www.gnu.org/software/cflow/

[4] http://www.gnu.org/software/cflow/manual/html_node
/Output-Formats.html

[5] http://www.gnu.org/software/cflow/manual/html_node/Emacs.html

[6] http://manju.cs.berkeley.edu/cil/

[7] http://www.gnu.org/software/cflow/manual/html_node/Makefiles.html