Is it getting any easier to understand other people's code?

BY GEORGE V. NEVILLE-NEIL

Code Spelunking Redux

IT HAS BEEN five years since I first wrote about code spelunking⁹ and though systems continue to grow in size and scope the tools we use to understand those systems are not growing at the same rate. In fact, I believe we are steadily losing ground. So why should we go over the same ground again? Is this subject important enough to warrant two articles in five years? I believe it is.

The oft-quoted Moore's Law about the increasing power of computers actually works against the code spelunker. The more powerful computers become, the more we demand that they do, which increases the complexity of the software that runs on them. Processor speeds increase and that means more lines of code can now be run in the same amount of time. Available memory gets larger so we can now keep more state or code in memory. Disks get larger and require less power (in the case of flash), and suddenly we're able to carry around what were once considered huge amounts data in our pockets. What was termed the "software crisis" in the 1970s has never really

abated, because each time software engineers came up with a new way of working that reduced complexity, the industry moved forward and demanded more.

Complexity increases in many directions, including lines of code, numbers of modules, and numbers of systems and sub-systems. More complex systems require more lines of code to implement. As they grow their systems, software teams often integrate more code from outside resources, which leads to complex interactions between systems that may not have been designed with massive integration in mind.

These numbers should not be surprising to any software engineer, but they are a cause for concern. Although it was unlikely that the numbers would shrink, all but one of them grew by more than 50%, and although the number of lines may have grown linearly, the interactions among the new components that these numbers represent have not grown in a linear fashion. If we assume that all modules in a system can interact freely with all other modules, then we have a system in which the potential number of interactions is expressed as $n(n-1)/2$, an equation that should be familiar to those who work in networking as it represents a fully connected network. If a system grows from 100 modules to 200 modules, a 100% growth rate, then the number of potential connections grows from 4,950 to 19,900, a 302% growth rate.

One reliable measure of the number of interfaces into a system is the number of system calls provided to user programs by an operating system kernel. Since the publication of my first article on code spelunking the Linux kernel has grown from just shy of 200 system calls to 313, an increase of more than 50% (see Table 1).⁷

Two New Tools

My first article on code spelunking covered several tools, including global,³ Cscope,¹ gprof,⁴ ktrace,⁸ and truss.¹¹ I continue to use these tools on a daily

basis but in the past five years two new tools have come to my attention that, although they may not have been specifically designed with code spelunking in mind, both make significant contributions to the field. The tools are Doxygen² and DTrace.⁶ Here, I discuss each tool and how it can help us understand large code bases.

Doxygen. Right at the top of the Doxygen Web page² we find the following: “Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D.” As the blurb says, Doxygen was designed with documenting source code in mind—and it is quite a good system for documenting source code so that the output is usable as man pages and manuals—but it has a few features that make it applicable to code spelunking, too.

What Doxygen does is read in all, or part, of a source tree, looking for documentation tags that it can extract and turn into nicely formatted output suitable for documenting a program. It can produce Unix man pages, LaTeX, HTML, RTF, PostScript, and PDF.

What is most interesting for the code spelunker is Doxygen’s ability to extract information from any source code by running pre-processors over the code in question. Doxygen is a static analysis tool in that it analyzes the source code of a program but does not look into the program state while it is running. The great thing about a static analysis tool is that it can be run at any time and does not require that the software be executing. In analyzing something like an operating system, this is extremely helpful.

The features that make Doxygen most relevant to our work are those related to how data is extracted from the source code. When you start out with the intention to document your own code with Doxygen you are already working with the system and very little extra needs to be done. If you’re code spelunking an unknown code base then you will need to be more aggres-

sive and manually turn on certain features in the Doxyfile, which is Doxygen's configuration file. These features are listed in Table 2.

The option "HAVE_DOT" is the most important one because it's what allows Doxygen to generate the most useful output for the code spelunker, including class, collaboration, call, and caller graphs. We'll now take a brief look at two of these types of graphs. The code that we're analyzing in this article is the TCP/IP stack of the FreeBSD Operating System. The BSD TCP/IP stack has been studied in the past¹² and continues to be studied by researchers working on the TCP/IP protocol suite.

For our examples we will look at a single function, `ip_output()`, which is called in various parts of the network

stack in order to send an IP datagram to the network. The `ip_output()` function is quite important to the stack because all normal packet transmissions flow through it. If a bug was found in this function, or if the API needed to be changed for some reason, it would be important to trace back all of the current users (callers) of the function. In Figure 1 we see the caller graph produced by Doxygen for `ip_output()`.

In Figure 1 we see that no fewer than 16 separate routines, in nearly as many modules, depend on the `ip_output()` function. To effect a fix or update the API all of these routines will need to be investigated. The red-bordered boxes in Figure 1 show nodes that had a greater number of incoming edges than could be shown comfort-

ably in the graph, which is a good indication that the function contained therein is an important component of the system.

The opposite of a caller graph is a call graph. A call graph is more familiar to users of the tools mentioned previously, such as Cscope and global, which allow the user to interactively move through the call graph of a function, jumping into and out of underlying functions while browsing the source code. Doxygen gives us a different way of interacting with the call graph. Figure 2 shows the call graph for the `ip_output()` function.

The call graph, like the caller graph, gives us a good visual overview of how the function fits into the overall system. Both of these figures function as maps

Figure 1. Caller graph for `ip_output()`.

from which we can derive clues as to how the software is structured. One clue that is relatively easy to see is that there is another hot spot in the packet output code, namely `tcp_output()`, which is called from seven different routines.

The kind of information that Doxygen can show comes at a price. Generating the graphs shown here, which required analyzing 136 files comprising 125,000 lines of code, took 45 minutes on a dual-core 2.5GHz Macbook Pro laptop. Most of the time was taken up by generating the call and caller graphs, which are by far the most useful pieces of information to a code spelunker.⁵

DTrace. One of the most talked about system tools in the last few years is DTrace, a project from Sun Microsystems released under the CDDL that has been ported to the FreeBSD and Mac OS/X operating systems. Regardless of whether the designers of DTrace were specifically targeting code spelunking when they wrote their tool, it is clearly applicable.

DTrace has several components: a command line program, a language, and a set of probes that give information about various events that occur throughout the system. The system was designed such that it could be run against an application for which the user had no source code.

DTrace is the next logical step in the line of program tracing programs that came before it, such as ktrace and truss. What DTrace brings to code spelunking is a much richer set of primitives, both in terms of its set of probes and the D language, which makes it easier for code spelunkers to answer the questions they have. A program like ktrace only shows the system calls that the program executes while it's running, which are all of the application's interactions with the operating system. On a typical OS these number in the low hundreds, and while they can give clues to what a complex piece of software is doing, they are not the whole story. Ktrace cannot trace the operating system itself, which is something that can now be accomplished using DTrace.

When people discuss DTrace they often point out the large number of probes available, which on Mac OS X is more than 23,000. This is somewhat misleading. Not all of the probes are

Table 1. Comparing the sizes of the systems as discussed in 2003 and today.

Program	Version	Files	Lines	% Chg Lines
Apache Web Server	1.3	471	158,332	
	2.2.8	1108	374,993	136%
Emacs	21	2586	1,317,915	34%
	22	2598	1,771,282	
FreeBSD Kernel	5.1	4758	2,140,517	
	7.0	6723	3,556,087	66%
Linux Kernel	2.4.20-8	12417	5,223,290	
	2.6.25-3	19483	8,098,992	55%
Python	2.2.3	1158	356,314	
	2.5.2	2379	910,573	155%

Table 2. Features in the Doxyfile.

Feature	Meaning
EXTRACT _ ALL	Extract everything you can from the source code.
SOURCE _ BROWSER	Create a full cross-reference of the source code.
CLASS _ DIAGRAMS	Create class diagrams and inheritance graphs.
HAVE _ DOT	Create useful code spelunking graphs.
CALL _ GRAPH	Makes a call graph following all function calls.
CALLER _ GRAPH	Outputs a graph of the caller dependencies.

Table 3. Providers available in Mac OS X.

Provider	Purpose
dtrace probes	related to dtrace itself
fbt	entry and exit points for functions
io	I/O probes
lockstat	Probes related to locking
plockstat	pthread lock related probes
proc	Process specific information
profile	Profiling and performance data
syscall	Information on system calls
vminfo	Virtual Memory probes

immediately usable, and in reality, having such an embarrassment of riches makes picking the most useful probes for a particular job difficult. A probe is some piece of code in an application, library, or the operating system that can be instrumented to record information on behalf of the user. The probes are broken down into several categories based on what they record. Each

probe is delineated by its Provider, Module, Function, and Name. Providers are named after systems such as io, lockstat, proc, profile, syscall, vminfo, and dtrace itself. There are several distinct providers available in Mac OS X, although naively printing them all will show you that several exist on a per-process basis. The per-process probes show information on core data within

Figure 2. Call graph for `ip_output()`.

the process as well as on locks. Some of the providers available on Mac OS X are shown in Table 3.

Probes are not only associated with providers but also with modules, which are the modules you want to

instrument, as well as with functions, which are also subject to observation. The name of a trace point specifies a single probe. All of these categories are put together in a DTrace script or command line to form a sort of address

that specifies what the engineer is trying to observe. The canonical form is provider:module:function:name, with an empty section indicating a wildcard. Although the two manuals from Sun are excellent references on DTrace,^{6, 10}

Figure 3. Abbreviated output from the execution of ls under DTrace.

```
> sudo dtrace -s calls.d -c ls
dtrace: script 'calls.d' matched
5906 probes

[output of ls command removed for brevity]

dtrace: pid 7008 has exited

strcoll          148
strcoll_1        148
__error          326
free             353
strcmp           381
wcwidth          614
__none__mbrtowc  662
mbrtowc          662
putchar          705
pthread_getspecific 1424

>
```

a quick example will demonstrate how it can be used for code spelunking.

When presented with a new and unknown system to spelunk one of the first things to find out is which services the program uses. Programs like ktrace and truss can show this type of information but DTrace extends this ability greatly. We will now find out what services the ls program requires to execute as well as which ones are used most often.

```
1: pid$target:::entry
2: {
3:   @[probefunc] = count();
4: }
```

The script here is written in the D language and should be relatively easy to decipher for anyone familiar with C. The script contains a single function, which counts the entry into any call that the ls program makes. Where a C programmer might find a function name and argument list we instead see what is called a predicate. The predicate is a way of selecting the probes that DTrace will record data for. The predicate on line 1 selects the entry into any call for the associated process. When the calls.d script is executed with dtrace in Figure 3, its pid\$ variable is replaced with the process ID of the program that is given after the -c command-line argument.

DTrace also allows the tracing of live processes by replacing -c with -p and the program name with a live process ID. Figure 3 gives abbreviated output

from the execution of ls under DTrace. Only the last several lines, those with high entry counts, are shown. From this snapshot we can see that ls does a lot of work with the string functions strcoll and strcmp, and if we were trying to optimize the program we might look first at where these functions were called.

With thousands of predefined probe points, and the ability to dynamically create probes for user processes, it's obvious that DTrace is the most powerful code spelunking tool developed in the last decade.

Continuing Challenges

In reviewing the tools mentioned here—as well as those that are not—a few challenges remain apparent. The first challenge is the move by some developers away from a tool-based approach to an all-in-one approach.

A tool-based approach can best be understood by looking at the programs available on any Unix-like system. The use of several programs, mixed and matched, to complete a task has several obvious benefits that are well documented by others. When working with large code bases, the downfalls of an all-in-one approach, such as an IDE, become a bit clearer. A system such as the FreeBSD kernel is already several hundred megabytes of text. Processing that code base with tools like Cscope and global in order to make it more easily navigable generates a further 175MB of data. Although 175MB of data may be small in comparison to the memory of the average desktops or laptops, which routinely come with 2GB to 4GB of RAM, storing all that state in memory while processing leads to lower performance in whatever tool is being used. The pipeline processing of data, which keeps in-memory data small, improves the responsiveness of the tools involved. Loading the FreeBSD kernel into Eclipse took quite a long time and then took up several hundred megabytes of RAM. I have seen similar results with other IDEs on other large code bases.

An even larger challenge looms for those who work on not only large, but heterogeneous, code bases. Most Web sites today are a melange of PHP or Python with C or C++ extensions, using MySQL or PostgreSQL as a database backend, all on top of an OS written in C. It is often the case that tracking

down particularly difficult problems requires crossing language barriers several times—from PHP into C++ and then into SQL, then perhaps back to C or C++. Thus far I have seen no evidence of tools that understand how to analyze these cross-language interactions.

The area that deserves the most attention is visualization. Of all the tools reviewed, only Doxygen generates interesting and usable visual output. The other tools have a very narrow, code-based focus in which the user is usually looking at only a small part of the system being investigated.

Working in this way is a bit like trying to understand the United States by staring at a street sign in New York. The ability to look at a high-level representation of the underlying system without the fine details would be perhaps the best tool for the code spelunker. Being able to think of software as a map that can be navigated in different ways, for instance, by class relations and call graphs, would make code spelunkers far more productive.

One last area that has not been covered is the network. Network spelunking, the ability to understand an application based on its network traffic, is still in a very nascent state, with tools like Wireshark being the state of the art. Many applications are already running online and to being able to understand and work with them at the network level is very important. ■

References

1. CScope Man Page; http://cscope.sourceforge.net/cscope_man_page.html.
2. Doxygen Web Site; <http://www.stack.nl/~dimitri/doxygen/>.
3. GNU GLOBAL Source Code Tag System. Tama Communications Corp., Apr. 21, 2008.
4. gprof; <http://www.gnu.org/manual/gprof-2.9.1/gprof.html>.
5. Graphviz Web Site; <http://www.graphviz.org/>.
6. How To Use DTrace. Sun Microsystems, 2005. Available on the Web at <http://www.sun.com/software/solaris/howtoguides/dtracehowto.jsp>.
7. HPC System Call Usage Trends. Terry Jones, Andrew Taufner, Todd Inglett Linux Clusters Institute 2007.
8. ktrace: standard tool on open source OSes.
9. Neville-Neil, G.V. Code spelunking: Exploring cavernous code basis. *ACM Queue* (Sept. 2003). ACM, NY.
10. Sun Microsystems. Solaris Dynamic Tracing Guide 2005; <http://docs.sun.com/app/docs/doc/817-6223>
11. Truss is available on Solaris.
12. Wright, G.R. and Stevens, W.R. *TCP/IP Illustrated, Vol. 2: The Implementation*. Addison-Wesley Professional. 1995.

George V. Neville-Neil (gnn@acm.org) is a columnist for *Communications* and *ACM Queue*, as well as a member of the *Queue* Editorial Board. He works on networking and operating system code and teaches courses on various subjects related to programming.

© 2008 ACM 0001-0782/08/1000 \$5.00