Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

OSTRA: Experiments With on-the-fly Source Patching

Arnaldo Carvalho de Melo
Mandriva Conectiva S.A.
acme@mandriva.com
acme@ghostprotocols.net

Abstract

OSTRA is an experiment on on-the-fly source patching, where codebases like the Linux kernel are "compiled" with occ, the OSTRA compiler, that inserts code to collect trace information to be post processed using tools available to generate fancy HTML and CSS2 callgraphs that show some profiling information, tables showing where specific data structure fields changed, graphs of all the observed internal state, etc.

1 Introduction

This paper talks about experiments in *source patching*, where tools are used to "compile" programs written in the C language, modifying the source code according to some criteria specified by the programmer.

Two experiments will be discussed, with tools developed by the author presented, shedding some light on the possibilities of source patching.

Ideas discussed with some fellow developers but not yet tried will also be presented, with the intent of—hopefully—having them finally tested in practice by interested readers.

2 Sparse

Sparse was (ab)used in the experiments described in this paper, both as a simple tokenizer in some tools and using some of its more advanced "semantic parsing" capabilities in other tools.

More information on sparse[1] can be obtained in its on-line source repository.

3 Experiment 1: initstr

I first got interested in source patching when trying to shrink the Linux kernel binary image by marking strings in initialization functions as `__initdata`, so that they would be put into the .init.data ELF section.

This involved the repetitive process of transforming code like:

```
int __init feature_init(void)
{
    if (alloc fails)
        panic("feature: not enough
memory!\n");
}
```

Into:

```
static char panic_msg[] __initdata
=
    "feature: not enough
memory!\n";

int __init feature_init(void)
{
    if (alloc fails)
        panic(panic_msg);
}
```

So I thought about doing this automatically, using some "pre-compiler," a tool to be used before the real compiler, that would modify the code for functions marked `__init`, inserting code that would mark the strings in arguments to functions as `__initdata`.

The initstr[2] tool was the result of this experiment, that to be really usable would require further work as there are cases where strings can not be blindly marked `__initdata` as they are referenced in kernel data structures, such as in the `kmem_cache_create` function, that would have to be modified to make a string duplicate of the received slab cache name, and also because Linus at the time thought sparse was not robust enough to be used in the process of building a production kernel image, perhaps this has changed and I encourage interested persons to try again as the tool was left available in my kernel.org site.

4 Experiment 2: Tracing

The initstr tool idea was later useful when the author was working on the his DCCP paper for OLS 2005[3], where, to illustrate refactorings done on the networking core to share TCP code with DCCP callgraphs were in demand.

To collect the information needed for such callgraphs the idea was to write tools that would find the functions of interest and insert code at the start and at the end of these functions, that would record in a ring buffer these events.

The criteria devised to identify the functions of interest were: functions that are "methods" of some "class", i.e. functions that receive as one of their parameters a pointer to some specified structure, for instance `struct sock`.

As the tool would have to look at each of the parameters of each of the parsed functions to see if they were a pointer of the specified class, another desired feature was added: to specify members of this class to be collected at entry and/or exit of the methods, so as to look at the internal state of the objects being traced at each trace point.

The following sections will talk about the tools written to achieve such goals.

4.1 ostra-grep

The first tool written was called ostra-grep, that, as suggested by its name, "greps" the code being compiled for functions that are methods of the specified class.

It was used as a replacement to sparse's "checker" tool, using the Linux kernel Makefile CHECKER parameter.

It just creates files with the name as the source file parsed plus a ".ostra" suffix, where each of the methods found would be recorded together with the names of the parameters that are of the class specified.

This mode of operation is interesting as it makes ostra-grep useful for other purposes, such as using another tool, ostra-kprobes, that uses a different approach for collecting the callgraph trace points, namely creating a kprobes module that hooks all the methods found.

4.2 ostra-patch

After the ostra-grep discovery pass another tool, ostra-patch, is used as a replacement for gcc, that looks if the file being compiled has methods to hook, short circuiting to gcc if not.

If the file has methods to hook ostra-patch will use sparse in its most basic tokenizer form, just to get to the methods found by ostra-grep and insert code at the start of the function, calling a trace entry collector function, `ostra_entry_hook()`, passing the pointer to the class in its parameter list and a identifiers for the source file and function.

It then looks for all the "exit points", i.e. all the return statements and the implicit one in void functions and inserts calls to `ostra_exit_hook()`, passing the same information passed to `ostra_entry_hook()` plus the "exit point" identifier, i.e. a sequential number telling which of the return statements was used in this specific function call, providing further useful information for the callgraph generator tool, ostra-cg.

Other criteria for specifying where to install trace points that can be implemented in the future is for *operator overloading*, that is to insert trace points when members of the class are being changed or plain referenced, when calls to something like `ostra_operator_foo_hook()` would be inserted.

The hook functions are defined in a separate file that has to be linked in.

5 Future Directions

Write it!

References

[1] `http://www.kernel.org/git/?p=devel/sparse/sparse.git`

[2] `http://www.kernel.org/pub/linux/kernel/people/acme/sparse/initstr.c`

[3] Arnaldo Carvalho de Melo, 2005. "DCCP on Linux", Ottawa Linux Symposium