Self Optimizing RTOS, a Proof of Concept

Alex Rosenfeld, Daniel Medina, Emilio Garcia, Leo Tomatsu, Young Jun Choi
Renato Mancuso, Clark Williams, Daniel Bristot de Oliveira

Goals

- Learn to profile/analyze applications in Linux
- Modify the Linux kernel to be able to read a new ELF section, and create a hello world example
- Present research ideas on how Linux could use the program metadata to optimize and run differently

Summary

What if your operating system could collect data on an application's performance, and optimize itself to run it faster? In Linux systems, all programs are run using Executable and Linkable Format (ELF) files, which are made up of sections containing data on how to run a program. We created a linux 4.14 kernel that can read a new ELF section that contains program metadata. We built some basic tools to trace apps and parse the resulting data. The results of the parser can be scanned for potential optimizations, which can then be written into an ELF file and observed by our modified kernel. The system will then be able to run the app in an optimized way.

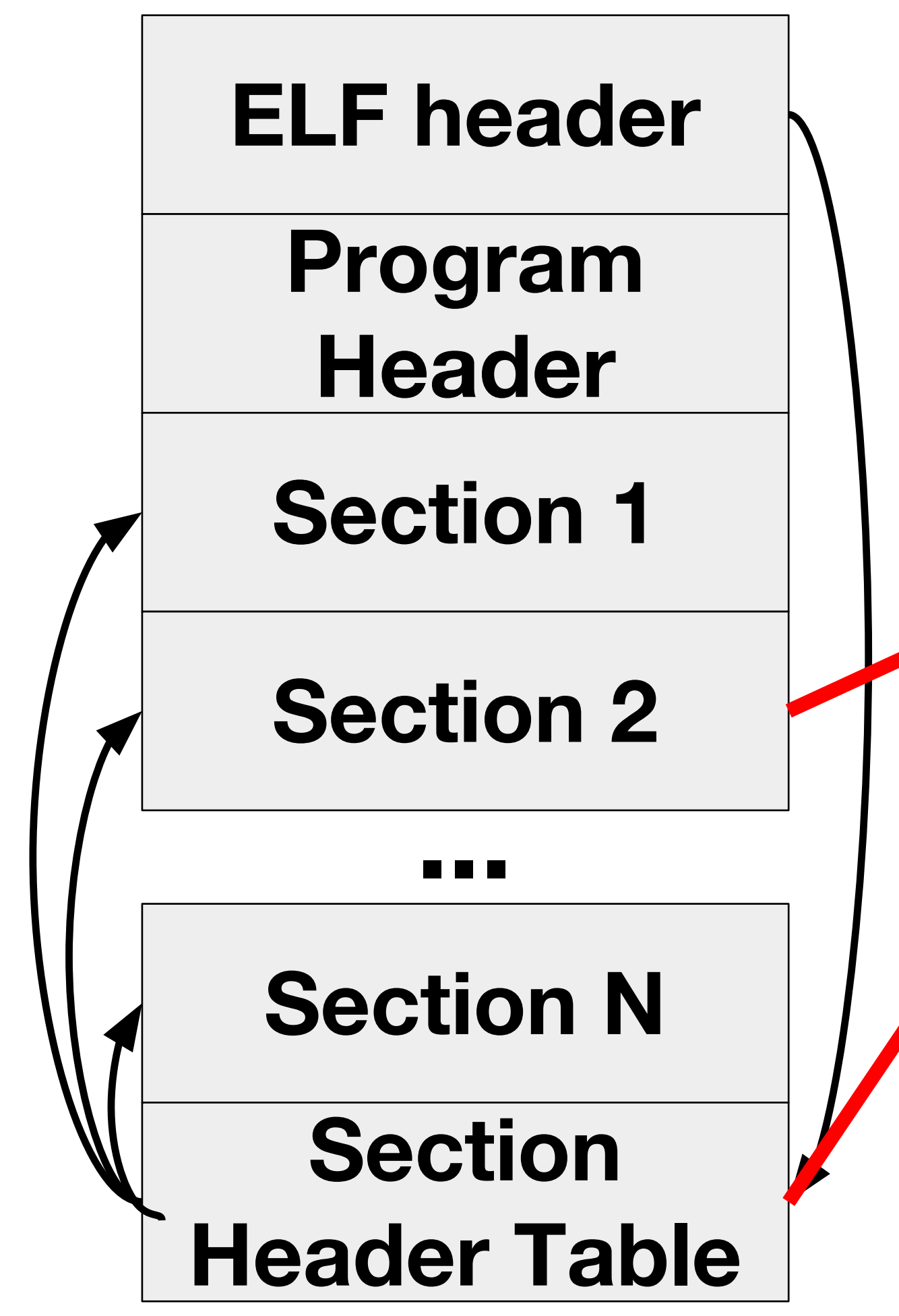
Next Steps

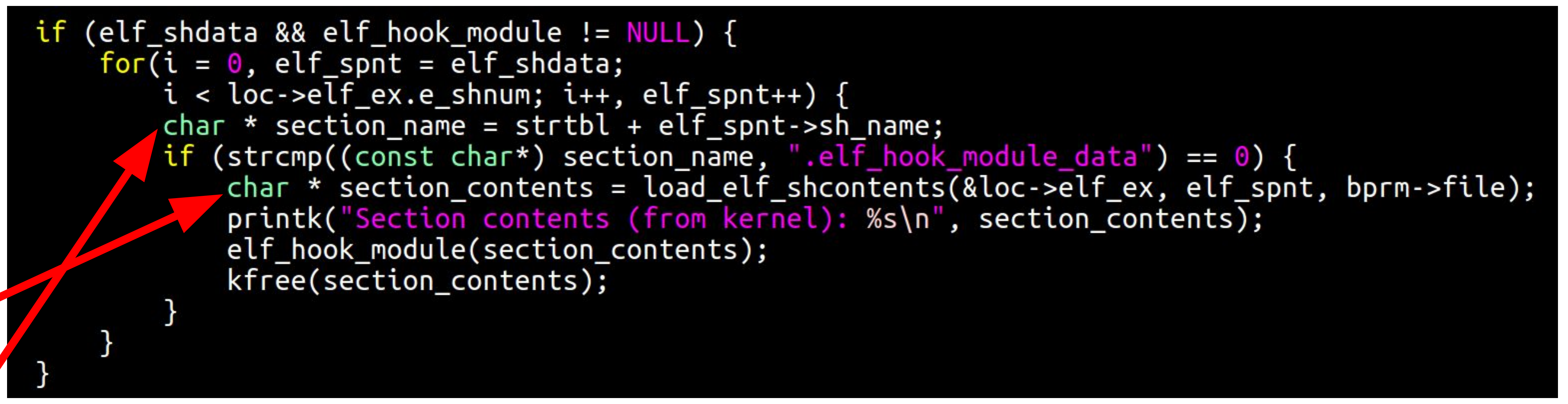
Combining our work on the Linux kernel with the tracing tools and knowledge we developed, the next logical step is to enable the OS to trace application performance, find optimizations, then write these to the application's new ELF section.

The following are examples of future use cases for our research:

1. Relocate high utilization kernel functions into scratchpad on ARM
2. Prevent cache evictions of high traffic memory regions using mlock
3. Defer Bottom Halves of unessential interrupts

Kernel Modification

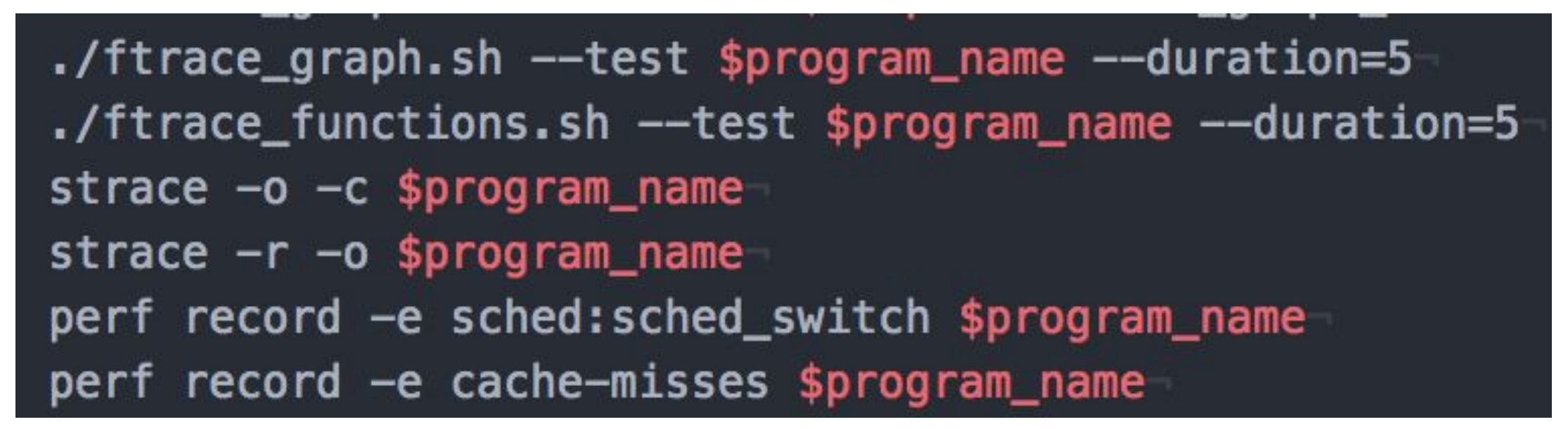




```
if (elf_shdata && elf_hook_module != NULL) {
    for(i = 0, elf_spnt = elf_shdata;
        i < loc->elf_ex.e_shnum; i++, elf_spnt++) {
        char * section_name = strtbl + elf_spnt->sh_name;
        if (strcmp((const char*) section_name, ".elf_hook_module_data") == 0) {
            char * section_contents = load_elf_shcontents(&loc->elf_ex, elf_spnt, bprm->file);
            printk("Section contents (from kernel): %s\n", section_contents);
            elf_hook_module(section_contents);
            kfree(section_contents);
        }
    }
}
```

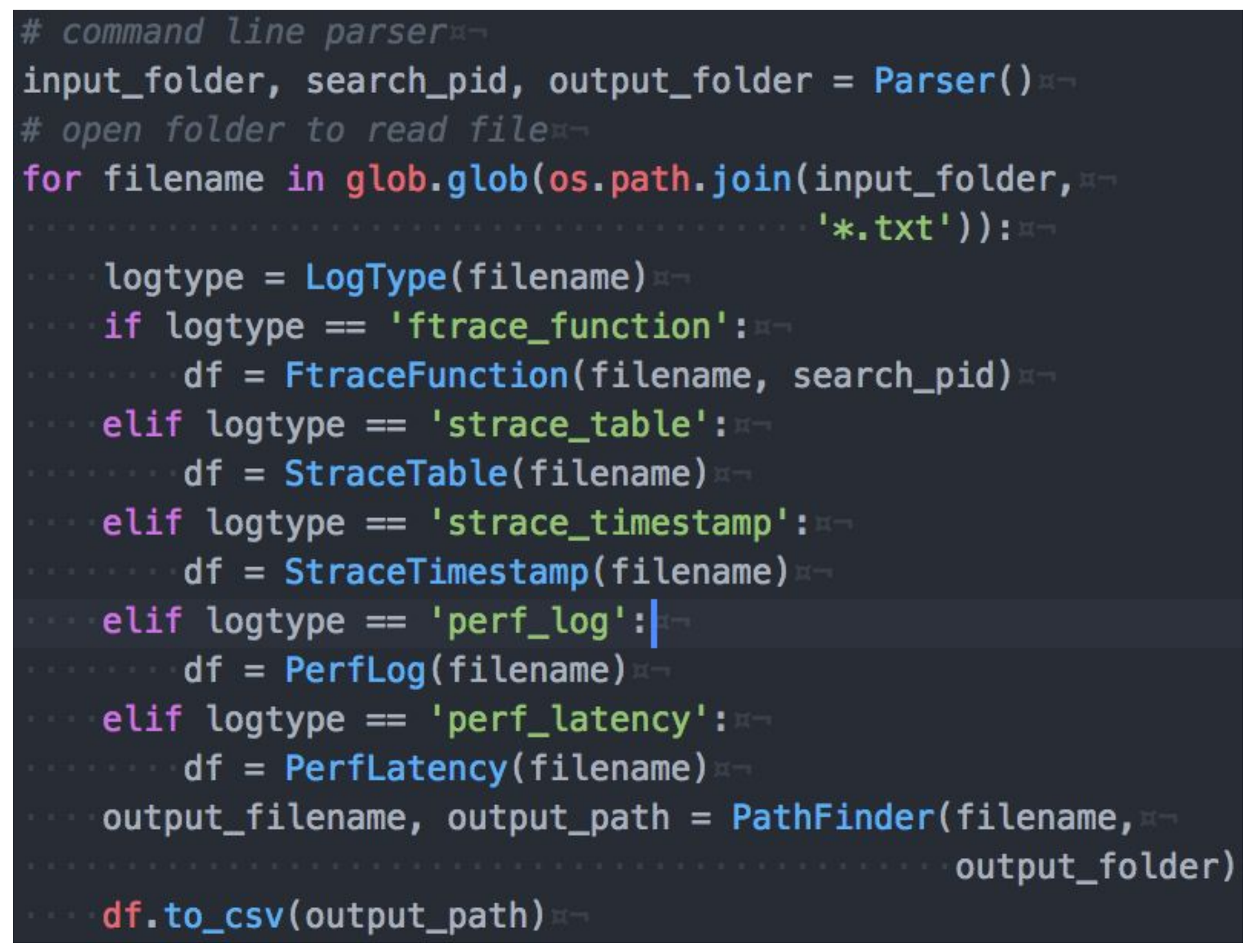
We modified how the kernel loads ELF files. In the function, we added a loop to find a new section and extract the data from this section, and pass it to a module. In order to show it works, we print the content of the section to the message buffer of the kernel.

Kernel Analysis



```
./ftrace_graph.sh --test $program_name --duration=5
./ftrace_functions.sh --test $program_name --duration=5
strace -o -c $program_name
strace -r -o $program_name
perf record -e sched:sched_switch $program_name
perf record -e cache-misses $program_name
```

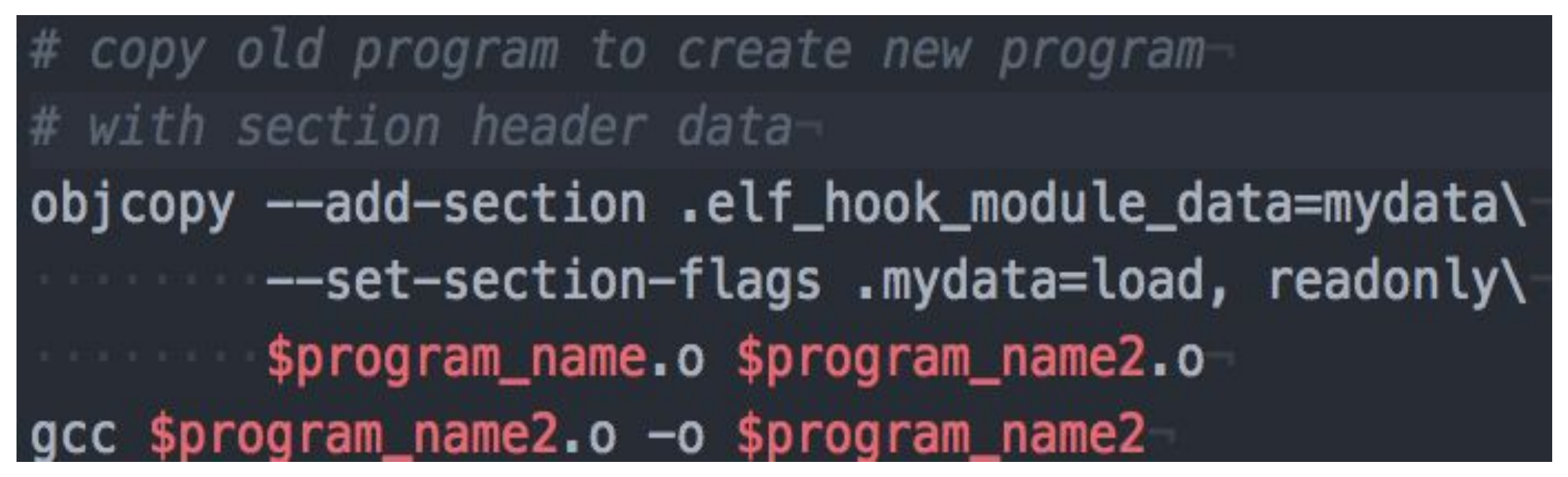
We ran trace functions as follows: Ftrace detected linux kernel functions. Strace recorded syscalls by a process. Perf showed system events associated with the target.



```
# command line parser
input_folder, search_pid, output_folder = Parser()
# open folder to read file
for filename in glob.glob(os.path.join(input_folder,
                                       '*.txt')):
    logtype = LogType(filename)
    if logtype == 'ftrace_function':
        df = FtraceFunction(filename, search_pid)
    elif logtype == 'strace_table':
        df = StraceTable(filename)
    elif logtype == 'strace_timestamp':
        df = StraceTimestamp(filename)
    elif logtype == 'perf_log':
        df = PerfLog(filename)
    elif logtype == 'perf_latency':
        df = PerfLatency(filename)
    output_filename, output_path = PathFinder(filename,
                                              output_folder)
    df.to_csv(output_path)
```

We wrote a python script to parse user input commands to specify the input folder with trace log files, program name under inspection, and the output folder. Each file inside the folder with be type checked to run the specified process to compute the logfile. Below is an example of an ftrace chart.

syscall	time	usecs/call	seconds	calls
nanosleep	33.73	847738	1.695475	2
gettimeofday	33.17	416863	1.667452	4
futex	33.03	227215	1.660506	7
write	0.02	42	0.001135	27



```
# copy old program to create new program
# with section header data
objcopy --add-section .elf_hook_module_data=mydata\
        --set-section-flags .mydata=load, readonly\
        $program_name.o $program_name2.o
gcc $program_name2.o -o $program_name2
```

To create an ELF file with our section, we utilized the objcopy tool. We copied our previous ELF file and added our custom section to contain our program metadata for kernel access.