TCCLK: A Port of the Tiny C Compiler into the Linux Kernel

James Whang
Northwestern University
sungyoonwhang2017@u.northwestern.edu

Taiwon Chung
Northwestern University
taichung2014@u.northwestern.edu

March 21, 2015

Abstract

Kernel development is difficult, because it involves too many steps for just the code to go through the kernel. Writing a Linux kernel module is a big pain, and the compilation and execution of a simple line of code such as: printk("I am a bug!") takes multiple unnecessary steps. We explored and implemented a way to address this problem by porting the Tiny C Compiler, written by Fabrice Bellard, into the kernel. The ported kernel C compiler is fully capable of executing "C scripts" and running kernel functions directly from the user level.

1 Introduction

1.1 The Tiny C Compiler

The Tiny C Compiler (TCC) is a fully functional C compiler with no external linker or assembler. Like its name suggests, TCC is tiny in size, and is only about 100 KB of executable when compiled in the user space, and in our port to the kernel, it becomes about 200 KB, which is a reasonable size for a kernel module. Furthermore, TCC is designed to be portable to many different platforms and architecture, which makes facilitates the port to kernel. Most importantly, TCC runs much faster than the GCC, which is ideal for the kernel environment where speed is important. [1]

1.2 Motivation

The initial motivation came from the fact that the barrier to entry for kernel development is high. Even getting the simplest kernel module that prints out "Hello world!" involves more steps than just calling printk function. In addition, this makes it difficult for even experienced kernel developers to do the simplest type of jobs. Figuring out how to set up init and deinit functions in a kernel module and writing a Makefile that works on your specific environment are some of initial hurdles that are required for new kernel developers to overcome in order to write a simple hello-world module. However, it would be nice if we can lower the bar of entry by somehow getting rid of those initial configuration steps so that developers can focus on main logic of the module.

Many script languages such as Python and Perl support the use of `eval()` , which executes another Python script inside a Python script. This is not supported in C, but it is possible to do this with TCC.

2 TCC Module Implementation

TCC module is a kernel module that uses libtcc, which is a library that contains the core functionality of TCC such as compilation, linking, and executing. This library has a struct called TCCState to store variables necessary for compiling and linking of codes given to it. It can be viewed as a separate execution environment for the code, which in a way resembles other language runtime virtual machines, such as the JVM. Our module is a port of this library and its functionality into the kernel module.

2.1 Design

Before going over the details of the port process, we go over the high-level design of the module and user-level service around it. When TCC module is initialized, it creates a character device named tcc_dev, which is used as an interface between the kernel and the user spaces. We provide a user-level service, tcc_service, which uses the tcc_dev and writes the C source code to be compiled and executed to the device as a string. When write call to the device is invoked from the user-level tcc_service, the kernel module then compiles and executes the program. After the program is executed, tcc_service closes the device and exits. After the execution, the user may execute the linux command `dmesg` to view any output of result to the memory.

Figure 1: Module-Service Design

2.2 Modifying libtcc

To make libtcc usable in kernel environment, we eliminated most of dependencies to C standard library with help from Prof. Dinda. First, we converted calls to stdlib functions into equivalent functions in kernel. For example, we replaced calls to printf and fprintf with printk. Then, we wrote stubs for functions that simply do not exist in kernel space. Some examples of stubs can be found below.

```
// Examples of stub code in tccglue.h
static inline FILE *fdopen(int fd, const char *mode)
    { printk("Write fdopen()\n"); return 0;}
static inline int fclose(FILE *f) { printk("Write
    fclose()\n"); return 0; }
static inline int fflush(FILE *f) { printk("Write
    fflush()\n"); return 0; }
static inline int fputc ( int character, FILE *
    stream ) { printk("Write fputc()\n"); return 0; }
static inline int fputs(const char *str, FILE *f) {
    printk("Write fputs()\n"); return 0;}
static inline size_t fwrite ( const void * ptr,
    size_t size, size_t count, FILE * stream ) {
    printk("Write fwrite()\n"); return 0;}
static inline int vfprintf ( FILE * stream, const
    char * format, va_list arg) { printk("Write
    vfprintf()\n"); return 0;}
static inline int unlink(const char p) {
    printk("Write unlink()\n"); return 0;}
```

On top of this, we had to disable opening of a static library file that libtcc tries to open at some point in compilation process, since opening a file in kernel space may not be a good idea. Also, this library file has a subset of libc, which is unnecessary to include since program given to this libtcc will be run in kernel space, so there is no need to link with user-level library.

We also added a few lines of codes in a function in tccelf.c that does undefined symbol resolution to look up symbols across the entire kernel so that program that user passes in can use any symbol inside the kernel. This basically enabled libtcc to link the given program with kernel. Below is the added code in tccelf.c.

```
// code for kernel symbol look-up in tccelf.c
#ifdef __KERNEL__
    void * func;
    func = __symbol_get(name);
    if (func != NULL) {
        sym->st_value = (addr_t)func;
        goto found;
    }
#endif
    tcc_error_noabort("undefined symbol '%s'", name);
```

2.3 Code Execution Process

As mentioned above, libtcc has TCCState struct to store any information necessary for compilation of a given program. Every compilation requires a new TCCState, so we create this right after call to write is invoked to TCC's character device. Then, we set the output type of compilation to memory, which means that after TCC is done compiling, it will output the compiled program to memory, instead of a file.

We can then compile and relocate the program to executable region of memory, while creating a symbol table in TCCState. The relocation step actually caused some issues with memory that could not be fully resolved, which is explained in Section 3 of this paper. Now that we have our program compiled and relocated in the executable part of memory, we look up the name of the function "main" in the symbol table, get a pointer to the function in return, and call the function. After we are done, we delete the state to prevent memory leakage. The complete process in C is shown below.

```
    TCCState *s;
    int (*func)();
    printk("Starting tcc_new\n");
    s = tcc_new();
    if (!s) {
        printk("Could not create tcc state\n");
        return 0;
    }
    printk("Starting tcc_set_output_type\n");
    // MUST BE CALLED before any compilation
    tcc_set_output_type(s, TCC_OUTPUT_MEMORY);

    printk("Starting tcc_compile_string\n");
    if (tcc_compile_string(s, user_program) == -1) {
      printk("Cannot compile program!\n");
      return 0;
    }
    printk("Starting tcc_relocate\n");
    // relocate the code
    if (tcc_relocate(s, TCC_RELOCATE_AUTO) < 0) {
      printk("Cannot relocate program\n");
      return 0;
    }
    printk("Starting tcc_get_symbol\n");
    // get entry symbol
    func = tcc_get_symbol(s, "main");
    if (!func) {
      printk("Cannot find main...\n");
      return 0;
    }
    printk("Calling the code!\n");
    int result = func();
    printk("Result is %d\n", result);
    tcc_delete(s);
```

Having the code copied, compiled, executed, and freed all at once limits the "lifetime" of the user-supplied kernel code. Refer to section 5.1 for further discussion of this problem.

2.4 Using tcc_service

tcc_service is the name of the user-level service written for interaction with the TCC kernel module. This user program supports two modes: user-input mode and file-input mode. As their names suggest, user-input mode takes input from the user directly through stdin, and file-input mode takes input from a file path specified by the user.

3 Results

We tested using a few functions in kernel, such as printk and panic, to see if script from user is getting linked successfully with the entire kernel, which was successful. However, global variables from the kernel, such as jiffies, were not accessible because this was considered as undeclared variable, which generated compilation error.

```
// this worked
int main() {
    printk("Hi!\n");
    panic();
    return 0;
}

//this did not
int main() {
    printk("Current jiffies: %lu.\n", jiffies);
    return 0;
}
```

We also tested if TCC module can recognize symbols that are exported from other modules use them. For example, if there is a module that has a function named "foo" that takes no input, and it has exported the symbol foo, we tested if we can call this function in user-generated script that looks like below.

```
// definition of foo in another module
static foo(void) {
    printk(KERN_INFO "Hello from another module");
    return 0;
}

// user script input to TCC module
int main() {
    printk("hi!\n");
    foo();
    return 0
}
```

We verified that foo indeed gets called, with a slight problem in reference counts of modules after exeucution, which is describe in the next section.

4 Existing Problems

4.1 NX-Protection Bit

NX-Protection bit is a hardware-enabled feature to protect the heap memory against any execution of code. From a security perspective, having the user be able to execute code directly from the heap is not ideal. Allowing this could result in stack/buffer overflow attacks.

For the purpose of this module, however, being able to execute directly from the memory region is necessary. Therefore, we need to be able to get around this.

TCC in the user-level service uses the system call `mprotect()`, which turns off execution-protection for the stack. Unlike the user-level TCC service, we allocate memory directly in the kernel heap. Furthermore, the system call `mprotect()` does not exist in the kernel.

The first attempt to get around this issue was to use the function `__vmalloc()` with appropriate flags. This kernel function allocates a page of memory, and it is possible to turn off the NX-protection bit when we allocate this page through this function call. The part of the code that deals with memory is shown below:

```
void * tcc_kmalloc(size_t n)
{
  return __vmalloc(n, GFP_ATOMIC, PAGE_KERNEL_EXEC);
}

void * tcc_krealloc(void *p, size_t n)
{
  return krealloc(p, n, GFP_ATOMIC);
}
```

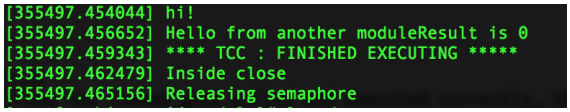
This successfully enabled the execution of code in the kernel heap memory, but caused a problem when we tried to free the memory. Freeing the memory allocated with the call to vfree does not work, because during tcc_relocate process shown in Section 2.3, the user's kernel code moves around with calls to krealloc. Before freeing any memory, vfree() does a sanity check by checking whether the pointer passed to it is a page address. Because the addresses have been changed by calls to krealloc, this is no longer true and causes the calls to vfree() to fail.

This is not a problem that we have been able to overcome so far. We will mention possible approaches to resolving this problem in the next section. As a temporary fix, we turned off the NX-protection bit setting from the test machine's GRUB configuration.

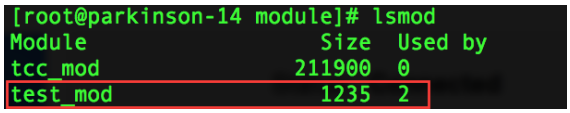
4.2 Working With Other Modules

While we verified an external symbol, foo, from a different module can be accessed and executed, the reference count for module that implements foo increases by 2 and never goes down even after removing TCC module, as shown in Figure 2 below. We suspect that this has something to do

with calling a function in kernel module from a user-level thread.



(a) dmesg



(b) lsmod

Figure 2: Results of running dmesg and lsmod after calling foo

5 Future Directions

5.1 Executable Memory Allocation

We need to address the issue with executable memory allocation problem mentioned in Section 4.1. To resolve this, we would need to have our own TCC kernel memory allocator, which basically allocates a number of pages that can be used by the TCCState. The calls to tcc_realloc() and tcc_free() should be handled by the TCC memory allocator. Once the kernel exits, it should free the memory that it allocated in the first place. This is a functionality that has not been implemented yet but should be in the near future.

5.2 Lifetime of User Code

Our current module takes the program given by user, compiles/executes it, and then throws it away. It does not save nor makes the user-written codes visible to any other module in kernel. To make TCC module more versatile in the future, we could add in functionality to insert user written functions into the kernel and make them visible to other kernel modules.

6 Conclusion

TCCLK has a fully working kernel module C compiler which is able to execute kernel functions directly from user level. Simple direct calls to kernel functions work without problems, such as panic() and printk(). It it also possible to call functions defined in other kernel modules.

7 Source Code

We hope to continue working on improving TCCLK and made a Git repository for TCCLK. The TCCLK Git repository is publicly available at http://github.com/jameswhang/tcclk .

References

[1] Fabrice Bellard. Tiny c compiler. http://bellard.org/tcc/, 2 2013.