

Lab 7

In this lab, we'll discover the wonders of self-modifying code.

Executing “data”

Consider the following assembly program:

```
.text
main:
    la    $a0, hello
    la    $t0, prog
    jr    $t0

.data
hello: .asciiz "Hello World\n"
.align 2
prog:  .word 0x34020004
       .word 0x0000000c
       .word 0x03e00008
```

It only has three instructions in the text segment and then jumps into the data segment. This is not normally allowed by `spim` since one of the main reasons to have separate text and data segments is to separate instructions from data, but I have provided executables `spim-unsafe` and `xspim-unsafe` that relax this restriction. (They are otherwise identical to `spim` and `xspim`.) Use `spim-unsafe` to execute the program above (conveniently pre-typed and available as `/home/courses/cs201/hello.s`).

You'll see that the program prints “Hello World” and then exits. Perhaps not too surprising since you can see the string in the data segment, but how is it done?

This program relies on that fact that machine instructions are actually sequences of zeros and ones, i.e. binary numbers. Remember that assembly language is just a way to present these instructions to make them easier for people to read (albeit not as easy to read as a high-level language like Java). Before the computer runs the instructions of a program, they are translated into binary by the assembler (or, in our case, translated into binary by `spim-unsafe`, which runs the machine instructions itself rather than passing them on to the actual hardware). Only the `text` segment is affected since the `data` segment does not normally contain instructions. For the program above, I converted some instructions to binary by hand and put them into the `data` segment as data for the program. (Actually, I cheated and used `xspim` to translate the instructions for me.) In fact, the three words beginning with the label `prog` correspond to the following assembly instructions:

```
li    $v0, 4
syscall
jr    $ra
```

The idea of treating instructions as data is fundamental to computer science. It is central to all the programming tools you've ever used. For example, a compiler is just a tool that takes one form of data (Java instructions) and converts them into another (Java bytecode). Your program may be instructions to you, but it is data to `javac`.

A particularly cool application of this idea is *self-modifying code*, programs that change themselves during execution. Although self-modifying code is something of an oddity now, it has been used for efficiency in

the past (this type of program can potentially be shorter since instructions can be “reused”) and it was (is?) used to copy-protect software. One such copy-protection scheme was to ask for a license number when the program starts. This number is used as a key to decrypt the rest of the program; without it, the program can not run.

Now you get to write a simple self-decrypting program. Copy the file `/home/courses/cs201/todecode.s` into your directory. This file is entirely a data segment containing a single word labeled `key`, followed by a sequence of words that form an encrypted program. To run this program, you’ll need to write a `main` function that decrypts it and then jumps to `begin`. (It is important that you perform this jump by loading the address into a register and then using `jr`.) To decrypt a word, you replace it with the xor of that word and the word labeled `key`. (Conveniently, MIPS assembly has an instruction `xor`.)

If you have extra time, write your own encrypted program. I wrote an encryption program and used `xspim` to stop right before it exited so I could read the instructions out of memory. Note that you can’t write to the text segment of a program so you’ll need to put the encrypted instructions into the data segment.