Department of Computing and Information Systems
COMP20006 Programming the Machine
Semester 1, 2012
Project 2

1 Aim

The main aim of this project is to give you experience writing a substantial piece of C code, using pointer data structures. The secondary aim is for you is to gain experience with version control and program testing.

2 Deadline

The deadline for this project is 11pm on Friday May 25th

This project is an individual project in which the submitted files must be your own individual work.

3 Task specification

Your task in this project is to write a C program named `virus.c`, which is intended to check a MIPS machine code program to see if it may contain a virus. The program will be controlled by command line options. By default (when called with no options) it should run only Stage E. Otherwise it should run stage A if it has `-A` in its options, stage B if it has `-B` in its options, etc. For example

```
virus -A -C -F
```

will run stages A, C and F in turn. If your program does not support a requested stage it should `exit(1)` after printing a message to `stdout` of the form

```
Stage ? not supported
```

where `?` is the unsupported stage, before outputting any information from previous supported stages. You may use the `getopt` function from `unistd.h` to manage the handling of options.

This is a staged project. Complete the stages in order.

3.1 Stage A

The input to this program from `stdin` is a sequence of zero or more hexadecimal numbers not including `0xffffffff`, followed by `0xffffffff` which is reserved for terminating the sequence. Each of the hexadecimal numbers except the last one is the encoding of a MIPS machine language instruction. You can assume there are no more than MAXSIZE=1000 instructions in the input. The instructions are assumed to start at address `0x00400000`.

For the purposes of this project we assume a virus gains control of the operating system when it makes a `syscall` with `$v0` set to 18.

The stage A virus checker should search the input MIPS code for the sequence of instructions

```
li $v0, 18
syscall
```

and print the line

`INFECTED(A)` *address*

where *address* is the address of the `syscall` instruction for each such pattern.

Consider the following example problem

```
[0x00400000] 0x8fa40000  lw $a0 0($sp)    # argc
[0x00400004] 0x27a50004  addiu $a1 $sp 4  # argv
[0x00400008] 0x24a60004  addiu $a2 $a1 4  # envp
[0x0040000c] 0x00041080  sll $v0 $a0 2
[0x00400010] 0x00c23021  addu $a2 $a2 $v0
[0x00400014] 0x0c10000f  jal main
[0x00400018] 0x00000000  nop
[0x0040001c] 0x3402000a  li $v0 10
[0x00400020] 0x0000000c  syscall # syscall 10 (exit)
beta:
[0x00400024] 0x34020012  li $v0, 18
[0x00400028] 0x0000000c  syscall
[0x0040002c] 0x34020012  li $v0, 18
[0x00400030] 0x00028021  move $s0, $v0
[0x00400034] 0x22100002  addi $s0, $s0, 2
[0x00400038] 0x0000000c  syscall
main:
[0x0040003c] 0x34020001  li $v0, 1
[0x00400040] 0x08100014  j label1
label2:
[0x00400044] 0x34020012  li $v0, 18
[0x00400048] 0x0000000c  syscall
[0x0040004c] 0x08100019  j label3
label1:
[0x00400050] 0x34040002  li $a0, 2
[0x00400054] 0x0000000c  syscall
[0x00400058] 0x08100011  j label2
label5:
[0x0040005c] 0x34020012  li $v0, 18
[0x00400060] 0x0810001e  j label6
label3:
[0x00400064] 0x0000000c  syscall
[0x00400068] 0x34020001  li $v0, 1
[0x0040006c] 0x34040002  li $a0, 2
[0x00400070] 0x0810001e  j label6
label7:
[0x00400074] 0x03e00008  jr $ra
```

```
label6:
[0x00400078] 0x0000000c  syscall
[0x0040007c] 0x10800002  beq $a0, $zero, label8
[0x00400080] 0x34020012  li $v0, 18
label8:
[0x00400084] 0x0000000c  syscall
[0x00400088] 0x0810001d  j label7
```

The output of stage A on the program above should be

```
INFECTED(A) 00400028
INFECTED(A) 00400048
INFECTED(A) 00400084
```

3.2 Stage B

Virus writers are more devious than that. Sometimes they separate the setting of the variable `$v0` from the `syscall` by a sequence of instructions that do not modify `$v0`.

The Stage B checker should look for a sequence of instructions

```
    li $v0, 18
    instructions
    syscall
```

where none of *instructions* writes to `$v0` or is a branch or jump instruction. For each such pattern (note *instructions* could be empty) it should output

```
    INFECTED(B) address
```

where *address* is the address of the `syscall` instruction.

In order to build this stage you are provided with a function

```
void registers_used(unsigned int opcode, unsigned int *reads, unsigned int *writes)
```

which given an opcode determines the registers read, and written by the opcode, and returns those by setting the bits of the unsigned ints pointed to by `reads` and `writes` respectively. It doesn't handle all MIPS instructions but at least: `addiu`, `addi`, `addu`, `add`, `andi`, `and`, `beq`, `bgez`, `bgtz`, `blez`, `bltz`, `bne`, `jalr`, `jal`, `jr`, `j`, `lb`, `lh`, `lui`, `lw`, `nop`, `nor`, `ori`, `or`, `sb`, `sh`, `sll`, `slt`, `sltu`, `slti`, `sltiu`, `sra`, `srl`, `subu`, `sub`, `sw`, `syscall`, `xori`, `xor`, which is sufficient to this project. This can be found in `/home/subjects/comp20006/local/project2/registers.c`. For example `registers_used(0x0230082a, &r, &w)` which is `slt $1, $17, $16` sets `r=0x00030000` and `w=0x00000002`. For simplicity for this project `registers_used` ignores the fact that `syscall` could write to `$v0` and `$a0`.

The output on the example program should be

```
INFECTED(B) 00400028
INFECTED(B) 00400038
INFECTED(B) 00400048
INFECTED(B) 00400084
```

3.3 Stage C

Virus writers can hide the virus by making it thread through code, for example

```
   li $v0, 18
   j  around
   li $v0, 10
back:
   syscall
around:
   j back
```

To overcome this you will build a control flow graph of the code. You need to calculate the basic blocks of the program, like in project 1 *except* leaders are defined as follows:

- All addresses jumped to by `j` and `jal` instructions are leaders;
- All addresses jumped to by branch instructions are leaders
- All addresses occurring after `j`, `jal`, `jr`, `jalr` and branch instructions are leaders.
- The address immediately after the sequence `li $v0, 10`, `syscall` is a leader (*this sequence represents the end of the program execution*).

Note that `syscall` does *not* create leaders afterwards, except for the last case above.

The program only needs to handle `jr` and `jalr` where the argument is `$ra` (`$31`). It should abort if this is not the case.

Once the basic blocks are defined you need to create a control flow graph, which links them together. You must use a structure defined as follows

```
typedef struct bb {
  unsigned int start_address;     // start address
  unsigned int end_address;       // end address
  int n_successors;               // number of sucessor blocks
  struct bb **successors;         // array of successor blocks
} BLOCK;
```

The `successors` field is an array of pointers to the successor blocks. The size of the array is given by `n_successors`. You will need to use `malloc` to get memory for storing blocks and the arrays of successors in each block.

The successors of a block are the blocks for the addresses where the execution could go to after the block. For blocks ending in `j` and `jal` this is the block of the address in the instruction. For blocks ending in branches, the successors are the next block and the block that could be branched too. For blocks ending in `jr $31` or `jalr $31` the successors are all blocks that appear after a `jal` or `jalr` instruction. *Note that we are approximating what the* `jr` *instruction can do by assuming it could jump to all possible instructions which store a return address in* `$ra`. For blocks ending in `syscall 10` there are no successors. For blocks ending in other instructions the successor is just the next block in memory.

Once you have defined the basic blocks of the program, and linked them together for stage C, you need to use the control flow graph to determine which basic blocks can actually be reached in execution of the program by starting from the first basic block (at `0x00400000`) and determining which are reachable. To do this you should add a new field to the basic block structure (`struct bb`) to record whether it is reachable.

Stage C output should output

> `INFECTED(C)` *address*

for each stage B style pattern which occurs solely within a reachable basic block, where *address* is the address of the `syscall` instruction.

The basic blocks for the program above and their successor blocks are

```
[1]  00400000 - 00400014 [4]
[2]  00400018 - 00400020 []
[3]  00400024 - 00400038 [4]
[4]  0040003c - 00400040 [6]
[5]  00400044 - 0040004c [8]
[6]  00400050 - 00400058 [5]
[7]  0040005c - 00400060 [10]
[8]  00400064 - 00400070 [10]
[9]  00400074 - 00400074 [2]
[10] 00400078 - 0040007c [11,12]
[11] 00400080 - 00400080 [12]
[12] 00400084 - 00400088 [9]
```

Note that only blocks [1,2,4,5,6,8,9,10,11,12] are reachable. The output should be

```
INFECTED(C) 00400048
```

3.4 Stage D

The real point of the control flow graph is to be able to track more complex control flow, like the example in the beginning of the previous section. In order to do so you need to record which blocks are always entered with `$v0` taking the value `18`. You should assume that initial execution at `0x0040000` may not have `$v0 = 18`.

To calculate this information you should initially assume that each block is always entered with `$v0 = 18`, and then analyse possible control flows to see if the block can be entered with this not the case. You should extend the `struct bb` to include a field to record this. You can assume any instruction that writes to `$v0` other than `li` sets it to an unknown value different from 18.

For the example program above block [8] `00400064 - 00400070` is the only one which is always entered with `$v0` taking the value `18`.

Once you have determined which blocks are always entered with `$v0 = 18`, then the output for stage D should give all reachable `syscall` instructions which are always reached with `$v0 = 18`.

The output for the example program is

```
INFECTED(D) 00400048
INFECTED(D) 00400064
```

3.5 Stage E

We can improve the virus checker so it finds not only definite problems but also suspicious code. Extend the analysis of stage D to find which blocks may be entered with `$v0` taking the value `18`. *Note the previous analysis looked for blocks which MUST be entered with* `$v0 = 18`. You should assume that on entry to the program (`0x00400000`) may be entered with `$v0 = 18`. You can assume any instruction that writes to `$v0` other than `li` sets it to an unknown value possibly equal to 18.

The analysis on the example program should find that blocks [1,2,4,8,12] may be entered with `$v0` taking the value `18`. Again you will need to extend the `struct bb` data structure.

The program should output a line

`INFECTED(E)` *address*

where *address* is the address of the `syscall` instruction that analysis determines is *always* reached with `$v0 = 18` and

`SUSPICIOUS(E)` *address*

when analysis determines the `syscall` can *sometimes* be reached with `$v0 = 18`.

The output for the example program is

```
INFECTED(E) 00400048
INFECTED(E) 00400064
SUSPICIOUS(E) 00400084
```

3.6 Optional Stage F

This stage is an extension for bonus marks. Improve the analysis of the program by keeping track for each register if it has a fixed possible value when it reaches an instruction, or what set of values it may take at that point.

For example consider the example program. At address `0x0040007c` it is always the case that `$v0 = 1` and `$a0 = 2` which means that the branch never occurs. This in turn means that at address `0x00400084` it is always the case that `$v0 = 18` and the program is infected at this point, not just suspicious.

The output should be more accurate version of that for stage E, marked with `F` as the stage name. There are plenty of ways to make the virus checking more accurate, for example handling register use by `syscall` better, or handling `jr $ra` more accurately.

4 Using subversion

You must use **subversion** to record your progress during the development of your code for the project. See Section 3 of the notes for more information on subversion. We have created a personal subversion repository directory for you to use, and checked it out as the directory `comp20006` in your home directory. To start the project, do the following commands

```
cd ~/comp20006
cp project1/submit/analyse.c project2/virus.c
svn add project2/virus.c
svn commit
```

which creates the project2 directory and starts your code from the code of project 1. When you complete a significant change, you should check in, with a descriptive log message.

5 Submission

When you are ready to submit a version of your `virus.c` you should

```
cd ~/comp20006/project2
cp virus.c submit/virus.c
svn add submit/virus.c
svn commit
```

to copy the working version into the submit directory. As before, your `comp20006/project2` directory is for your work-in-progress, and your `comp2006/project2/submit` directory is for submitted work.

That is all you have to do to submit your work. At the deadline, we will make a copy of the relevant part of your subversion repository, and find the snapshot of all files that you most recently committed in `comp20006/project2/submit`. The markers will mark this version of `virus.c`. If you do not copy `virus.c` to this directory, or if you forget to execute `svn add` or `svn ci` on it, there will be nothing to mark, and you will get a zero. **Note it is your responsibility to ensure that svn is working for you.**

The marking process will start by checking

- whether your program has been properly checked in,
- whether it compiles without any errors and warnings, and
- whether its executable generates the right outputs for a small set of test cases.

This will all be done by a simple software tool. We will make this tool available to you in `/home/subjects/comp20006/local/project2`, together with the expected output generator and some test cases.

While we will normally look at only the last version of the program in your submit directory, we reserve the right to look at the program's development history in cases of disputed authorship, and we will be actively looking for submissions that are too similar. Your submission is supposed to reflect your own unaided work.

Marks will be out of 20 with a possibility of a bonus marks for improved analysis in stage F, which is optional. You will be assessed on your understanding of the project, as evidenced by the design of your solution; your programming style with respect to clarity, logical structure, and modularity; the robustness of your program; and your standard of documentation, as well as correctness of the program.

6 Extensions

You may request an extension for one of two reasons.

The first reason is that there is some factor outside your control that is affecting your ability to work on this project. In such cases, the department typically asks for whatever documentation is practicable under the circumstances (such as a note from a medical practitioner), but if the extension is granted, there is no penalty. You need to apply for such extensions *before* the deadline.

The other reason is that you simply haven't got far enough in your work, and think you can do significantly better given a bit of extra time. Late submissions of this kind will incur a penalty of 1 mark per day (or part thereof) for the first two days after the deadline and a penalty of 2 marks per day (or part thereof) after that. Note if you do not request this extension *before the deadline* we will use the version in the submit directory at the due date.

Note that machine breakdowns (within reason) and projects in other subjects are part of life, and you must learn to deal with them. One way you can do this is by starting work on your projects as soon as possible. Therefore they will not be acceptable as an excuse for not having a project completed by the due date.

7 Questions

If you have any questions about any aspect of the project, you should post them on the LMS discussion forum instead of sending email, so that other students can also read the answers.