Assignment 6, Spring 2008
CS 351, Defense Against the Dark Arts
Removing Obfuscating Jumps with Phoenix

Purpose

This assignment will further your abilities to use Phoenix to de-obfuscate virus code. Specifically, Phoenix will be used to detect and remove obfuscating jump instructions as found in a source-code metamorphic virus.

Prerequisites to Review

You should know how to create a Phoenix plug-in phase from your previous Phoenix assignment. The only other prerequisites for this assignment are to understand the nature of obfuscating jump instructions in virus code, and to understand the *basic block* structure of a program in Phoenix IR.

A metamorphic virus could change its signature pattern in each generation by rearranging sequences of code into convoluted "spaghetti code" using numerous jump instructions. For example, a straightforward sequence of five instructions, called *inst1* through *inst5*, could become:

```
        goto label1
label4: inst4
        goto label5
label2: inst2
        goto label3
label1: inst1
        goto label2
label3: inst3
        goto label4
label5: inst5
```

Notice that the instructions *inst1* through *inst5* are still executed in the proper order. The functionality of the virus is unchanged, but its signature has changed, which will defeat simple, pattern-matching anti-virus scanners.

A basic block is a sequence of instructions terminating with a jump or branch instruction, which does not contain any jumps or branches except the terminating instruction. Compilers break a program up into basic blocks to facilitate certain analyses. In the code above, the first basic block would consist of a single instruction, the `goto label1` instruction. The second basic block would contain the `label4`, `inst4`, and `goto label5` instructions. There are six basic blocks represented in the above code (notice that the last basic block has not terminated yet; any following instructions would be included in it until a branch or jump or function return instruction is encountered).

The basic idea of this assignment is to use Phoenix to detect this spaghetti code, rearrange the basic blocks into the natural order of execution (in which case the jumps will be jumping to the very next instruction and will be unneeded), and then allow later Phoenix optimizations to remove the unneeded jumps. This sequence of operations will undo the jump obfuscation in a virus, restoring the signature of the virus to enable its detection.

Assignment Details

1. On a PC with Phoenix installed, generate a Phoenix c2 Plugin using the Visual Studio wizard. Have your phase execute immediately after the creation of SSA form. Call the project `StraightenJumps`. You will be working within the file `StraightenJumps.cpp`.

2. Add code to the `Execute()` method to satisfy the requirements listed in the Requirements section below.

3. Test your code on the test case provided on the web page for this assignment.

4. Copy and paste your entire `Execute()` method into a Microsoft Word document called `hw6.doc`. Put the usual comments at the top giving your name and the assignment due date (April 2, 2008).

5. Create a ZIP archive with the two files, `hw6.doc` and `StraightenJumps.cpp`. Call the ZIP archive `hw6.zip`. E-mail this ZIP file as an attachment to `cs351-dada@cs.virginia.edu`. Make the subject line be: `Submission: Assignment 6`.

Requirements

1. For each instruction in the `FuncUnit` being compiled, determine if the instruction is a potentially obfuscating jump instruction. In order to determine this, observe the following: An obfuscating jump is an unconditional branch to a label that has no fall-through into that label. Examine the code example given previously. The only way to reach the labels used by the obfuscating jumps is to follow those jumps; you cannot fall into the label from the previous instruction, because the previous instruction is an unconditional branch, and unconditional branches have no fall-through path.

   In order to detect these conditions, you will need to look at the Phoenix documentation for the instructions (`class Phx::IR::Instr`). You need to examine the Properties of this class and find out the following:
   - how to determine that an instruction is a branch instruction;
   - how you can know that it is an unconditional branch instruction;
   - how you can track down the instruction that the unconditional branch jumps to (the *branch target*);
   - how you can find the basic block associated with each instruction;
   - how you can determine whether the branch target basic block already follows the basic block containing the unconditional branch; and
   - how you can determine that the branch target does not have a path that falls into it.

   Once you are able to write the if-statement that checks all these conditions, you need to know how to rearrange the basic blocks of the function into a more natural order.

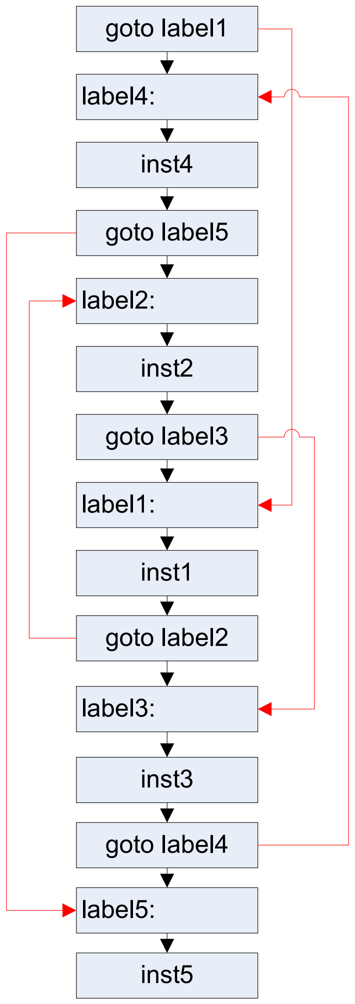
2. Each instruction in Phoenix IR form is part of a basic block, and each basic block has a corresponding node in the control flow graph for the function. Each function (Phoenix class `FuncUnit`) has a control flow graph associated with it. Examine the Phoenix documentation to:
   - determine how you find the control flow graph associated with each `FuncUnit`;
   - find a method in the control flow graph class that will allow you to move a basic block to a location after another basic block.

   (Note that both of these items are obtained through inheritance in the Phoenix class hierarchy. You will need to examine the parent classes of the `FuncUnit` and its flow graph in order to answer these questions.)

   Now, you are ready to write the code that will move the branch target basic block to the location right after the obfuscating jump's basic block.

   As you read the Phoenix documentation, you might notice that basic blocks are arranged in a doubly linked list, using `Next` and `Prev` links, and instructions are arranged in the same kind of doubly linked list. In fact, the for loop that iterates over all instructions in your `FuncUnit`, which was generated by the `c2` Plug-in wizard, uses the linked list of instructions to iterate over all instructions in the function. However, you do NOT want to try to rearrange code by directly manipulating these linked lists. In order to maintain integrity of all data structures, you want to move code around at the abstraction level of the control flow graph. This also greatly simplifies your job. A single move of a control flow graph node handles numerous linked list updates on your behalf.

The linked list of instructions will remain unchanged as you move control flow graph nodes. The control flow graph is just a set of pointers linking the basic blocks of the program. In the figure below (corresponding to the example code on Page 1), the black arrows are the links in the doubly linked list of instructions, while the red arrows link the basic blocks. Changing basic block links does not automatically change instruction links. When Phoenix reaches the code layout phase, it will alter the instruction links to order the instructions of the function in the same order as the basic block order implied by the flow graph links. This phase is late in the compilation process.



3. After moving the control flow graph node for a branch target, the obfuscating jump instruction could be removed with the `Phx::Instr::Remove()` method. However, there are requirements within Phoenix that make such a removal difficult to perform without generating warning messages and assertions. For example, at this phase of compilation, Phoenix will require that the label instruction also be removed, and that profiling data associated with each instruction be cleared before the instruction is removed. It is simpler to leave the obfuscating jumps in the code. Now that these jumps only goto the very next basic block in the flow graph, they will be recognized as useless and removed by a later phase of Phoenix. This technique is often employed in optimizing compilers: If a later optimization will clean up a certain fragment of code, there is no point in duplicating functionality in the present optimization.

4. You are essentially doing an insertion sort over the nodes of the flow graph. An insertion sort cannot fully sort all inputs in a single pass. Therefore, you will need to iterate the above process using an outer loop that terminates when no more changes have been made to the flow graph.

5. Insert calls to dump the flow graph of the `FuncUnit` before and after the *for* loop that iterates over all instructions. The Phoenix documentation will reveal how to dump the flow graph of a `FuncUnit`. This will demonstrate the effectiveness of your code.

6. After you build your Visual Studio project, you should be able to run your code over the code sample provided on the web pages (called **spaghetti.cpp**) with the following command:

   ```
   cl.exe -d2plugin:C:\<YourPath>\StraightenJumps\debug\StraightenJumps.dll spaghetti.cpp
   ```

   where `<YourPath>` is the path to where project `StraightenJumps` is located.

   IMPORTANT: You must run the above command from a Phoenix RDK Command Prompt - Debug window. You may redirect the output of the command if you wish to capture your IR dumps into a file for examination. Correct output is given on the web page for the assignment.

7. Your solution should produce a straight sequence of code that has the same functionality as the original spaghetti code, with the obfuscating jumps now jumping to their successor basic blocks.