Programming Languages and Compilers
Qualifying Examination
Monday, September 18, 2006

For this exam, the student was instructed to answer 4 of 6 questions. What follows are the questions that were answered.

Question 1: Whole Program Analysis

When we study dataflow analysis, we often consider only analyses that involve a single function; i.e., the analysis works on a single control-flow graph with no function calls.

This is clearly not realistic.

Describe at least two ways to do whole-program dataflow analysis; i.e., dataflow analysis that safely computes dataflow information for a whole program that includes multiple functions. Be sure to say what features you are assuming (e.g., global variables, reference parameters, pointers). You don't have to include all features, but don't make your language trivial, either.

Illustrate both of your methods with a specific dataflow problem (e.g., constant propagation, reaching definitions, points-to analysis).

Question 2: Loop Unrolling and Software Pipelining

Part (a):

An important loop optimization is loop unrolling. Give a *simple* example that illustrates the optimizations that loop unrolling can enable. How is the degree of unrolling determined? What is done if the number of loop iterations is not known in advance?

Part (b):

Software pipelining is often used as a substitute for explicit loop unrolling. Show how your example from Part (a) could be software-pipelined. What factors limit the amount of speedup obtainable by software pipelining? How are loops with an unknown number of iterations handled?

Question 3: Points-to Analysis

This question is about points-to analysis for a language like C that allows pointers, pointers-to-pointers, etc.

Assume that our language includes assignment statements, if-then-else statements, and while loops (but no function calls, no pointer arithmetic, and no heap-allocated storage). Also assume that the assignment statements have been normalized so that there is at most one pointer dereference per statement (e.g., the following are OK: `x = y; x = *y; *x = y;`, but not `*x = *y; **x = y`).

Part (a):

Describe how to do a flow-insensitive points-to analysis for this language.

Part (b):

Now define a flow-sensitive points-to analysis for the language by providing the following:

1. The dataflow functions for each kind of node in the control-flow graph (the enter node, the different kinds of assignment, if, and while).
2. The meet operator (used to combine dataflow facts at join points in the control-flow graph).

Part (c):

Why is it interesting to know whether the dataflow functions for a flow-sensitive analysis are distributive?

Are the dataflow functions that you wrote for Part (b) all distributive? (Justify your answer.)

Question 4: Security

Part (a):

Languages like C that do not guarantee array-bounds checking and that allow pointer arithmetic can lead to programs that are vulnerable to certain kinds of malicious attacks. Explain how a malicious user can exploit a buffer-overrun vulnerability in a program.

Part (b):

Consider the program shown below.

```
main(int argc, char * argv[]) {
    char header[2048], buf[1024], *ptr;
    int counter;
    FILE *fp;
    ...
    ptr = fgets(header, 2048, fp);
    copy_buffer(header);
    ptr = fgets(buf, 1024, fp);
    copy_buffer(buf);
}
void copy_buffer(char * buffer) {
    char copy[20];
    strcpy(copy, buffer);
}
```

How could a malicious user exploit the buffer overrun in this program ro execute the system call `system("open /etc/passwd")`? Assume that the malicious user controls the contents of the file pointed to by `fp`.

Based on your answer, discuss why buffer-overrun attacks belong to a class of attacks called *code-injection* attacks.

Part (c):

Address-space randomization is a common technique to thwart code-injection attacks. *Stack randomization* is a specific kind of address-space randomization in which the compiler randomizes the stack frame for each function; i.e., entries in the stack frame appear in a random order. Assume that a different random order is chosen each time a procedure p is compiled. Does the attack you described in Part (b) still work? (Justify your answer.)

Part (d):

Briefly describe how a compiler would implement stack randomization. Be sure to consider how separate compilation could work.