Programming Languages and Compilers
Qualifying Examination

Monday, February 6, 2012

Answer 4 of 6 questions.

GENERAL INSTRUCTIONS

1. Answer each question in a separate book.
2. Indicate on the cover of *each* book the area of the exam, your code number, and the question answered in that book. On *one* of your books list the numbers of *all* the questions answered. *Do not write your name on any answer book.*
3. Return all answer books in the folder provided. Additional answer books are available if needed.

POLICY ON MISPRINTS AND AMBIGUITIES

The Exam Committee tries to proofread the exam as carefully as possible. Nevertheless, the exam sometimes contains misprints and ambiguities. If you are convinced that a problem has been stated incorrectly, mention this to the proctor. If necessary, the proctor will contact a representative of the area to resolve problems during the *first hour* of the exam. In any case, you should indicate your interpretation of the problem in your written answer. Your interpretation should be such that the problem is nontrivial.

```
 (1) main(int argc, char* argv[]){
 (2)    char header[2048], buf[1024], *ptr;
 (3)    int counter;
 (4)    FILE *fp;
 (5)    ...
 (6)    ptr = fgets(header, 2048, fp);
 (7)    copy_buffer(header);
 (8)    ptr = fgets (buf, 1024, fp);
 (9)    copy_buffer(buf);
(10) }
(11)
(12) void copy_buffer(char *buffer){
(13)    char copy[20];
(14)    strcpy(copy, buffer);
(15) }
```

Figure 1: Example Program

Question 1 (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 buffer overrun vulnerability in a program. Why are programs written in languages such as Java not vulnerable to these vulnerabilities?

Part (b): Consider the program shown in Figure 1. How could a malicious user exploit the buffer overrun in this program, to execute the system call `system(''exec /bin/sh'')`? Assume that the malicious user controls the contents of the file which `fp` points to.

Part (c): One technique to address the buffer-overrun vulnerability is to make the stack *non executable*. Explain why this addresses the buffer-overrun vulnerability. Discuss the run-time overhead of this mitigation technique.

Part (d): Another technique for addressing the buffer-overrun vulnerability is to keep a auxiliary stack. At a function call site, the return address is pushed on the auxiliary stack. Before returning to that call site, the program verifies that the top of the auxiliary matches the actual return address. Give details about this technique in the context of the program shown in Figure 1. Explain why this technique addresses the buffer-overrun problem.

Question 2 (Functional Languages and Tail Recursion).

This question concerns implementation techniques for recursive functions in functional languages.

Part (a): Consider the following recursive function definition, which returns the length of a list:

Length : **IntList** $\rightarrow$ **Int**
Length(*list*) =
    cases *list*
        nil : 0
        cons(*hd*, *tail*) : 1 + Length(*tail*)
    end

For a call Length(*list*), where *list* is of length n, (i) how many calls are made, and (ii) how deep does the stack grow?

Part (b): Consider the following function definition, which returns true or false depending on whether x is a member of *list*:

MemberOf : **Int** $\times$ **IntList** $\rightarrow$ **Boolean**
MemberOf(x, *list*) =
    cases *list*
        nil : *false*
        cons(*hd*, *tail*) : **if** x = *hd* **then** *true* **else** MemberOf(x, *tail*)
    end

Note that function MemberOf is tail recursive.

1. Explain what optimization or optimizations are possible for tail-recursive functions.

2. For a call MemberOf(x, *list*), where *list* is of length n, (i) how many calls are made, and (ii) how deep does the stack grow? Explain why.

3. In general, how deep does the stack grow for your proposed implementation of tail-recursive functions? Explain why.

Part (c): "Continuation-passing style" is a paradigm for writing functional programs in which functions receive explicit "continuation" arguments that are invoked within the function instead of returning from the function. In other words, each "return" from a function looks like a call on another function.

> **Remark**. The functional definition obtained as the meaning of a program in a language whose denotational-semantics definition uses "continuations" is one example of a function in continuation-passing style; this question happens to be about functions written directly in continuation-passing style. **End Remark**.

Suppose that we rewrite the Length function from Part (a) as follows:

continuation = **Int** $\rightarrow$ **Int**
Length$'$: **IntList** $\times$ **continuation** $\rightarrow$ **Int**
Length$'$(*list*, k) =
    cases *list*
        nil : $k(0)$
        cons(*hd*, *tail*) : Length$'$(*tail*, $\lambda z.k(1+z)$)
    end

Length : **IntList** $\rightarrow$ **Int**
Length(*list*) = Length$'$(*list*, $\lambda z.z$)

In particular, function Length$'$ is tail recursive and thus the method you described in Part (b) should apply.

1. In terms of the number of calls made and/or the depth of the stack, did we "get something for free" by turning Length into tail-recursive form (by transforming it into continuation-passing style)? For instance, for a call Length(*list*), where *list* is of length n, (i) how many calls are made, and (ii) how deep does the stack grow?

2. In general, by transforming a function into continuation-passing style do we necessarily gain performance benefits (i.e., measured in number of calls and/or depth of the stack)? If not, suggest a way in which a performance benefit could be gained for the modified version of Length.

Question 3 (Dominators).

The notion of *dominance* can be useful for code analysis and optimization. This question asks you to define, compute, and use dominators.

Part (a): Let n and m be nodes of a control-flow graph. Define what it means for n to dominate m.

Part (b): Define each of the following properties and say whether or not the property holds for the dominance relation (justify your answers):

- transitivity
- reflexivity
- symmetry
- anti-symmetry

Part (c): A straightforward representation of the dominance relation for the nodes of a control-flow graph requires N^2 space in the worst case, where N is the number of nodes in the graph. Describe a representation that is commonly used and that requires less space. How much space does your representation require in the worst case?

Part (d): Define a dataflow analysis that can be used to compute the dominance relation for the nodes of a control-flow graph.

Part (e): Describe at least two analyses and/or optimizations that require knowing the dominance relation for the nodes of a control-flow graph.

Question 4 (Prolog).

Part (a): In Prolog, assume that `edge(X, Y)` is true when `X` and `Y` are graph nodes for which there is a directed edge from `X` to `Y`. Consider the following alternatives for defining a `path(X, Z)` relation which is true when there is a path from `X` to `Z`:

Alternative I:

```
path(X, X).
path(X, Z) :- edge(X, Y), X \= Y, path(Y, Z).
```

Alternative II:

```
path(X, X).
path(X, Z) :- path(X, Y), X \= Y, edge(Y, Z).
```

Alternative III:

```
path(X, Y) :- edge(X, Y).
path(X, Z) :- edge(X, Y), X \= Y, path(Y, Z).
```

Alternative IV:

```
path(X, X).
path(X, Y) :- edge(X, Y).
path(X, Z) :- path(X, Y), X \= Y, path(Y, Z), Y \= Z.
```

Are these alternatives equivalent in correctness and performance? If not, which alternative(s) is/are preferable and why?

Part (b): Define a unary `cycle` relation in Prolog such that `cycle(X)` is true if and only if `X` is part of a nontrivial graph cycle. You may use `edge` and `path` in your definition. If you use `path`, say which of the alternatives given above you are assuming.

Part (c): What is the worst-case asymptotic time complexity of your answer to part (b)? For simplicity, assume that constant time is required for each Prolog fact look-up (including `edge`), unification operation, or backtracking step.

Question 5 (Threads).

For purposes of this question, assume that an entire Java program is statically available at compile time. Reflection, native methods, and dynamic class loading are not used.

Consider memory accesses (loads and stores) in a multi-threaded Java program. If an access could coincide with an access of the *same* location by a *different* thread, then we call this a "potentially-shared access"; otherwise, we call this a "definitely-unshared access".

We may be interested in determining whether a given access is potentially shared or definitely unshared. A trivial static shared-access analysis might simply report the following:

1. All loads or stores of local variables are definitely unshared.

2. All loads or stores of instance fields are potentially shared.

This analysis conservatively over-approximates the set of potentially-shared accesses.

Part (a): Propose two alternative static shared-access analyses. Like the trivial analysis given above, your alternatives must conservatively over-approximate the set of potentially-shared accesses. However, your alternatives must be strictly more precise. Briefly describe your two analyses. You need not present every tiny detail, but you should give enough information to clearly demonstrate that your approaches are feasible. You may assume that your analysis is operating in the context of an optimizing Java compiler: the full Java syntax has been reduced to some simpler intermediate representation, other standard compiler analyses are available, etc.

Part (b): How do your two analyses compare in terms of precision? Are they equivalent? Is one strictly more precise than the other? Or does each beat the other in different cases? If your analyses are not equivalent, then give examples of code fragments where your analyses behave differently. (However, examples *alone* do not constitute a complete answer.)

Part (c): Briefly describe three interesting ways in which the results of a static shared-access analysis could be used. These might involve optimization, bug detection, interactive development environments (IDEs), or any other aspect of software development.

Your answers may be suitable for use with *any* static shared-access analysis, or you may require that the analyses have more specific qualities. If you do have special analysis requirements, clearly identify those as part of your answer.

Question 6 (Fixed Points).

Let S be a finite set of size n. Let $F : 2^S \to 2^S$ be a function.[1] A set $X \in 2^S$ is called a fixed point of a function F iff $F(X) = X$. Define $\mu X.F(X)$ to be the least fixed point of F and $\nu X.F(X)$ to be the greatest fixed point of F.

Part (a): Under what conditions is a function guaranteed to have a least fixed point and how can that fixed point be computed? Do those conditions hold for all functions F of type $2^S \to 2^S$ as defined above?

Part (b): Let $G = (S, E)$ be a labeled directed graph where S is the set of vertices and $E \subseteq S \times S$ is the set of edges. Given a subset S_1 of S, let $Reach(S_1)$ be the set of vertices s such that there exists a path from s to vertex $s' \in S_1$. Describe a work-list algorithm for computing $Reach(S_1)$.

Part (c): Define a function F such that F's least fixed point is $Reach(S_1)$. (Don't just make F ignore its argument and compute $Reach(S_1)$.)

Give a small example graph, identify set S_1, and show how to compute F's least fixed point for your example using the fixed-point-finding technique that you described in Part (a).

Does F also have a greatest fixed point? If yes, what is it? If no, why not?

Part (d): The duals of fixed-point formulas are defined as follows:

$$\begin{aligned} D(\mu X.F(X)) &= \nu X.\neg F(\neg X) \\ D(\nu X.F(X)) &= \mu X.\neg F(\neg X) \end{aligned}$$

For a set $S_1 \in 2^S$, $\neg S_1$ denotes the set $S - S_1$. Let F be the function that you gave for Part (c). What does the formula $D(\mu X.F(X)))$ represent?

[1] Recall that 2^S is the power-set of S.