Programming Languages and Compilers
Qualifying Examination

Fall, Sept 20, 2010

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.

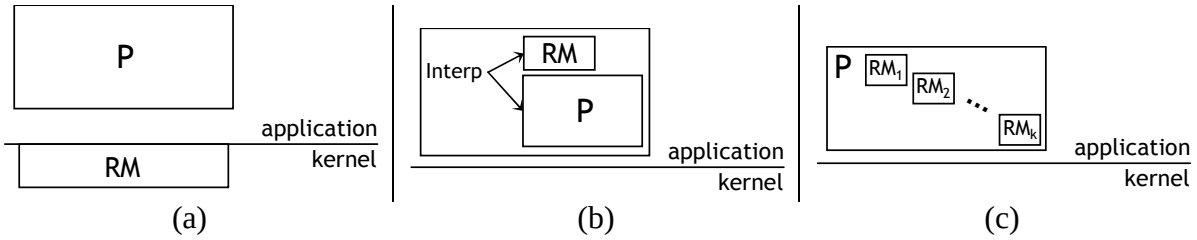

Figure 1: Three types of reference monitors: (a) OS-level reference monitor; (b) interpreted application-level reference monitor; (c) in-lined application-level reference monitor.

Question 1 (Reference monitors).

This question concerns the use of reference monitors in OS kernel and application code.

Part (a)
One class of *security policies* can be expressed using *security automata*: automata with calls as the alphabet (where calls are sometimes qualified by specific values of actual parameters). Describe (e.g., by means of a diagram) a security automaton for the policy that prohibits execution of a *Send* operation after any *FileRead* operation has been executed.

Part (b)
Using the automaton you used to answer Part (a), give an example of an infinite sequence of operations σ for which each of the prefixes of σ is accepted by the automaton. In other words, the infinite sequence avoids ever falling into the error state.

Part (c)
The security automaton discussed in Parts (a) and (b) would be suitable as the basis for an OS kernel-level reference monitor (see Figure 1(a)). Reference monitors could also be used at application level (see Figures 1(b) and 1(c)).

Using application-level reference monitors opens up new possibilities because the alphabet of visible events is different. That is, for OS kernel-level reference monitors, the only events visible are system calls (perhaps further qualified by specific values of actual parameters), whereas application-level events and data are visible to application-level reference monitors.

Give an example of an application-level event that would be interesting—from a security perspective—to use for reference monitoring. (Note: before writing your answer to this part, please read Part (d); it is not our intention that you restrict your answer to just reference monitors that are pure finite-state automata.)

Part (d)
For the example from Part (c), give a specification of it in some suitable notation. (Note: Various kinds of specification are acceptable; we are leaving it up to you because it may not be possible to express the property with something as simple as a finite-state automaton.)

Part (e)
Describe a compilation/rewriting strategy whereby an application-level reference monitor can be in-lined with the code (as hinted at in Figure 1(c)). Include a discussion of the potential kinds of

optimizations that could be performed.

Question 2 (Loops).

One task usually performed by an optimizing compiler is to identify the loops in each function. This question explores loop identification. Parts (b) and (c) talk about natural loops. If you are familiar with Bourdoncle Components, you can use them in place of natural loops for parts (b) and (c).

Part (a): Why is it useful for an optimizing compiler to identify loops?

Part (b): A compiler could identify natural loops or strongly-connected components (SCCs). What is a natural loop? How are natural loops different from SCCs? Give an example to illustrate the difference.

Part (c): What are the advantages and disadvantage of using natural loops versus SCCs in an optimizing compiler? (Note: the efficiency of computing natural loops vs strongly connected components is not the issue.)

Question 3 (Dataflow analysis).

Recall that we can use Kildall's lattice framework to define an instance of a (forward, intraprocedural) dataflow problem for a given CFG as follows:

- We define a complete lattice of dataflow facts with no infinite chains.
- We define a monotonic dataflow function for each CFG node.
- We define a special "initial" fact that holds at the start of the function.

Given this, an algorithm that produces the solution to the dataflow problem (using the CFG) is the following:

```
for the entry node e, initialize e.after to the special initial fact
for all other nodes n, initialize n.before = n.after = TOP
repeat {
    simultaneously compute all n.after
    (by applying n's dataflow function to n.before),
    and at the same time, compute all n.before
    (as the meet of all predecessors' after values)
} until no change
```

This algorithm finds the greatest fixed point for the set of equations that define the "before" and "after" facts for all CFG nodes. You can think of one iteration of the loop as an application of a function `f`. Then the loop above computes

```
f(TOP)
f(f(TOP))
f(f(f(TOP)))
 ...
```

until a fixed point is found. That fixed point is a value `v_0` $=$ `f`^{k}`(TOP)`; i.e., k applications of `f` to `TOP`.

This question is about doing incremental dataflow analysis: i.e., assume that you have used the above algorithm to solve a dataflow problem. Now someone edits the program, and you want the solution for the updated CFG, but you don't want to start over and compute the new solution from scratch.

To simplify this, we'll assume that the shape of the CFG is not changed (it has the same nodes and edges as before). However, the code in some of the nodes, and therefore the dataflow functions associated with those nodes, has changed. This means that the function `f` discussed above has changed to a new function `f_new` (which is still monotonic).

Part (a): Here's one way to compute the new solution (remember that `v_0` is the solution to the original problem):

```
v_1 = f_new(v_0) meet v_0
v_2 = f_new(v_1) meet v_1
v_3 = f_new(v_2) meet v_2
...
until no change
```

Prove (i) that this process is guaranteed to terminate, and (ii) that it will produce a safe solution; i.e., one that is less than or equal to the greatest fixed point of function `f_new`.

Part (b): While the loop above is guaranteed to terminate and to find a safe solution, it is not guaranteed to find the greatest fixed point of `f_new`. Show that this is true by supplying the following:

- a dataflow problem
- a simple CFG
- the solution to the dataflow problem (before and after facts for each CFG node)
- a change to the code at one CFG node
- the greatest solution to the new dataflow problem (the solution that would be found by solving the new problem from scratch)
- the (different) solution that would be found using the incremental approach described above in Part (a).

Question 4 (Fixpoints).

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 $\mu X.F(X)$ as the least fixed point of F and $\nu X.F(X)$ as the greatest fixed point of F.

- **Part (a):** Give a formal definition for $\mu X.F(X)$ and $\nu X.F(X)$, and provide conditions under which these fixpoints exist and are unique.

- **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 a vertex s'. Describe a work-list algorithm for computing $Reach(S_1)$.

- **Part (c):** Provide a fixpoint formula for $Reach(S_1)$ (evaluating your formula should yield $Reach(S_1)$).

- **Part (d):** For a set $S_1 \in 2^S$, the set $S - S_1$ is denoted by $\neg S_1$. The dual of a formula ϕ (denoted by $D(\phi)$) is defined as follows:

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

  Let β be the formula that you gave for part (c). What does the formula $D(\beta)$ represent?

- **Part (e):** What is the relationship between a formula ϕ and its dual $D(\phi)$? Justify your answer

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

Question 5 (Security).

Part (a): Describe memory-corruption vulnerabilities for C programs, and give two specific examples of memory-corruption vulnerabilities.

Part (b): In taint analysis, any variable whose value can be controlled by an adversary (for example, a buffer whose contents are read from a network) is given a tainted attribute. The taint attribute is then propagated throughout the program. Describe a static-analysis technique for taint analysis of C programs. Use an example to demonstrate your answer. At the end of executing your static-analysis technique, each variable instance in the program should be marked as tainted or untainted. In your description, include a definition of initially tainted variable instances.

Part (c): Describe a source-to-source transformation, which takes a C program as input and outputs another C-program where the taint attribute is dynamically propagated during the execution of the program.

Part (d): Describe how taint analysis can be used to thwart exploits that use memory corruption vulnerabilities.

Question 6 (Exception Checking).

Consider exception handling in a simplified Java-like language. Exceptions must be instances of the `Throwable` class or some subclass of `Throwable`. An exception is raised (or thrown) using a `throw` statement:

`throw` e`;`

Exceptions are caught using `try`/`catch` statements. Let s and s_i represent statements and let T_i and id_i represent identifiers. Assume the following simplified syntax for `try`/`catch` statements:

```
try
   s
catch (T1  id1)
   s1
...
catch (Tn  idn)
   sn
```

In the text that follows, we refer to s as the body of the `try` clause, and s_i as the bodies of the corresponding `catch` clauses.

When an exception is thrown, the run-time environment locates the closest dynamically enclosing `try`/`catch` statement. It searches through the `catch` clauses of that statement in order until it finds one with associated type T_i that is a superclass of the dynamic type of the thrown value. The corresponding `catch` clause s_i is then evaluated with id_i bound to the thrown value. If no `catch` clause has a suitable associated type, then the exception propagates outward to the next closest dynamically enclosing `try`/`catch` statement. Propagation continues until a suitable `catch` clause is found or until the last of the enclosing `try`/`catch` statements has been exhausted.

Assume that we have a typing environment Γ with which we can make judgments as given in fig. 2. For example, the following might be part of the type checking rules for statements. It enforces the restriction that only `Throwable` instances can be thrown:

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \sqsubseteq \texttt{Throwable}}{\Gamma \vdash \texttt{throw } e\texttt{;} : \texttt{void}}$$

As a second example, a sequence of n statements type checks if each individual statement type checks:

$$\frac{\Gamma \vdash s_1 : \texttt{void} \qquad \Gamma \vdash s_2 : \texttt{void} \qquad \cdots \qquad \Gamma \vdash s_n : \texttt{void}}{\Gamma \vdash s_1\ s_2\ \cdots\ s_n : \texttt{void}}$$

Java requires that each method declare the set of exception classes that it may propagate out to callers. If the exception declaration includes a class T, then thrown instances of T or any T subclass might propagate out from the method without being caught.

[You may be aware of Java's distinction between checked and unchecked exceptions. Disregard unchecked exceptions for purposes of this question. All exceptions are checked exceptions that must be declared in the method signature if not caught within the method body.]

variable lookup judgments:

$\Gamma(id) = \tau$

Interpretation: id is a bound identifier with type τ.

subclassing judgments:

$\Gamma \vdash A \sqsubseteq B$

Interpretation: A and B are the names of classes and A is a subclass of B. The "$\sqsubseteq$" relation is reflexive and transitively closed.

method lookup judgments:

$\Gamma \vdash A.id : \tau_1 \times \cdots \times \tau_n \rightarrow \rho$

Interpretation: A is the name of a class with method id taking arguments of types τ_i and yielding a result of type ρ. Assume that the implicit `this` argument is represented by τ_1.

type checking judgments for expressions: $\Gamma \vdash e : \tau$

Interpretation: e is an expression that type checks and has static type τ.

type checking judgments for statements: $\Gamma \vdash s$: `void`

Interpretation: s is a statement or block of statements that type checks.

Figure 2: General forms of judgments provided by a typing environment Γ.

Part (a):

Modify the general forms of (1) method lookup judgments, (2) type checking judgments on expressions, and (3) type checking judgments on statements to yield any additional information that would be required to verify that checked exception declarations are correct. Do not write complete type checking rules. Merely show the general forms that these three kinds of judgment would have and state the interpretation of each as in fig. 2.

Part (b):

Write a type checking rule for the binary "`+`" operator that uses the modified judgment forms you proposed in part (a). For simplicity, assume that this operator only applies to a pair of arguments of type `int`.

Part (c):

Write a type checking rule for `try`/`catch` statements that uses the modified judgment forms you proposed in part (a). You may use just one rule or multiple rules as you see fit. Your rule(s) should enforce the following restrictions:

1. Identifier id_i for a caught exception is bound with type T_i within the corresponding `catch` clause body s_i.

2. Each `catch` clause must have the possibility of actually catching some exceptions raised in the body of the `try` clause and not caught by any preceding `catch` clause in the same `try`/`catch` statement.

3. Only instances of `Throwable` or a `Throwable` subclass can be caught.

Include comments clearly explaining which parts of your rule(s) enforce each of the three restrictions above.

Note that it is legal for `catch` clause bodies to raise new exceptions of their own. Any such exception is dispatched starting with the next enclosing `try`/`catch` statement. Later `catch` clauses in the current `try`/`catch` statement are not considered.