Assignment 4
Information Flow

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Due **Wednesday**, October 30, 2024
90 points + 20 points extra credit

Your solution should be handed in as a file `hw4.pdf` to Gradescope. If at all possible, write your solutions in LaTeX. The handout `hw4-safety.zip` includes the LaTeX sources for Lectures 11 and 12 and the necessary style files which provide some examples for rules, derivations, and proofs. Because we are one day late to post the assignment, it is due to Wednesday, instead of Tuesday. You may use up to two late days as usual.

1 Implicit Flows [60 points + 20 points extra credit]

Consider adding a new construct to our language SAFETINY, $\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta$. Note that in SAFETINY all commands are safe, but we have $\mathbf{test}\ P$ which aborts if P is false. We do not consider division, memory read/write, or **assert**. In order to simplify matters further, we also exclude loops from consideration, but see the extra credit tasks at the end of this problem.

$\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta$ is supposed to execute as follows:

1. Execute α in the current state ω
2. If α **does not abort**, the $\mathbf{try}/\mathbf{catch}$ construct finishes in the poststate of α
3. If α **aborts**, we continue by executing β in the prestate ω. In this case, the poststate of β will be the poststate of $\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta$.

$\mathbf{try}/\mathbf{catch}$ is not easy to implement efficiently in a compiler since we have to either save the prestate ω, or track assignments so we can roll back the state when a test fails. As we see in Task 2, it is not so difficult in an interpreter.

Here are some examples:

$$
\begin{array}{lcl}
\mathsf{eval}\ \omega\ (\mathbf{try}\ \mathbf{test}\ \bot\ \mathbf{catch}\ x := 0) & = & \omega[x \mapsto 0] \\
\mathsf{eval}\ \omega\ (\mathbf{try}\ \mathbf{test}\ \top\ \mathbf{catch}\ x := 0) & = & \omega \\
\mathsf{eval}\ \omega\ (\mathbf{try}\ \mathbf{test}\ \bot\ \mathbf{catch}\ \mathbf{test}\ \bot) & & \text{aborts} \\
\mathsf{eval}\ \omega\ (\mathbf{try}\ (x := 0\ ;\ \mathbf{try}\ (\mathbf{test}\ x > 0)\ \mathbf{catch}\ x := 1)\ \mathbf{catch}\ x := 2) & = & \omega[x \mapsto 1]
\end{array}
$$

Task 1 (10 points) Give a semantic definition of $\omega[\![\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]\!]\nu$ and $\mathbf{test}\ P$ that models the intended behavior based on the informal description above. You should model a program that aborts (and is not caught) as one that has no poststate.

With this understanding, we can update our definition of eval so that it explicitly returns either a state ν or abort. We show the cases for sequential composition, **skip**, and assignment.

$$\begin{array}{lcll}
\mathsf{eval}\ \omega\ (\alpha\ ;\ \beta) & = & \mathsf{abort} & \text{if } \mathsf{eval}\ \omega\ \alpha = \mathsf{abort} \\
\mathsf{eval}\ \omega\ (\alpha\ ;\ \beta) & = & \mathsf{abort} & \text{if } \mathsf{eval}\ \omega\ \alpha = \mu \text{ and } \mathsf{eval}\ \mu\ \beta = \mathsf{abort} \\
\mathsf{eval}\ \omega\ (\alpha\ ;\ \beta) & = & \nu & \text{if } \mathsf{eval}\ \omega\ \alpha = \mu \text{ and } \mathsf{eval}\ \mu\ \beta = \nu \\
\mathsf{eval}\ \omega\ (\mathbf{skip}) & = & \omega & \\
\mathsf{eval}\ \omega\ (x := e) & = & \omega[x \mapsto c] & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e = c
\end{array}$$

Task 2 (15 points) Complete the definition of eval with the cases for conditionals, tests, and **try**/**catch**.

Task 3 (10 points) Conjecture an axiom of equivalence for $[\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]Q$, or explain briefly why you believe no such axiom is possible in dynamic logic (as we have constructed it so far). Note that your axiom only needs to be sound in the language without loops.

Task 4 (5 points) Give an example of a security policy Σ_0 and program α_0 demonstrating that **test** and **try**/**catch** create a new possibility for implicit information flow. For this question, you should work with a security lattice with just two elements H and L with $\mathsf{L} \sqsubset \mathsf{H}$ and the definition of termination-insensitive noninterference from Lecture 11.

Task 5 (10 points) Prove that your example from the previous task violates termination-insensitive noninterference, that is, $\Sigma_0 \models \alpha_0$ secure is **not** true.

In order to prevent the implicit flows enabled by **try**/**catch** we introduce a new ghost variable *handler* into the information flow type system. The security level of *handler* should be that of the **catch** that would be invoked should the current program abort. It should be $\bot$ (the least element of the security lattice) at the beginning of evaluation.

Task 6 (10 points) Give rules for **try**/**catch** and **test** in the information flow type system. You do not need to prove their soundness.

The remainder of Problem 1 is for extra credit.

In order to support loops, we assume a global bound b on the number of iterations for each while loop, after which it aborts. For example, with $\mathsf{b} = 0$ the program aborts if it ever attempts to enter the body of a loop, with $\mathsf{b} = 1$ each loop $\mathbf{while}\ P\ \alpha$ is equivalent to $\mathbf{if}\ P\ \mathbf{then}\ (\alpha\ ;\ \mathbf{test}\ (\neg P))\ \mathbf{else}\ \mathbf{skip}$.

Task 7 (5 bonus points) Give a semantic definition of $\omega[\![\mathbf{while}\ P\ \alpha]\!]\nu$ that models the intended behavior of bounded loops based on the informal description above.

Task 8 (5 bonus points) Complete the definition eval by providing a clause for **while** loops bounded by b. Feel free to use auxiliary functions.

In order the conjecture suitable axioms in dynamic logic assume that each loop $\mathbf{while}\ P\ \alpha$ with loop invariant J is written explicitly as $\mathbf{while}^{\mathsf{b}}_J\ P\ \alpha$.

Task 9 (10 bonus points) Conjecture axioms in dynamic logic for $\mathbf{while}^{\mathsf{b}}_J$ and **try**/**catch**, or explain why you think that bounded loops and **try**/**catch** cannot be axiomatized in the framework of dynamic logic (as we have constructed it so far).

2 Declassification [30 points]

Consider the formulation of termination-insensitive noninterference in the presence of declassification under the two-level security lattice ($\mathsf{L} \sqsubset \mathsf{H}$).

> We define $\Sigma \models \alpha$ secure where α contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$
> iff $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ and for all $x \in \mathsf{use}\ e$ implies $x \notin \mathsf{maydef}\ \alpha$ and $\mathsf{eval}\ \omega_1\ e = \mathsf{eval}\ \omega_2\ e$
> imply $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

Task 10 (20 points) Assume you are given a security policy Σ and a program α that contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$. Show how to construct a formula R in dynamic logic such that the validity of R implies $\Sigma \models \alpha$ secure. Your starting point should be the construction in Section 2 of Lecture 12. A key question will be how to ensure the condition on uses and possible definitions of variables in α, ideally without extending dynamic logic.

Task 11 (5 points) Give an example policy Σ_0 and program α_0 that is **not** secure due to incorrect use of declassification. Show the encoding of the example in dynamic logic and demonstrate that it is not valid. This is usually done most directly via a counterexample.

Task 12 (5 points) Give an example policy Σ_1 and a program α_1 that uses declassification and is secure. Show the encoding of the example in dynamic logic and demonstrate that it is valid. This is usually done most directly by constructing a weakest precondition, if the formula is in the class that permits it. You don't need to show intermediate steps.