Assignment 4
Information Flow
Sample Solution

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 $\mathbf{assert}$. 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.

The definition of **test** remains unchanged.

$$\begin{array}{lll} \omega[\![\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]\!]\nu & \text{iff} & \omega[\![\alpha]\!]\nu \\ & & \quad \text{or (there is no } \nu \text{ such that } \omega[\![\alpha]\!]\nu) \text{ and } \omega[\![\beta]\!]\nu \\ \omega[\![\mathbf{test}\ P]\!]\nu & \text{iff} & \omega \models P \text{ and } \nu = \omega \end{array}$$

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**.

$$\begin{array}{lcll} \mathsf{eval}\ \omega\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) & = & \mathsf{eval}\ \omega\ \alpha & \text{if } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \top \\ \mathsf{eval}\ \omega\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) & = & \mathsf{eval}\ \omega\ \beta & \text{if } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \bot \\ \\ \mathsf{eval}\ \omega\ (\mathbf{test}\ P) & = & \omega & \text{if } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \top \\ \mathsf{eval}\ \omega\ (\mathbf{test}\ P) & = & \mathsf{abort} & \text{if } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = \bot \\ \\ \mathsf{eval}\ \omega\ (\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta) & = & \nu & \text{if } \mathsf{eval}\ \omega\ \alpha = \nu \\ \mathsf{eval}\ \omega\ (\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta) & = & \mathsf{eval}\ \omega\ \beta & \text{if } \mathsf{eval}\ \omega\ \alpha = \mathsf{abort} \end{array}$$

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.

There are multiple reasonable conjectures. We choose one using $\langle\alpha\rangle\top$ because by definition that is true if and only if there exists a poststate for α. Conversely, $[\alpha]\bot$ is true if and only if α has no poststate.

$$[\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]Q \leftrightarrow (\langle\alpha\rangle\top \rightarrow [\alpha]Q) \wedge (\neg\langle\alpha\rangle\top \rightarrow [\beta]Q)$$

We can use the identity $\neg\langle\alpha\rangle R \leftrightarrow [\alpha]\neg R$ to eliminate the diamond if we wish. For example:

$$[\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]Q \leftrightarrow ([\alpha]\bot \vee [\alpha]Q) \wedge ([\alpha]\bot \rightarrow [\beta]Q)$$

Further, if $[\alpha]\bot$ then also $[\alpha]Q$, so we can simplify the first conjunct:

$$[\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]Q \leftrightarrow [\alpha]Q \wedge ([\alpha]\bot \rightarrow [\beta]Q)$$

One last possibility:

$$[\mathbf{try}\ \alpha\ \mathbf{catch}\ \beta]Q \leftrightarrow (\neg[\alpha]\bot \wedge [\alpha]Q) \vee ([\alpha]\bot \wedge [\beta]Q)$$

We cannot omit $\neg[\alpha]\bot$ here, since $[\alpha]Q$ will be true if α has not poststate.

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.

$$\alpha_0 = \mathbf{try}\ (\mathbf{test}\ x > 0\ ;\ y := 1)\ \mathbf{catch}\ y := 0$$

with $\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L})$.

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.

Let

$$\begin{array}{rclcrclcl} \omega_1 & = & (x \mapsto 0, y \mapsto 17) & \qquad & \mathsf{eval}\ \omega_1\ \alpha_0 & = & (x \mapsto 0, y \mapsto 0) & = & \nu_1 \\ \omega_2 & = & (x \mapsto 1, y \mapsto 17) & & \mathsf{eval}\ \omega_2\ \alpha_0 & = & (x \mapsto 1, y \mapsto 1) & = & \nu_2 \end{array}$$

We have $\Sigma_0 \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ but $\Sigma_0 \vdash \nu_1 \not\approx_{\mathsf{L}} \nu_2$

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.

In a conditional **if** P **then** α **else** β we can raise the level of the pc to that of P. For the **try**/**catch** we don't have the condition directly available, but we can "guess" it. As long as the security level of any test remains below that, the resulting code should be secure.

$$\frac{\Sigma(pc) \sqsubseteq \ell \qquad \Sigma[handler \mapsto \ell] \vdash \alpha\ \mathsf{secure} \quad \Sigma[pc \mapsto \ell] \vdash \beta\ \mathsf{secure}}{\Sigma \vdash \mathbf{try}\ \alpha\ \mathbf{catch}\ \beta\ \mathsf{secure}}\ \mathbf{try}F$$

$$\frac{\Sigma \vdash P : \ell \quad \ell \sqsubseteq \Sigma(handler)}{\Sigma \vdash \mathbf{test}\ P\ \mathsf{secure}}\ \mathbf{test}F$$

There are multiple other possibilities that are less permissive. Requiring the level of each test to be $\bot$ (the minimum element of the security lattice) it will be secure, but prevent some programs that are allowed with the rules above.

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 $\mathbf{try/catch}$, or explain why you think that bounded loops and $\mathbf{try/catch}$ 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 ($\mathsf{L} \sqsubset \mathsf{H}$).

> Assume α contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$ where $x \in \mathsf{use}\ e$ implies $x \notin \mathsf{maydef}\ \alpha$.
>
> For such programs we define $\Sigma \models \alpha$ secure iff
> whenever $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$
> and $\mathsf{eval}\ \omega_1\ e = \mathsf{eval}\ \omega_2\ e$
> then $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

Task 10 (20 points) Assume you are given a security policy Σ and a program α that contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$ satisfying the use/maydef condition. 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.

As before, we can encode $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ with $Q = \bigwedge_{\Sigma(x)=\mathsf{L}}(x = x')$. If that is still true after executing $\alpha\ ;\ \alpha'$, part of the property is encoded. We also want to ensure that the evaluations of e in the initial state are equal. We can say this just with $e = e'$, where e' is the result of renaming all variables of e to their primed versions. Since we assume the use/maydef condition, we do not attempt to capture this. In fact, this would be difficult to do correctly. We then have:

$$R = (e = e' \wedge \bigwedge_{\Sigma(x)=\mathsf{L}} (x = x') \rightarrow [\alpha\ ;\ \alpha'] \bigwedge_{\Sigma(x)=\mathsf{L}} (x = x'))$$

Task 11 (5 points) Give an example policy Σ_0 and program α_0 that is **not** secure due to incorrect use of declassification, that is, the def/mayuse condition is violated. Show the encoding of the example in dynamic logic and determine whether it is valid. Explain your finding.

An incorrect use of declassification would be

$$\alpha_0 = (x_1 := 0 \,;\, y := \mathbf{declassify}_{\mathsf{L}}(x_1 + x_2))$$

where $\Sigma_0 = (x_1 : \mathsf{H}, x_2 : \mathsf{H}, y : \mathsf{L})$. We intend to declassify the sum of x_1 and x_2, but really we declassify x_2. The corresponding formula

$$x_1 + x_2 = x_1' + x_2' \land y = y' \to [\alpha_0 \,;\, \alpha_0']\, y = y'$$

is not valid because the weakest precondition

$$\mathsf{wlp}\ (\alpha_0 \,;\, \alpha_0')\ (y = y') = (0 + x_2 = 0 + x_2')$$

and

$$x_1 + x_2 = x_1' + x_2' \land y = y' \to x_2 = x_2'$$

is not valid. While not necessarily guaranteed by the encoding, the fact that this is not valid is comforting because it expresses that the programs leaks the value of x_2, even though the declassification was intended only to declassify the sum $x_1 + x_2$. It seems that in this and many other cases (and perhaps in general—I am not sure) the encoding in dynamic logic lets us reason about exactly what we can deduce from the assumptions of the noninterference theorem, including the declassified expressions.

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.

If we remove the assignment to x_1 we get

$$\alpha_1 = (y := \mathbf{declassify}_{\mathsf{L}}(x_1 + x_2))$$

with $\Sigma_1 = \Sigma_0$. Then the formula

$$x_1 + x_2 = x_1' + x_2' \land y = y' \to [\alpha_1 \,;\, \alpha_1']\, y = y'$$

is valid. We calculate

$$\mathsf{wlp}\ [\alpha_1 \,;\, \alpha_1']\, (y = y') = (x_1 + x_2 = x_1' + x_2')$$

This means we have to verify that

$$(x_1 + x_2 = x_1' + x_2' \land y = y') \to x_1 + x_2 = x_1' + x_2'$$

is valid, which it is.