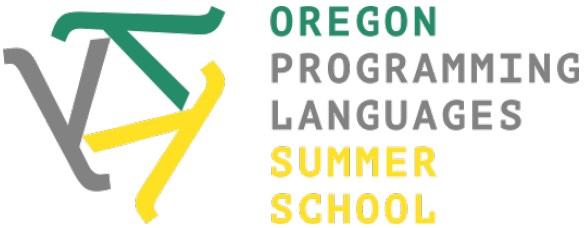

Information Flow Type Systems — Limin Jia

Lecture 2 - *June 24, 2025*

1 Final thoughts from previous lecture

Recall the example of the Strava heatmap, which clearly is a security violation.

- An individual might not mind their anonymized data being shared with Strava. The problem only arises when the data is from a sensitive location.
- So, here, we have two classifications of sensitivity: the data itself, and the fact that the aggregated data reveals information about a sensitive area. Therefore, even aggregate data might not conform to noninterference.
- This is an example of our security properties not appropriately handling the declassification of information.
- From this, we see that information flow security is a *global* property and stronger than access control, which is just *local*.
- We want to establish local policies that guarantee non-interference.

2 Noninterference for concurrent and nondeterministic systems

Non-interference is a property of a system, and it can be defined as the absence of dependency between secret inputs and public outputs. In other words, a system satisfies non-interference if any set of inputs that differ *only* on secret values have indistinguishable public outputs.

Consider a nondeterministic system. We can formulate a new definition of non-interference for this system.

Definition (Possible Noninterference) Given two sets of inputs that differ only in secret inputs, any public outcome for the first inputs is a possible public outcome for the second inputs.

- This definition is okay for truly nondeterministic programs. However, true nondeterminism is impossible in practice. For one, any concurrent program's execution will depend on the scheduler. Most schedulers are deterministic, i.e. round-robin.

As an example of how this definition fails, consider the program in Figure 1 spread across two processes.

```
P1
if(secret == 0) long_computation;
print 0

P2
print 1
```

Figure 1: Example security flaw for concurrent programs

If both processes are running on a round-robin system, and we pass in 0 as the secret, it will almost always print 1 followed by 0.

3 Approaches

Suppose we are working with the following security labels: S (secret) and P (public). To ensure a valid information flow, we must consider the approach to adopt to prevent any invalid flows. One possibility is to enhance our type checker to assign security labels to terms in addition to just types. This is a static approach because the type checker catches errors during compile time. However, it is too *conservative* and may reject programs that are actually okay — false positives —. In other words, static analysis over-approximates security-related behaviors.

An alternative solution is to use a runtime monitor. This checks every step of the running program and either allows it to continue or terminates the program if the execution violates some security property. This approach is dynamic because infractions are caught during runtime, just before execution.

4 Information flow type system

We will work with the IMP language.

$$
\begin{array}{rl}
\textbf{Values} & v ::= x \mid n \\
\textbf{Terms} & e ::= v \mid e_1 \ bop \ e_2 \mid uop \ e \\
\textbf{Commands} & c ::= skip \mid halt \mid x := e \mid if \ e \ then \ c_1 \ else \ c_2 \mid while \ e \ do \ c \mid c_1; c_2 \\
\textbf{Stores} & \sigma ::= \bullet \mid \sigma, x \mapsto v \\
\textbf{Security Labels} & \ell ::= L \mid H
\end{array}
$$

In this terse language, values are variables (x) or numerical constants (n). Terms can be a value, or the binary and unary operators (bop/uop). Here, σ is a store that maps variables to values, representing our runtime state. ℓ is the syntax category for secrecy labels, on top of which there exists a partial order $\sqsubseteq$, e.g. $L \sqsubseteq H$.

For our semantics, we imagine a program e in a store σ will result in a value v. Our programs evaluate like $(\sigma, c) \rightarrow (\sigma', c')$. Also, the meaning of an expression in a store is a value, $[\![e]\!]_\sigma = v$, since terms do not have effects.

Contexts

$\boxed{\Gamma := \overline{x : \ell}}$ A context Γ is a list of secrecy labels (i.e. L, H) assignments to variables. Normally, we would say something like $x : int \ \ell$. For the sake of simplicity, we will omit the concrete type part (e.g. int) and focus on labels (ℓ) from now on.

Security typing

The judgment $\boxed{\Gamma \vdash e : \ell}$ asserts that a expression e has label ℓ under the context Γ. Take the typing rule for bop as an example:

$$\frac{\Gamma \vdash e_1 : \ell_1 \qquad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \, bop \, e_2 : \ell_1 \sqcup \ell_2}$$

We can observe that the term has $e1 \ \ textbop \, e2$, which guarantees that the resulting label is the greatest upper bound among $l1$ and $l2$. To materialize this, let us consider that $l_1 = L$ and $l_2 = H$. An operation involving H should be treated as H.

Typing commands

The judgment $\boxed{\Gamma, pc \vdash c}$ checks if a command c respects the security context pc — a security label — in context Γ. The label pc represents the current security label of the systems. We start the execution of a program by assigning the lowest label ($\bot$) to pc. Let us define the type checking rule for if statements:

$$\frac{\Gamma \vdash b : \ell \qquad \Gamma, pc \sqcup \ell \vdash c_1 \qquad \Gamma, pc \sqcup \ell \vdash c_2}{\Gamma, pc \vdash if\ b\ then\ c_1\ else\ c_2}$$

In this rule, b has the security label ℓ. Notice how c_1 and c_2 are type-checked under a combined label $pc \sqcup \ell$. This guarantees that we respect the higher label among pc and ℓ. The purpose of this inference rule is that the type-checking for a branching statement will ensure that the label for either branch respects the combined label $pc \sqcup \ell$. This avoids leaks that could occur if b has a higher label than its branches.

Another example, for assignment:

$$\frac{\Gamma \vdash e : \ell \qquad \Gamma(x) = \ell \qquad pc \sqsubseteq \ell}{\Gamma, pc \vdash x := e}$$

This states that when we assign a term to a variable, the label of the variable and the term must be compatible and also respect the current pc.

Consider the following rule. We will examine how it violates non-interference:

$$\frac{\Gamma \vdash e : \ell_1 \qquad \ell_1 \sqsubseteq \ell_2}{\Gamma \vdash e : \ell_2}$$

When combined with the previous rule, this rule violates non-interference. Namely, we are able to derive e as a high (ℓ_2) value from this rule, but using the previous rule (with x as e, pc as ℓ_1, and ℓ as ℓ_2), we have that x is in a low context (pc, or ℓ_1).

Attacker label

Let ℓ_a denote the label of an attacker.

ℓ_a Equivalence

We define $v_1 \approx_{\ell_a} v_2 : \ell$ by

$$\frac{\ell \sqsubseteq \ell_a \qquad v_1 = v_2}{v_1 \approx_{\ell_a} v_2 : \ell}$$

If the attacker is at a higher level than the values, the values must be *identical* in order for them to be *indistinguishable*.

$$\frac{\ell \not\sqsubseteq \ell_a}{v_1 \approx_{\ell_a} v_2 : \ell}$$

Otherwise, every value is indistinguishable to the attacker.

ℓ_a Equivalence of stores

Similarly, we define $\Gamma \vdash \sigma_1 \approx_{\ell_a} \sigma_2$ by

$$\Gamma \vdash \bullet \approx_{\ell_a} \bullet$$

$$\frac{\Gamma \vdash \sigma_1 \approx_{\ell_a} \sigma_2 \qquad v_1 \approx_{\ell_a} v_2 : P(x)}{\Gamma \vdash \sigma_1, x \to v_1 \approx_{\ell_a} \sigma_2, x \to v_2}$$

Lemma *(Noninterference of terms).* If $\Gamma \vdash e : \ell$, and $\Gamma \vdash \sigma_1 \approx_{\ell_a} \sigma_2$, then $[\![e]\!]_{\sigma_1} \approx_{\ell_a} [\![e]\!]_{\sigma_2} : \ell$.

Sequencing

$$\begin{aligned} \textbf{Evaluation Context} \quad & E ::= \Box \mid \Box; c \\ \textbf{Stack} \quad & s ::= \cdot \mid E \triangleright s \\ \textbf{State} \quad & \Sigma ::= (\sigma, s, c) \end{aligned}$$

To actually run programs, we evaluate a sequence of commands. Think of the stack as containing continuations. The sequencing rule for pushing to the stack is

$$(\sigma, s, c_1; c_2) \to (\sigma, (\Box; c_2) \triangleright s, c_1).$$

Likewise, for pop, when there is no command to execute immediately ($SKIP$) but there is some command pending in the stack ($(\Box; c) \triangleright s$), we move it out to be the currently executing command.

$$(\sigma, (\Box; c) \triangleright s, SKIP) \to (\sigma, s, c)$$

Here's an additional rule for SKIP, when we have no immediate command nor pending command in the stack, we can halt the machine.

$$(\sigma, \Box, SKIP) \to (\sigma, HALT)$$

Lemma (*Noninterference for Commands*). If $\Gamma, \bot \vdash c$, and $\Gamma \vdash \sigma_1 \approx_{\ell_a} \sigma_2$, and $(\sigma_i, \Box, c) \to^* (\sigma_i', HALT)$ (for $i = 1, 2$) then $\Gamma \vdash \sigma_1' \approx_{\ell a} \sigma_2'$.

We can prove this by induction on the number of steps.