**15-316: Software Foundations of Security and Privacy** **Spring 2017**

# Lecture 13: Information Flow Types

*Lecturer: Jean Yang* *Based on material from: Matt Fredrikson*

Prior to this class, you may have thought that type systems were good only for making sure values were well-formed in memory and what we said they were. It turns out that type systems can be used for proving all kinds of interesting properties about programs. We can statically prevent implicit flows using types, and we can prove it!

First, we will prove soundness for the information typing rules we saw last lecture. Recall that the premise of this type system is that we wanted to be able to *statically* rule out information flow violations. We are interested in *decentralized information flow control*, where rather than having a centralized authority that is in charge of *declassifying* sensitive information. With both process-based information flow and with our type system, the programmer associates *labels* with information describing the permissions, and then either a *reference monitor* or a *type checker* prevents disallowed flows.

Previously, we discussed how the *noninterference* property we want with information is actually a much more permissive property than is actually checked most of the time. Simply *reading* a sensitive value does not violate noninterference; a program must compute a sensitive values in a way that restricted-access viewer can tell the difference between two different sensitive inputs. Systems usually enforce a more conservative version of this property. For instance, Flume enforced a conservative, process-level version of information flow using a reference monitor. (Important! Make sure you understand why non-conservative information flow is not a safety property, and thus cannot be enforced using reference monitors.)

Last class, we began to define an information flow type system for a simple imperative language. While this language seems simple, research for proving interesting properties about very real languages (such as Java) often starts out with these small models! Last class we defined the *abstract syntax*, *operational semantics*, and *typing rules* for the language. This class, we will prove *soundness* for the typing rules: that programs that leak information do not type-check.

## 13.1 Review of Definitions

Recall that we have been working with the following simple imperative language.

$$
\begin{array}{lcl}
a \in \mathbf{AExp} & ::= & n \in \mathbb{Z} \mid x \in \mathbf{Var} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
b \in \mathbf{BExp} & ::= & \mathbf{T} \mid \mathbf{F} \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid a_1 = a_2 \mid a_1 \leq a_2 \\
c \in \mathbf{Com} & ::= & \mathbf{skip} \mid x := a \mid c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c
\end{array}
$$

Last class, we defined an *operational semantics* for our language, corresponding to the rules necessary to define an interpreter. States are maps $\sigma : \mathbf{Var} \mapsto \mathbb{Z}$. We describe expression evaluation with the big-step relation $\langle \sigma, a \rangle \Downarrow n$. For instance, for **AExp** we have:

$$
\frac{}{\langle \sigma, n \rangle \Downarrow n} \quad \frac{}{\langle \sigma, x \rangle \Downarrow \sigma(x)} \quad \frac{\langle \sigma, a_1 \rangle \Downarrow n_1 \quad \langle \sigma, a_2 \rangle \Downarrow n_2 \quad n = n_1\ \mathbf{op}\ n_2}{\langle \sigma, a_1\ \mathbf{op}\ a_2 \rangle \Downarrow n}
$$

We describe command evaluation with the big-step relation $\langle \sigma, \rangle \Downarrow 1c\sigma_2$:

$$\frac{\langle \sigma, a\rangle \Downarrow n}{\langle \sigma, x := a\rangle \Downarrow \sigma[x \mapsto n]} \quad \frac{\langle \sigma_1, c\rangle \Downarrow \sigma_2}{\langle \sigma_1, \textbf{skip}; c\rangle \Downarrow \sigma_2} \quad \frac{\langle \sigma_1, c_1\rangle \Downarrow \sigma_1' \quad \langle \sigma_1', c_2\rangle \Downarrow \sigma_2}{\langle \sigma_1, c_1; c_2\rangle \Downarrow \sigma_2}$$

$$\frac{\langle \sigma, b\rangle \Downarrow \textbf{T} \quad \langle \sigma, c_1\rangle \Downarrow \sigma_2}{\langle \sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\rangle \Downarrow c_2} \quad \frac{\langle \sigma, b\rangle \Downarrow \textbf{T} \quad \langle \sigma, c_2\rangle \Downarrow \sigma_2}{\langle \sigma, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\rangle \Downarrow c_2}$$

$$\frac{\langle \sigma, b\rangle \Downarrow \textbf{F}}{\langle \sigma, \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma} \quad \frac{\langle \sigma, b\rangle \Downarrow \textbf{T} \quad \langle \sigma_1, c\rangle \Downarrow \sigma_1' \quad \langle \sigma_1', \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_2}{\langle \sigma_1, \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_2}$$

## 13.2 Typing Rules

Now that we have a language, we can define types to prevent information leaks in this language! We define $L = (SC, \leq, \sqcup, \sqcap, \bot)$ to be a security lattice. A type environment $\Gamma : \textbf{Var} \mapsto SC$ for a program $c$ maps each variable in $c$ to a program label, and additional contains a mapping for the program counter **pc**. We have the following notation:

- $\Gamma \vdash e : \ell$ means expression $e$ has label $\ell$ under $\Gamma$.
- $\Gamma \vdash c$ means $c$ is well-typed under $\Gamma$.
- Environment $(\Gamma, x :: \ell)$ gives $x$ type $\ell$, preserves rest of $\Gamma$.

To give some context, all these definitions are because we eventually want to prove that well-typed programs preserve non-interference. Recall that we define non-interference in terms of states that are equivalent under a given security label:

- **State Equivalence.** Two states $\sigma_1$, $\sigma_2$ are $\ell$-equivalent to an observer of class $\ell \in SC$ under $\Gamma$, written $\sigma_1 \approx_{\ell,\Gamma} \sigma_2$ (and abbreviated as $\sigma_1 \approx \sigma_2$) if and only if
$$\forall x \in \textbf{Var}.\Gamma(x) \leq \ell \Rightarrow \sigma_1(x) = \sigma_2(x) \tag{13.1}$$
- **Noninterference.** A program $c$ satisfies noninterference at class $\ell$ under $\Gamma$ if $\ell$-equivalent initial states lead to $\ell$-equivalent final states:
$$\forall \sigma_1, \sigma_2.\sigma_1 \approx_\ell \sigma_2 \wedge \langle \sigma_1, c\rangle \Downarrow \sigma_1' \wedge \langle \sigma_2, c\rangle \Downarrow \sigma_2' \Rightarrow \sigma_1' \approx_\ell \sigma_2' \tag{13.2}$$

Last class, we discussed the following rules for typing expressions:

$$\textsc{Var}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash n : \bot} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \textbf{T} : \bot} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \textbf{F} : \bot} \qquad \textsc{Bin}\ \frac{\Gamma \vdash a_1 : \ell_1 \quad \Gamma \vdash a_2 : \ell_2}{\Gamma \vdash a_1 \textbf{ op } a_2 : \ell_1 \sqcup \ell_2}$$

We have the following rules for typing commands:

$$\textsc{Skip}\ \frac{}{\Gamma \vdash \textbf{skip}} \qquad \textsc{Comp}\ \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \qquad \textsc{Asgn}\ \frac{\Gamma \vdash a : \ell \quad \ell \sqcup \Gamma(\textbf{pc}) \leq \Gamma(a)}{\Gamma \vdash x := a} \qquad \textsc{While}\ \frac{\Gamma \vdash b : \ell \quad \ell' = \Gamma(\textbf{pc}) \sqcup \ell \quad \Gamma, \textbf{pc} :: \ell' \vdash c}{\Gamma \vdash \textbf{while } b \textbf{ do } c}$$

$$\textsc{If}\ \frac{\Gamma \vdash b : \ell \quad \ell' = \Gamma(pc) \sqcup \ell \quad \Gamma, pc :: \ell' \vdash c_1 \quad \Gamma, pc :: \ell' \vdash c_2}{\Gamma \vdash \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2}$$

## 13.3 Type System in Action

For a command typing example, consider the expression **if** $p = g$ **then** $o := 1$ **then** $o := 2$, under the environment $\Gamma = p :: H, g :: L, o :: L, pc :: L$. We should not be able to type this expression, because it permits an *implicit flow*! Let us look at how the typing rules prevent this from happening. Our goal is to type the following:

$$\textsc{If}\ \frac{}{\Gamma \vdash \textbf{if}\ p = g\ \textbf{then}\ o := 1\ \textbf{then}\ o := 2} \tag{13.3}$$

First, we type the binary expression p=g, which gets assigned label $H$, and so our new **pc** label gets bumped up to be at least $H$:

$$\textsc{If}\ \frac{\textsc{Bin}\ \dfrac{\cdots}{\Gamma \vdash p = g : H} \quad \Gamma(\textbf{pc}) \sqcup H}{\Gamma \vdash \textbf{if}\ p = g\ \textbf{then}\ o := 1\ \textbf{then}\ o := 2} \tag{13.4}$$

For this expression to be well-typed, the assignment of $o$ under the condition needs to be well-typed. Let's try to type it:

$$\textsc{If}\ \frac{\textsc{Bin}\ \dfrac{\cdots}{\Gamma \vdash p = g : H} \quad \Gamma(\textbf{pc}) \sqcup H \quad \dfrac{\textsc{Asgn}\ \dfrac{\textsc{Int}\ \dfrac{}{\Gamma \vdash 1 : L} \quad L \sqcup H \leq \Gamma(o)}{\Gamma, \textbf{pc} :: H \vdash o := 1}}{\Gamma \vdash \textbf{if}\ p = g\ \textbf{then}\ o := 1\ \textbf{then}\ o := 2} \tag{13.5}$$

This doesn't work! We have $\Gamma(o) = L$, and we need $L \sqcup H \leq \Gamma(o)$. The guard on the condition raises the **pc** label and then propagating it, making it so the *type system* can prevent the implicit leak.

**Another case.** Now consider what happens if instead, we had the expression **if** $p = g$ **then** $o := 1$ **then** $o := 1$. In this case, both branches are the same, so it should not cause an information leak to do the assignment under the conditional. However, our type system does not know this, and would still forbid it.

## 13.4 Soundness

Now we will prove *soundness* for our type system. In a logical system, soundness is the property that the inference rules only prove properties that are valid with respect to the semantics. In our type system, soundness is the property of *non-interference* that we designed the type system to enforce. We now want to show that if a program is well-typed, then it preserves non-interference.

More formally, the type system is sound if whenever conditions 1-3 hold for program $c$ and type environment $\Gamma$, then $c$ has noninterference (*i.e.*, the final states $\sigma_1' \approx_\ell \sigma_2'$ for any starting states $\sigma_1 \approx_\ell \sigma_2$):

1. $\Gamma \vdash c$
2. $\langle \sigma_1, c \rangle \Downarrow \sigma_1', \langle \sigma_2, c \rangle \Downarrow \sigma_2'$
3. $\sigma_1 \approx_\ell \sigma_2$

## 13.5 Key Lemmas

Our proof of soundness is based on two key lemmas, which we will prove first. The Simple Security Lemma talks about the security of *reads*: an expression never reads what it is supposed to read. The Confinement Lemma talks about secure *writes*: commands never write to somewhere that they are not supposed to write to.

Something that makes our proofs convenient is that, in our formalism, expressions, commands, and the *derivations* of evaluating them are all trees made up of smaller expressions, commands, and derivations. This allows us to prove properties by *induction*, where the base cases are judgments with no assumptions. We prove each of these lemmas by induction on the structures of expressions and commands, respectively. We then use these lemmas to prove soundness, by induction on the structure of the derivation.

**Lemma 13.1 (Simple Security)** *Expressions never read variables above their typed class: if* $\Gamma \vdash e : \ell$*, then for every variable* $x$ *appearing in* $e$*,* $\Gamma(x) \leq \ell$*.*

**Proof:** By induction on the structure of $e$:

- Base cases $n$, $\mathbf{T}$, and $\mathbf{F}$ are trivial.
- Base case $x$: we have $\Gamma \vdash x : \ell$.
- Case $e_1$ **op** $e_2$: by BIN, we have $\Gamma \vdash e_1 : \ell_1$ and $\Gamma \vdash e_2 : \ell_2$. By induction, we have $\forall x \in e_1.\Gamma(x) \leq \ell_1$ and $\forall x \in e_2.\Gamma(x) \leq \ell_2$. Then $\Gamma(x) \leq \ell_1 \sqcup \ell_2 = \ell$ for all $x$ in $e = e_1$ **op** $e_2$.

■

**Lemma 13.2 (Confinement)** *Commands never write to variables below* ***pc****'s typed class: if* $\Gamma \vdash c$*, then for every variable* $x$ *assigned in* $c$*,* $\Gamma(\mathbf{pc}) \leq \Gamma(x)$*.*

**Proof:** By induction on the structure of $c$:

- Base case **skip** is trivial.
- Base case $x := a$: we have $\Gamma \vdash a : \ell$. By ASGN, $\ell \sqcup \Gamma(\mathbf{pc}) \leq \Gamma(x)$, so $\Gamma(\mathbf{pc}) \leq \Gamma(x)$ by $\leq$-transitivity.
- Case $c_1; c_2$ follows directly by induction.
- Case **while** $b$ **do** $c$: suppose $\Gamma \vdash b : \ell$. By WHILE, we have that $\Gamma, \mathbf{pc} :: (\ell \sqcup \Gamma(\mathbf{pc})) \vdash c$. By induction, we have that $\forall x \in c.\ell \sqcup \Gamma(\mathbf{pc}) \leq \Gamma(x)$. By $\leq$-transitivity, $\forall x \in c.\Gamma(\mathbf{pc}) \leq \Gamma(x)$.
- The case for **if** is similar to **while**.

■

Now we can prove the soundness theorem.

**Theorem 13.3** *The type system is sound if whenever conditions 1-3 hold for program c and type environment* $\Gamma$*, then c has noninterference (*i.e.*, the final states* $\sigma_1' \approx_\ell \sigma_2'$ *for any starting states* $\sigma_1 \approx_\ell \sigma_2$*):*

1. $\Gamma \vdash c$

2. $\langle \sigma_1, c\rangle \Downarrow \sigma_1', \langle \sigma_2, c\rangle \Downarrow \sigma_2'$
3. $\sigma_1 \approx_\ell \sigma_2$

The proof sketch is that we can use Simple Security to argue about identical evaluation, and we can use Confinement to argue about $\ell$-equivalent updates. We will do the **while** case for the proof below:

**Proof:** By induction on the derivation of $\langle \sigma_1, c\rangle \Downarrow \sigma_1'$. Suppose $\langle \sigma_1, \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_1'$, and typing ends with the rule:

$$\frac{\Gamma \vdash b : \ell_1 \quad \ell_2 = \Gamma(\textbf{pc}) \sqcup \ell_1 \quad \Gamma, \textbf{pc} :: \ell_2 \vdash c}{\Gamma \vdash \textbf{while } b \textbf{ do } c} \text{ WHILE}$$

Now there are two cases: either $\ell_2 \leq \ell$, and the new **pc** for the conditional is below our permitted security class, or $\ell_2 > \ell$, and the new **pc** is above our permitted security class.

In the case $\ell_2 \leq \ell$,

- By Simple Security, $\Gamma(x) \leq \ell_2 \leq \ell$ for all $x$ in $b$.
- Since we know $b$ does not read variables above $\ell$, we can apply (3) to get $\sigma_1(x) = \sigma_2(x)$ for all $x$ in $b$, so $\langle \sigma_1, b\rangle \Downarrow v$ and $\langle \sigma_2, b\rangle \Downarrow v$.
- If $v = \textbf{F}$, then we do not evaluate the body, and $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$. We can invoke (3) again.
- If $v = \textbf{T}$, then we can induct over the derivation tree:

  $$\frac{\langle \sigma_1, b\rangle \Downarrow \textbf{T} \quad \langle \sigma_1, c\rangle \Downarrow \sigma_1'' \quad \langle \sigma_1'', \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_1'}{\langle \sigma_1, \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_1'} \quad \frac{\langle \sigma_2, b\rangle \Downarrow \textbf{T} \quad \langle \sigma_2, c\rangle \Downarrow \sigma_2'' \quad \langle \sigma_2'', \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_2'}{\langle \sigma_2, \textbf{while } b \textbf{ do } c\rangle \Downarrow \sigma_2'}$$

  Then $\sigma_1'' \approx_\ell \sigma_2''$ by induction, and $\sigma_1' \approx_\ell \sigma_2'$ also by induction.

For the case $\ell_2 > \ell$, the condition has a higher security typed class than our desired class $\ell$, so we are worried about implicit flows of the condition. We use Confinement to argue about $\ell$-equivalent updates:

- By Confinement, $\ell_2 \leq \Gamma(x)$ for all $x$ assigned in $c$.
- For $x$ assigned in $c$, $\Gamma(x) \not\leq \ell$, since $\ell_2 > \ell$.
- For every $x$ in $c$ where $\Gamma(x) \leq \ell$, $\sigma_1(x) = \sigma_1'(x)$ and $\sigma_2(x) = \sigma_2'(x)$.
- By (3), we have $\sigma_1 \approx_\ell \sigma_2$.

■