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

Lecture Notes on Information Flow Types I

Matt Fredrikson

Carnegie Mellon University

Lecture 12

1 Introduction

In the previous lecture we had a formal introduction to information flow policies. We saw why they are different than safety properties, in that they cannot be enforced or even checked just by looking at a single execution trace. While there are ways of approximating information flow with safety properties, most notably with dynamic taint analysis [SAB10], doing so will result in missing some real flows.

The property that captures information flow is called *non-interference*, and is shown in Definition 1. Recall that the big-step transition relation $\langle \omega, \alpha \rangle \Downarrow \omega'$ used in the definition denotes the fact that by executing α starting in ω it is possible to terminate in state ω'. Additionally, the policy Γ is a mapping from variables to security labels, which we said are either $\mathtt{H}$ (high security) or $\mathtt{L}$ (low security). We write specific policies out as lists, so the policy that maps x to $\mathtt{L}$ and y to $\mathtt{H}$ is $(x : \mathtt{L}, y : \mathtt{H})$. If Γ is a policy, possibly containing a mapping for x, then $(\Gamma, x : \mathtt{L})$ is the policy that maps x to $\mathtt{L}$ and everything else y to $\Gamma(y)$.

Finally, to properly define secure information flow we needed to formalize a concept of ℓ-equivalent states. Let ω_1 and ω_2 be states, and Γ be a policy. Then we say that ω_1 and ω_2 are ℓ-equivalent, where $\ell \in \{\mathtt{L}, \mathtt{H}\}$ is a security label, if and only if all of the variables where $\Gamma(x) = \ell$ have the same value in ω_1 and ω_2.

$$\forall x. \Gamma(x) = \ell \rightarrow \omega_1(x) = \omega_2(x) \tag{1}$$

We use the notation $\omega_1 \approx_{\Gamma,\ell} \omega_2$ to denote the relation in Equation 1, and if the choice of Γ is clear from the context, we will abbreviate it as $\omega_1 \approx_{\ell} \omega_2$.

Definition 1 (Non-interference). Let α be a program and Γ a policy associating security labels to all of the variables in α. Then α satisfies non-interference under Γ if and only if executing α under L-equivalent states leads to final states that are also L-equivalent. More precisely,

$$\forall\omega_1,\omega_2.\omega_1 \approx_{\Gamma,\mathtt{L}} \omega_2 \land \langle\omega_1,\alpha\rangle \Downarrow \omega_1' \land \langle\omega_2,\alpha\rangle \Downarrow \omega_2' \to \omega_1' \approx_{\Gamma,\mathtt{L}} \omega_2' \tag{2}$$

where ω_1 and ω_2 range over the set of possible program states.

Equation 2 captures the idea that whatever the values of high-security variables labeled H may be, they should not influence the values that the low-security values labeled L variables take when the program terminates. If such influence exists, then the program does *not* satisfy non-interference. In this way, non-interference equates information flow with the tendency of one variable to influence that of another.

Today we will dive deeper into information flow. We will start by squaring away some preliminaries, namely how the big-step transition relation is defined for our simple imperative language. Then we will introduce a type system [VIS96] that is able to enforce Definition 1 with respect to a particular policy. That is, only programs that satisfy non-interference can be typed according to this system. Along the way we'll generalize the set of security labels that are allowed to appear in the policy Γ, to allow for richer policies that just labeling variables as either H or L.

2 Big-step semantics

In previous lectures we have defined the semantics of languages in two different ways. When we studied safety properties, we defined the *trace semantics* in terms of the set of traces that a program can execute. Later, we looked at the *small-step semantics*, which define how each step of a computation proceeds by modifying state. These both offer a similar view of a program's exeuction, because they characterize the sequence of intermediate states between the initial and final states.

The big-step semantic relation $\langle\omega,\alpha\rangle \Downarrow \omega'$ used in Definition 1 is different. Rather than "recording" the sequence of transitions that a program takes as it executes, this way of defining the semantics simply relates the initial state ω with a possible final state ω'. If the language is deterministic, as in the case of our simple imperative language, then there will only ever be one possible final state for each initial state of a given program. Also noteworthy is the fact that when using big-step semantics, there is no direct way to refer to divergent behavior of programs. If α diverges from initial state ω, then there is no ω' for which we could write $\langle\omega,\alpha\rangle \Downarrow \omega'$.

Why bother with another way of defining semantics at this point? As we will see when we define the type system and prove that it enforces non-interference, the structure of the big-step semantics definition closely mirrors that of the typing rules. This symmetry makes proving things about the typesystem simpler and easier to follow, highlighting the correspondence between the syntactic typing rules and the underlying semantics of the language.

Expressions. Let's proceed with defining the big-step semantics of our language. Recall that the syntax of terms is defined by the following constructors.

$$e, \tilde{e} \ ::= \ x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e}$$

We will simplify this a bit by abstracting away the functions $+$ and $\cdot$, and instead just use the function symbol $\odot$ to stand for either. The big-step semantics of a term relates an initial state ω and term e with a final value v, because that is how terms are evaluated in states. We will use the symbol $\Downarrow_{\mathbb{Z}}$ to denote the big-step relation on terms, where the $\mathbb{Z}$ subscript reminds us of the fact that we evaluate *arithmetic* terms to *integer* values.

We can begin defining the semantics of terms. We will do so using a set of rules that resemble the proof rules we have used previously. The most straightforward rule is the one for constants, which is shown in Equation 3. This rule has no conditions, so it always applies for any constant c and state ω, and simply says that the value of a constant is the constant itself.

$$\frac{}{\langle \omega, c \rangle \Downarrow_{\mathbb{Z}} c} \tag{3}$$

The next rule covers terms that consist of just a variable x, and is shown in Equation 4. Intuitively, we want a variable to evaluate to the integer that ω maps it to. The rule reflects this in its antecedent, which requires that $\omega(x) = v$. The entire rule can be read "if $\omega(x) = v$ (i.e. ω maps x to value v), then the term x evaluates to v in state ω."

$$\frac{\omega(x) = v}{\langle \omega, x \rangle \Downarrow_{\mathbb{Z}} v} \tag{4}$$

The next and final rule shown in Equation 5 for arithmetic terms covers operations, abstractly represented by symbol $\odot$. The antecedent refers to the big-step semantic relation for terms $\Downarrow_{\mathbb{Z}}$ to refer to the values of each operand $e, \tilde{e}$, and the consequent defines the value of the operation to be the operation applied to the value of each operand.

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v_1 \quad \langle \omega, \tilde{e} \rangle \Downarrow_{\mathbb{Z}} v_2}{\langle \omega, e \odot \tilde{e} \rangle \Downarrow_{\mathbb{Z}} v_1 \odot v_2} \tag{5}$$

Arithmetic terms aren't the only sort of term-like object that programs can use. Conditional and while-loop commands can also refer to formulas P to test when deciding which branch to take. We will think of these as terms as well, but we will distinguish them from arithmetic terms by defining a separate big-step transition relation $\Downarrow_{\mathbb{B}}$ to reflect the fact that they evaluate to Boolean values $\mathit{true}, \mathit{false}$ rather than integers. With this in mind, their definitions are similar to those for arithmetic expressions, and we won't belabor the matter with a tedious explanation of each. The rules are shown in

Equation 6

$$\frac{}{\langle\omega,\texttt{true}\rangle \Downarrow_{\mathbb{B}} \mathit{true}} \quad \frac{}{\langle\omega,\texttt{false}\rangle \Downarrow_{\mathbb{B}} \mathit{false}} \quad \frac{\langle\omega,P\rangle \Downarrow_{\mathbb{B}} b}{\langle\omega,\texttt{!}P\rangle \Downarrow_{\mathbb{B}} \neg b}$$

$$\frac{\langle\omega,P\rangle \Downarrow_{\mathbb{B}} b_1 \quad \langle\omega,Q\rangle \Downarrow_{\mathbb{B}} b_2}{\langle\omega,P \;\texttt{\&\&}\; Q\rangle \Downarrow_{\mathbb{B}} b_1 \wedge b_2} \quad \frac{\langle\omega,P\rangle \Downarrow_{\mathbb{B}} b_1 \quad \langle\omega,Q\rangle \Downarrow_{\mathbb{B}} b_2}{\langle\omega,P \;\texttt{||}\; Q\rangle \Downarrow_{\mathbb{B}} b_1 \vee b_2} \tag{6}$$

$$\frac{\langle\omega,e\rangle \Downarrow_{\mathbb{Z}} v_1 \quad \langle\omega,\tilde{e}\rangle \Downarrow_{\mathbb{Z}} v_2}{\langle\omega,P \;\texttt{==}\; Q\rangle \Downarrow_{\mathbb{B}} v_1 = v_2} \quad \frac{\langle\omega,e\rangle \Downarrow_{\mathbb{Z}} v_1 \quad \langle\omega,\tilde{e}\rangle \Downarrow_{\mathbb{Z}} v_2}{\langle\omega,P \;\texttt{<=}\; Q\rangle \Downarrow_{\mathbb{B}} v_1 \le v_2}$$

Commands. Now we come to the commands. We will no longer use the $\texttt{assert}(Q)$ command, as it was useful primarily for thinking about safety properties and we are concerned with information flow now. We will also forget for the moment about memory accesses and state, and focus only on assignments to variables. So the language that we will consider is shown in Equation 7.

$$\alpha,\beta \;::=\; x := e \mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta \mid \alpha;\beta \mid \texttt{while}(Q)\,\alpha \tag{7}$$

The big-step semantics for the language are actually a bit more intuitive than the trace and small-step semantics. Recall that $\langle\omega,\alpha\rangle \Downarrow \omega'$ denotes the fact that when executing α starting in ω, it is possible to end in ω'. So the rule for assignment is shown in Equation 8, and it simply says that if the right-hand side e evaluates to v in ω, then the final state after the assignment will be the same as ω but with x mapping to v. Notice that to reason about the value v of the right-hand side, the big-step relation for arithmetic expressions $\Downarrow_{\mathbb{Z}}$ is used.

$$\frac{\langle\omega,e\rangle \Downarrow_{\mathbb{Z}} v}{\langle\omega,x := e\rangle \Downarrow \omega\{x \mapsto v\}} \tag{8}$$

The remaining rules for commands refer to the big-step command transition rule recursively. The one for composition is shown in Equation 9, and its antecedent defines an intermediate state ω_1 for after α executes, which β begins executing in to finally reach ω'. But this intermediate state is not present in the transition $\langle\omega,\alpha;\beta\rangle \Downarrow \omega'$ of the composition, and is "forgotten" as the only thing that matters is the final state ω'.

$$\frac{\langle\omega,\alpha\rangle \Downarrow \omega_1 \quad \langle\omega_1,\beta\rangle \Downarrow \omega'}{\langle\omega,\alpha;\beta\rangle \Downarrow \omega'} \tag{9}$$

The rules for conditionals are shown in Equation 10. Note that there are two of them, one for the case where the condition P evaluates to *true*, and another for when it evaluates to *false*. Everything else is the same between the two rules, except that the final state is related to either the "if" branch α or the "then" branch β.

$$\frac{\langle\omega,P\rangle \Downarrow_{\mathbb{B}} \mathit{true} \quad \langle\omega,\alpha\rangle \Downarrow \omega'}{\langle\omega,\texttt{if}(P)\,\alpha\,\texttt{else}\,\beta\rangle \Downarrow \omega'} \quad \frac{\langle\omega,P\rangle \Downarrow_{\mathbb{B}} \mathit{false} \quad \langle\omega,\beta\rangle \Downarrow \omega'}{\langle\omega,\texttt{if}(P)\,\alpha\,\texttt{else}\,\beta\rangle \Downarrow \omega'} \tag{10}$$

Finally, the while-loop command has two rules as well, as shown in Equation 11. The first covers the case where P evaluates to *false*, and says that the final state is the same as the initial. The other covers P evaluating to *true*, and defines the final state as that of executing the body α once, and then executing the loop immediately afterwards. You may notice the resemblence to the axioms we used when reasoning about `while` commands earlier in the semester.

$$\frac{\langle \omega, P\rangle \Downarrow_{\mathbb{B}} \mathit{false}}{\langle \omega, \mathtt{while}(P)\, \alpha\rangle \Downarrow \omega} \quad \frac{\langle \omega, P\rangle \Downarrow_{\mathbb{B}} \mathit{true} \quad \langle \omega, \alpha; \mathtt{while}(P)\, \alpha\rangle \Downarrow \omega'}{\langle \omega, \mathtt{while}(P)\, \alpha\rangle \Downarrow \omega'} \tag{11}$$

3 Enforcing non-interference with types

We now turn to developing a type system that enforces Definition 1. What does it mean for a type system to *enforce* a security policy like non-interference? To understand this, think about types as you understand them from programming languages like ML.

When we associate the type `int` with a variable x in our program, we are telling the compiler that we expect x to only hold data representing signed machine integers. Thus, the compiler should expect us to perform operations on x and make use of it in ways that are consistent with machine integers. So for example performing addition on x with other `int` data would be perfectly normal and correct behavior, but attempting to invoke x as though it were a function or append a string to x would not. The compiler would tell us that this is a type error, and refuse to produce executable code for our program.

In the case above, the compiler checked the types of data used in operations and prevented us from performing an operation on `int` data that has no well-defined meaning. If it had not, and instead compiled our program after trying to infer some reasonable intention on our part (as in the case of languages like Javascript and Python) regarding the nonsensical integer operations, then who knows what behavior would result? This form of *type safety* gives us modest assurances about how our program will behave when its commands are consistent with the types that we associate to data.

We wish to ensure that our programs satisfy non-interference, and the types that we associate with variables label them as either `H` ("high security") or `L` ("low security"). Perhaps the type-checker can confirm that all of the computations that are done on our data are consistent with these labelings, so that only programs satisfying non-interference will successfully typecheck, and those that do not will be rejected. This is the goal that we will pursue in this section, starting with expressions and moving to commands.

3.1 Assigning types to expressions

Recall that our policy Γ assigns labels `L`,`H` to variables. The meaning of such an assignment is that the data stored in the variable is either of no concern to the security policy, i.e. should not contain secret data (`L`), or is allowed to contain secret data (`H`). In our language, variables are a type of expression and expressions represent data, so it is natural to extend this labeling to other expressions as well. The notation that we use to say that an expression e has label ℓ in policy Γ is shown in Equation 12.

$$\Gamma \vdash e : \ell \tag{12}$$

Notice the familiar sequent notation, which should suggest to you that Γ can be thought of as our assumptions, and $e : \ell$ is what we want to prove from these assumptions.

So what rules do we use to prove these judgements? Let's start from the simple cases, and work our way up to more complicated ones. The simplest possible expression are constants, which in our language can be integers c or Boolean constants `true`, `false`. What security type do we assign to constants? Perhaps there is a conceivable scenario in which considering program constants as secret `H` data makes sense, but in general we would like to assume that an attacker is allowed to see the syntax of a program. So we will assume that constants are always `L`, reflected in the rules ConstL, TrueL, and FalseL below.

$$(\text{ConstL})\ \frac{}{\Gamma \vdash c : \mathtt{L}} \quad (\text{TrueL})\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{L}} \quad (\text{FalseL})\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{L}}$$

Variable expressions are also easy to reason about, as their labels follow immediately from the policy as shown in rule Var.

$$(\text{Var})\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

Now moving on to operations on expressions, we will treat arithmetic and Boolean expressions the same, again abstracting operators with $\odot$. But we will treat binary operations that take two operands differently from unary operations like $\neg$ that take a single operand by giving them separate rules. First consider the case of unary operations, shown in UnOp below.

$$(\text{UnOp})\ \frac{\Gamma \vdash e : \ell}{\Gamma \vdash \odot\, e : \ell}$$

If the operand in this expression has, for example, type `L` then it means that the term does not contain any secret information. After applying a fixed operation to it, containing no other sources of data, can it have secret information afterwards? Certainly not, and the same goes for operands labeled `H`, so UnOp carries the type of its operand to the full expression.

On the other hand, binary operators are different. If an expression consisting of a binary operation has two operands that are labeled `L` under Γ, then by the same reasoning above the result should also have this label. This is reflected in BinOpL. On the other hand, if either of its operands are labeled `H`, denoting that the expression carries secret data, then will the result? The answer is that it depends, because the expression could evaluate to a constant as in $x + -x$. But without considering such cases in detail, we ultimately want our type system to reject *any* program that has an information flow from H to L, and one could certainly arise from a binary operation with one H operand. So BinOpH below says that if *either* operand is H, then the result is as well.

$$\text{(BinOpL)}\;\dfrac{\Gamma\vdash e:\mathtt{L}\quad \Gamma\vdash\tilde{e}:\mathtt{L}}{\Gamma\vdash e\odot\tilde{e}:\mathtt{L}}\qquad \text{(BinOpH)}\;\dfrac{\Gamma\vdash e:\ell_1\quad \Gamma\vdash\tilde{e}:\ell_2\quad \ell_1=\mathtt{H}\text{ or }\ell_2=\mathtt{H}}{\Gamma\vdash e\odot\tilde{e}:\mathtt{H}}$$

That pretty much does it for assigning security types to the expressions in our language. Let's see a short example of the rules in action to get a better feel for them. Suppose that we want to check that the expression $x+1\le y$ has type H in the policy $\Gamma=(x:\mathtt{H},y:\mathtt{L})$. To do so, we must construct a proof of the judgement in Equation 13.

$$x:\mathtt{H},y:\mathtt{L}\vdash x+1\le y:\mathtt{H} \tag{13}$$

We construct a proof using the typing rules that we have discussed so far.

$$\text{BinOpH}\;\dfrac{\text{BinOpH}\;\dfrac{\text{Var}\;\dfrac{*}{x:\mathtt{H},y:\mathtt{L}\vdash x:\mathtt{H}}\quad \text{ConstL}\;\dfrac{*}{x:\mathtt{H},y:\mathtt{L}\vdash 1:\mathtt{L}}}{x:\mathtt{H},y:\mathtt{L}\vdash x+1:\mathtt{H}}\quad \text{Var}\;\dfrac{*}{x:\mathtt{H},y:\mathtt{L}\vdash y:\mathtt{L}}}{x:\mathtt{H},y:\mathtt{L}\vdash x+1\le y:\mathtt{H}}$$

3.2 Generalizing security labels

Looking at BinOpL and BinOpH, we might ask ourselves whether it is possible to express all of the necessary logic in a single rule for binary operations. For example, our intuition about "high" and "low" security suggests some structure that might be useful in reasoning about these cases uniformly. Perhaps we can define an ordering on the security labels, denoted by $\sqsubseteq$. The natural way to order elements would be as shown in Equation 14.

$$\mathtt{L}\sqsubseteq\mathtt{H},\mathtt{L}\sqsubseteq\mathtt{L},\ \text{and}\ \mathtt{H}\sqsubseteq\mathtt{H},\ \text{but}\ \mathtt{H}\not\sqsubseteq\mathtt{L} \tag{14}$$

We might build on this ordering, the define a notion of "least upper bound", which given two security labels ℓ_1,ℓ_2 corresponds to the label that is at least as "large" as both (according to $\sqsubseteq$), but equal to one of them. We denote this operation $\sqcup$, and reason that it satisfies the equalities in Equation 15.

$$\begin{aligned}\mathtt{L}\sqcup\mathtt{L}\ &=\mathtt{L}\\ \mathtt{L}\sqcup\mathtt{H}\ &=\mathtt{H}\\ \mathtt{H}\sqcup\mathtt{L}\ &=\mathtt{H}\\ \mathtt{H}\sqcup\mathtt{H}\ &=\mathtt{H}\end{aligned} \tag{15}$$

Then with this in mind, the rules BinOpL and BinOpH can be combined into one BinOp rule shown below.

$$\text{(BinOp)}\;\dfrac{\Gamma\vdash e:\ell_1\quad \Gamma\vdash\tilde{e}:\ell_2}{\Gamma\vdash e\odot\tilde{e}:\ell_1\sqcup\ell_2}$$

We will prove that this is correct later, but for now convince yourself that it is not completely broken by replacing the BinOpH rules in the proof above with BinOp, and verifying that the result stays the same.

Now that we have thought about some general concepts related to our security labels, perhaps we can define even more labels to denote different levels of security. For example, instead of just $\mathtt{L}$ and $\mathtt{H}$, perhaps we also want a $\mathtt{M}$ type so that we can label data as not-quite of the utmost secrecy or importance, but not available to the entire world as we might imagine $\mathtt{L}$ to be. We could define the ordering in Equation 16, assuming all of the reflexive relations $\ell \sqsubseteq \ell$ hold and nothing else.

$$\mathtt{L} \sqsubseteq \mathtt{M} \sqsubseteq \mathtt{H} \tag{16}$$

Of course, we also want the ordering to be transitive, so that we can conclude $\mathtt{L} \sqsubseteq \mathtt{H}$ because $\mathtt{L} \sqsubseteq \mathtt{M}$ and $\mathtt{L} \sqsubseteq \mathtt{H}$.

Alternatively, we could define a set of labels $\mathtt{U}_1, \ldots, \mathtt{U}_n$ for users 1 through n, and let $\mathtt{H}$ be the system administrator's label and $\mathtt{L}$ be a "guest" label for temporary untrusted system users. Then we may want the ordering

$$\text{For all } 1 \le i \le n, \mathtt{L} \sqsubseteq \mathtt{U}_i \sqsubseteq \mathtt{H} \tag{17}$$

So any user can read "guest" data, and the administrator can read any user's data as well as "guest". But we don't want users to read each other's data, so for any i, j we want $\mathtt{U}_i \not\sqsubseteq \mathtt{U}_j$.

We can generalize our classification labels to consist of *security lattices* $(L, \sqsubseteq, \sqcup, \bot, \top)$. The components of a security lattice are as follows.

- A set of elements L. In our examples, $L = \{\mathtt{L}, \mathtt{H}\}$, $L = \{\mathtt{L}, \mathtt{M}, \mathtt{H}\}$, and $L = \{\mathtt{L}, \mathtt{U}_1, \ldots, \mathtt{U}_n, \mathtt{H}\}$.
- A *partial order* $\sqsubseteq$ between elements of L, which is a reflexive ($\forall \ell . \ell \sqsubseteq \ell$), transitive ($\forall \ell_1, \ell_2, \ell_3 . \ell_1 \sqsubseteq \ell_2 \land \ell_2 \sqsubseteq \ell_3 \to \ell_1 \sqsubseteq \ell_3$), and anti-symmetric ($\forall \ell_1, \ell_2 . \ell_1 \sqsubseteq \ell_2 \land \ell_2 \sqsubseteq \ell_1 \to \ell_1 = \ell_2$) relation between elements of L.
- A least upper-bound operator $\sqcup : L \times L \to L$ defined on all pairs from L.
- A *least element* $\bot$ such that $\bot \sqsubseteq \ell$ for all $\ell \in L$.
- A *greatest element* $\top$ such that $\ell \sqsubseteq \top$ for all $\ell \in L$.

The labels that we started out with, $L = \{\mathtt{L}, \mathtt{H}\}$ indeed comprise a lattice with $\bot = \mathtt{L}$ and $\top = \mathtt{H}$. It should come as no surprise that we can generalize the definition of non-interference to arbitrary security lattices, and doing so is an excellent exercise.

3.3 Type-checking commands

Now that we know how to assign types to expressions, we can move on to the more interesting question of how to check that a given command satisfies non-interference under a policy Γ. We will proceed as before, developing a set of rules that we can later prove give us this property. The judgements that the rules derive is slightly different than with expressions, though. Keeping in mind that the security types L, H denote the fact that a given data element, e.g. an expression in our language, carries information of a particular security level, it does not make sense to write something like the following:

$$\Gamma \vdash \texttt{if}(x \le 0)\, y := 1\, \texttt{else}\, y := 0 : \texttt{H}$$

We have not thought of programs as any sort of data element capable of carrying information by themselves, and however intriguing such an idea may seem, doing so would distract us from our goal of enforcing non-interference at the moment.

Instead, we have thought of programs as objects that compute, i.e., by applying operations to data and moving it between variables. Our goal is to make it impossible to use the typing rules to construct a proof that a program satisfies non-interference when it doesn't. So the judgements that our rules for commands will use takes the form shown in Equation 18, and should be read as "under policy Γ, program α is well-typed". We will set up the rules so that "well-typed" implies "satisfies non-interference", but we'll say more on this later.

$$\Gamma \vdash \alpha \tag{18}$$

Now we proceed to design typing rules for each of the commands in our language.

Assignments. To design a typing rule for assignment commands, we must ask ourselves what conditions might result in a violation of Definition 1. More intuitively, what conditions on Γ and the command $x := e$ would result in a flow of information from variables labeled H to those labeled L?

Certainly, if the target of the assignment x is labeled L and the expression on the right-hand side is labeled H, then such a flow will occur. More generally, if $\Gamma(x) = \ell_1$ and $\Gamma \vdash e : \ell_2$, where $\ell_2 \not\sqsubseteq \ell_2$, then an information flow will occur when the value of e is assigned to x. Perhaps we can write the following rule, shown in Equation 19.

$$\frac{\Gamma \vdash e : \ell_1 \quad \ell_1 \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e} \tag{19}$$

This rule says that if the label of e as no larger (i.e., "more secret") than that of x, then the assignment is well-typed. Does this work? What about the "implicit" information flows we discussed last lecture, such as that in the following judgement?

$$x : \texttt{H}, y : \texttt{L} \vdash \texttt{if}(x)\, y := 1\, \texttt{else}\, y := 0 \tag{20}$$

We know that this doesn't satisfy non-interference, but it seems as though we might not reject it using the rule shown in (19).

In order to reject programs with implicit flows, we will keep track of the security label of a distinguished "program counter" variable pc in Γ. Later when we design rules for conditionals and loops, we will make sure to account for this part of the policy so that whenever the control flow is currently influenced by H-labeled data, then $\Gamma(\texttt{pc}) = \texttt{H}$.

But for now we will just assume that this has been properly accounted for in the policy, and use it to define our rule for assignment.

$$\text{(Asgn)}\ \frac{\Gamma \vdash e : \ell_1 \quad \ell_1 \sqcup \Gamma(\texttt{pc}) \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e}$$

The rule Asgn says that if the label of x is at least as large as the larger of the label for the right-hand side and $\texttt{pc}$, then the assignment is well-typed. Thinking through a few cases, this means that whenever $\Gamma(\texttt{pc}) = \texttt{H}$, then $\Gamma(x)$ must be $\texttt{L}$ because $\texttt{H} \sqcup \ell = \texttt{H}$ and $\texttt{H} \not\sqsubseteq \texttt{L}$. Really, if either of e or $\texttt{pc}$ is $\texttt{H}$, then in order for the assignment to be well-typed then x must also be $\texttt{H}$. This seems like what we want, so we move to the next command.

Composition. Compared to the surprisingly nuanced reasoning we had to do for assignment commands, composition is relatively easy to think about. If we want to reason that a program $\alpha;\beta$ is well-typed, then the only rule that makes any sense is to require that both α and β also be well-typed on their own.

$$\text{(Comp)}\ \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha;\beta}$$

The rule Comp above says exactly this.

Conditionals. Now we come to conditionals. Unlike the previous cases, conditionals raise the possibility of influencing control flow (represented in our type system by $\Gamma(\texttt{pc})$) on $\texttt{H}$-labeled data. We need to account for this in the typing rule, so that when we reason about whether the branches are well-typed we properly account for whether the $\texttt{pc}$ might carry $\texttt{H}$ data.

Recall that we treat policies similarly to updateable maps, so that if Γ is a policy, possibly containing a mapping for x, then $(\Gamma, x : \texttt{L})$ is the policy that maps x to $\texttt{L}$ and everything else y to $\Gamma(y)$. Then to carry the type of the Boolean expression Q in the guard of a conditional to the contexts of its branches, we want to use a context that maps $\texttt{pc}$ to the least upper bound of the current $\texttt{pc}$, and the type of e.

$$\text{(If)}\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\texttt{pc}) \quad \Gamma, \texttt{pc} : \ell' \vdash \alpha \quad \Gamma, \texttt{pc} : \ell' \vdash \beta}{\Gamma \vdash \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta}$$

The rule If above does this. Looking at the antecedent, we first type the guard expression Q as ℓ, and then compute the least upper bound ℓ' of ℓ and the current label of $\texttt{pc}$. We then check that both branches are well-typed under the policy where $\Gamma(pc) = \ell'$. If this is the case, then the conditional is well-typed.

While Loops. The last command that we need to derive a rule for is our looping construct $\texttt{while}$. Much like in the case of conditionals, while loops can leak information from $\texttt{H}$ data to the program counter through their conditional test. Not surprisingly, the rule for while loops can use the same approach as that for conditionals, as shown in While below.

$$\text{(While)}\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\texttt{pc}) \quad \Gamma, \texttt{pc} : \ell' \vdash \alpha}{\Gamma \vdash \texttt{while}(Q)\, \alpha}$$

It may seem strange that the rule for `while` is in some ways simpler than the one for `if` statements, as this was certainly not the case when we discussed axioms for proving safety. However, the type of reasoning that this type system does about program behaviors is significantly more "coarse" than what we did when proving arbitrary postconditions, and in this case the only special consideration that we need to account for is whether the loop flows `H` data to `L` state through the program counter.

References

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.