Notes on Information Flow Control

CS 5430

May 8, 2017

Enforcing Information Flow Policies

The goal of *information flow control* is to enforce IF policies associated with variables in a program. Assume there is a mapping Γ from variables to labels, which represent desired IF policies. The enforcement mechanism should ensure that a program and the accompanied mapping Γ satisfy noninterference.

For these notes, we consider the following definition of noninterference for confidentiality:

$$\text{if } M_1 =_L M_2, \text{ then } C(M_1) =_L C(M_2).$$

where label L tags *public* variables (i.e., their values are allowed to flow to everyone) and H tags *secret* variables (i.e., their values are allowed to flow only to some principals). According to the above statement of noninterference, when executing program C twice, by keeping the same initial values of public variables and possibly changing the initial values of secret variables, the values stored in public variables at termination should not change.

Fixed versus Flow-sensitive Γ

An enforcement mechanism for IF policies may use a *fixed* or a *flow-sensitive* mapping Γ for the analysis of a program. Using a fixed Γ means that labels on variables remain always the same during analysis. The enforcement mechanism *checks* whether the program and the particular Γ satisfy noninterference. For example, consider $\Gamma(y) = H$, $\Gamma(x) = L$ and assignment

$$x := y. \tag{1}$$

The mechanism would check whether the particular Γ and the particular assignment satisfies noninterference. The mechanism would deduce that noninterference is actually not satisfied, and thus the assignment would be *rejected*. If, instead, $\Gamma(y) = L$ and $\Gamma(x) = H$, then the assignment would be *accepted* by the mechanism, because now noninterference is satisfied.

If an enforcement mechanism uses a flow-sensitive mapping Γ, then Γ may change during the analysis of a program. This means that labels on variables may change during analysis. Here, the enforcement mechanism *deduces* labels

$$
\begin{array}{llll}
\textit{(Expressions)} & e & ::= & n \mid x \mid e_1 + e_2 \\
\textit{(Commands)} & c & ::= & \textbf{skip} \mid x := e \mid c_1; c_2 \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ end} \mid \\
 & & & \textbf{while } e \textbf{ do } c \textbf{ end}
\end{array}
$$

Figure 1: Syntax

on variables, such that the program and mapping Γ satisfy noninterference. Consider again assignment (1). At the beginning of the analysis we may have $\Gamma(x) = H$ and $\Gamma(y) = L$, but after analyzing this assignment, the mechanism would set $\Gamma(y)$ to be H. Thus, noninterference is satisfied.

Static versus Dynamic mechanisms

The enforcement mechanism for information flow control may be applied to programs before or after execution. A *static* mechanism performs checking and/or deduction of labels *before* execution. A *dynamic* mechanism performs checking and/or deduction of labels *during* execution. There are also *hybrid* mechanisms that combine techniques from static and dynamic mechanism to achieve the best of both worlds.

A static mechanism with fixed Γ

We examine a static mechanism for information flow control, which uses a fixed mapping Γ. So, the mechanism only needs to check whether a given program and a given mapping Γ satisfies noninterference.

Programs are written in a simple imperative language, whose syntax is presented in Figure 1. According to this syntax, an expression e is either a constant n, or a variable x, or the application of an operator to expressions $e_1 + e_2$. A command c is either a **skip**, which has no effect, or an assignment, or a sequence of commands, or an "if"-statement, or a "while"-statement. Next, we examine how the enforcement mechanism decides whether a command and a mapping Γ satisfy noninterference.

Consider first the assignment below:

$$x := y. \tag{2}$$

Here the value in y *explicitly* flows to x. Whoever learns the value of x, they also learn the value of y. So, the restrictions imposed by $\Gamma(x)$ on where x is allowed to flow had better be at least as many as those imposed by $\Gamma(y)$.

According to noninterference, label H imposes more restrictions than L, because variables tagged with H are allowed to flow only to variables tagged with H, however variables tagged with L are allowed to flow to any variable. So, the mechanism accepts assignment (2) if $\Gamma(x) = H$ and $\Gamma(y) = L$, or if $\Gamma(x) = H$

and $\Gamma(y) = H$, or if $\Gamma(x) = L$ and $\Gamma(y) = L$. However, the assignment is rejected if $\Gamma(x) = L$ and $\Gamma(y) = H$.

We assume there is a *restrictiveness* relation $\sqsubseteq$ that compares labels in terms of the restrictions they impose. We write $\ell \sqsubseteq \ell'$ to denote that ℓ' is *at least as restrictive as* ℓ. Relation $\sqsubseteq$ should be: reflexive, transitive, and antisymmetric. There is a *bottom* label $\bot$ that is less restrictive than all other labels, and a *top* label $\top$ that is more restrictive than all other labels.

Restrictiveness relation $\sqsubseteq$ essentially define allowed flows between labels. If $\ell \sqsubseteq \ell'$, then values in variables tagged with ℓ are allowed to flow to variables tagged with ℓ'. For the set $\{L, H\}$ of confidentiality labels, relation $\sqsubseteq$ is defined to be:

$$L \sqsubseteq L,\ L \sqsubseteq H,\ H \sqsubseteq H. \tag{3}$$

Notice that $L \sqsubseteq H$ is in accordance with the confidentiality policies represented by labels L and H, because values in public variables (i.e., tagged with L) may flow to secret variables (i.e., tagged with H). According to (3), L is the bottom label $\bot$ and H is the top label $\top$.

The static mechanism accepts assignment (2), if $\Gamma(y) \sqsubseteq \Gamma(x)$ holds. Consider now assignment $x := y + z$ that causes both y and z to explicitly flow to x. This assignment is accepted if $\Gamma(y) \sqsubseteq \Gamma(x)$ and $\Gamma(z) \sqsubseteq \Gamma(x)$ hold. So, $\Gamma(x)$ should be at least as restrictive as both $\Gamma(y)$ and $\Gamma(z)$.

For each pair of labels ℓ and ℓ', there should exist label $\ell \sqcup \ell'$, such that:

- $\ell \sqcup \ell'$ is at least as restrictive as both ℓ and ℓ' (i.e., $\ell \sqsubseteq \ell \sqcup \ell'$, $\ell' \sqsubseteq \ell \sqcup \ell'$), and
- there is no other such label ℓ'' that is less restrictive than $\ell \sqcup \ell'$ (i.e., if $\ell \sqsubseteq \ell''$ and $\ell' \sqsubseteq \ell''$, then $\ell \sqcup \ell' \sqsubseteq \ell''$).

Label $\ell \sqcup \ell'$ is then called the *join* of ℓ and ℓ'. Operator $\sqcup$ is associative and commutative. The set of labels and relation $\sqsubseteq$ define a *lattice*, with join operator $\sqcup$.

For example, the set $\{L, H\}$ of confidentiality labels and relation $\sqsubseteq$ is a lattice, where the join operator $\sqcup$ is defined as:

$$L \sqcup L = L,\ L \sqcup H = H,\ H \sqcup H = H.$$

Notice that equality $L \sqcup H = H$ is in accordance with the confidentiality policies represented by labels L and H, because the combination of a public (i.e., tagged with L) and a secret value (i.e., tagged with H) can be safely considered as a secret value.

So, assignment $x := y + z$ is accepted if $\Gamma(y) \sqcup \Gamma(z) \sqsubseteq \Gamma(x)$. Defining $\Gamma(y + z)$ to be $\Gamma(y) \sqcup \Gamma(z)$, we can simply write $\Gamma(y + z) \sqsubseteq \Gamma(x)$.

Consider, now, the *if*-statement below:

$$\begin{array}{l} \textbf{if } z > 0 \textbf{ then} \\ \quad \textbf{if } y > 0 \textbf{ then } x := 1 \textbf{ else } x := 2 \textbf{ end} \\ \textbf{else} \\ \quad x := 3 \\ \textbf{end} \end{array} \tag{4}$$

Here, values in z and y *implicitly* flow to x. This is because the value of x indicates the truth values of *guards* $z > 0$ and $y > 0$. For example, if x is 2, then it can be deduced that $z > 0$ is *true* and $y > 0$ is *false*. However, if x is 3, then it can be deduced that $z > 0$ is *false*, but nothing can be deduced about the truth value of $y > 0$. So, assignments $x := 1$ and $x := 2$ reveal information about both guards $z > 0$ and $y > 0$, while assignment $x := 3$ reveals information only about $z > 0$. Thus, we say that the execution of assignments $x := 1$ and $x := 2$ is *controlled* by guards $z > 0$ and $y > 0$, while the execution of assignment $x := 3$ is controlled by only guard $z > 0$.

The set of guards that control the execution of a command is called the *context* of that command. Because the execution of a command may reveal information about its context (e.g., assignment $x := 1$ in (4) reveals information about guards $z > 0$ and $y > 0$), the enforcement mechanism uses a *context label* *ctx* to represent the sensitivity of the information conveyed by a context. In example (4), we saw that the context of $x := 1$ involves guards $z > 0$ and $y > 0$. The context label *ctx* that represents the sensitivity of $z > 0$ and $y > 0$ is the combination of the sensitivity of z and the sensitivity of y. So, the context label *ctx* for $x := 1$ in (4) is $\Gamma(z) \sqcup \Gamma(y)$. The label $\Gamma(x)$ of x had better be at least as restrictive as *ctx*, otherwise information could be implicitly leaked from the context to x. For example, if $\Gamma(x) = H$, $\Gamma(y) = L$, and $\Gamma(z) = H$, then the program is accepted. However, if $\Gamma(x) = L$, $\Gamma(y) = L$, and $\Gamma(z) = H$, then the program is rejected.

Up until now we examined how the static enforcement mechanism, with fixed Γ, analyses particular commands. Next, we define this enforcement mechanism as a typing system and explain how it can analyze any possible command. This typing system addresses explicit and implicit flows using the techniques introduced above.

Typing system

We employ a static type system to enforce noninterference. Here types are labels. There is a fixed mapping Γ from variables to types (i.e., labels). The typing system consists of typing rules for

- deducing types for expressions, given types of variables in these expressions,
- deciding whether each command in a program is type correct.

If a program is type correct according to the typing rules, then it is proved that the program satisfies noninterference.

Typing rules for expressions

Typing rules for expressions use judgment $\Gamma \vdash e : \ell$ to denote that expression e has type ℓ according to mapping Γ. We give one typing rule for each possible expression that may occur in a program, given syntax in Figure 1.

No information about secret variables is revealed by a **constant** n, because its value remains the same for all possible execution of a program. So, a constant can be safely tagged with bottom label $\bot$, which equals to L when considering confidentiality labels $\{L, H\}$. The typing rule for constants is:

$$\Gamma \vdash n : \bot.$$

The type of a **variable** is the label that Γ maps this variable to:

$$\Gamma \vdash x : \Gamma(x).$$

The type of an **expression** $e+e'$ should be at least as restrictive as the type of e and the type of e'. So, it suffices for the type of $e+e'$ to be the join of the type of e and the type of e':

$$\Gamma \vdash e + e' : \Gamma(e) \sqcup \Gamma(e').$$

Typing rules for commands

Typing rules for commands use judgment $\Gamma, \mathit{ctx} \vdash c$ to denote that according to mapping Γ and context label ctx, command c is type correct.

We give one typing rule for each possible kind of command that may occur in a program, given syntax in Figure 1.

An **assignment** $x := e$ is type correct if the explicit flow from e to x and the implicit flow from the context of that assignment to x are allowed. In particular, $\Gamma(x)$ should be at least as restrictive as $\Gamma(e)$ (to prevent explicit flows) and at least as restrictive as ctx (to prevent implicit flows). So, we write:

$$\begin{aligned} &\Gamma, \mathit{ctx} \vdash x := e \\ &\qquad \text{if } \Gamma \vdash e : \ell \\ &\qquad \text{and } \ell \sqcup \mathit{ctx} \sqsubseteq \Gamma(x) \end{aligned}$$

We use the *inference rule* below to represent the above statement:

$$\frac{\Gamma \vdash e : \ell \quad \ell \sqcup \mathit{ctx} \sqsubseteq \Gamma(x)}{\Gamma, \mathit{ctx} \vdash x := e}$$

Here, the judgments above the line are called the *premises* of the inference rule, and the judgment below the line is called the *conclusion* of the inference rule.

The typing rule for **"if"**-statement is responsible for constructing the correct context label under which the branches of this statement should be type checked. In particular, statement **if** e **then** c **else** c' **end** is type correct if c and c' are type correct in a context label augmented with the type of e:

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup \mathit{ctx} \vdash c \quad \Gamma, \ell \sqcup \mathit{ctx} \vdash c'}{\Gamma, \mathit{ctx} \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' \textbf{ end}}$$

A **"while"**-statement **while** e **do** c **end** is type correct if c is type correct in a context augmented with e:

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup \mathit{ctx} \vdash c}{\Gamma, \mathit{ctx} \vdash \textbf{while } e \textbf{ do } c \textbf{ end}}$$

A **sequence** statement $c; c'$ is type correct if c and c' are type correct:

$$\frac{\Gamma, \mathit{ctx} \vdash c \quad \Gamma, \mathit{ctx} \vdash c'}{\Gamma, \mathit{ctx} \vdash c; c'}$$

This typing system can be used to enforce labels from an arbitrary lattice (non just H and L labels), for either confidentiality or integrity.

Example

Consider the program below and a static mapping Γ from variables to labels.

$$\textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end};\ y := z. \tag{5}$$

We follow the typing rules introduced above to deduce restrictions between labels $\Gamma(x)$, $\Gamma(y)$, $\Gamma(z)$, such that the program is type correct. If Γ satisfies these restrictions between labels, then program (5) is type correct. Otherwise, program (5) is not type correct.

The context of program (5) is empty, because no guard controls the execution of (5). So, the context label ctx for (5) can be set to the bottom label $\bot$. We want to deduce the relation between labels $\Gamma(x)$, $\Gamma(y)$, $\Gamma(z)$ such that we can prove the following judgment:

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end};\ y := z. \tag{6}$$

According to the typing rule of sequence statement, (6) can be proved if the following hold:

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end} \tag{7}$$
$$\Gamma, \bot \vdash y := z. \tag{8}$$

According to the typing rule of assignment, (8) can be proved if:

$$\Gamma(z) \sqcup \bot \sqsubseteq \Gamma(y)$$

which can be rewritten as:

$$\Gamma(z) \sqsubseteq \Gamma(y) \tag{9}$$

because $\Gamma(z) \sqcup \bot = \Gamma(z)$.

From the typing rule of "if"-statement, (7) can be proved if:

$$\begin{aligned} &\Gamma \vdash x > 0 : \Gamma(x) \\ &\Gamma, \bot \sqcup \Gamma(x) \vdash z := 1 \\ &\Gamma, \bot \sqcup \Gamma(x) \vdash z := 2 \end{aligned}$$

where the last two judgments can be rewritten as:

$$\Gamma, \Gamma(x) \vdash z := 1 \tag{10}$$

$$\Gamma, \Gamma(x) \vdash z := 2. \tag{11}$$

According to the typing rule of assignment, (10) and (11) can be proved if:

$$\Gamma(x) \sqsubseteq \Gamma(z). \tag{12}$$

So, program (5) is type correct, under the bottom context label $\bot$, if restrictions (9) and (12) hold between labels $\Gamma(x)$, $\Gamma(y)$, and $\Gamma(z)$. For instance, if $\Gamma(x) = L$, $\Gamma(y) = H$, and $\Gamma(z) = L$, then restrictions (9) and (12) hold. However, if $\Gamma(x) = L$, $\Gamma(y) = L$, and $\Gamma(z) = H$, then restriction (9) does not hold, and thus program (5) is not type correct.

Table 1 summarizes the deduction steps we followed above as a proof tree. At the bottom of the proof tree is the judgment that needs to be proved. At the top of the proof tree reside the restrictions that need to hold between labels $\Gamma(x)$, $\Gamma(y)$, and $\Gamma(z)$.

$$\dfrac{\dfrac{\Gamma \vdash x > 0 : \Gamma(x) \quad \dfrac{\Gamma(x) \sqsubseteq \Gamma(z)}{\Gamma, \Gamma(x) \vdash z := 1} \quad \dfrac{\Gamma(x) \sqsubseteq \Gamma(z)}{\Gamma, \Gamma(x) \vdash z := 2}}{\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}} \quad \dfrac{\Gamma(z) \sqsubseteq \Gamma(y)}{\Gamma, \bot \vdash y := z}}{\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end};\ y := z}$$

Table 1: Proof tree

Exercise

Decide whether the following programs are type correct, given the corresponding mapping Γ.

Program 1:

$$\textbf{while } b \neq 0 \textbf{ do } t := b;\ b := a \ mod\ b;\ a := t \textbf{ end}$$

where $\Gamma(a) = \Gamma(b) = \Gamma(t) = H$.

Program 2:

$$\textbf{if } l1 = l2 \textbf{ then } l3 := 1 \textbf{ else } h := 0 \textbf{ end}$$

where $\Gamma(l1) = \Gamma(l2) = \Gamma(l3) = L$ and $\Gamma(h) = H$.

Program 3:

$$\textbf{if } l = h \textbf{ then } l := 0 \textbf{ else skip end}$$

where $\Gamma(l) = L$ and $\Gamma(h) = H$.