Notes on Information Flow Control for CS 5430

Elisavet Kozyri

January 5, 2019

Restrictions and Access Control

Data are associated with *restrictions*. These restrictions are usually expressed in terms of confidentiality (e.g., who can read data), in terms of integrity (e.g., how much trusted data is), or in terms of privacy (e.g., what operations can be applied on data).

Access control has been widely used to specify and enforce restrictions on data. However, access control alone is not enough. Consider a document *doc* associated with an access control policy P (e.g., only Alice can read *doc*). Assume that some computation is applied to *doc* producing several new documents that needs to be used later. What should be the access control policy on these new documents? A human should make this decision and should manually associate the desired policies to the new documents.

This manual work becomes even harder, when we scale up to a system that stores multiple pieces of data that are owned by multiple users, and supports rich interactions between data of different users. So, access control alone does not seem suitable for the *Big Data* era. Also, access control alone cannot prevent leaking information through metadata, shared resources, and other *covert channels*, which are channel not intended to convey information.

Information Flow Policies

Information Flow (IF) policies are proposed to address the limitations of access control. An IF policy associated with a piece of data d specifies restrictions on d, and on all data derived from d. For example, an IF policy for confidentiality could specify: value v and all its derived values is allowed to be read at most by Alice. Equivalently, we say that v is allowed *to flow* only to Alice.

An enforcement mechanism for IF policies automatically deduces the restrictions for derived data. For example, if a document *doc* is allowed to flow only to Alice, then any document derived from *doc* is allowed to flow only to Alice, too. Otherwise, the IF policy on *doc* would be violated. When documents with different policies are combined to produce a new document, that new document is associated with a policy that satisfies all policies of the constituent documents.

Labels for IF policies

We use *labels*, which are syntactic objects, to represent IF policies. *Classifications* can be used as labels for information flow policies. For example, labels U, C, S, and TS (i.e., Unclassified, Confidential, Secret, and Top Secret) or labels L and H (i.e., Low confidentiality, High confidentiality) can be used to represent IF policies. If some document *doc* is associated with C, it means that *doc* and all documents derived from *doc* are considered Confidential (i.e., tagged with C). *Sets of principals* can also be used as labels. For example, if some document *doc* is associated with $\{Alice\}$, it means that *doc* and all documents derived from *doc* can be read at most by Alice.

More expressive labels

The labels discussed above represent policies that are more restrictive than necessary, because they impose the same restrictions to all possible derived data. However, there are practical cases where restrictions on outputs of some operations are different (e.g., fewer or more) than the restrictions on inputs of these operations.

Consider, for example, operation *maj* that tallies votes for an election:

$$x := maj(\nu_1, \nu_2, \ldots, \nu_n).$$

Input plaintexts are usually considered *secret* (i.e., of high confidentiality), but the result x of the election is usually considered *public* (i.e., of low confidentiality). If each vote ν_i is tagged with label H (which represents that ν_i is of high confidentiality), then the output x, which is derived from all these votes, is required to be tagged with H, too. But this means that x cannot be considered public. So, there is a mismatch between the restrictions imposed by the labels of inputs to the output, and the desired restrictions to the output.

Similarly, for the encryption operation:

$$x := Enc(y; k).$$

Input votes are usually considered secret, but the ciphertext x is usually considered public. However, tagging y and k with H, implies that x is tagged with H. Again, there is a mismatch between the imposed and the desired restrictions on x.

Other operations may cause the restrictions imposed on outputs to increase comparing to restrictions imposed on inputs. For example, the list of students in CS and the list of addressed in Ithaca may be public, but the mapping of students to home addresses should not be public. So, here, if the list of students in CS and the list of addresses in Ithaca are tagged with L, then the resulting list will be tagged with L. Thus, fewer restrictions than desired will be applied to the resulting list.

So, there is a need for IF policies and labels to express how restrictions on derived data may change based on applied operations, or based on events that occur during execution, or based on ownership of this data.

Noninterference

Consider inputs and outputs of a program being tagged with label H or L. Inputs tagged with H are allowed to flow only to outputs tagged with H. Equivalently, inputs tagged with H are not allowed to flow to outputs tagged with L. This implies that *changing inputs tagged with H should not cause changes on outputs tagged with L.* This requirement is an instantiation of *noninterference*. Noninterference is a *semantic* guarantee that should be offered by the enforcement mechanism of IF policies. Access control does *not* offer a similar semantic guarantee.

Consider, for example, the program below

$$h' := h + l;\ l' := l + 1$$

where variables h, h' are tagged with H and variables l, l' are tagged with L. Here, l and h model inputs, while l' and h' model outputs. This program satisfies noninterference because changing values in h does not cause values in l' to change. However, program

$$l' := h * 2$$

does not satisfy noninterference because changing h causes l' to change. So, here h is *leaked* to l'.

More carefully, noninterference states that if two initial (or input) memories M_1, M_2 agree on variables tagged with L (i.e., $M_1 =_L M_2$), and if program C is executed on M_1 and M_2 to termination, then the corresponding outputs $C(M_1)$ and $C(M_2)$ should also agree on variables tagged with L (i.e., $C(M_1) =_L C(M_2)$). Specifically:

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

The above statement of noninterference handles only programs that terminate. What if a program does not terminate depending on inputs tagged with H? Consider the following example:

$$\begin{array}{l} \textbf{while } h > 5 \textbf{ do } \{\textbf{skip}\}; \\ l' := 4 \end{array} \tag{1}$$

where command **skip** does nothing. If $h > 5$ is *false*, then l' becomes 4. If $h > 5$ is *true*, then no value is assigned to l'. Principals observing l' either observe 4 being assigned to l', or no value being assigned to l', depending on $h > 5$. So, $h > 5$ is leaked to principals observing l'.

Termination sensitive noninterference strengthens noninterference by requiring the termination behavior of the problem to not depend on secret values:

If $M_1 =_L M_2$, then

C terminates on M_1 iff C terminates on M_2 and

$C(M_1) =_L C(M_2)$.

Program (1) does not satisfy termination sensitive noninterference, but the following program satisfies termination sensitive noninterference:

$$\textbf{while } l > 5 \textbf{ do } \{\textbf{skip}\};$$
$$l' := 4$$

When information flow policies relax restrictions on derived data, and thus allow leaking information to the output of a certain operation, the previous statements of noninterference do not hold. Consider, for example, an information flow policy that allows an input h tagged with H to flow to an output l' tagged with L only through the operation *mod* 2. According to that policy principals observing l' are allowed to learn whether h is even or not, but they are not allowed to learn anything else about h. So, program

$$l' := h \ mod \ 2 \tag{2}$$

satisfies this policy, but program

$$l' := h \ mod \ 2 + h \tag{3}$$

does not, because this last program leaks more information to l' than just h *mod* 2. For this particular policy, noninterference should be restated as:

If $M_1 =_L M_2$ and $M_1(h) \ mod \ 2 = M_2(h) \ mod \ 2$, then $C(M_1) =_L C(M_2)$.

Notice that this statement of noninterference is satisfied by program (2), but it is not satisfied by (3).

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

$$x := y. \tag{4}$$

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 on variables, such that the program and mapping Γ satisfy noninterference. Consider again assignment (4). 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.

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

Figure 1: Syntax

Consider first the assignment below:

$$x := y. \tag{5}$$

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 (5) 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:

$$L \sqsubseteq L,\ L \sqsubseteq H,\ H \sqsubseteq H. \tag{6}$$

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 (6), L is the bottom label $\bot$ and H is the top label $\top$.

The static mechanism accepts assignment (5), 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:

```
if z > 0 then
   if y > 0 then x := 1 else x := 2 end
else                                          (7)
   x := 3
end
```

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 (7) 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 (7), 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 (7) 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 $\Gamma, ctx \vdash c$ 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:

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

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

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

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:

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

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

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

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

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

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.

$$\textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}; \; y := z. \tag{8}$$

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 (8) is type correct. Otherwise, program (8) is not type correct.

The context of program (8) is empty, because no guard controls the execution of (8). So, the context label ctx for (8) 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:

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}; \; y := z. \tag{9}$$

According to the typing rule of sequence statement, (9) can be proved if the following hold:

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end} \tag{10}$$

$$\Gamma, \bot \vdash y := z. \tag{11}$$

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

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

which can be rewritten as:

$$\Gamma(z) \sqsubseteq \Gamma(y) \tag{12}$$

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

From the typing rule of "if"-statement, (10) can be proved if:

$$\Gamma \vdash x > 0 : \Gamma(x)$$

$$\Gamma, \bot \sqcup \Gamma(x) \vdash z := 1$$

$$\Gamma, \bot \sqcup \Gamma(x) \vdash z := 2$$

where the last two judgments can be rewritten as:

$$\Gamma, \Gamma(x) \vdash z := 1 \tag{13}$$

$$\Gamma, \Gamma(x) \vdash z := 2. \tag{14}$$

According to the typing rule of assignment, (13) and (14) can be proved if:

$$\Gamma(x) \sqsubseteq \Gamma(z). \tag{15}$$

So, program (8) is type correct, under the bottom context label $\bot$, if restrictions (12) and (15) 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 (12) and (15) hold. However, if $\Gamma(x) = L$, $\Gamma(y) = L$, and $\Gamma(z) = H$, then restriction (12) does not hold, and thus program (8) 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 $\Gamma(z)$.

Noninterference for any label ℓ

The static type system we introduced for information flow control is *sound*. This means that if a program is type correct, then the program satisfies noninterference. Equivalently, this type system does not accept programs that violate noninterference.

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

Table 1: Proof tree

Up until now we have defined noninterference in terms of a simple lattice, which consists of a set of labels $\{L, H\}$ and a restrictiveness relation $\sqsubseteq$. In practice, more than two labels are needed to represent desired information flow policies. So, now, we generalize the statement of noninterference to describe allowed flows between labels of an arbitrary lattice.

Given a lattice that consists of a set Λ of labels and restrictiveness relation $\sqsubseteq$, we express noninterference with respect to a label $\ell \in \Lambda$. Consider a principal who is allowed to read variables tagged with that label ℓ. Then, this principal is also allowed to read any variable tagged with a label in set $Low(\ell) = \{\ell' \mid \ell' \sqsubseteq \ell\}$. This is because, by the definition of relation $\sqsubseteq$, label ℓ' is allowed to flow to label ℓ. However, this principal should not be allowed to read any variable tagged with a label ℓ'' not in $Low(\ell)$, because ℓ'' is not allowed to flow to ℓ. So, each label ℓ' in $Low(\ell)$ is considered "low" with respect to ℓ and each label ℓ'' not in $Low(\ell)$ is considered "high" with respect to ℓ."High" inputs with respect to ℓ should not flow to "low" outputs with respect to ℓ. This requirement should hold for any label $\ell \in \Lambda$, so noninterference can be generalized as:

$$\forall \ell : \; M_1 =_\ell M_2 \Rightarrow c(M_1) =_\ell c(M_2), \tag{16}$$

where $M_1 =_\ell M_2$ denotes equality on all variables tagged with a label in $Low(\ell)$, and $c(M_1) =_\ell c(M_2)$ denotes equality on all outputs tagged with a label in $Low(\ell)$.

The static type system is sound with respect to statement (16) of noninterference. Statement (16) can be used for either confidentiality or integrity. In fact, statement (16) is oblivious to the exact meaning of labels and the policies they represent. Noninterference in (16) only depends on the restrictiveness relation $\sqsubseteq$ defined on the set Λ of labels. Consequently, the typing system can be used to enforce noninterference in (16) for any information flow policies, provided these policies are represented by labels where relation $\sqsubseteq$ and join operator $\sqcup$ are defined.

Limitations of the static typing system

Programs accepted by the static typing system may leak sensitive information through their termination behavior. Consider the program below:

$$\begin{array}{l} \textbf{while } s \neq 0 \textbf{ do skip end}; \\ p := 1 \end{array}$$

where s is a secret variable (i.e., $\Gamma(s) = H$), and p is a public variable (i.e., $\Gamma(p) = L$). The final value of p is the output of the program. Command **skip** has no effect.[1] If $s \neq 0$ is *true*, then the program does not terminate, and thus $p := 1$ is never executed. So, no public output is generated. However, if $s \neq 0$ is *false*, then the program terminates, by assigning 1 to p. So, one public output is generated. Consequently, the termination behavior of the program is used as a covert channel to leak $s \neq 0$ to public outputs!

The typing rule for the **while**-statement has to be strengthened, if leaking through termination needs to be prevented. For example, the typing rule could accept a **while**-statement only when the type of its guard expression is the bottom label $\bot$:

$$\frac{\Gamma(e) = \bot \quad \Gamma, ctx \vdash c}{\Gamma, ctx \vdash \textbf{while } e \textbf{ do } c \textbf{ end}} \tag{17}$$

Under this restriction, the termination behavior of the program does not depend on sensitive information, and thus the covert channel due to termination is avoided. However, the enforcement mechanism becomes overly *conservative*; more secure programs, such as

$$\textbf{while } s > 0 \textbf{ do } s := s + 1 \textbf{ end},$$

are now rejected. Researchers in information flow control either chose to ignore the covert channel due to termination, to avoid making their mechanism conservative, or try to find a more precise rule than (17).

The current static type system is already conservative enough, because there are programs that satisfy noninterference but they are not type correct. Consider, for example, the program below:

$$\textbf{if } x > 0 \textbf{ then } y := 1 \textbf{ else } y := 1 \textbf{ end}$$

where the confidentiality labels for variables are fixed: $\Gamma(x) = H$ and $\Gamma(y) = L$. This program satisfies noninterference, because x does not flow to y, but it is not type correct, because $\Gamma(x) \not\sqsubseteq \Gamma(y)$. Consider another example:

$$\textbf{if } 1 = 1 \textbf{ then } y := 1 \textbf{ else } y := x \textbf{ end} \tag{18}$$

with the same confidentiality labels as above. Again, this program satisfies noninterference, because x does not leak to y, but it is not type correct, because $\Gamma(x) \not\sqsubseteq \Gamma(y)$. These two programs are examples of *false positives* for the static type system: they satisfy noninterference but they are not type correct.

It is impossible to built an enforcement mechanism for information flow control that accepts exactly those programs that satisfy noninterference. This is because the halting problem, which is undecidable, can be reduced to the information control problem. Consider the following statement:

$$\textbf{if } s > 1 \textbf{ then } c; p := 2 \textbf{ else skip end}$$

[1] Command **skip** cannot be used to leak sensitive information, thus the typing system always accepts it: $\Gamma, ctx \vdash \textbf{skip}$.

where s is a secret variable and p is a public variable. If a mechanism could precisely decide whether this statement is secure, then this mechanism could decide whether command c terminates (if the statement is secure, it means that c does not terminate, but if the statement is not secure, than c terminates), which is impossible. So, false positives are unavoidable for information flow control mechanisms.

The effort of researchers has been focused on proposing enforcement mechanisms with as few false positives as possible. One way to decrease false positives is for the mechanism to use run time information. These mechanisms are called dynamic, because they analyze programs during execution.

Dynamic enforcement mechanisms

A dynamic mechanism checks and/or deduces labels on variables during execution. If an assignment $x := e$ is executed, and if Γ is fixed, then the mechanism checks whether relation $\Gamma(e) \sqcup ctx \sqsubseteq \Gamma(x)$ holds and halts the execution when the check fails. If an assignment $x := e$ is executed, and if Γ is flow sensitive, then the mechanism deduces label $\Gamma(x)$ to be $\Gamma(e) \sqcup ctx$. When execution enters a conditional command, the mechanism augments ctx with the label of the guard.

Under a dynamic mechanism, statement (18) would be accepted. This is because, at any execution, only assignment $y := 1$ is executed, and this assignment does not cause any illegal explicit or implicit flows. So, statement (18), which is rejected by the static type system, would be accepted by a dynamic mechanism.

However, dynamic mechanisms are notorious for introducing leakage of information through the checks they perform and the labels they deduce during execution. The example below demonstrates how sensitive information may be leaked through deduced labels:

$$\begin{array}{l} x := 0; \\ \textbf{if } h > 0 \textbf{ then } x := 1 \textbf{ else skip end}; \\ y := x \end{array} \tag{19}$$

Assume the dynamic mechanism uses a flow sensitive mapping Γ, which is initialized as: $\Gamma(x) = L$, $\Gamma(y) = L$, and $\Gamma(h) = H$. Assume also that the final value of y is the output of the program. If $h > 0$ is *false*, then $\Gamma(x)$ remains L, and thus, $\Gamma(y)$ becomes L, at termination. So, the value assigned to y will be a public output. If $h > 0$ is *true*, then $\Gamma(x)$ becomes H, and thus, $\Gamma(y)$ becomes H, at termination. So, no public output will be generated. Thus, the value of $h > 0$ is leaked to public outputs.

Notice that in example (19) the value of $h > 0$ always flows to the value of x. This is because the final value of x is 1 when $h > 0$ is *true*, and 0 when $h > 0$ is *false*. So, at termination, x should always be tagged with H. However, the dynamic mechanism of the example above failed to tag x with H when $h > 0$

was *false*, because during execution no assignment to x was performed in the context of $h > 0$. So, that dynamic mechanism missed to capture the indirect flow from h to x, when $h > 0$ was *false*, because x did not appeared as a target variable[2] in the the taken branch; x was a target variable in the untaken branch.

One way to capture indirect flows to target variables of untaken branches, is to perform an *on-the-fly* static analysis of untaken branches. When the execution (and the analysis) of the taken branch of a conditional statement finishes, the mechanism can analyze the untaken branch, without executing it, to ensure all implicit flows from the context to target variables are captured. Considering again example (19). When $h > 0$ is *false*, and after the taken branch (i.e., **skip**) is executed, an on-the-fly static analysis can be applied to the untaken branch $x := 1$. Specifically, $\Gamma(x)$ would be deduced to be $\Gamma(1) \sqcup \Gamma(h)$, which is label H. So, for every execution, x is tagged with H. Subsequently, for every execution, y is tagged with H, and thus there will always be no public output. Thus, $h > 0$ is no longer leaked to public outputs.

A dynamic enforcement mechanism may leak sensitive information when it decides to halt an execution due to a failed label check. Consider the program below:

$$
\begin{array}{l}
p := 0; \\
\textbf{if } s > 0 \textbf{ then } p := 1 \textbf{ else } s := 1 \textbf{ end}; \\
p := 2
\end{array}
$$

where fixed mapping Γ is: $\Gamma(p) = L$ and $\Gamma(s) = H$. Assume that the final value of p is the (public) output of the program. If $s > 0$ is *true*, then execution is halted to prevent s from implicitly flowing to p through assignment $p := 1$. So, no public output is generated. If $s > 0$ is *false*, then execution terminates normally, and thus one public output is generated. Here, a public output is generated depending on whether the execution is halted or not, which in turn is a decision that depends on $s > 0$. Consequently, $s > 0$ is leaked to public outputs. Current research is focused on designing dynamic mechanisms that do not leak sensitive information when they decide to halt an execution, without making the mechanism too conservative.

As a conclusion, dynamic mechanisms have both advantages and disadvantages comparing to static mechanisms for information flow control. A dynamic mechanism may be less conservative (less false positives) than a static mechanism, but a dynamic mechanism adds run time overhead. Also, a dynamic mechanism may introduce new covert channels due to label deduction and halting decisions.

[2]For any assignment $x := e$, x is called the target variable.