Notes on Information Flow Control (2)

CS 5430

May 9, 2017

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.

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:

$$\forall \ell : \ M_1 =_\ell M_2 \Rightarrow c(M_1) =_\ell c(M_2), \tag{1}$$

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 (1) of noninterference. Statement (1) can be used for either confidentiality or integrity. In fact, statement (1) is oblivious to the exact meaning of labels and the policies they represent. Noninterference in (1) 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 (1) 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:

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

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 $\bot$:

$$\frac{\Gamma(e) = \bot \quad \Gamma, ctx \vdash c}{\Gamma, ctx \vdash \textbf{while } e \textbf{ do } c \textbf{ end}} \tag{2}$$

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 (2).

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:

$$\textbf{if } 1 = 1 \textbf{ then } y := 1 \textbf{ else } y := x \textbf{ end} \tag{3}$$

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

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:

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

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 (3) 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 (3), 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:

$$\begin{aligned} &x := 0;\\ &\textbf{if } h > 0 \textbf{ then } x := 1 \textbf{ else skip end};\\ &y := x \end{aligned} \tag{4}$$

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 (4) 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 (4). 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

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

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.

Exercises

Problem 1:
(*i*) Give a program that is accepted by both the static type system and the dynamic analysis (with on-the-fly static analysis).
(*ii*) Give another program that satisfies noninterference, but it is rejected by both the static type system and the dynamic analysis (with on-the-fly static analysis).
(*ii*) If a program is accepted by the static type system, will this program be necessarily accepted by the dynamic analysis, too?

Problem 2: Consider a dynamic mechanism with on-the-fly static analysis and flow sensitive Γ. What are the confidentiality labels that tag variables in the program below, immediately after the execution of $y := 2$? What are the confidentiality labels that tag variables when the program terminates? Assume Γ is initialized as: $\Gamma(x) = H$, $\Gamma(y) = L$, $\Gamma(z) = H$.

```
$x := 1;$
$y := 2;$
**if** $z > 0$ **then** $y := 1$ **else** $x := 2$ **end**
```

Problem 3: Consider a purely dynamic mechanism that decides to halt the execution before entering conditional commands with sensitive guard expressions (i.e., the label of the guard expression is not $\perp$). Can this mechanism leak sensitive information when deciding to halt an execution?