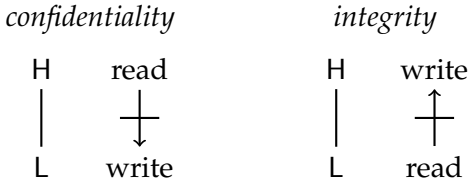
Lecture Notes on Declassification

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Lecture 12
October 3, 2024

1 Introduction

Before we move on the core topic of today's lecture, namely declassification, we briefly talk about another use of information flow. So far we have focused on *confidentiality*, that is, preventing flow reading high security data and writing them to low security variables. The dual, *integrity*, prevents reading low security data and writing them to high security variables. The prevents an attacker from violating the integrity of high security information.



Reasoning about integrity is dual to what we have developed so far for confidentiality and can be addressed with the same techniques.

In view of the examples showing incompleteness of the type system, one wonders if it is possible to reason about information flow with more precision. In other words, can we perhaps use dynamic logic at the cost of potentially incurring an undecidable problem? The answer is yes, and we will think this through in Section 2.

After that (in Sections 3 and 4) we consider some common scenarios where the kind of information flow control we have discussed so far is too strict, and we need to allow some information to be leaked. But hopefully not too much!

2 Information Flow in Dynamic Logic

Can we reason about information flow more precisely than in the type system by employing dynamic logic? Since noninterference is not a property of a single exe-

cution of a program, it seems at first that $P \to [\alpha]Q$ would be insufficient, because we always just reason about the poststate of a single execution of α.

Let's review the definition, restricting ourselves here to just high (H) and low (L) security levels.

> We define $\Sigma \models \alpha$ secure
> iff for all ω_1 and ω_2, $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ implies $\Sigma \vdash \mathsf{eval}\ \omega_1\ \alpha \approx_{\mathsf{L}} \mathsf{eval}\ \omega_2\ \alpha$

The notation in the conclusion is meant to imply that both evaluations terminate.

Maybe we start with how to represent $\Sigma \models \omega_1 \approx_{\mathsf{L}} \omega_2$. For each variable in the program α there should be two versions. The low-level versions must be equal before the program is executed, but no such constraint is imposed on the high-level versions.

Let's take the example

$$\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b$$

where $\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L})$ and a and b are constants. We create two versions of each variable x, x', y, and y'. The condition that the low-security variables must be equal is represented by

$$y = y'$$

Cool. How do we reason about the evaluation of α in the two different initial states? One solution is the run them sequentially, but rename one of them to use only the primed variables. So:

$$\begin{aligned} y = y' \to [&(\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b)\ ; \\ &(\textbf{if } x' = 0 \textbf{ then } y' := a \textbf{ else } y' := b)]\ Q(x, x', y, y') \end{aligned}$$

But what should the postcondition be? It should once again express that the low-security variables must be equal. So it is the same as the precondition! It will automatically talk about the values of these variables after evaluation for two reasons: (1) the two versions of the program compute over different variables, and (2) $[\alpha]Q$ means that Q must be true in every possible poststate of α (of which there is at most one). If there is none, then any postcondition is provable and the program is considered secure.[1]

So in summary, we define for all variables occurring in the program:

$$Q = \bigwedge_{\Sigma(x)=\mathsf{L}} (x = x')$$

and then prove noninterference by proving

$$Q \to [\alpha\ ;\ \alpha']Q$$

[1] We missed that point in lecture, but fortunately it works out. But one would still have to consider the issue of loop invariants.

where α' is the renaming of α by priming all variables.

Let's apply this technique in our example $\alpha = (\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b)$ by computing the weakest liberal precondition of $y = y'$ with respect to

$$\mathsf{wlp}\ (\alpha\ ;\ \alpha')\ (y = y')$$

We do this by first constructing $\mathsf{wlp}\ \alpha'\ (y = y')$.

$$\begin{array}{ll} & \mathsf{wlp}\ (\textbf{if } x' = 0 \textbf{ then } y' := a \textbf{ else } y' := b)\ (y = y') \\ = & (x' = 0 \to \mathsf{wlp}\ (y' := a)\ (y = y')) \wedge (x' \neq 0 \to \mathsf{wlp}\ (y' = b)\ (y = y')) \\ = & (x' = 0 \to y = a) \wedge (x' \neq 0 \to y = b) \\ = & Q(x', y) \end{array}$$

Now we use $Q(x', y)$ as a postcondition for (the unrenamed) α.

$$\begin{array}{ll} & \mathsf{wlp}\ (\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b)\ Q(x', y) \\ = & (x = 0 \to \mathsf{wlp}\ (y := a)\ Q(x', y)) \wedge (x \neq 0 \to \mathsf{wlp}\ (y' = b)\ Q(x', y)) \\ = & (x = 0 \to Q(x', a) \wedge (x \neq 0 \to Q(x', b)) \\ = & R(x, x') \end{array}$$

Let's work out what this is (with some small simplifications):

$$\begin{array}{lll} R(x, x') & \leftrightarrow & \quad (x = 0 \wedge x' = 0 \to a = a) \\ & & \wedge \quad (x = 0 \wedge x' \neq 0 \to a = b) \\ & & \wedge \quad (x \neq 0 \wedge x' = 0 \to b = a) \\ & & \wedge \quad (x \neq 0 \wedge x' \neq 0 \to b = b) \end{array}$$

The first and last conjunct are true, since $a = a$ and so we obtain

$$\begin{array}{lll} R(x, x') & \leftrightarrow & \quad (x = 0 \wedge x' \neq 0 \to a = b) \\ & & \wedge \quad (x \neq 0 \wedge x' = 0 \to b = a) \end{array}$$

If the constants a and b are equal, this is valid (regardless of x and x') and if a and b are not equal, this is not valid because we can pick x and x' to be different and falsify it. Note that our precondition $y = y'$ is irrelevant here since y and y' are both assigned to by the program.

So we conclude that the program

$$\textbf{if } x = 0 \textbf{ then } y := a \textbf{ else } y := b$$

is secure if and only if the constants a and b are equal.

3 Checking PINs

Imagine there is a variable pin holding a personal identification number (in lieu of a password) and a variable $guess$ holding the user's "guess". The job of the

following small program is to match the the user's guess against the pin and set the variable $auth$ to 1 (user authenticated) or 0 (not authenticated).

$$\textbf{if } guess = pin \textbf{ then } auth := 1 \textbf{ else } auth := 0$$

Let's see what the security level of these three variables need to be.

pin**:** Clearly, this shouldn't be leaked so $pin : \mathsf{H}$

$guess$**:** This is the user input, so it is low security $guess : \mathsf{L}$

$auth$**:** This needs to be exposed to the user so they know if they are logged in or not. So $auth : \mathsf{L}$.

At this point we can see that this does **not** satisfy our definition of noninterference and is therefore not secure according to the only reasonable policy. It is easy to cook up an example of two states where the guesses are the same, but lead to different outcomes depending on the pin. In the type system, the problem is reflected in that the comparison is high security and therefore the two branches must be checked with $pc = \mathsf{H}$. But they write to the low security variable $auth$.

In order to account for this kind of situation, Volpano and Smith [2000] introduced a **match** construct into the language, comparing a guess to a secret password. We use a new kind of formula $\textbf{match}(e_1, e_2)$ with the typing rule

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash \textbf{match}(e_1, e_2) : \ell_1 \sqcap \ell_2}\ \textbf{match}F$$

For this rule to make sense we need to generalize the semilattice of security levels we had so far to a *lattice*, where there is a *greatest lower bound* operation $\ell_1 \sqcap \ell_2$ (often pronounced "*meet*"). For the **match** expression we lower the security level of the result, but only as far as necessary so the result is below ℓ_1 and ℓ_2.

Recasting our particular example

$$\textbf{if } \textbf{match}(guess, pin) \textbf{ then } auth := 1 \textbf{ else } auth := 0$$

it is now well-typed because we have

$$pin : \mathsf{H}, guess : \mathsf{L}, auth : \mathsf{L} \vdash \textbf{match}(guess, pin) : \mathsf{L}$$

Of course, without any change to our definition of noninterference, it will still not be semantically secure.

This raises several questions:

1. Can we relax our definition of noninterference to account for **match**?
2. What are the consequences of adding it to the language? We intend the effect to be somehow confined, but we might be making a mistake and suddenly all secrets may be leaked.

3. How can we square the fact that the match operation doesn't preserve confidentiality with the fact that it is commonly used and "seems to be fine" (ignoring some practical issues like insecure passwords).

One particularly worrisome aspect of this construct might be the following program:

$$guess := 0 \ ; \ \textbf{while} \ \neg \textbf{match}(guess, pin) \ guess := guess + 1$$

If pins are known to be nonnegative, this program would allow us to determine the password!

The reason this seems to be okay in practice is that each time around the loop, unless the pin has been determined, the attacker only rules out a single possible pin. If the size of the pin is, say, 256 bits, then if the pins are uniformly distributed it would take something like 2^{255} guesses on average to identify the password (which is clearly not feasible).

Volpano and Smith [2000] turn this into a theorem that states that a polynomial attacker can determine a k-bit integer (drawn from a uniform distribution) with probably of at most $(\mathrm{poly}(k) + 1)/2^k$. They also point out that it is essential that the security level of $\textbf{match}(e_1, e_2)$ is $\ell_1 \sqcap \ell_2$ and cannot always be simply of low security (L, or $\bot$ in the general case).

4 Explicit Declassification

When we generalize away from the **match** construct, the situation becomes a lot more complex, even if the new rule is deceptively simple:

$$\frac{}{\Sigma \vdash \textbf{declassify}_\ell(e) : \ell} \ \textbf{declassify} F$$

$\textbf{declassify}_\ell(e)$ is an expression (where ℓ defaults to $\bot$ in general and L in our typical example), but if we had a corresponding formula we could model the **match** with

$$\textbf{if} \ \textbf{declassify}(guess = pin) \ \textbf{then} \ auth := 1 \ \textbf{else} \ auth := 0$$

However, the generality comes at a price, namely that there no simple reasoning principles covering the many applications of declassification.

There are many dimension to declassification, and we recommend Sabelfeld and Sands [2009] for a broad discussion of the many relevant issues. A key question is *what* is being declassified, and some thought needs to be given *why*. One class of examples are aggregates, like averages. For example, in this course we won't reveal to you the scores of others, but we might reveal class averages. Of course, if there were only two students, and you were one of them, you could infer the other students score if you knew the average. Another example is given by

releasing (possibly anonymized) samples. Neither of these can be done without declassification.

What is being declassified is of critical importance in determining the impact and what an attacker might learn. For example, imagine that instead of **match** we had **compare** that reveals the result of comparing a high security and a low security variable, say, with less-or-equal. Unlike the **match** construct can then learn the pin by using binary search through the space of possibilities.

Can we say anything generically about the **declassify** construct that is useful? In other words, can we generalize noninterference to take declassification into account? In some circumstances we can, but the general definition may still require a lot of work in each case to prove something about how much the attacker can learn.

So let's assume we have a program α with a single occurrence of a construct $\mathbf{declassify}_{\mathsf{L}}(e)$. We would like to express that the attacker can essentially only learn about e, but nothing else. We express this by weakening the noninterference definition to allow the attacker not only know the value of the low-security variables, but also the value of the expression e.

> We attempt to define $\Sigma \models \alpha$ secure where α contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$ iff $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ and eval ω_1 e = eval ω_2 e imply $\Sigma \vdash$ eval ω_1 $\alpha \approx_{\mathsf{L}}$ eval ω_2 α

Unfortunately, this definition is not quite correct. For example, consider the following program that attempts to declassify an average of $x_1, \ldots, x_n$, all high security variables as $\mathit{average}$: L.

$$
\begin{array}{l}
x_2 := x_1 \ ; \\
x_3 := x_2 \ ; \\
\ldots \\
x_n := x_{n-1} \ ; \\
\mathit{average} := \mathbf{declassify}_{\mathsf{L}}((x_1 + x_2 + \cdots + x_n)/n)
\end{array}
$$

Note that this program leaks the value of x_1. That's because in the condition eval ω_1 e = eval ω_2 e we evaluate e in the original environments, but in this example we assign to the free (high security) values of e before declassifying the aggregate. So we modify our definition to:

> We define $\Sigma \models \alpha$ secure where α contains a single occurrence of $\mathbf{declassify}_{\mathsf{L}}(e)$ iff $\Sigma \vdash \omega_1 \approx_{\mathsf{L}} \omega_2$ and for all $x \in$ use e implies $x \notin$ maydef α and eval ω_1 e = eval ω_2 e imply $\Sigma \vdash$ eval ω_1 $\alpha \approx_{\mathsf{L}}$ eval ω_2 α

With this definition we can prove that the inference rules with the additional rules for declassification are *sound*.

Theorem 1 *If* $\Sigma \vdash \alpha$ secure *and* α *contains exactly one instance of* $\mathbf{declassify}(e)$ *and for all* $x \in$ use e *implies* $x \notin$ maydef α, *then* $\Sigma \models \alpha$ secure *according to the preceding definition.*

Proof: See Sabelfeld and Myers [2003]. □

References

Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *International Symposium on Software Security (ISSS 2003)*, pages 174–191. Springer LNCS 3233, November 2003.

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

Dennis M. Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In M. N. Wegman and T. W. Reps, editors, *Symposium on Principles of Programming Languages*, pages 268–276, Boston, Massachusetts, January 2000. ACM.