Lecture Notes on
Beyond Safety Properties

15-316: Software Foundations of Security & Privacy
Matt Fredrikson

Lecture 11
February 19, 2026

1 Introduction

Recall that a *trace* of a program is the (potentially infinite) sequence of states that make up its computation. A *safety property* of a trace is defined as one whose violation can be determined from a finite prefix. A *liveness property* is one whose violation may depend on the whole infinite trace. Operations such as division by zero or out-of-bounds memory access are examples of safety properties. We can prove safety in dynamic logic via propositions of the form $P \rightarrow [\alpha]\top$. Examples of liveness properties would be that a server responds to a query or that a lock acquired in a concurrent computation is eventually released. We can prove liveness properties in dynamic logic via propositions of the form $P \rightarrow \langle\alpha\rangle Q$ (pronounced "*diamond alpha Q*"). Recall that the formula $\langle\alpha\rangle Q$ is true if there is a way to reach a final state such that Q is true. This implies that, among other things, loops appearing in the computation must be proved terminating. We have deemphasized the diamond modality, not investigating its properties.

There are techniques for transforming liveness properties into safety properties. For example, we can require that a server respond within a certain number of steps or milliseconds. However, it may still be difficult to enforce such transformed liveness properties, and it may be even more difficult to take appropriate corrective action.

Today we will start analyzing an important class of security policies, called *information flow policies*, that go beyond both safety and liveness properties in the sense that we cannot determine if they are violated by analyzing a single program trace.

2 Information Flow, Informally

When you log in to your favorite banking site, you would like to be able to see information about your own account, but you should not be able to see anyone else's. In other words, we don't want information to flow from other accounts to the program serving you. In our small imperative language we model this using *high security variables* and *low security variables*. Reading from a high security variable and writing the value to a low security variable would be a violation of our information flow policy. It would be slightly more realistic to consider reading and writing from memory, but it would be more complex without changing the fundamental ideas we study.

As a small running example we consider the following program.

$$\begin{array}{l} x := 1 \ ; \\ y := x + 5 \ ; \\ z := y - 1 \end{array}$$

We consider x to be a high security variables, while y and z are classified as low security. We write this information flow policy as

$$x : \mathsf{H}, y : \mathsf{L}, z : \mathsf{L}$$

Intuitively, the program above would not satisfy our security policy: we read from x (high security) and then write $x + 5$ to y, which is low security. In fact, we can exactly recover x as $y - 5$, so we gain perfect information about a secret.

Here is a trace of this program, assuming all variables initially have value 0.

$$\begin{array}{rl} & (x = 0, y = 0, z = 0) \\ \Rightarrow & (x = 1, y = 0, z = 0) \\ \Rightarrow & (x = 1, y = 6, z = 0) \\ \Rightarrow & (x = 1, y = 6, z = 5) \end{array}$$

Now consider the following alternative program shown on the right.

$$\begin{array}{l|l|l} (x = 0, y = 0, z = 0) & x := 1 \ ; & x := 1 \ ; \\ (x = 1, y = 0, z = 0) & y := x + 5 \ ; & y := 6 \ ; \\ (x = 1, y = 6, z = 0) & z := y - 1 & z := 5 \\ (x = 1, y = 6, z = 5) & & \end{array}$$

We see that both programs have *exactly the same trace*, but the first one violates the policy (information flows from x to y and then to z) while no information flows at all on the right.

This shows that information flow is not a property of a single trace of a program, but requires something more. We'll see what in the next lecture. Meanwhile, we'll try to intuit a program analysis that ensures adherence to an information flow policy. In the next lecture, we will check if it accomplishes what a semantic definition of information flow demands.

3 Tracking Security Levels

We now consider ways analyze programs with the goal of proving that a given information flow policy is respected by a program. Because we haven't rigorously defined what that is, the remainder of this lecture is rather speculative. In the next lecture we will nail it down precisely.

For now, we imagine a security policy is given by an assignment of security levels to variables, like H for *high* and L for *low*. We use Σ as a map from variables to security levels. We write $x : \ell$ if the variable x has security level ℓ. Our system of inference rules derive

$$\Sigma \vdash \alpha \text{ secure}$$

which expresses that, given the security policy Σ, the program α is secure. We name the inference rules as $nameT$, where T stands for *taint* (see Section 6).

Assignment. At the root of the system is that an assignment $x := e$ is a violation of the security policy if $x : \mathsf{L}$ and $e : \mathsf{H}$. The security level of an expression is the maximal level of the variables occurring in it. That is, we also define $\Sigma \vdash e : \ell$, meaning that expression e has security level ℓ.

Variables just have the level prescribed by the policy, and constants are always of low security.

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}T \qquad \frac{}{\Sigma \vdash c : \mathsf{L}}\ \mathsf{const}T$$

For a binary operator, we take the maximal security level of the constituents, where $\mathsf{H} > \mathsf{L}$.

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2 \quad \ell = \max(\ell_1, \ell_2)}{\Sigma \vdash e_1 + e_2 : \ell}\ +T$$

For an assignment, there are several possible combinations. The first: we can always write to a high security variable, since this does not represent a flow from high to low. An example of this could be appending to a (secure) log file using a low-security value.

$$\frac{\Sigma(x) = \mathsf{H}}{\Sigma \vdash x := e \text{ secure}}\ :=T_1$$

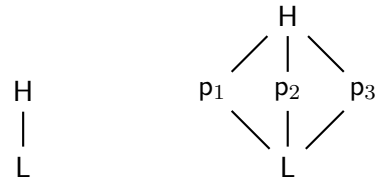
If x is of low security, then e also has to be (or we fail).

$$\frac{\Sigma(x) = \mathsf{L} \quad \Sigma \vdash e : \mathsf{L}}{\Sigma \vdash x := e \text{ secure}}\ :=T_2 \qquad \frac{\textbf{no rule for} \quad \Sigma(x) = \mathsf{L} \quad \Sigma \vdash e : \mathsf{H}}{\Sigma \vdash x := e \text{ secure}}$$

4 A Lattice of Security Levels

Before we go further, we generalize the security levels from just two (high and low) to potentially multiple ones. We imagine them being arranged in a *lattice*, where information is allowed to flow upwards but not downwards. I think that technically we just need a *join-semilattice*, although different authors make slightly different assumptions.

Below are two examples. This first just has the high and low security levels we have been using. The second is one where we imagine three principals, p_1, p_2, and p_3, say bank account holders. They cannot see high security values (level H) and they cannot see each other's data since information can only flow up but not down. The lowest level L is "public" (anyone can see it).



A join-semilattice is defined by a partial order $\ell_1 \sqsubseteq \ell_2$ (ℓ_1 is a lower security level than ℓ_2) and the operation of $\ell_1 \sqcup \ell_2$ (the *least upper bound* of two security levels). We also need a least element $\bot$ which is the unit of $\sqcup$ and is below every other level. We will not go into details regarding all the algebraic laws of the semilattice, but here are some from properties of the least upper bound and the least element.

$$\bot \sqcup \ell = \ell \sqcup \bot = \ell$$

$$\bot \sqsubseteq \ell$$

$$\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$$
$$\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$$
$$\ell_1 \sqsubseteq \ell \text{ and } \ell_2 \sqsubseteq \ell \text{ implies } \ell_1 \sqcup \ell_2 \sqsubseteq \ell$$

We can generalize the rules so far from two levels to a lattice of security levels.

$$\frac{\Sigma(x) = \ell}{\Sigma \vdash x : \ell}\ \mathsf{var}T \qquad \frac{}{\Sigma \vdash c : \bot}\ \mathsf{const}T \qquad \frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2 \quad \ell = \ell_1 \sqcup \ell_2}{\Sigma \vdash e_1 + e_2 : \ell}\ +T$$

$$\frac{\Sigma \vdash e : \ell \quad \ell \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e\ \mathsf{secure}}\ :=T$$

The last rule expresses succinctly that information can only flow from lower to higher levels of security, and not in any other circumstances.

5 Tracking Security Levels, Continued

With assignment specified, we move on to other language constructs.

Sequential Composition. We check both subprograms independently with respect to the same security policy Σ.

$$\frac{\Sigma \vdash \alpha \text{ secure} \quad \Sigma \vdash \beta \text{ secure}}{\Sigma \vdash \alpha \,;\, \beta \text{ secure}} \,;T$$

At this point we have enough to check that one of our two example programs is secure while the other one is not. We use the two-element lattice with $\mathsf{H} \sqsupset \mathsf{L}$ and the security policy

$$\Sigma_0 = (x : \mathsf{H}, y : \mathsf{L}, z : \mathsf{L})$$

We construct the following derivation bottom-up, failing at the second assignment as expected. We elide some rule names for the sake of brevity.

$$\dfrac{\dfrac{\dfrac{}{\Sigma_0 \vdash 1 : \mathsf{L}} \quad \mathsf{L} \sqsubseteq \Sigma_0(x) = \mathsf{H}}{\Sigma_0 \vdash x := 1 \text{ secure}} :=T \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Sigma_0(x) = \mathsf{H}}{\Sigma_0 \vdash x : \mathsf{H}} \quad \dfrac{}{\Sigma_0 \vdash 5 : \mathsf{L}} \quad \mathsf{H} = \mathsf{H} \sqcup \mathsf{L}}{\Sigma_0 \vdash x + 5 : \mathsf{H}} +T \qquad \text{fails} \quad \mathsf{H} \sqsubseteq \Sigma_0(y) = \mathsf{L}}{\Sigma_0 \vdash y := x + 5 \text{ secure}} :=T \qquad \dots}{\Sigma_0 \vdash (y := x + 5 \,;\, z := y - 1) \text{ secure}} \,;T}{\Sigma_0 \vdash (x := 1 \,;\, y := x + 5 \,;\, z := y - 1) \text{ secure}} \,;T$$

Conditionals. Conditionals are interesting. The condition may have a security level, but we ignore that for now because it doesn't perform an assignment.

$$\frac{\Sigma \vdash \alpha \text{ secure} \quad \Sigma \vdash \beta \text{ secure}}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}} \,\textbf{if}T$$

Loops. Loops are similar to conditionals.

$$\frac{\Sigma \vdash \alpha \text{ secure}}{\Sigma \vdash \textbf{while } P \ \alpha \text{ secure}} \,\textbf{while}T$$

6 Taint Analysis

The rules we have so far can be used for *taint checking*. We think of high security variables as being sources of taint and we track how their values are propagated throughout a program. If a tainted value reaches a variable that is of low security,

the program can be rejected or aborted as insecure. This can be done statically (so insecure programs are never executed) or dynamically (say, with an extra taint bit attached to memory locations or values).

If a tainted value reaches a low-security value, we definitely have a violation of the (for now informal) security policy. We don't even have to declare the security level of all variables, because the analysis can infer them. For more on taint analysis, we recommend Schwartz et al. [2010].

However, there are some obvious situations where a flow of information does occur, but taint analysis will not discover it. We recommend you think about possible holes in the system before moving on.

7 Indirect Flows

Consider the following simple program, where $x : \mathsf{H}$ and $y : \mathsf{L}$.

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

With our policy so far, this program checks! Both assignments to y are with constants, which have low security level. It should be clear that, intuitively, information (illegally!) flows from x to y because x is tested in the condition, and x has a high security level.

In order to track this kind of *indirect flow* we need to somehow "remember" which tests we have performed to get to the current point in the program, and whether these tests involved high security variables. For example, the program above would be safe if both x and y were of the same security level (whether high or low).

Our solution is based on the seminal paper by Volpano et al. [1996], although the details of the formalization vary.

For that purpose we introduce a *ghost variable* named pc. It is called a ghost variable because it is not allowed to appear in the program, only in our analysis rules. We usually assume that the program starts executing at the lowest level of security ($\bot$ in general, in many of our examples L). Let's get right to the critical rule. We now call the rules $name F$, where F suggests *flow*.

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \text{ secure} \quad \Sigma' \vdash \beta \text{ secure}}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}} \textbf{if} F$$

Let's tease apart what's in this rule. First, we determine the security level of P (which will end up being the least upper bound of all variables occurring in P). If the current program is at the lowest security level, that would be the level in which we have to check α and β. But if we have already branched on conditions before, we might have to pick a higher one. So $\ell' = \Sigma(pc) \sqcup \ell$ is the least upper bound of the current pc and the test P. We update the security level of the pc to ℓ' and then check the branches.

In our example, we have $x : \mathsf{H}, y : \mathsf{L} \vdash x = 0 : \mathsf{H}$, so both branches are checked with $pc : \mathsf{H}$.

Now we have to reconsider the other constructs to take the pc into account.

Assignment. For assignment $x := e$, we have to take the least upper bound of the level of e and level of pc and compare the result to the level of x. This rules out our motivating example, as it should.

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \text{ secure}} :=F$$

Sequential Composition. It seems like as we proceed through the program the security level of the pc keeps going up, as in the rule for conditionals. Does it ever go down? Not explicitly, but when we have processed both branches of a conditional and proceed with the program that follows it, the security level implicitly reverts to what it was before.

$$\frac{\Sigma \vdash \alpha \ \mathsf{secure} \quad \Sigma \vdash \beta \ \mathsf{secure}}{\Sigma \vdash \alpha \ ; \ \beta \ \mathsf{secure}} \ ;F$$

For example, the following program is secure. Even though the assignments to y are checked at a high security level (with $pc : \mathsf{H}$), the assignment to z is checked with $pc : \mathsf{L}$.

$$\begin{array}{l}(x : \mathsf{H}, y : \mathsf{H}, z : \mathsf{L}, pc : \mathsf{L}) \\ x := 1 \ ; \\ \textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } y := 0 \ ; \\ z := 5\end{array}$$

Loops. Loops are similar to conditionals in the sense that we may have to upgrade the security level of the pc in the loop body. Somehow we lose information about the failed test when we exit the loop. We have to see if this is really sound in the next lecture. Let's mark it as suspicous for now.

$$\frac{\Sigma \vdash P : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \ \mathsf{secure}}{\Sigma \vdash \textbf{while } P \ \alpha \ \mathsf{secure}} \ \textbf{while}F \ ?$$

Tests. Tests also seem a bit strange. Do we need to check that the security level of P is lower than the pc or not?

$$\frac{\Sigma \vdash P : \ell \quad \ell \sqsubseteq \Sigma(pc)}{\Sigma \vdash \textbf{test } P \ \mathsf{secure}} \ \textbf{test}F \ ?$$

Formulas. The security level of a formula is easy to determine. We show some sample rules. In the rule $\top F$ the "$\bot$" is the least element of the lattice, not falsehood (an unfortunate clash of notation).

$$\frac{\Sigma \vdash e_1 : \ell_1 \quad \Sigma \vdash e_2 : \ell_2}{\Sigma \vdash e_1 \leq e_2 : \ell_1 \sqcup \ell_2} \leq F \qquad \frac{}{\Sigma \vdash \top : \bot} \ \top F \qquad \frac{\Sigma \vdash P : \ell_1 \quad \Sigma \vdash Q : \ell_2}{\Sigma \vdash P \wedge Q : \ell_1 \sqcup \ell_2} \wedge F$$

Let's read the last rule:

> *The formula $P \wedge Q$ has security level $\ell_1 \sqcup \ell_2$ if P has security level ℓ_1 and Q has security level ℓ_2.*

If we wrote this as a function seclev, it would something like

$$\mathsf{seclev}\ \Sigma\ (P \wedge Q) = \mathsf{lub}\ (\mathsf{seclev}\ \Sigma\ P)\ (\mathsf{seclev}\ \Sigma\ Q)$$

References

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 *Symposium on Security and Privacy (2010)*, pages 317–331, Oakland, California, May 2010. IEEE.

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.