So far we have defined a wide range of programming language constructs. We will now see an example of how these can be used in an application domain, specifically security. Language-based security is the field of defining, analyzing, and enforcing security properties by building analysis and enforcement structures into programming languages themselves. Today we will be specifically focused on information flow control (IFC), a technique for tracking and controlling flows of information through a program.

IFC is most commonly used to prevent data leakage by tracking and constraining data flows based on confidentiality. That is the context in which we will discuss it today. Notably, identical ideas and techniques can operate with integrity [Biba 1977] (avoiding data corruption), availability [Zheng and Myers 2005] (understanding how crashes in one part of a system impact others), data consistency in a distributed system [Milano and Myers 2018] (ensuring data from weakly consistent stores doesn't pollute strongly consistent ones), and more. It has been shown to be effective in helping audit the security of code, even code written by malicious developers [Ernst et al. 2014], it is used to secure isolation in parts of Firefox [Narayan et al. 2020], has been proposed for internal use in operating systems [Zeldovich et al. 2008], and can even help detect and eliminate speculative execution vulnerabilities [Guarnieri et al. 2020; Zagieboylo et al. 2019].

1 Security Labels

To define, analyze, and enforce the security of a program, we need to know the security policies of that code. For IFC systems, we do that by attaching security labels to the inputs and outputs of the program. To be useful for tracking and controlling flows of information, we need to know when information labeled with one policy is allowed to influence (flow to) some output. We also need to know how to label computations with inputs that have different labels.

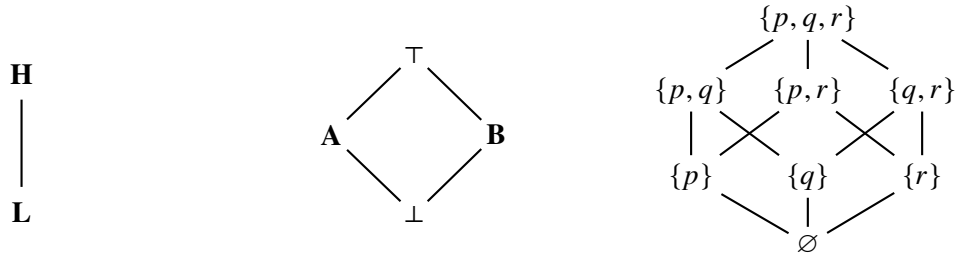
To accomplish these goals, we require the security labels $\mathcal{L}$ form a *lattice*[1] A lattice is a set with a partial order—we will write $\sqsubseteq$ and call it "flows to"—and join (least upper bound) and meet (greatest lower bound) operations, that we denote using $\sqcup$ and $\sqcap$, respectively.[2] That is, $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$ must satisfy the following rules.

- Flows to ($\sqsubseteq$) is a partial order; it is reflexive, transitive, and anti-symmetric.
- The join of two labels, $\ell_1 \sqcup \ell_2$, is their least upper bound. That is, it is an upper bound of the two labels—$\ell_1, \ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$—and it flows to every upper bound—if $\ell_1, \ell_2 \sqsubseteq \ell$ then $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$. More succinctly, for all ℓ_1, ℓ_2, ℓ,
$$\ell_1 \sqcup \ell_2 \sqsubseteq \ell \iff \ell_1 \sqsubseteq \ell \text{ and } \ell_2 \sqsubseteq \ell.$$
- The meet of two labels, $\ell_1 \sqcap \ell_2$, is their greatest lower bound:
$$\ell \sqsubseteq \ell_1 \sqcap \ell_2 \iff \ell \sqsubseteq \ell_1 \text{ and } \ell \sqsubseteq \ell_2.$$

The simplest lattices are the empty and singleton lattices, but those are uninteresting from a security perspective, so we will ignore them. The next simplest lattice, shown visually in Figure 1a, is a two-point lattice consisting of "high" (**H**) and "low" (**L**) where $\mathbf{L} \sqsubseteq \mathbf{H}$, but not vice versa. This is a good lattice for building intuition where you think of **H** as "high confidentiality" or "secret" and **L** as "low confidentiality" or "public." All of the math we will discuss works with arbitrary lattices, but those can be harder to build intuition with.

[1] These are order-theory lattices, not to be confused with geometric lattices, which are the ones sometimes used in cryptography.

[2] Order theory often uses $\leq$, $\vee$, and $\wedge$ for ordering, join, and meet, respectively. We use the square versions to avoid confusion with numeric less than and logical conjunction and disjunction.



(a) Two-point Lattice (b) Four-point Lattice (c) Subset lattice

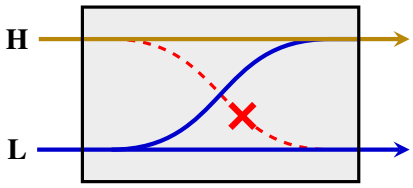
Figure 1: Examples of lattices

Another common example of lattices include four-point lattices (Figure 1b), where **A** and **B** are not related to each other. Here $\top$ is the most-secret data that nobody can read, $\bot$ is the most-public data that everyone can read, and **A** and **B** specify data that only one party can read.

Subset lattices of permissions (e.g., Figure 1c) are also good examples. Here the labels $\mathcal{L}$ are the subsets of a permission set $\{p, q, r\}$. Flows to is simply set inclusion $\subseteq$, and join and meet are union and intersection, respectively. In a security context, a label might represent the permissions required to see a piece of data.

2 Noninterference

The simplest and most standard of security definitions for IFC systems is *noninterference*, which says an attacker should learn nothing about high (secret) inputs just from looking at low (public) outputs. Intuitively, noninterference says that high inputs cannot influence, or "interfere with," low outputs in any way. The following picture depicts this intuition graphically.



Low inputs may freely influence both high and low outputs (the blue lines) and high inputs may freely flow to high outputs (the yellow line). However, flows are not allowed from high inputs to low outputs (the crossed-out dotted red line).

Noninterference is a semantic property. That is, its definition considers only the behavior of a program, not its source code (which contains information about other possible behaviors). That poses a challenge for formalizing this intuition about noninterference. How can we know that a low output did or did not depend on a high input when running the program? For instance, the program $\lambda h.\, h * 2$ produces the output 6 when run on input 3, but so too does the program $\lambda h.\, 6$.

The trick is to notice that the output of the first program changes when we change the high input h, while the output of the second does not. That is, we need to run the program twice! Specifically, a program is noninterfering if running it multiple times with different high inputs (but the same low inputs) will always produce the same high outputs.

Notably, an interfering program only needs to leak information *sometimes*. For instance, a program that takes high and low integers and multiplies them together is interfering, despite the fact that it will never leak information if the low value is 0. Because there is *some* low input (say, 1) where running it with that low input and different high inputs produces different results, the program is interfering (insecure).

We formalize noninterference using a notion of *indistinguishability*. Two values are indistinguishable at some label ℓ, often denoted $\approx_\ell$, if an observer at level ℓ cannot tell them apart. We then say that a program is noninterfering at ℓ if it produces indistinguishable outputs when run with indistinguishable inputs.

Definition 1. A program p is *noninterfering at* ℓ if, for any inputs v_1 and v_2 such that $v_1 \approx_\ell v_2$, whenever $p(v_1) \longrightarrow^* w_1$ and $p(v_2) \longrightarrow^* w_2$, it must be the case that $w_1 \approx_\ell w_2$.

What constitutes input and output varies by setting, as does the definition of indistinguishability. For instance, an imperative language modeling a system where an attack can read portions of the memory during execution might define the input as an initial memory, the outputs as the sequence of memory states, and $\approx_\ell$ by comparing the portion of the memory the attacker can read. A functional language might consider functions on one variable where the variable is the input and the computed value is the output. In other settings, other definitions may be appropriate.

3 An Information Flow Language

We will now show one way to make the discussion above precise. To do that, we define an information flow calculus that is a slightly simplified version of the Dependency Core Calculus (DCC) [Abadi et al. 1999]. Specifically, we omit the fixed point operator of DCC, meaning that our calculus will be strongly normalizing. This choice substantially simplifies reasoning, allowing us to use simpler definitions and techniques. It also avoids the need to decide whether to model nontermination behavior itself as an information channel.

Our calculus will be STLC with sums, pairs, and a special tagged value to describe the security label on values. We denote tagged types $\mathrm{T}_\ell\ \tau$, and tag values by wrapping them in $\mathsf{wrap}_\ell\ v$. To compute on a tagged value, we must first unwrap it, using an unwrap expression that unwraps a value, binds it to a variable x, and substitutes the unwrapped value in for x in the body. The types and expressions of this language are as follows. We include integers for simplicity of examples.

$$
\begin{array}{lcl}
\ell & \in & \mathcal{L} \\
\tau & ::= & \mathsf{unit} \mid \mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathrm{T}_\ell\ \tau \\
e & ::= & () \mid n \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{proj}_i\ e \\
 & \mid & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x).\, e_1 \mid \mathsf{inr}(y).\, e_2 \\
 & \mid & \mathsf{wrap}_\ell\ e \mid \mathsf{unwrap}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

The semantic rules for the language are identical to STLC with sums and pairs, plus the following rules for wrap and unwrap, formalizing the intuition described above.

$$
E ::= \cdots \mid \mathsf{wrap}_\ell\ E \mid \mathsf{unwrap}\ x = E\ \mathsf{in}\ e
\qquad
\frac{}{\mathsf{unwrap}\ x = (\mathsf{wrap}_\ell\ v)\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]}
$$

Note that the semantics does not attempt to check that an unwrapped values is used in accordance with the security policy specified in the removed tag. That will be the job of the type system, which we will see later.

As a side note, this calculus is a *coarse grained* IFC system, one in which security-relevant values are wrapped, must be unwrapped to be used, and the entire computation in which they are used is then tainted by any restrictions present in the removed label. There are also *fine grained* systems that attach a label to every type instead of creating a new type constructor and the typing rules specify how they combine.

3.1 Examples

This language allows us to write some simple examples of both secure and insecure programs. For simplicity, we will use untagged values to represent fully public information.

As one example, a constant function $\lambda h.\, 6$ would be noninterfering—it returns the same result regardless of its input. The simplest insecure (interfering) program is one that takes a tagged secret value, unwraps it, and simply returns the same value in the clear (or wrapped in a much less secret tag).

$$\lambda h.\, \mathsf{unwrap}\ x = h\ \mathsf{in}\ x$$

This is an example of an *explicit* flow, where secret data flows directly to the output of the program. We could have modified the value in some simple ways, say by adding, subtracting, or multiplying by other values, and it would still be insecure. For instance, if $h : \mathrm{T}_{\mathbf{H}}\ \mathsf{int}$ and $l : \mathrm{T}_{\mathbf{L}}\ \mathsf{int}$, the program

$$\begin{aligned}\lambda hl.\, &\mathsf{unwrap}\ x = h\ \mathsf{in}\\ &\mathsf{unwrap}\ y = l\ \mathsf{in}\\ &x * y\end{aligned}$$

remains interfering; there exist pairs of inputs that are indistinguishable to a low observer and produce different outputs.

A more subtle type of leak is an *implicit* flow, where a secret impacts the control flow of the program, and the output depends on that control flow. For instance, consider the following example program of type $\mathrm{T}_{\mathbf{H}}\ (\mathsf{unit} + \mathsf{unit}) \to \mathrm{T}_{\mathbf{L}}\ \mathsf{int}$.

$$\begin{aligned}\lambda h.\, \mathsf{unwrap}\ x = h\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}&\\ \mathsf{inl}(\_).\ \mathsf{wrap}_{\mathbf{L}}\ 0&\\ \mid \mathsf{inr}(\_).\ \mathsf{wrap}_{\mathbf{L}}\ 1&\end{aligned}$$

This program only ever returns wrapped constants, and constants never leak information, right? However, because the control flow depends on the value of the secret input, it returns a *different* constant depending on the value of that input, thereby leaking precisely what the secret is. Indeed, to a low observer, this program will produce a different result when given $\mathsf{wrap}_{\mathbf{H}}\ (\mathsf{inl}())$ versus $\mathsf{wrap}_{\mathbf{H}}\ (\mathsf{inr}())$ ($\mathsf{wrap}_{\mathbf{L}}\ 0$ and $\mathsf{wrap}_{\mathbf{L}}\ 1$, respectively).

We will see in Section 4 how we can use a type system to rule out these sorts of programs while still allowing numerous (though not all) secure ones.

3.2 Defining Indistinguishability

If we had wrapped the outputs in the second example above with the *high* label **H** instead of the low one, the outputs would have been marked just as secret as the inputs and we would like to consider that program secure. Defining that distinction formally requires a precise notion of indistinguishability that captures the idea that a low observer can tell apart $\mathrm{T}_{\mathbf{L}}\ 0$ from $\mathrm{T}_{\mathbf{L}}\ 1$ (or just unwrapped 0 and 1), but cannot tell apart $\mathrm{T}_{\mathbf{H}}\ 0$ from $\mathrm{T}_{\mathbf{H}}\ 1$.

To define ℓ-equivalence (our notion of indistinguishability), we look at the various types in our language. Units and integers are clearly only indistinguishable if they are equal. For products, each component must be equal, and for sums, it must be the same injection (inl or inr), and the value must be the same. The other two type constructors—functions and tagged types—things are a little more complicated.

For tagged types, the label matters. Consider whether values $\mathsf{wrap}_{\ell'}\ v_1$ and $\mathsf{wrap}_{\ell'}\ v_2$ of type $\mathrm{T}_{\ell'}\ \tau$ are indistinguishable at label ℓ. If $\ell' \sqsubseteq \ell$, that means ℓ should be able to see the values, and it makes sense to require v_1 and v_2 to be indistinguishable. If $\ell' \not\sqsubseteq \ell$, however, the whole point of tagged values is to make them opaque! In particular, we want *any* pair of internal values to be equivalent. For technical reasons, we will require that v_1 and v_2 behave properly as values of type τ, which we will denote $v_1 \in \lfloor \tau \rfloor$ and $v_2 \in \lfloor \tau \rfloor$, but otherwise place no restrictions on them. For simplicity, we will define this as just being well-typed with type τ—that is, $v \in \lfloor \tau \rfloor \stackrel{\triangle}{\iff} \vdash v : \tau$—but many definitions use a more complicated purely semantic notion of "behaves as a value of type τ." We will provide the definition of the type system in Section 4.

Functions are, perhaps, the most complicated. For two functions to be indistinguishable to ℓ, they should behave the same way whenever given to inputs that are indistinguishable to ℓ. In essence, they need to look

noninterfering in their execution! This sort of definition starts to feel circular very quickly, and the way out of the hole is to use a logical relation.

Unlike the logical relation we used to prove normalization, this is a *binary* logical relation. That is, it relates two terms, not one. Also, in addition to being parameterized by a type, the relation is also parameterized by a label to indicate the level of indistinguishability. To make our lives easier, we actually define two relations: one on expressions, denoted $\mathcal{E}[\![\tau]\!]_\ell$, and one on values, denoted $\mathcal{V}[\![\tau]\!]_\ell$. The expression relation simply says that, if the expressions evaluate to values, those values must be related.

$$(e_1, e_2) \in \mathcal{E}[\![\tau]\!]_\ell \stackrel{\triangle}{\iff} e_1, e_2 \in \lfloor\tau\rfloor \wedge \forall v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \Rightarrow (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\ell$$

The value relation precisely captures the intuition described above about how to define indistinguishability.

$$\frac{}{((), ()) \in \mathcal{V}[\![\mathsf{unit}]\!]_\ell} \qquad \frac{}{(n, n) \in \mathcal{V}[\![\mathsf{int}]\!]_\ell} \qquad \frac{(v_1, w_1) \in \mathcal{V}[\![\tau_1]\!]_\ell \qquad (v_2, w_2) \in \mathcal{V}[\![\tau_2]\!]_\ell}{((v_1, v_2), (w_1, w_2)) \in \mathcal{V}[\![\tau_1 \times \tau_2]\!]_\ell}$$

$$\frac{(v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\ell}{(\mathsf{inl}\ v_1, \mathsf{inl}\ v_2) \in \mathcal{V}[\![\tau_1 + \tau_2]\!]_\ell} \qquad \frac{(v_1, v_2) \in \mathcal{V}[\![\tau_2]\!]_\ell}{(\mathsf{inr}\ v_1, \mathsf{inr}\ v_2) \in \mathcal{V}[\![\tau_1 + \tau_2]\!]_\ell}$$

$$\frac{\forall (v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\ell.\, (e_1[x \mapsto v_1], e_2[x \mapsto v_2]) \in \mathcal{E}[\![\tau_2]\!]_\ell}{(\lambda x.\, e_1, \lambda x.\, e_2) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]_\ell}$$

$$\frac{(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\ell}{(\mathsf{wrap}_{\ell'}\ v_1, \mathsf{wrap}_{\ell'}\ v_2) \in \mathcal{V}[\![\mathrm{T}_{\ell'}\ \tau]\!]_\ell} \qquad \frac{\ell' \not\sqsubseteq \ell \qquad v_1 \in \lfloor\tau\rfloor \qquad v_2 \in \lfloor\tau\rfloor}{(\mathsf{wrap}_{\ell'}\ v_1, \mathsf{wrap}_{\ell'}\ v_2) \in \mathcal{V}[\![\mathrm{T}_{\ell'}\ \tau]\!]_\ell}$$

Note that, while these are presented as inference rules to make them easier to read, this is *not* an inductive definition! The logical relation is defined by recursion over the type constructors. For sum types and tagged types, there are two rules, meaning the relation $\mathcal{V}[\![\tau_1 + \tau_2]\!]_\ell$ is the union of the relations defined by the two rules, and similarly for $\mathcal{V}[\![\mathrm{T}_\ell\ \tau]\!]_\ell$.

One really interesting property about this logical relation is that it forms a *partial equivalence relation* (PER). A PER is a binary relation that is symmetric and transitive, but not necessarily reflexive. Notably, if $a \equiv b$ and $\equiv$ is a PER, then $b \equiv a$ by symmetry and $a \equiv b \equiv a$ by transitivity. That means that a PER behaves like an equivalence relation (including reflexivity) on any values that relate to *anything*, but some values may be unrelated to anything (including themselves). Checking symmetry and transitivity of the indistinguishability relation above is straightforward.

Proving that it is not reflexive comes down to looking at the definition for the function case coupled with the semantics for unwrap. A function that takes a wrapped value, unwraps it, will not be related to itself. There exists a pair of inputs (two different values that have been wrapped) that are indistinguishable, but produce a distinguishable result when fed into the function. Indeed, proving that a function is noninterfering is equivalent to proving that it is indistinguishable from itself!

4 Enforcing Noninterference

There are a variety of ways to enforce noninterference, but we will focus on a type-based approach. That is, we will give our calculus a type system such that every well-typed program is noninterfering. Most of the typing rules are unmodified from STLC with sums and pairs—this is one of the benefits of our coarse-grained IFC approach. However, we still need typing rules for wrap and unwrap. The typing rule for wrap is straightforward.

$$[\textsc{Wrap}]\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{wrap}_\ell\ e : \mathrm{T}_\ell\ \tau}$$

Things are a bit more complicated for unwrap, however. We said in Section 3 that the type system was responsible for ensuring that data is used in accordance with its security label. Since unwrap is the language construct that removes the labels and allows computation on the values, the typing rule for it must enforce the security policies. In particular, the typing rule should require that computation using data with label ℓ should produce output that respects (at least) the restrictions specified by ℓ.

We could accomplish that goal by requiring the output type be tagged with label ℓ (or some label that ℓ flows to). That would be safe, but unnecessarily restrictive. For instance, unit values carry no information, so there is no need to wrap them. Similarly, pair values already enforce any restrictions enforced by both types—the type $(\mathrm{T}_{\ell_1}\ \tau_1) \times (\mathrm{T}_{\ell_2}\ \tau_2)$ enforces all restrictions of $\ell_1 \sqcup \ell_2$. We therefore define a new relation, know as a *protection relation* relating labels and types: $\ell \triangleleft \tau$. This relation means that values of type τ will respect (at least) the restrictions specified by ℓ. It is defined by induction on the type as follows.

$$\frac{}{\ell \triangleleft ()} \qquad \frac{\ell \sqsubseteq \ell'}{\ell \triangleleft \mathrm{T}_{\ell'}\ \tau} \qquad \frac{\ell \triangleleft \tau}{\ell \triangleleft \mathrm{T}_{\ell'}\ \tau} \qquad \frac{\ell \triangleleft \tau_1 \qquad \ell \triangleleft \tau_2}{\ell \triangleleft \tau_1 \times \tau_2} \qquad \frac{\ell \triangleleft \tau_2}{\ell \triangleleft \tau_1 \to \tau_2}$$

The first, second, and fourth rules are straightforward. The third says that, if τ already includes all restrictions of ℓ, then adding more is safe. The last rule recognizes that functions can only be used by applying them, so they inherently enforce all restrictions of their outputs.

Note that there is no protection rule for $\tau_1 + \tau_2$. That is because it doesn't protect any label at all! Simply knowning which side of the sum we are on is information, so it must be wrapped in a label to satisfy any policy. A protection rule for $\tau_1 + \tau_2$ would be like a protection rule for an unwrapped boolean.

This new protection relation is now sufficient to define the typing rule for unwrap.

$$[\textsc{Unwrap}]\ \frac{\Gamma \vdash e_1 : \mathrm{T}_\ell\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \ell \triangleleft \tau_2}{\Gamma \vdash \mathsf{unwrap}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

4.1 Examples Revisited

We can use this new understanding to look back at the examples from Section 3.1 to see how the typing rules rule out interfering programs, while allowing their secure counterparts.

Recall the first (secure) example: $\lambda h.\, 6$. In this case, we never unwrap—or otherwise examine—the secret input h, so the new typing rules do not impede us. However, consider the *insecure* example

$$\lambda h.\, \mathsf{unwrap}\ x = h\ \mathsf{in}\ x$$

where we assumed $h : \mathrm{T}_{\mathbf{H}}\ \mathsf{int}$. In this case, the Unwrap rule requires that the output type—here int—protect the label of the unwrapped value—here **H**. That is, it requires $\mathbf{H} \triangleleft \mathsf{int}$. However, there is no protection rule for raw integers, so this protection relation does not exist, meaning the term is not well-typed—exactly what we were hoping for.

Similarly, the example of the more complicated implicit flow suffers from the same flaw. By unwrapping both high and low arguments and multiplying them together, the two uses of Unwrap require the return type τ to satisfy $\mathbf{H} \triangleleft \tau$ and $\mathbf{L} \triangleleft \tau$, respectively. We again have $\tau = \mathsf{int}$, so those protections do not hold and the program, correctly, fails to type check.

Finally, we can look at the implicit flow case:

$$\begin{aligned}\lambda h.\, \mathsf{unwrap}\ x = h\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\\ \mathsf{inl}(\_).\ \mathsf{wrap}_{\mathbf{L}}\ 0\\ \mid \mathsf{inr}(\_).\ \mathsf{wrap}_{\mathbf{L}}\ 1\end{aligned}$$

where $h : \mathrm{T}_{\mathbf{H}}\ (\mathsf{unit} + \mathsf{unit})$. Again, UNWRAP requires $\mathbf{H} \triangleleft \tau$ for the return type τ. The [MATCH] rule (see Lecture 24) says that the return type of a match statement is the return type of each of its branches (which must be the same as each other). Here, both branches return a value of type $\mathrm{T}_{\mathbf{L}}\ \mathsf{int}$. Inspecting the protection rules tell us that $\mathbf{H} \triangleleft \mathrm{T}_\ell\ \tau$ if and only if either $\mathbf{H} \sqsubseteq \ell$ or $\mathbf{H} \triangleleft \tau$. Since neither of these is the case when $\ell = \mathbf{L}$ and $\tau = \mathsf{int}$, this program again fails to type check.

Notably, a secure version of this program could return its output wrapped at label $\mathbf{H}$ instead of $\mathbf{L}$. The outputs would then be indistinguishable to a low observer, so we would hope our type system would allow such a program. And indeed, it does. If the output type were changed to $\mathrm{T}_{\mathbf{H}}\ \mathsf{int}$, then the second protection rule proves that $\mathbf{H} \triangleleft \mathrm{T}_{\mathbf{H}}\ \mathsf{int}$, and the program is well-typed.

4.2 Proving Noninterference

To prove our language enforces noninterference, we prove the fundamental theorem of our logical relation. As with normalization, we cannot directly prove that $\vdash e : \tau$ implies $e \in \mathcal{E}[\![\tau]\!]_\ell$, but instead we must include a substitution of the type of the context. Because we have a binary logical relation instead of a unary one, we instead extend $\mathcal{V}[\![\cdot]\!]_\ell$ to contexts to be a *binary* relation on substitutions.

$$\begin{aligned}(\gamma_1, \gamma_2) \in \mathcal{V}[\![\Gamma]\!]_\ell \iff{}& \mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma_1) = \mathrm{dom}(\gamma_2) \\ &\wedge\ \forall x \in \mathrm{dom}(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\Gamma(x)]\!]_\ell\end{aligned}$$

From here, we can state the fundamental theorem.

Theorem 1 (Fundamental Theorem)**.** *If $\Gamma \vdash e : \tau$, then for any γ_1 and γ_2 such that $(\gamma_1, \gamma_2) \in \mathcal{V}[\![\Gamma]\!]_\ell$, $(\gamma_1(e), \gamma_2(e)) \in \mathcal{E}[\![\tau]\!]_\ell$.*

Proving this theorem requires three simple lemmas. The first two should be familiar from the proof of strong normalization of STLC. The third formalizes the intuition that the protection relation works properly. That is, any two values of protected types are indistinguishable to someone who cannot see the protected label. The proofs of all three lemmas are left as exercises.

Lemma 1 (Substitution)**.** *If $\Gamma \vdash e : \tau$ and $(\gamma_1, \gamma_2) \in \mathcal{V}[\![\Gamma]\!]_\ell$, then $\gamma_i(e) \in \lfloor \tau \rfloor$ for both $i = 1, 2$.*

Lemma 2 (Preservation of Indistinguishability)**.** *If $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$ and $e_i \longrightarrow^* e_i'$, for both $i = 1, 2$, then $(e_1, e_2) \in \mathcal{E}[\![\tau]\!]_\ell$ if and only if $(e_1', e_2') \in \mathcal{E}[\![\tau]\!]_\ell$.*

Lemma 3 (Hidden Values)**.** *If $\vdash v_1 : \tau$ and $\vdash v_2 : \tau$ and $\ell' \triangleleft \tau$, then for any ℓ, either $\ell' \sqsubseteq \ell$ or $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\ell$.*

Proof of Theorem 1. This is a proof by induction on the derivation of $\Gamma \vdash e : \tau$.

Cases UNIT **and** INT**:** Here $e = ()$ or n, so $\gamma_1(e) = \gamma_2(e) = e$, and the inclusion in $\mathcal{V}[\![\tau]\!]_\ell$ is by definition.

Case VAR**:** By the definition of $(\gamma_1, \gamma_2) \in \mathcal{V}[\![\Gamma]\!]_\ell$.

Case APP**:** Here $e = e_1\ e_2$, and $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$. By induction,

$$(\gamma_1(e_1), \gamma_2(e_1)) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]_\ell \qquad \text{and} \qquad (\gamma_1(e_2), \gamma_2(e_2)) \in \mathcal{E}[\![\tau_1]\!]_\ell.$$

By definition of $\mathcal{E}[\![\cdot]\!]_\ell$, that means if $\gamma_i(e_1) \longrightarrow^* f_i$ and $\gamma_i(e_2) \longrightarrow^* v_i$ for both $i = 1, 2$, then both $(f_1, f_2) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]_\ell$ and $(v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\ell$. By inspection on the definition of $\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\ell$, we know that $f_i = \lambda x.\ e_i'$ for both $i = 1, 2$ and that $(e_1'[x \mapsto v_1], e_2'[x \mapsto v_2]) \in \mathcal{E}[\![\tau_2]\!]_\ell$. Since

$$\gamma_i(e_1\ e_2) = \gamma_i(e_1)\ \gamma_i(e_2) \longrightarrow^* f_i\ v_i \longrightarrow e_i'[x \mapsto v_i]$$

for both $i = 1, 2$, Lemma 2 finishes the case.

Case ABS**:** Here we have $e = \lambda x.\, e'$ and $\tau = \tau_1 \to \tau_2$. We need to show that for any $(v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\ell$, that applying $\gamma_i(e)$ to v_i produces indistinguishable results.

Let $\gamma_i' = \gamma_i[x \mapsto v_i]$ for both $i = 1, 2$. Using the same argument as in the ABS case of the normalization proof in Lecture 31,

$$\gamma_i(\lambda x.\, e')\ v_i \longrightarrow \gamma_i'(e').$$

Moreover, we also have $(\gamma_1', \gamma_2') \in \mathcal{V}[\![\Gamma, x{:}\tau_1]\!]_\ell$. Therefore, by induction, $(\gamma_1'(e'), \gamma_2'(e')) \in \mathcal{E}[\![\tau_2]\!]_\ell$. A single-step application of Lemma 2 completes the case.

Case WRAP**:** Here $\tau = \mathrm{T}_{\ell'}\ \tau'$ and $e = \mathsf{wrap}_{\ell'}\ e'$. By induction, $(\gamma_1(e'), \gamma_2(e')) \in \mathcal{E}[\![\tau']\!]_\ell$. That means, if they step to values v_1 and v_2, then $(v_1, v_2) \in \mathcal{V}[\![\tau']\!]_\ell$, so $(\mathsf{wrap}_{\ell'}\ v_1, \mathsf{wrap}_{\ell'}\ v_2) \in \mathcal{V}[\![\mathrm{T}_{\ell'}\ \tau']\!]_\ell$. By the definition of $\mathcal{E}[\![\mathrm{T}_{\ell'}\ \tau']\!]_\ell$, this completes the case.

Case UNWRAP**:** Here $e = (\mathsf{unwrap}\ x = e_1\ \mathsf{in}\ e_2)$.

Sub-case $\ell' \sqsubseteq \ell$**:** The induction hypothesis for the premise $\Gamma \vdash e_1 : \mathrm{T}_{\ell'}\ \tau_1$ of UNWRAP proves that $(\gamma_1(e_1), \gamma_2(e_1)) \in \mathcal{E}[\![\mathrm{T}_{\ell'}\ \tau_1]\!]_\ell$, meaning $\gamma_i(e_1)$ evaluates to $\mathsf{wrap}_{\ell'}\ v_i$ for both $i = 1, 2$, and

$$(\mathsf{wrap}_{\ell'}\ v_1, \mathsf{wrap}_{\ell'}\ v_2) \in \mathcal{V}[\![\mathrm{T}_{\ell'}\ \tau_1]\!]_\ell.$$

Unfolding the definition of $\mathcal{E}[\![\cdot]\!]_\ell$, it suffices to show that

$$((\gamma_1 - \{x\})(e_2)[x \mapsto v_1], (\gamma_2 - \{x\})(e_2)[x \mapsto v_2]) \in \mathcal{E}[\![\tau_2]\!]_\ell.$$

Note that, as before, $(\gamma_i - \{x\})(e_2)[x \mapsto v_i] = (\gamma_i[x \mapsto v_i])(e_2)$.

Because $\ell' \sqsubseteq \ell$, inversion on the fact that $(\mathsf{wrap}_{\ell'}\ v_1, \mathsf{wrap}_{\ell'}\ v_2) \in \mathcal{V}[\![\mathrm{T}_{\ell'}\ \tau_1]\!]_\ell$ proves that $(v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\ell$, proving $(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in \mathcal{V}[\![\Gamma, x{:}\tau_1]\!]_\ell$. Induction on the original typing derivation thus proves

$$((\gamma_1[x \mapsto v_1])(e_2), (\gamma_2[x \mapsto v_2])(e_2)) \in \mathcal{E}[\![\tau_2]\!]_\ell,$$

completing the sub-case.

Sub-case $\ell' \not\sqsubseteq \ell$**:** For both $i = 1, 2$, Lemma 1 gives us that $\gamma_i(e) \in \lfloor\tau\rfloor$, and therefore type preservation proves that if $\gamma_i(e) \longrightarrow^* v_i$, then $v_i \in \lfloor\tau\rfloor$. Lemma 3 and the existing assumptions that $\ell' \not\sqsubseteq \ell$ and $\ell' \triangleleft \tau$ combine to show $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\ell$, as needed.

Case PAIR**:** By induction, the first and second components of $\gamma_1(e)$ and $\gamma_2(e)$ are related by $\mathcal{E}[\![\tau_1]\!]_\ell$ and $\mathcal{E}[\![\tau_2]\!]_\ell$, respectively. Therefore, if they step to values, the components of those values must be related by $\mathcal{V}[\![\tau_1]\!]_\ell$ and $\mathcal{V}[\![\tau_2]\!]_\ell$ as well. The definition of $\mathcal{V}[\![\tau_1 \times \tau_2]\!]_\ell$ completes the case.

Case PROJ**:** By induction, using a similar argument as the previous case.

Cases INL **and** INR**:** By induction, using a nearly identical argument to the PAIR case.

Case MATCH**:** By induction, using the definition of $\mathcal{V}[\![\tau_1 + \tau_2]\!]_\ell$ to ensure that both executions take the same branch of the match statement, and then using the same logic as the sub-case of UNWRAP where $\ell' \sqsubseteq \ell$ from there. □

To avoid the need for complicated logical relations, noninterference theorems are often phrased in a simpler and more intuitive form: if inputs are not visible, outputs are, and the outputs have an easily-comparable type (e.g., int or unit + unit), then the running the program with different inputs must produce identical outputs. This formulation follows as a corollary from Theorem 1 above.

Corollary 1 (Noninterference)**.** *Assume* $x{:}\tau \vdash e : \mathrm{T}_\ell\ \mathsf{int}$ *where* $\ell' \triangleleft \tau$*. Let* v_1, v_2 *such that* $\vdash v_1 : \tau$ *and* $\vdash v_2 : \tau$*, and* $e[x \mapsto v_1] \longrightarrow^* w_1$ *and* $e[x \mapsto v_2] \longrightarrow^* w_2$*. Then either* $\ell' \sqsubseteq \ell$ *or* $w_1 = w_2$*.*

Proof. By Lemma 3, either $\ell' \sqsubseteq \ell$, in which case we are done, or $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\ell$. Theorem 1 then guarantees that $(e[x \mapsto v_1], e[x \mapsto v_2]) \in \mathcal{E}[\![\mathrm{T}_\ell\ \mathsf{int}]\!]_\ell$. Unfolding the definition, this means $(w_1, w_2) \in \mathcal{V}[\![\mathrm{T}_\ell\ \mathsf{int}]\!]_\ell$, which, by inspection, is only possible when $w_1 = w_2$. □

There are several other ways to define noninterference, other ways to enforce it, and many ways to prove it. If you are interested, there is a relatively large body of literature on various IFC systems with different structures, goals, definitions, and proof techniques.

References

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, January 1999. doi: 10.1145/292540.292555.

Kenneth J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp, Bedford, MA, 1977. URL https://apps.dtic.mil/sti/pdfs/ADA039324.pdf.

Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *21st ACM Conference on Computer and Communication Security (CCS '14)*, November 2014. doi: 10.1145/2660267.2660343.

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *41st IEEE Symposium on Security and Privacy (S&P '20)*, May 2020. doi: 10.1109/SP40000.2020.00011.

Mae Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, June 2018. doi: 10.1145/3192366.3192375.

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *29th USENIX Security Symposium (USENIX Security '20)*, August 2020.

Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *32nd IEEE Computer Security Foundations Symposium (CSF '19)*, June 2019. doi: 10.1109/CSF.2019.00026.

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, April 2008. URL https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/zeldovich/zeldovich.pdf.

Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW '05)*, June 2005. doi: 10.1109/CSFW.2005.16.