Lecture Notes on
Relaxing Noninterference

Matt Fredrikson

Carnegie Mellon University
Lecture 15

1 Introduction

In the previous two lectures, we have developed and studied a type system that enforces *non-interference* (Definition 1). Non-interference is a powerful information flow property, stipulating that variables assigned to designated "high security" classes have no influence over the values held in variables of lower-security classes. It is useful for establishing a number of security goals, such as *confidentiality* and *integrity*.

The former goal concerns the flow of secret information out of a program through channels that may be observable to untrusted entities, and is perhaps the most commonly-cited use of non-interference in security. The latter goal, integrity, concerns the flow of untrusted information into a program, and is a key property to establish when one wants to eliminate the possibility of untrusted entities influencing critical parts of the program.

Definition 1 (Non-interference). Let α be a program and Γ a type environment associating security labels to all of the variables in α. Then α satisfies non-interference under Γ if and only if executing α under L-equivalent states leads to final states that are also L-equivalent. More precisely,

$$\forall \omega_1, \omega_2.\omega_1 \approx_{\Gamma,\mathrm{L}} \omega_2 \wedge \langle \omega_1, \alpha \rangle \Downarrow \omega_1' \wedge \langle \omega_2, \alpha \rangle \Downarrow \omega_2' \rightarrow \omega_1' \approx_{\Gamma,\mathrm{L}} \omega_2' \tag{1}$$

where ω_1 and ω_2 range over the set of possible program states.

However, the stricture of non-interference can be problematic for many applications. Consider a program that authenticates users using passwords, using a database of credentials stored in local memory. It seems natural to apply non-interference to such a program to ensure that the secret credentials don't inadvertently leak out through public channels. A quick analysis of this proposal however indicates that it is doomed to

failure, because of the necessary implicit flow that informs users of whether the password they provided is correct.

Our intuition tells us that there should be a way to formalize the security properties of such programs, as they have been used for decades without serious issue[1]. In today's lecture, we will see one such approach for *relaxing* non-interference to allow for selected flows that have been deemed in advance to pose an insignificant threat to security.

2 Relaxing non-interference: a simple example

Let us continue with the example of a password-checking program, but because the programming language we work with only handles integer types, we will view passwords as personal identification numbers (PINs). To make things concrete, we will for now confine our discussion to the following program fragment that checks a user-provided "guess" against the value stored in a variable holding the correct PIN.

$$auth := 0; \texttt{if}(guess = pin)\, auth := 1 \tag{2}$$

This program uses the variable $auth$ to hold the result of the authentication check, and will be communicated to the user to inform them of whether they are granted access to the system. So $auth$ must be L-typed, as must $guess$ which we assume holds the value of a variable provided by the user. But pin should be H-typed, because we don't want the correct value leaking to users who haven't provided the correct PIN. What happens if we try to type this in the environment $\Gamma = (auth : \text{L}, guess : \text{L}, pin : \text{H})$? Obviously we can't do it, because the assignment to $auth$ in the H-typed scope will leak some information about pw.

Why can't we just type $auth$ as H? The point of an authentication routine like the one above is to determine when a user is among the list allowed to access the system. We must expect that some un-authentic users will attempt access, and when they do, it is unavoidable that our routine will let such users know that they have failed to authenticate. Thus, the value of auth needs to be typed L, or we will have no way of communicating this result back over a safe channel.

To address this problem in the special context of such authorization checks, in 1999 Dennis Volpano and Geoff Smith introduced a match expression into the information flow type system we've studied in this class [VS00]. They introduced a corresponding typing rule that allows the result of a match expression to flow to the class corresponding to the least of its given operands.

$$(\text{Match})\ \ \frac{\Gamma \vdash e : \ell_1 \quad \Gamma \vdash \tilde{e} : \ell_2}{\Gamma \vdash \texttt{match}(e, \tilde{e}) : \ell_1 \sqcap \ell_2}$$

Where $\ell_1 \sqcap \ell_2$ denotes the *greatest lower bound* of ℓ_1 and ℓ_2. Operationally, match just performs an equality test of its arguments. This makes it easy to rewrite our program

---

[1]There are indeed good arguments against using passwords for user authentication, but they mostly rest on the fact that everyday users tend to select poor passwords that are easy to guess. This is separate and orthogonal to the information flow security of password authentication programs.

from before.

$$auth := 0; \texttt{if}(\texttt{match}(guess, pin)) \, auth := 1 \qquad (3)$$

Now we see that the assignment to auth will happen in a low context, because $\Gamma \vdash$ match(guess, pw) : L $\sqcap$ H, and L $\sqcap$ H = L.

Understanding leakage. Is this a good idea? The programs that our new type system with Match verifies no longer need to satisfy noninterference. Now the question of whether a well-typed program protects secret information is more subtle, because we don't have the straightforward all-or-nothing definition of noninterference to rely on.

Although our intuition (and decades of experience) tells us that breaking noninterference in the way that many authorization checks do is probably fine, there may be other programs we could write in this new language that aren't so well-behaved. For example, we could write the following program:

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

If an attacker with unsavory goals can manage to run this program on the system holding pin, then the final value of guess will leak the *entire* contents of pin!

We know that adding match to our language makes it *possible* for a program to leak the contents of an H-typed variable into an L variable, *but perhaps this does not imply that doing so is feasible.* In this case, the attacker may need to let the program run for time $O(2^k)$, assuming pin is a k-bit secret. Intuitively it seems that this complexity is the best an attacker can do given match, but can we formalize this intuition to get a guarantee that bounds the attacker's complexity when attempting such a leak?

Essentially, we want to state a guarantee that any attacker who uses match to learn the secret value requires time exponential in the size of the secret. But there are a few subtleties we should think about when postulating this guarantee.

Deterministic vs. non-deterministic attackers. Although we've only talked about deterministic semantics for our language, it's good to be explicit about limitations when stating a formal guarantee. In particular, if we were to extend the language with non-deterministic commands, then keeping our relaxed rule for match would be a bad idea. The attacker could then non-deterministically choose a value for guess, compute the value of match(*guess*, *pin*), and if it returns true, learn the secret in constant time.

Distribution of secrets. You may already be familiar with the fact that users tend to pick bad passwords that are far easier to guess than a naive bound based on plain bit-length would suggest. Formally, we characterize this fact in terms of the probability distribution corresponding to passwords selected by users in the "real world". The fact that this distribution is not uniform, i.e., does not assign equal probability to all possible passwords, is what makes password-guessing easier than $O(2^k)$ in practice.

Formalizing real-world distributions is hard. To make our guarantee generalize to any type of secret, we won't make any assumptions about the distribution of

values for the secret, so our formal statement will need to be *qualitative* and *universal*. In other words, we'll characterize the complexity of an attacker who attempts to copy a secret from *any* distribution (i.e., universal), and we'll only worry about whether this attacker can succeed all the time or some of the time (i.e., qualitative).

Secret size. We assume that the secret value is stored in the memory of a real machine, so it must have a finite length of k bits, for some k. For any fixed k, there exists a polynomial-time attacker who can brute-force the secret using the program above. So our guarantee will need to refer to an attacker who attempts to learn a secret of *any* size in polynomial time.

Putting this all together, we can state our guarantee as the following theorem, due to Volpano and Smith [VS00].

Theorem 2. *Let* $\Gamma = (s : \text{H}, o : \text{L})$ *and* α *be a deterministic program such that:*

1. $\Gamma \vdash \alpha$ *in the type system that includes* `match`.

2. $\langle \omega, \alpha \rangle \Downarrow \nu$ *for* $\omega(s) = v, \omega(o) \neq v, \nu(o) = v$.

In other words, α *always succeeds at transferring the value stored in* s *at the initial state into* o *at the final state. Then there exists some state* $\hat{\omega}$ *where:*

1. $\hat{\omega}(s)$ *requires at least* k *bits to represent*

2. *Evaluating* $\langle \hat{\omega}, \alpha \rangle$ *results in greater than* $poly(k)$ *evaluations of* `match`.

In other words, the type system prevents polynomial-time attacks on some H *state.*

Proof. We'll begin with the intuition behind the proof. The only way for the value of s to influence the value of o is via calls to `match`, and each call either eliminates one possible value of s from the adversary's list of candidates to consider, or confirms the correct value. A k-bit variable encodes 2^k possible values, so we just need to choose k large enough that α can't make enough calls to `match` to distinguish between all the possible values.

Suppose that α runs in time $poly(k)$. Choose k large enough such that $2^k > poly(k)+1$. Note that α can only call `match` at most $poly(k)$ times, so there must be a pair of k-bit values $x \neq y$ such that α doesn't evaluate $\text{match}(o, s)$ in either state $\omega(o) = x, \omega(o) = y$. Then because α is deterministic, if α is to end in $\nu(o) = x$ when started in $\nu(s) = x$, it must also end in $\nu(o) = x$ when started in $\nu(s) = y$. Thus the assumption that α runs in $poly(k)$ is in conflict with the assumptions made of α in the theorem statement. $\square$

Question. *Theorem 2 makes a fairly specific, and one might argue narrow, statement about the complexity of attacking a secret using* `match`. *Are you convinced by this result? Could you make the result stronger without changing the operational semantics of* `match`? *How might you make small changes to* `match` *to allow for a stronger statement? Are there assumptions that one might make about an implementation of* `match` *(ideally ones that can be realized in practice) to strengthen the result?*

3 Declassification: A Taxonomy

The Match rule is a simple example of a *declassification* mechanism. In general, we can think of a richer space of mechanisms for making information flow protection less rigid than noninterference. One useful way of characterizing declassification mechanisms places them along several dimensions [SS09]: *what* information is released, *where* controls the release, *who* is allowed to see it, and *when* the release occurs.

What. The `match` construct only allows us to declassify one particular type of information: the result of an equality comparison between a (potentially) `H` and `L` variable. This is a type of *partial* information about the `H` variable, but we could imagine other forms of partial information we may want to release, such as other comparisons, aggregates, and samples.

Where. We might also imagine releasing information without sacrificing control over where it could eventually end up. The type system we have already discussed employs a security lattice to do this in one way; a partial order on lattice elements denotes how information is allowed to flow between variables (i.e., $\ell_1 \sqsubseteq \ell_2$ means that information in an ℓ_1 variable can flow to one typed ℓ_2, but not the other way around). Apart from controlling "where" in terms of levels, we can also think of doing so in terms of code location. By defining which parts of the code are allowed to read certain pieces of information, we can ensure that potentially untrusted parts aren't able to leak them any further.

Who. The information flow type system we've discussed doesn't explicitly differentiate between principals, but one could design a system that tracks who owns a particular piece of information. With this information, it would then be possible to specify that certain types of declassification are allowed when the owner of the data requests them, but not on behalf of any other users. These systems can be further extended with delegation for additional flexibility. However, one must be careful to ensure that such mechanisms can't be abused by attackers.

When. Time can play a nuanced role in declassification. In the example we discussed at the beginning of lecture, we essentially relied on an argument about timing to justify the safety of `match`. Namely, because the secret won't be leaked in polynomial time, we decided it was safe to release partial information. Another nuanced application of timing in declassification mechanisms might crop up in the form of a probabilistic guarantee, which states that a secret will only be released with some small probability. Essentially, this is an argument that secrets will be leaked infrequently (assuming the distribution used aligns well enough with realistic assumptions). Finally, timing can play a more obvious role, such as with policies that dictate the release of information relative to the occurrence of other events on the system. For example, a digital media retailer might use a policy which states that the DRM key for a particular title can be released to the user once payment has been confirmed.

We won't explore every dimension of this taxonomy in greater detail today, and will instead focus primarily on the **what**. However, you should consider approaches that we've already discussed, in addition to those we'll discuss later on, in the context of this taxonomy. Doing so will highlight the primary differences between different approaches to information protection and what they are trying to achieve, which can be helpful when trying to distinguish important high-level (and perhaps generally-applicable) ideas from incidental low-level details.

4 Formalizing Leakage

Attacker model.   The notion of observability is central to precisely defining an information flow attacker. We assume that the adversary is able to observe and influence certain aspects of the program and its execution, and we're interested in understanding exactly what the attacker can deduce about the secret parts of the initial program state.

- As we did before when we formalized noninterference, we'll define a security lattice $L = (SC, \sqsubseteq, \sqcup, \sqcap, \bot)$. To keep things simple, we'll use the two-point lattice $SC = \{L, H\}$ where $L = \bot \neq H$, so $L \sqsubseteq H$ and $H \not\sqsubseteq L$.

- We assume that the attacker knows the code of the program that is executing. It may seem that this gives our attacker quite a bit of power that may be unrealistic in some cases, but it frees us from needing to consider whether the attacker is able to learn this information using other means.

- We assume that there are two moments in time at which the attacker can make observations about the program state: before the program is run (i.e., the initial state ω), and after it finishes executing (i.e., the final state ω'). We won't worry about nonterminating programs for the time being, although termination behavior could be relevant to information flow.

- We associate the attacker with the lattice element L, and assume that the attacker can observe any L-typed variable in ω and ω'. We'll write ω_L and ω'_L to denote the portions of the initial and final states, respectively, that correspond only to the L variables.

- We provision our attacker with the ability to make arbitrary assignments to the L-typed variables in the initial state ω.

- Finally, we place no immediate constraints on the time or space complexity of the attacker. In certain situations we may attempt to characterize how powerful an attacker needs to be, in terms of time and space resources, but unless otherwise stated, we assume that the attacker has unlimited resources.

To summarize, our attacker has total knowledge of the program being executed, along with the ability to decide which values L-typed variables take in the initial state, and to observe the values of L-typed variables in the initial and final states. The attacker's goal is to determine which values the H-typed variables take in the initial state.

Feasible sets and indistinguishability. We've introduced the big-step operational semantics previously, characterizing it in terms of the following relations for expression and programs, respectively:

$$\langle \omega, e \rangle \Downarrow v \qquad\qquad \langle \omega, \alpha \rangle \Downarrow \omega'$$

Given a state ω mapping variables to values and expression e (resp. program α), evaluation results in a value v (resp. state ω'). When relating the results of different evaluations, it becomes unwieldly to use this notation, as it requires introducing new "temporary" variables to hold the result of each evaluation.

We'll simplify matters a bit by introducing a new notation:

$$\mathsf{Ev}(\omega, e) \qquad\qquad \mathsf{Ev}(\omega, \alpha)$$

$\mathsf{Ev}(\omega, e)$ (resp. $\mathsf{Ev}(\omega, \alpha)$) refers to the value (resp. state) obtained by evaluating e (resp. α) in ω.

The only way in which our attacker has to go about learning the H-typed values is by comparing their observations of the L-typed parts of the initial and final states, ω and $\mathsf{Ev}(\omega, \alpha)$ with their knowledge of the program's semantics, and deducing which values of the initial H variables are **feasible**, or consistent with her observations and the program semantics. Note that because programs are deterministic, once the intiial H values are known, the attacker can deduce them for any other point in the program's execution.

We formalize this idea by defining the **feasible set** given a program α, context Γ, and initial state ω, using the notation $\Omega_\Gamma(\alpha, \omega)$ for shorthand. This is nothing more than the set of initial states that *agree with the attacker's knowledge of the* L *values of the initial and final states*:

$$\Omega_\Gamma(\alpha, \omega) = \{\omega' \ : \ \omega' \approx_{\mathrm{L}} \omega \text{ and } \mathsf{Ev}(\omega', \alpha) \approx_{\mathrm{L}} \mathsf{Ev}(\omega, \alpha)\}$$

The feasible set characterizes all of the attacker's knowledge of what the initial state, and in particular the H part of the initial state (because the rest is known), could be given the available information. Intuitively, any state in the feasible set will lead to *exactly* the same observations in the initial and final states, and so the attacker has no way of determining which of these states the program actually started in. From the attacker's perspective, they are *indistinguishable* from each other.

Example 3. Let's go back to the `match` construct from earlier, and work out the feasible set. Suppose that we have a very simple program that consists of a single evaluation of `match`.

$$o := \mathtt{match}(l, h) \tag{4}$$

We'll assume that $\Gamma = (o : \mathrm{L}, l : \mathrm{L}, h : \mathrm{H})$, so the attacker can set the value of l to whatever they likes in the initial environment, and observe the value of o afterwards. The goal, of course, is to learn the value h holds in the initial state.

Suppose that we, playing the role of an attacker, choose to run the program in an environment where $\omega(l) = v$, and in the final environment ν observe that $\nu(o) = 0$. While this does not allow us to deduce the exact value of $\omega(h)$, we haven't come away

completely empty-handed, because we know that $\omega(h) \neq v$. We can deduce this by reasoning counterfactually, supposing what *would have* happened in either situation:

- In the case where $\omega(h) = v$, reasoning by our knowledge of the program and the operational semantics tells us that:

$$\langle \omega, \mathtt{match}(l, h) \rangle \Downarrow \nu \text{ where } \nu(o) = 1$$

In other words, if $\omega(h) = v$ in the initial state, then we would expect to see $\nu(o) = 1$ in the final state.

- In the case where $\omega(h) = v'$, where $v' \neq v$, reasoning by our knowledge of the program tells us that:

$$\langle \omega, \mathtt{match}(l, h) \rangle \Downarrow \nu \text{ where } \nu(o) = 0$$

This is exactly what we observed when we ran the program, so we conclude that $\omega(h) \neq v$.

So although we didn't learn the whole value of $\omega(\mathtt{h})$, we were able to rule exactly one potential value out as impossible given our observations. In this case, after running the program a single time in an initial state where $\omega(l) = v$, we end up with the feasible set:

$$\Omega_\Gamma(\alpha, \omega) = \{\omega' \ : \ \omega'(h) \neq v\}$$

On the other hand, by our reasoning above, if we had observed $\omega'(\mathtt{o}) = 1$, then we would have had:

$$\Omega_\Gamma(\alpha, \omega) = \{\omega' \ : \ \omega'(h) = v\}$$

The feasible sets and their corresponding deductions are depicted in the diagram below.
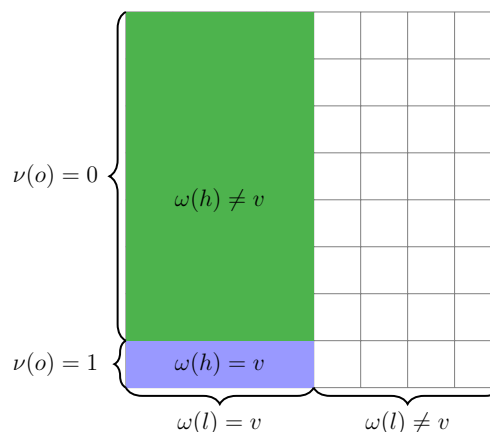
Each region of the diagram depicts a set of possible initial states, the notations on curly braces correspond to observations that the attacker can make, and the text inside each region corresponds to deductions that the attacker can make about $\mathtt{H}$-typed variables. Obviously, the attacker knows that $\omega(l) = v$, which allows elimination of the right half of the figure. After setting $\omega(l) = v$ in the initial state and observing $\nu(o) = 1$ in the final state, the attacker can conclude that $\omega(h) = v$, or $\omega(h) \neq v$ in the case where $\nu(o) = 0$. ∎

Example 4. Recall the definition of noninterference from an earlier lecture:

$$\forall \omega_1, \omega_2. \omega_1 \approx_\text{L} \omega_2 \wedge \langle \omega_1, P \rangle \Downarrow \omega_1' \wedge \langle \omega_2, P \rangle \Downarrow \omega_2' \implies \omega_1' \approx_\text{L} \omega_2'$$

In the notation introduced today, we can simply write:

$$\forall \omega_1, \omega_2. \omega_1 \approx_\text{L} \omega_2 \implies \mathsf{Ev}(\omega_1, P) \approx_\text{L} \mathsf{Ev}(\omega_2, P)$$

What is the feasible set for any program that satisfies noninterference? Looking at the definition, notice the universal quantifier which says that *all* L-equivalent initial states will lead to exactly the same final-state observation, so executing the program won't allow us to eliminate any more possible states from the feasible set than those not satisfying $\omega_I \approx_\text{L} \omega_\text{L}$. This leaves us with the set:

$$\Omega_\Gamma(\alpha, \omega) = \{\omega' \;:\; \omega' \approx_\text{L} \omega\}$$

which does not eliminate any possible values for H variables, so no information is leaked. ∎

Notice in the first example two very different feasible sets. In one case, where `match` returns 0, the attacker is left with many feasible values; in fact, only one value is eliminated, so there are $2^n - 1$ elements assuming n bits to represent the H part of ω.

In the second case, there is exactly one feasible element that remains, so the attacker has complete knowledge of the H state. The cardinality of the feasible set is one way of measuring the degree to which P leaks information about H state, quantitatively. Indeed, this is why we were able to prove the theorem from before: `match` does not give the attacker a reliable way of obtaining a small feasible set, so it is difficult to obtain much information about the H initial state.

Question. *What would happen if instead of allowing* `match`*, we allowed an expression* `compare`(a_1, a_2) *which returns 1 iff* $a_1 < a_2$*, to be given the type* $\ell_1 \sqcap \ell_2$*? Can the attacker reliably obtain feasible sets that are small enough to effeciently learn the secret?*

5 Explicit Declassification

Suppose that we extend our language with a more general-purpose declassification mechanism, by means of an expression called `declassify`. The expression `declassify` takes a single argument, which is another expression, and works as follows.

1. Operationally, `declassify` simply returns the value of its argument.

2. In the type system, `declassify` always types as the least element $\bot$, which in our running example two-label lattice is L, so it does not prevent leaking its value to any variable.

The typing logic is formalized in the rule below.

$$(\text{Declass}) \; \frac{}{\Gamma \vdash \texttt{declassify}(e) : \bot}$$

We could also consider generalizing even further, supporting a `declassify`$_\ell$ expression for each label ℓ in the lattice. The corresponding typing rule would be as shown below.

$$\text{(DeclassL)} \quad \frac{}{\Gamma \vdash \texttt{declassify}_\ell(e) : \ell}$$

To keep things simple, we'll stick with the former expression `declassify` for the rest of the lecture. Notice that this mechansim generalizes the sort of functionality we obtained from `match`.

$$auth := 0; \texttt{if}(\texttt{declassify}(guess = pin))\ auth := 1$$

But we can also do other useful things with it. For example, suppose we have a set of variables $s_1, \ldots, s_{100}$ containing the salaries of 100 employees. Before, if we wanted to extract any useful information from this data while still obtaining any degree of information security regarding its entire contents, there were few options. Noninterference wouldn't allow us to release an aggregate like the average, so our type system wouldn't allow the program to run. But if we wrap an expression computing the average in `declassify`, we are allowed to save the result to `L` variables.

However, this flexibility is sufficiently powerful that we can also do very bad things, like simply declassifying variable expressions containing secret data. Thus, when using `declassify`, it's important to understand what's being leaked. We can reason in terms of feasible sets and indistinguishability to figure this out.

Indistinguishability and declassification.   The `declassify` construct gives us an "escape hatch" through which we can selectively relax the stricture of noninterference. Citing again the definition of noninterference, let's think about what `declassify` changes:

$$\forall \omega_1, \omega_2. \omega_1 \approx_\text{L} \omega_2 \implies \textsf{Ev}(\omega_1, P) \approx_\text{L} \textsf{Ev}(\omega_2, P)$$

Noninterference says that whenever our initial states are indistinguishable on `L` variables, then our final states will be too. In correspondence with this notion, we assume that an attacker can observe all the `L` variables in both states.

With `declassify`, we make a different assumption. Namely, by typing all `declassify` expressions as `L`, we assume that the attacker is able to observe the value of all such expressions. In other words, given two states ω_1, ω_2:

- If their values differ in a way that can be observed through a `declassify` escape hatch, then we want to allow it.

- However, `declassify` doesn't allow *arbitrary* leaks. If there is a difference between the `H` variables in ω_1, ω_2 that can't be detected through the value of `declassify`, i.e. does not result in different values of the expressions wrapped in `declassify`, then the attacker won't be able to distinguish them.

Armed with this intuition, we can formalize the indistinguishability guarantee that `declassify` gives us. Namely, we can *weaken* noninterference by adding a condition

to the antecedent of the implication. For a program with a single `declassify` wrapping an expression e, perhaps we can say that:

$$\forall \omega_1, \omega_2. \omega_1 \approx_L \omega_2 \wedge \mathsf{Ev}(\omega_1, e) = \mathsf{Ev}(\omega_2, e) \implies \mathsf{Ev}(\omega_1, P) \approx_L \mathsf{Ev}(\omega_2, P) \qquad (5)$$

The property shown in Equation 5 is more subtle than it may seem. To see why, consider the following program.

$$
\begin{aligned}
&s_2 := s_1; \\
&s_3 := s_1; \\
&\vdots \\
&s_{100} := s_1 \\
&o := \texttt{declassify}((s_1 + s_2 + \cdots + s_{100})/100)
\end{aligned}
\qquad (6)
$$

When this program terminates, avg will contain exactly the initial value of s_1. However, given two initial states ω_1, ω_2 in which $\mathsf{Ev}(\omega_1, s_1 + \cdots + s_{100}) = \mathsf{Ev}(\omega_2, s_1 + \cdots + s_{100})$, it will not be the case that $\mathsf{Ev}(\omega_1, \alpha) = \mathsf{Ev}(\omega_2, \alpha)$.

The problem is due to the fact that the `H` variables appearing in the `declassify` expression were assigned before being used in the `declassify`, so that the declassified value differs from the indistinguishability condition in Theorem 5. So to ensure that the leakage characterized in Equation 5 aligns with our intuition of what `declassify` means, we need to require that certain variables not change from their initial values prior to their use in a `declassify`.

Theorem 5. *Let P be a deterministic program such that:*

1. *$\Gamma \vdash P$ in the type system that includes* **declassify**.

2. *P contains exactly one instance of* **declassify**(e), *over expression e.*

3. *None of the variables mentioned in e are assigned within the program before the* **declassify** *occurs.*

Then the following holds: $\forall \omega_1, \omega_2. \omega_1 \approx_L \omega_2 \wedge \mathsf{Ev}(\omega_1, e) = \mathsf{Ev}(\omega_2, e) \implies \mathsf{Ev}(\omega_1, P) \approx_L \mathsf{Ev}(\omega_2, P)$. *In other words, whenever the initial states are indistinguishable under the declassification expression, then the resulting final states will be indistinguishable as well.*

Proof. Proving this theorem is a good exercise that you should consider doing when preparing for the final exam. If you have trouble, ask the course staff for some hints or consult Sabelfeld and Myers [SM04]. $\square$

This theorem formalizes the intuition we developed above. The additional requirement imposed by (3) means that if we wanted to obtain the protection guaranteed by Theorem 5 using a type system, then we would need to design the rules so that *only* instances of `declassify` whose constituent variables have never been assigned can be given the type `L`.

References

[SM04] A. Sabelfeld and A. Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security*, 2004.

[SS09] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. In *Proceedings of the 18th IEEE Computer Security Foundations Symposium (CSF)*, 2009.

[VS00] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 2000.