15-316: Software Foundations of Security and Privacy **Spring 2017**

Lecture 17: Practical Information Flow: Relaxing Noninterference

Lecturer: Matt Fredrikson

17.1 Declassifying an Information Flow: A Familiar Example

Recall from the midterm exam the question about adding a typing rule for `match`, which potentially allowed typing the result of a equality test over a high variable as L. The motivation for such an allowance is apparent: the information flow type system we've discussed so far enforces noninterference, which is a strong and rigid guarantee that prevents *any* information leaking from H to L. If we insist on noninterference, then it's difficult to get anything useful done whenever H-typed data is involved.

It isn't hard to find examples of programs we'd like to write in an information flow-safe language, for which noninterference is too rigid a guarantee. One very important such class of programs are those that authenticate users based on a shared secret like an encryption key, or in an even more basic form, a password or PIN code. For example, consider a fragment of code that checks a secret PIN number against a user-provided guess.

```
auth := 0;
if guess == pin then
  auth := 1;
```

What happens if we try to type this in the environment $\Gamma = (\texttt{auth} :: \mathsf{L}, \texttt{guess} :: \mathsf{L}, \texttt{pw} :: \mathsf{H})$? Obviously, we can't do it, because the assignment to `auth` in the H-typed scope will leak some information about `pw`.

Why not just type `auth` 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 inauthentic 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 equality-based authorization checks, in 1999 Dennis Volpano and Geoff Smith introduced the **match** expression into the information flow type system we've studied in this class. The typing rule that they used was essentially the same one you provided for the midterm question:

$$\textsc{Match}\quad \frac{\Gamma \vdash a_1 : \ell_1 \quad \Gamma \vdash a_2 : \ell_2}{\Gamma \vdash \mathbf{match}(a_1, a_2) : \ell_1 \sqcap \ell_2}$$

Where $\ell_1 \sqcap \ell_2$ denotes the *greatest lower bound* of ℓ_1 and ℓ_2. This makes it easy to rewrite our program from before.

```
auth := 0;
if match(guess , pin) then
  auth := 1;
```

Now we see that the assignment to `auth` will happen in a low context, because $\Gamma \vdash \mathbf{match}(\texttt{guess}, \texttt{pw}) : \mathsf{L}$.

Understanding leakage. Is this a good idea? The programs that our new type system verifies don't necessarily satisfy noninterference. Now the question of whether a well-typed program protects secret information is much more subtle, because we don't have the straightforward all-or-nothing definition of noninterference to refer to.

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;
while !match(guess, pin) do
  guess := guess + 1;
```

If an attacker can manage to run this program on the system holding `pin`, then the final value of `guess` will leak the *entire* contents of `pin`! **Is this a problem?**

We know that adding **match** to our language makes it *possible* for a program to leak the contents of an H-typed variable into one typed L, 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 the correct 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 raw 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. 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 (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 [1].

Theorem 17.1 *Let* $\Gamma = (\mathsf{s} :: \mathsf{H}, \mathsf{o} :: \mathsf{L})$ *and* P *be a deterministic program such that:*

1. $\Gamma \vdash P$ *in the type system that includes* **match**.
2. $\langle [\mathsf{s} \mapsto v, \mathsf{o} \mapsto w], P\rangle \Downarrow [\mathsf{s} \mapsto u, \mathsf{o} \mapsto v]$ *for all* k*-bit values* v, w *and some value* u.

There exists an integer k *and values* v, w *such that evaluating* $\langle [\mathsf{s} \mapsto v, \mathsf{o} \mapsto w], P\rangle$ *results in greater than* $poly(k)$ *evaluations of* **match**.

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 P can't make enough calls to **match** to distinguish between all the possible values.

Suppose that P runs in time $poly(k)$. Choose k large enough such that $2^k > poly(k) + 1$. Note that P can only call **match** at most $poly(k)$ times, so there must be a pair of k-bit values $x \neq y$ such that P doesn't evaluate **match**(o, s) in either environment $[\mathsf{o} \mapsto x, \ldots], [\mathsf{o} \mapsto y, \ldots]$. Then if P is to end in $[\mathsf{o} \mapsto x]$ when started in $[\mathsf{s} \mapsto x]$, it must also end in $[\mathsf{o} \mapsto x]$ when started in $[\mathsf{s} \mapsto y]$ because it is deterministic. ■

17.2 Declassification: A Taxonomy

match is a relatively simple example of a **declassification** mechanism. In general, we can think of a richer space of potential ways of making information flow protection less rigid than noninterference. One useful way of characterizing declassification mechanisms places them along several dimensions: *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. Mechanisms that differ in this regard occupy different points in the "what" dimension.

Where. We can also release information so that we maintain control over where it may 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., $\ell_1 \leq \ell_2$ means that information in an ℓ_1 variable can flow to one types 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 (do any examples come to mind?). 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, such as occurred with the confused deputy attack from two lectures ago.

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 (entire) 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.

17.3 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 $L = (SC, \leq, \sqcup, \sqcap, \bot)$. To keep things simple, we'll use the two-point lattice $SC = \{\mathsf{L}, \mathsf{H}\}$ where $\mathsf{L} = \bot \neq \mathsf{H}$, so $\mathsf{L} \leq \mathsf{H}$ and $\mathsf{H} \not\leq \mathsf{L}$.
- 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 *any* observations about the program state: before the program is run (i.e., the initial state σ), and after it finishes executing (i.e., the final state σ'). 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 σ and σ'.
- We provision our attacker with the ability to make arbitrary assignments to the L-typed variables in the initial state σ.
- 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 arbitrary resources to spend.

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.

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

$$\langle \sigma, e \rangle \Downarrow v \qquad\qquad \langle \sigma, P \rangle \Downarrow \sigma'$$

Given a state σ mapping variables to values and expression e (resp. program P), evaluation results in a value v (resp. state σ'). 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:

$$\mathsf{Ev}(\sigma, e) \qquad\qquad \mathsf{Ev}(\sigma, P)$$

$\mathsf{Ev}(\sigma, e)$ (resp. $\mathsf{Ev}(\sigma, P)$) refers to the value (resp. state) obtained by evaluating e (resp. P) in σ.

Feasible sets and indistinguishability. The only way in which our attacker has to go about learning the H-typed values is by comparing her observations of the L-typed parts of the initial and final states, σ_{L} and σ'_{L} with her knowledge of the program's semantics, and deducing which values of the H variables are **feasible**, or consistent with her observations and the program semantics.

We'll formalize this idea by defining the **feasible set** given a program P, context Γ, and observations $\sigma_{\mathsf{L}}, \sigma'_{\mathsf{L}}$, using the notation $\Sigma_{P,\Gamma}(\sigma_{\mathsf{L}}, \sigma'_{\mathsf{L}})$ for shorthand. This is nothing more than the set of initial states that *agree with the attacker's knowledge*:

$$\Sigma_{P,\Gamma}(\sigma_{\mathsf{L}}, \sigma'_{\mathsf{L}}) = \{\sigma_I \mid \sigma_i \approx_{\mathsf{L}} \sigma_{\mathsf{L}} \wedge \Downarrow (\sigma_I, P) \approx_{\mathsf{L}} \sigma'_{\mathsf{L}}\}$$

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 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. In this way, they are **indistinguishable** from each other.

Example 17.2 *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 = match(l, h)
```

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

Suppose that we, playing the role of an attacker, choose to run the program in an environment where $\sigma(\mathtt{l}) = v$, and in the final environment σ' observe that $\sigma'(\mathtt{o}) = 0$. While this does not allow us to deduce the exact value of $\sigma(\mathtt{h})$, we haven't come away completely empty-handed, because we know that $\sigma(\mathtt{h}) \neq v$. We can deduce this by reasoning counterfactually, supposing what would have *happened in either situation:*

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

$$\langle \sigma, \mathbf{match}(\mathtt{l}, \mathtt{h}) \rangle \Downarrow [\mathtt{o} \mapsto 1, \mathtt{h} = v, \mathtt{l} = v]$$

  In other words, if $\sigma(\mathtt{h}) = v$ in the initial state, then we would expect to see $\sigma'(\mathtt{o}) = 1$ in the final state.

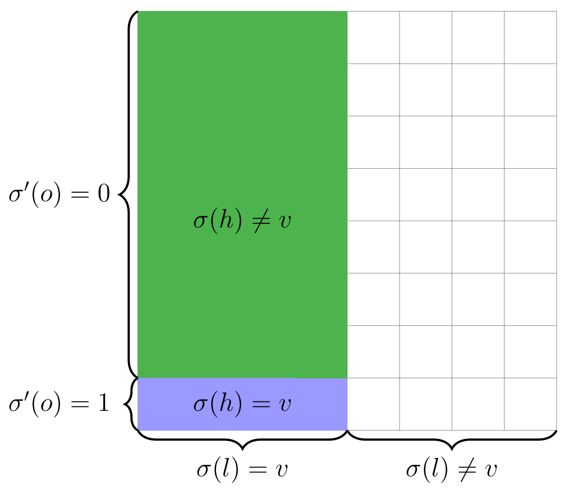

Figure 17.1: Graphic depiction of the attacker's knowledge of the initial state space after observing the result of a **match** evaluation. Each region of the diagram depicts a set of possible initial states, the notations on curly braces correspond to observations that the attacker can make, and the text inside each region corresponds to deductions that the attacker can make about H-typed variables. Obviously, the attacker knows that $\sigma(l) = v$, which allows her to eliminate the right half of the figure. After setting $\sigma(l) = v$ in the initial state and observing $\sigma'(o) = 1$ in the final state, the attacker can conclude that $\sigma(h) = v$, or $\sigma(h) \neq v$ in the case where $\sigma'(o) = 0$.

- *In the case where* $\sigma(\mathtt{h}) = v'$*, where* $v' \neq v$*, reasoning by our knowledge of the program and the operational semantics tells us that:*

  $$\langle \sigma, \mathbf{match}(\mathtt{l}, \mathtt{h})\rangle \Downarrow [\mathtt{o} \mapsto 0, \mathtt{h} = v', \mathtt{l} = v]$$

  This is exactly what we observed when we ran the program, so we conclude that $\sigma(\mathtt{h}) \neq v$*.*

So although we didn't learn the whole value of $\sigma(\mathtt{h})$*, 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* $\sigma(\mathtt{l}) = v$*, we end up with the feasible set:*

$$\Sigma_{P,\Gamma}(\sigma_{\mathsf{L}}, \sigma'_{\mathsf{L}}) = \{\sigma_I \mid \sigma_I(\mathtt{l}) = v \wedge \sigma_I(\mathtt{h}) \neq v\}$$

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

$$\Sigma_{P,\Gamma}(\sigma_{\mathsf{L}}, \sigma'_{\mathsf{L}}) = \{\sigma_I \mid \sigma_I(\mathtt{l}) = v \wedge \sigma_I(\mathtt{h}) = v\}$$

The feasible sets and their corresponding deductions are depicted in Figure 17.3. ■

Example 17.3 *Recall the definition of noninterference from an earlier lecture:*

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_{\mathsf{L}} \sigma_2 \wedge \langle \sigma_1, P\rangle \Downarrow \sigma'_1 \wedge \langle \sigma_2, P\rangle \Downarrow \sigma'_2 \implies \sigma'_1 \approx_{\mathsf{L}} \sigma'_2$$

In the notation introduced today, we can simply write:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_{\mathsf{L}} \sigma_2 \implies \mathsf{Ev}(\sigma_1, P) \approx_{\mathsf{L}} \mathsf{Ev}(\sigma_2, P)$$

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* $\sigma_I \approx_{\mathsf{L}} \sigma_{\mathsf{L}}$*. This leaves us with the set:*

$$\Sigma_{P,\Gamma}(\sigma_{\mathsf{L}}, \sigma'_{\mathsf{L}}) = \{\sigma_I \mid \sigma_I \approx_{\mathsf{L}} \sigma\}$$

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 an n-bit representation of the H part of σ.

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?*

17.4 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 L, so it does not prevent leaking its value to any variable.

This is formalized in the two rules given below. We give them the same name, although this should not lead to confusion as the left rule is for the operational semantics, and the right rule for the type system.

$$\textsc{Declass}\ \frac{\langle \sigma, e\rangle \Downarrow v}{\langle \sigma, \mathbf{declassify}(e)\rangle \Downarrow v} \qquad \textsc{Declass}\ \Gamma \vdash \mathbf{declassify}(e) : \mathsf{L}$$

Notice that this mechansim generalizes the sort of functionality we obtained from **match**.

```
auth := 0;
if declassify(guess = pin) then
  auth := 1;
```

But we can also do other useful things with it. For example, suppose we have a set of variables `s1`, . . . , `s100` 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 pass the program.

Now, with **declassify**, we can safely release statistics like the average.

```
avg := declassify((s1 + s2 + ... + s100)/100);
```

However, this flexibility is sufficiently powerful that we can also do very bad things, like simply declassifying variable expressions containing secret data.

```
avg := declassify(password);
```

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:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_{\mathsf{L}} \sigma_2 \implies \mathsf{Ev}(\sigma_1, P) \approx_{\mathsf{L}} \mathsf{Ev}(\sigma_2, P)$$

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 σ_1, σ_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 σ_1, σ_2 that can't be detected through the value of **declassify**, then the attacker won't be able to distinguish it.

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** over expression e, we can say that:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_{\mathsf{L}} \sigma_2 \wedge \mathsf{Ev}(\sigma_1, e) = \mathsf{Ev}(\sigma_2, e) \implies \mathsf{Ev}(\sigma_1, P) \approx_{\mathsf{L}} \mathsf{Ev}(\sigma_2, P) \tag{17.1}$$

Does this always hold? Consider the following example.

Example 17.4 *Consider the program in Figure 17.2. When this program terminates,* `avg` *will contain exactly the value of* `s1`*. However, given two initial states σ_1, σ_2 in which $\mathsf{Ev}(\sigma_1, \mathtt{s1} + \ldots + \mathtt{s100}) = \mathsf{Ev}(\sigma_2, \mathtt{s1} + \ldots + \mathtt{s100})$, it will not be the case that $\mathsf{Ev}(\sigma_1, P) = \mathsf{Ev}(\sigma_2, P)$. In fact, $\mathsf{Ev}(\sigma_1, P) = \mathsf{Ev}(\sigma_1, \mathtt{s1})$, and likewise for $\mathsf{Ev}(\sigma_2, P)$, and it could be the case that* `s1` *takes different values in these states (e.g., the values of $\mathtt{s1}, \ldots, \mathtt{s100}$ could be permuted in the two states).*

The problem is due to the fact that the H variables appearing in the ***declassify*** *expression were assigned before being used in* ***declassify****, so that the declassified value differs from the indistinguishability condition in Theorem 17.5.*

```
s2 := s1;
s3 := s1;
...
s100 := s1;
avg := declassify((s1 + s2 + ... + s100)/100);
```

Figure 17.2: Program that does not satisfy the leakage guarantee of Equation 17.1. We assume that Γ assigns H to s1, . . . , s100, and L to avg.

So, to ensure that the leakage characterized in Equation 17.1 holds, we need to require that certain variables not change from their initial values prior to their use in a **declassify**.

Theorem 17.5 *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:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_{\mathsf{L}} \sigma_2 \wedge \mathsf{Ev}(\sigma_1, e) = \mathsf{Ev}(\sigma_2, e) \implies \mathsf{Ev}(\sigma_1, P) \approx_{\mathsf{L}} \mathsf{Ev}(\sigma_2, P)$$

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 difficulty, ask the course staff for some hints or consult Sabelfeld and Myers [3]. ■

This theorem formalizes the intutition we developed above. The additional requirement imposed by (3) means that if we wanted to obtain the protection guaranteed by Theorem 17.5 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.

Question. *How would you go about designing such a type system, and formally proving that it provides the guarantee stated in Theorem 17.5?*

Further reading

[1] D. Volpano, and G. Smith, "Verifying secrets and relative secrecy," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2000.

[2] A. Sabelfeld, and D. Sands, "Declassification: Dimensions and Principles," in *Proceedings of the 18th IEEE Computer Security Foundations Symposium*, 2009.

[3] A. Sabelfeld, and A. Myers, "A Model for Delimited Information Release," in *Proceedings of the International Symposium on Software Security*, 2004.