Information Flow Type Systems — Limin Jia

Lecture 3 - *June 25, 2025*

1 Non-interference Proofs

We will extend our type system from the last lecture and introduce state and functions.

$$\begin{array}{rl} \textbf{values} & v ::= x \mid n \mid (f(x:\tau):pc = e)^{\ell} \\ \textbf{terms} & e ::= v \mid e\ e \mid e := e \mid \mathrm{ref}^{\tau}(e) \mid !e \mid \mathrm{let}\ x = e\ \mathrm{in}\ e \\ \textbf{types} & \tau ::= \mathrm{int}^{\ell} \mid \mathrm{ref}^{\ell}\tau \mid ([pc]\tau \to \tau)^{\ell} \end{array}$$

Here, τ is a type with a secrecy label. Note that

- Integers (n) have a label;
- References carry two labels, ℓ, which is for the reference itself (pointer label), and τ contains an additional label for the term which is being referenced.

For functions,

- both the input type (first τ) and return types (second τ) have a label, and the function as an object has a label (ℓ).
- we need a label for the context (pc) in which we can call the function.

Here are some inference rules to enforce the creation of only good information flow systems.

Allocate new location

$$\frac{\Gamma, pc \vdash e : \tau}{\Gamma, pc \vdash \mathrm{ref}^{\tau}(e) : \mathrm{ref}^{pc}\tau}$$

Create a function

$$\frac{\Gamma, x : \tau, f : ([pc]\tau \to s)^{\ell}, pc \vdash e : s}{\Gamma, pc' \vdash (f(x : \tau) : pc = e)^{\ell} : ([pc]\tau \to s)^{\ell}}$$

Function application The current PC, pc', is the context in which the function is being called. We need it to be a lower or equal level to pc, the context required by the function.

$$\frac{\Gamma, pc' \vdash e_1 : ([pc]\tau \to s)^{\ell} \qquad \Gamma, pc' \vdash e_2 : \tau \qquad pc' \sqcup \ell \sqsubseteq pc}{\Gamma, pc' \vdash e_1\ e_2 : s}$$

2 Downgrading

Sometimes, we need to leak secret information or endorse untrusted data. Some examples:

- Checking passwords, releasing aggregated data
  - Sometimes necessary for the application to work;
  - Other times, judged sufficiently safe to release (i.e. heatmaps).
- Using public APIs, downloading programs
  - Need to endorse untrusted data;
  - Imagine downloading a program from the internet. Most OS's will emit a warning telling the user that "this is an untrusted application" before letting you run. Users frequently will want to endorse that program and run it anyway.

To allow safe downgrading, we need to relax our requirements on non-interference.

Delimited Release Suppose we allow $x := \text{declassify}(e, \ell)$ in our code. We want a notion of non-interference that still holds in the presence of declassifications. With this in mind, we formally say that command c is secure as follows.

Suppose c contains $\text{declassify}(e_1, \ell_1), \ldots, \text{declassify}(e_n, \ell_n)$. Then, c is secure if $\forall \ell_a, \forall \sigma_1 \approx_{\ell_a} \sigma_2$,

$$\begin{aligned} &\text{if } (\forall i \in [1, n], [\![e_i]\!]_{\sigma_1} \approx_{\ell a} [\![e_i]\!]_{\sigma_2} : \ell_i), \text{ and} \\ &\quad (\sigma_1, c) \to^* (\sigma_1', skip) \text{ and } (\sigma_2, c) \to^* (\sigma_2', skip) \\ &\text{then } \sigma_1' \approx_{\ell a} \sigma_2'. \end{aligned}$$

Intuitively, this says that we do not leak more information than what we intended, i.e. the $e_1, \ldots, e_n$, and that we still maintain non-interference otherwise.

Example We can write

$$x = \text{declassify}(y \bmod 2, L)$$

This does not directly reveal the exact value of y, but it does reveal the parity of y. Now, the attacker has acquired some information about y, but is unable to discern between the two "bags" (even and odd).

3 Knowledge-based non-interference

Here is a more intuitive definition of non-interference, based on knowledge: we never want to allow the attacker to refine their knowledge of our secrets.

Let us define the attacker's knowledge as the set of all possible values of a variable from the perspective of an attacker. Knowledge-based non-interference means that we should prevent the attacker from reducing the space of possibilities (meaning the set of possible values after execution is a proper subset of the set before execution).

Here are some examples of failing strict non-interference. Suppose x is public and y is secret.

```
1 x := y
```

As soon as we do the assignment, the attacker immediately knows the value of y, so their knowledge is refined.

```
1 if y > 0 then x := 1 else x := 0
```

Now, the attacker's knowledge is refined because they know whether y is positive or negative. It's less refined than the previous case, which refines down to a single value. Neither of these examples satisfies strict non-interference. However, this is not always desirable; maybe we judge that it is acceptable to reveal some information related to our secrets.

```
1 x := declassify(y mod 4, L)
```

We *want* to allow this, since we declassified the information.

Possible formal definition of knowledge. Knowledge is the set of states that are indistinguishable from the current state, possibly after executing a command.

$$K[\![\sigma, c]\!] = \{\sigma' \mid \sigma \approx_{\ell a} \sigma', (\sigma, c) \to^* \sigma_f, (\sigma', c) \to^* \sigma'_f, \sigma_f \approx_{\ell a} \sigma'_f\}$$
$$K_{init}(\sigma) = \{\sigma' \mid \sigma \approx_{\ell a} \sigma'\}$$

Here, the security condition (for strict non-interference) is

$$K[\![\sigma, c]\!] \supseteq K_{init}(\sigma).$$

We can see that the security condition is violated when $K[\![\sigma, c]\!] \subset K_{init}(\sigma)$, i.e. when the attacker's knowledge is refined in the execution of command c.

4 Trace semantics

Additionally, we may want to output information. Using this as motivation, we extend the language with command output(e, ℓ). Now, a program execution produces an execution trace; each step produces a possibly empty action α where

$$\alpha ::= \varepsilon \mid \text{declassify}(e, \ell) \mid \text{output}(e, \ell)$$

Now, we define a program trace as a sequence of states along with the action:

$$T = (\sigma_0, c_0) \xrightarrow{\alpha_1} (\sigma_1, c_1) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} (\sigma_n, c_n)$$

The attacker can observe a program trace and possibly gain information from it.

Formally, an attacker's knowledge from an execution trace is characterized by

$$K[\![c, T]\!] = \{\sigma \mid \sigma_0 \approx_{\ell_a} \sigma, T \approx_{\ell_a} T'\},$$

where $T = (\sigma_0, c) \to^*$(rest of trace), and $T' = (\sigma, c) \to^*$(rest of trace).

Gradual release We would like to declassify some information. Thus, we define a notion of gradual release, which says the attacker cannot refine knowledge unless a release (i.e. declassify is invoked) happens. This is stated as

$$K[\![c, T \xrightarrow{\alpha} \Sigma]\!] \supseteq K[\![c, T]\!]$$

when $\alpha \neq \text{declassify}(-)$.

We don't want to allow the attacker to rule out possibilities really quickly. For the password checker, the attacker was allowed to rule out all passwords with a particular prefix, allowing it to refine its knowledge too quickly.

5 Examples

Let h be a highly sensitive variable and l a public variable.

```
l := h; output(l, L);
```

This (above) should be clearly bad, since we are trying to output l as low, revealing the value of h, which was never declassified.

```
l := h; l := 0; output(l, L)
```

```
l := declassify (h, L);
output (l, L);
```

```
l := declassify (h, L);
output (h, L);
```

The above three should be morally okay (though may be rejected in the type system).

```
b := declassify (h>0, L)
if (b) l:= l; output(l,L)
else l:= 0; output (l, L)
```

This one is OK, since the attacker is only able to refine h by half—whether it's greater than or less than 0—which is exactly what we explicitly declassified.

```
if (h > 0) l := 1; output(l, L)
else l := 0; output(l, L)
```

This one should be ruled out, since $h > 0$ hasn't been declassified.

6 Progress knowledge

```
while (h>0) do skip;
l := 1;
output(1, L)
```

Should we rule out this program or not? The knowledge the attacker can obtain is related to the possible early termination of the program when the while guard is not satisfied.

If we do want to rule it out, we can include it in the model of attacker knowledge. However, if we want to make it OK to leak knowledge in this manner, we can do that as well.

7 Robust declassification

```
debug:L := false
secret:H
x:L

setDebug(f:L) {debug := f}
main() {
    if (debug)
        x := declassify(secret, L)
    else
        skip
    output(x, L)
}
```

Who controls the declassification? This program is wrong because anyone is allowed to call `setDebug`. The attacker should not be able to influence declassification.

We want

$$K[\![T]\!] = K[\![T \setminus A]\!].$$

That is, the information determinable by the attacker shouldn't be affected by any actions the attacker could take.

In this example, we have

$$K[\![\text{setDebug}(\mathit{true}), \sigma]\!] = \{\text{secret} = n\},$$

and

$$K[\![\sigma]\!] = \{\text{secret} = 0, \text{secret} = 1, \dots\}.$$

They are not equal, so this program violates the condition that the attacker should not be able to influence declassification.

8 Runtime monitors

One advantage of a dynamic approach is that it can allow behaviors that might be impossible with a static approach. For example, while the program is running, the runtime monitor tracks whether we are in a high or low context, the current state of variables, and the actual program being executed. With the use of a runtime monitor, we can choose whether to taint variables or not, which means promoting low variables to high if necessary.

With tainting x is tainted because it is assigned to a high variable.

$$\boxed{\begin{array}{l} L \\ x^{L} \mapsto 1, y^{H} \mapsto 2, z^{L} \mapsto 3 \\ x := y \end{array}} \mapsto \boxed{\begin{array}{l} L \\ x^{H} \mapsto 2, y^{H} \mapsto 2, z^{L} \mapsto 3 \\ SKIP \end{array}}$$

x is tainted because it is assigned in a high context.

$$\boxed{\begin{array}{l} H \\ x^{L} \mapsto 1, y^{H} \mapsto 2, z^{L} \mapsto 3 \\ x := z \end{array}} \mapsto \boxed{\begin{array}{l} H \\ x^{H} \mapsto 3, y^{H} \mapsto 2, z^{L} \mapsto 3 \\ SKIP \end{array}}$$

Non-sensitive upgrade (NSU) With a strict runtime monitor, both programs would be aborted as we do not allow tainting.

Tainting might lead us to a situation known as PC creep, where we end up promoting too many things and getting stuck in a high context.

9 Implicit leaks

```
1 x^H
2 y^L := true;
3 z^L := true;
4 if (x) y := false;
5 if (y) z := false;
6 output(z,L)
```

In this example, we are leaking the value of x, even though we are not using an output or a declassification operation with x directly. Independent of the value of x, after execution, z, a low value, is the same value as x, a high value. Thus, we are leaking x, but actually, nothing technically wrong happened; a type checker is not able to detect that we are leaking information about x through the branch *not* taken.

10 Gradual typing

We can adopt both approaches (static and dynamic) simultaneously to achieve the best of both worlds by making typing by the user optional. A type system will only check parts that are typed, and for the untyped terms, refine the dynamic types during execution. If we face a type error during runtime, we abort the program.

Formally, we want parts of the program to be statically typed

$$x : \tau\ \ell,$$

and parts to be dynamically

$$x : \tau\ ?.$$

with typing rules

$$? \sqsubseteq {}_g\ell \qquad\qquad \ell \sqsubseteq {}_g?$$

The ? label means "all possible label intervals on the lattice." Gradually, we will refine the range that the label lives in. For example, we could have

$$x^H \mapsto (H, H)\text{true}, y^? \mapsto (L, H)\text{true}$$

After running

```
1 if x then y^? := true
```

then the type of y is refined to

$$y^? \mapsto (H, H)\text{true}.$$

Static gradual guarantee. Programs that type check with precise type annotations will also type check with less precise types.

Dynamic gradual guarantee. Programs that run to completion with precise type annotations will also run to completion with less precise types.

For example, if this more precise program type checks or runs to completion,

```
1 x^H := true;
2 if x then y^H := true;
```

then this less precise program will also type check or run to completion:

```
1 x^H := true;
2 if x then y^? := true
```

Example. Gradual systems are tricky. Consider the following program.

```
1 x^H
2 y^? := true^?;
3 z^L := true^L;
4 if (x) y := false;
5 if (y) z := false;
6 output(z,L)
```

Let's consider both cases of whether y is low or high. If y is low, this could be an implicit information leak. If y is high, then the assignment to z when branching on y is bad.

To modify this to work,

```
1 x^H
2 y^? := true^H;
3 if (x) y := false;
4 output(y,H)
```

Hybrid approach In a hybrid approach, we update the labels of the writes in both branches upon entering the branch. If the labels do not agree after joining both sides of the branch, reject. This approach satisfies gradual guarantees, but requires more effort from the monitor.

Consider the above code. In either branch, we need y to have a high label—if x is true, then the write requires y to be high, and if not then the `output(y, H)` requires y to be high. Thus, this should be valid.