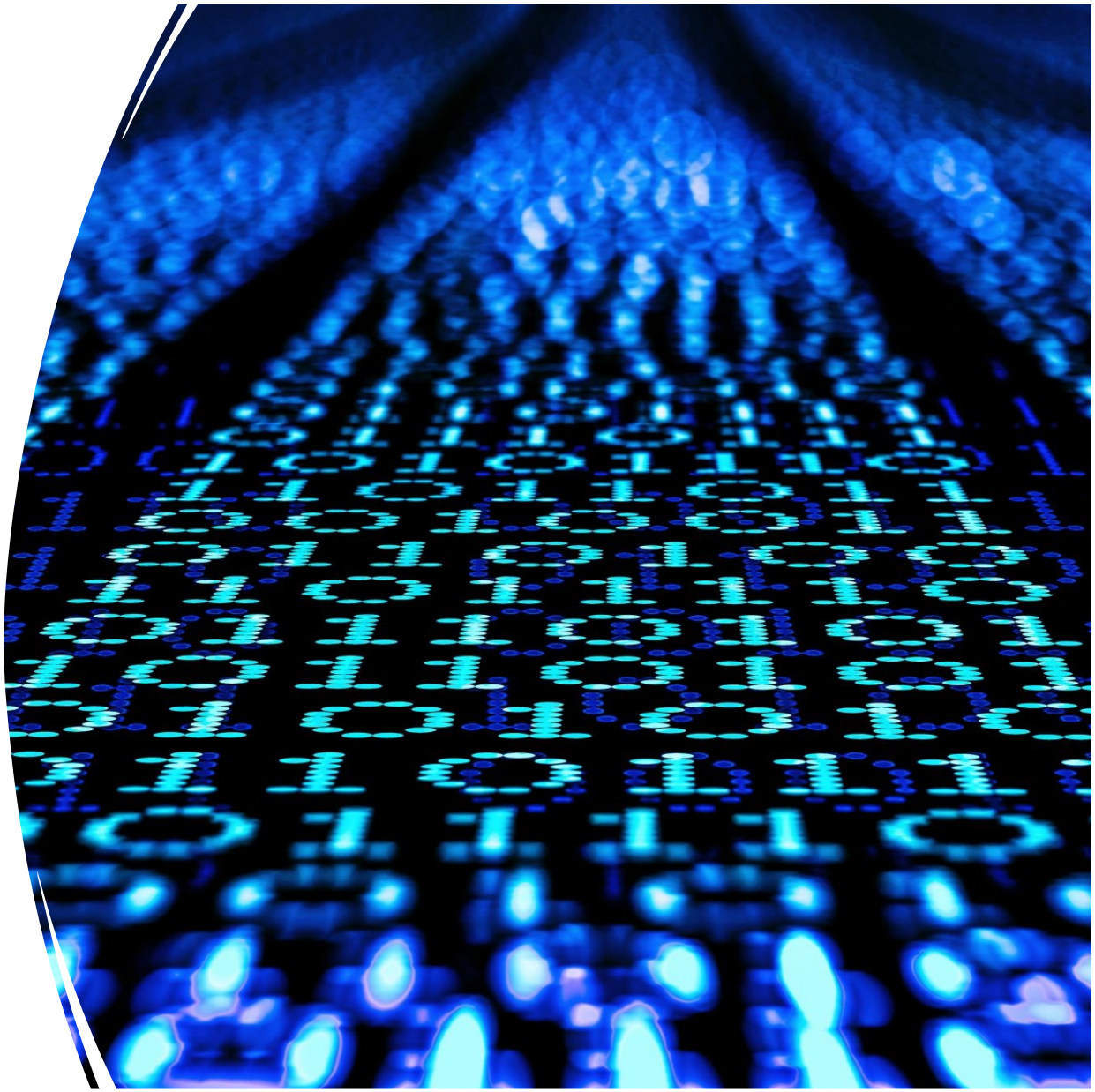
Dynamic Information Flow

Goal: decrease false positives over static mechanisms through the use of run-time information



Key Features

3 Key features

Interpreter of language has to check whether or not instructions are safe. We label data with a security level. Runtime mech. has to be designed in a way that tracks the flow of info and how it propagates. Then run time data structure has to have a mechanism to enforce policies to avoid violations

- Dynamic Information Flow tracks **security labels** attached to values during execution.
- **Labeling Data**:
  - Each value in memory or in a variable gets a security label.
- **Tracking Labels**:
  - As data flows through the program (assignments, arithmetic, control flow), the labels propagate.
- **Enforcing Policies**:
  - When a value labeled secret is about to be written to a public output, the system raises an error or blocks the operation.

8

Note: In static approach you have their associative between var → s.levels is fixed (mapped in Γ). Here label associated to variables can change.

Example

secret = "password" // **label**: **High**
x = 0 // **label**: **Low**

Operations we have to execute to ensure enforcement

if secret == "1234" then x = 1 // **tracking**: x inherits label **High**

print(x) // **Enforcement**: Attempt to print High data to Low output
// violation!

Runtime data structure checking labels and flow inside interpreter to check policy.

- Since X=1 maps x as high even if 1 is low, the runtime data structure we use needs to have notion of context to keep track of implicit flows.

Issues

1. We will have overhead if we check things at runtime
2. We need something in the runtime data structure that keeps track of implicit flow.
3. Statically, this was done for variables, but at runtime you have different granularities to choose!

- **Granularity**: Whether to track at the level of variables, objects, or memory addresses.
- **Implicit flows**: Data can flow via control structures (like if, for), not just direct assignments.
- **Performance overhead**: Runtime tracking can slow down execution.

Issues

our approach! Location, even functions in functional PL; you can also for objects assign labels to methods and fields.

- **Granularity**: Whether to track at the level of variables, objects, or memory addresses.
  - Every value in the program is assigned a **security label,** Low or High
- **Implicit flows**: Data can flow via control structures (like if, for), not just direct assignments.
  - Context We need to have a notion of context
- **Performance overhead**: Runtime tracking can slow down execution.
  - Execution Monitor

↳ We use EM: an entity in parallel executed with a program that only interact with security related events.

Execution monitor for Dynamic Information flow

Label Propagation: As the program executes, the monitor **tracks how labels flow** through assignments, function calls, and control structures.

If the current instruction is

y = x + 1 and x is H,

then y becomes H.

EM has to keep track of the flow of data based on the execution of code.

EM marks y as H, in parallel with the interpreter that runs values y location and puts x+1.

EM works at level of security level

Execution monitor for Dynamic Information flow

Control Flow Tracking (to handle implicit flows):
The monitor maintains the context cxt, the abstraction of the **program counter label,** that reflects the sensitivity of the current control context.

If a High value is used as a guard of the condtional , cxt becomes High, and all assignments in that branch are tagged accordingly.
When execution exits the conditional command, ctx is restored.

EM should have internal value to store the value of the context.
But we also need to restore an old value after executing an if then else. EM should work with a stack or similar structure to keep track of old values.

Execution monitor for Dynamic Information flow

Easiest thing: at a certain point you will have sensitive operations such as assignments. You check if operation is allowed based on policy, if not, ①

Enforcement (Checks):
Before sensitive operations (e.g., assignment or output to a Low channel), the monitor checks:

Is the value's label allowed to flow to the destination's label?

①

If not, the operation is **blocked**, or the program is **halted**.

Fixed or flow sensitive labels?

When an assignment **x := e** is executed the run-time monitor

- either check whether $\Gamma(\mathbf{e}) \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})$ holds (fixed Γ). • Security level is fixed once and for all for all prog. variables
- or deduce $\Gamma(\mathbf{x})$ such that $\Gamma(\mathbf{e}) \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})$ holds (flow-sensitive Γ). Or flow sensitivity ↘

Example considered before has labels associated to labels are dynamically evolving (we talk about flow sensitivity).

$\Gamma(e)$ returns security level of e, a bit of an abuse.

Assume a dynamic enforcement mechanism with fixed Γ, where $\Gamma(\mathtt{h}) = \mathrm{H}$ and $\Gamma(\mathtt{l}) = \mathrm{L}$.

Fixed mapping, h is high level value
l is low,
Fixed by security policy Γ,

```
if h=0 then h:=2 else h:=3;  l:=h
```

$ctx = \mathrm{L}$
We assume ctx is Low.

$ctx = \mathrm{H}$
Check: $ctx \sqcup \Gamma(2) \sqsubseteq \Gamma(\mathtt{h})$

Guard is H, so context becomes H

$ctx = \mathrm{L}$
Check: $ctx \sqcup \Gamma(\mathtt{h}) \sqsubseteq \Gamma(\mathtt{l})$

Execution blocks!

We restore previous ctx lvl after guard

Discussion

Under dynamic analysis the command

if 0=0 then l:=2 else l:=h

is always executed to completion,

1. because dynamic check **Γ(2) ⊔ Γ(0=0) ⊑ Γ(l)** always succeeds,
2. because branch **l:=h** is never taken.

The static type system rejects this program before execution, even though the program is secure!

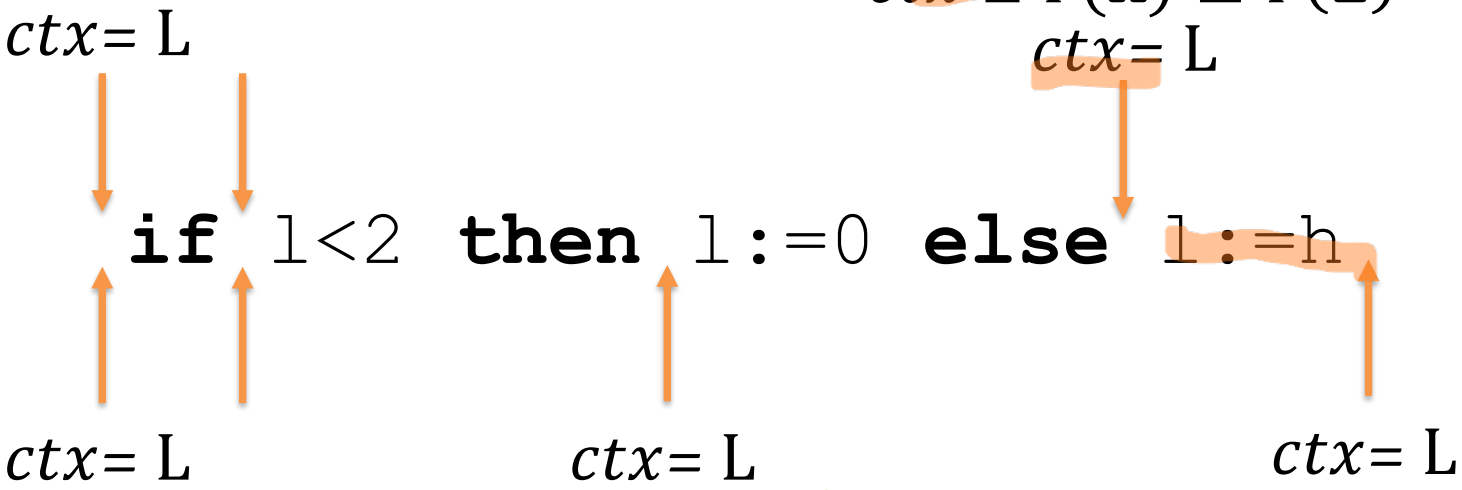
• Fixed policy again

Assume a dynamic enforcement mechanism with fixed Γ, where $\Gamma(\mathtt{h}) = \mathrm{H}, \Gamma(\mathtt{l}) = \mathrm{L}$.

Execution blocks!

Check: 👎

$ctx \sqcup \Gamma(\mathtt{h}) \sqsubseteq \Gamma(\mathtt{l})$



Check: 👍

$ctx \sqcup \Gamma(\mathtt{0}) \sqsubseteq \Gamma(\mathtt{l})$

For program

if l<2 then l:=0 else l:=h

If **l<2** holds, then command is executed to termination, because **Γ(0)⊔ Γ(l<2) ⊑ Γ(l)** succeeds.

If **l<2** does not hold, then command is blocked before executing **l:=h**, because **Γ(h)⊔ Γ(l<2) ⊑ Γ(l)** does not succeed.

Is this program accepted by the static type system?

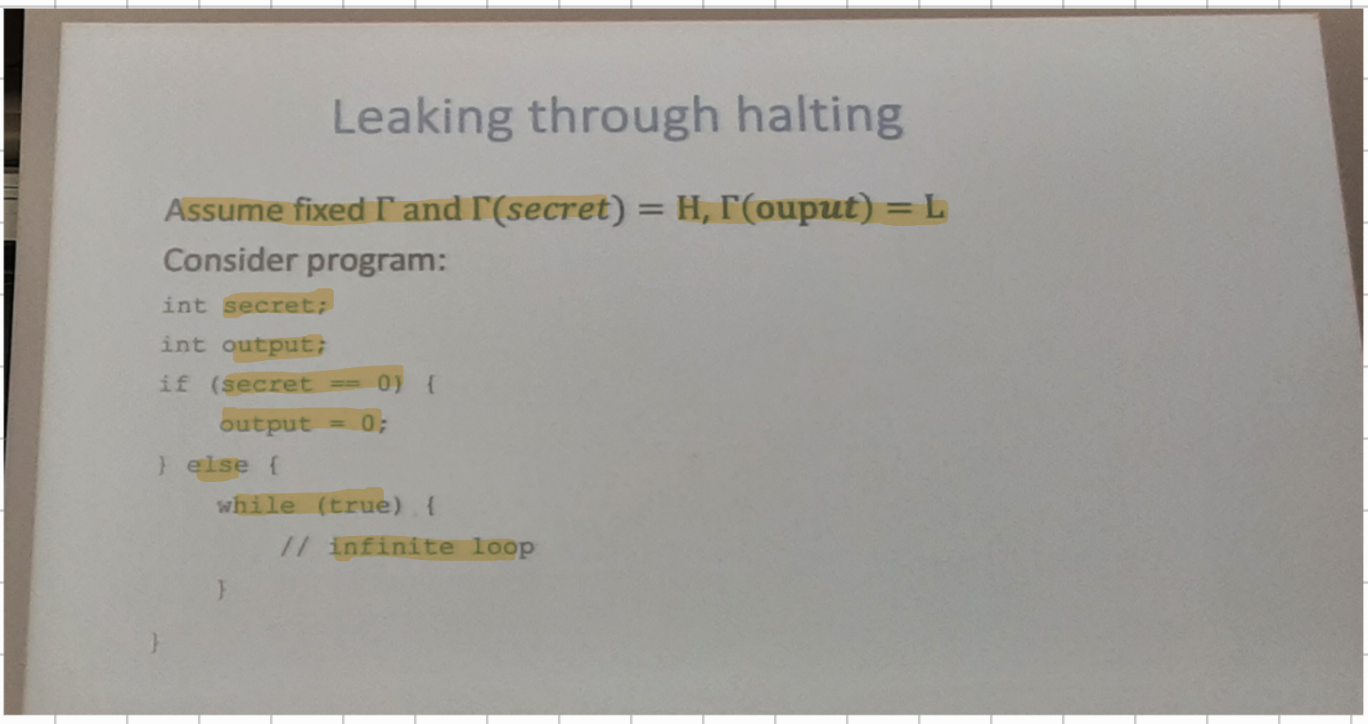
The static type system rejects this program before execution: all executions of this program are rejected.

We don't reject all of them, only some of them.

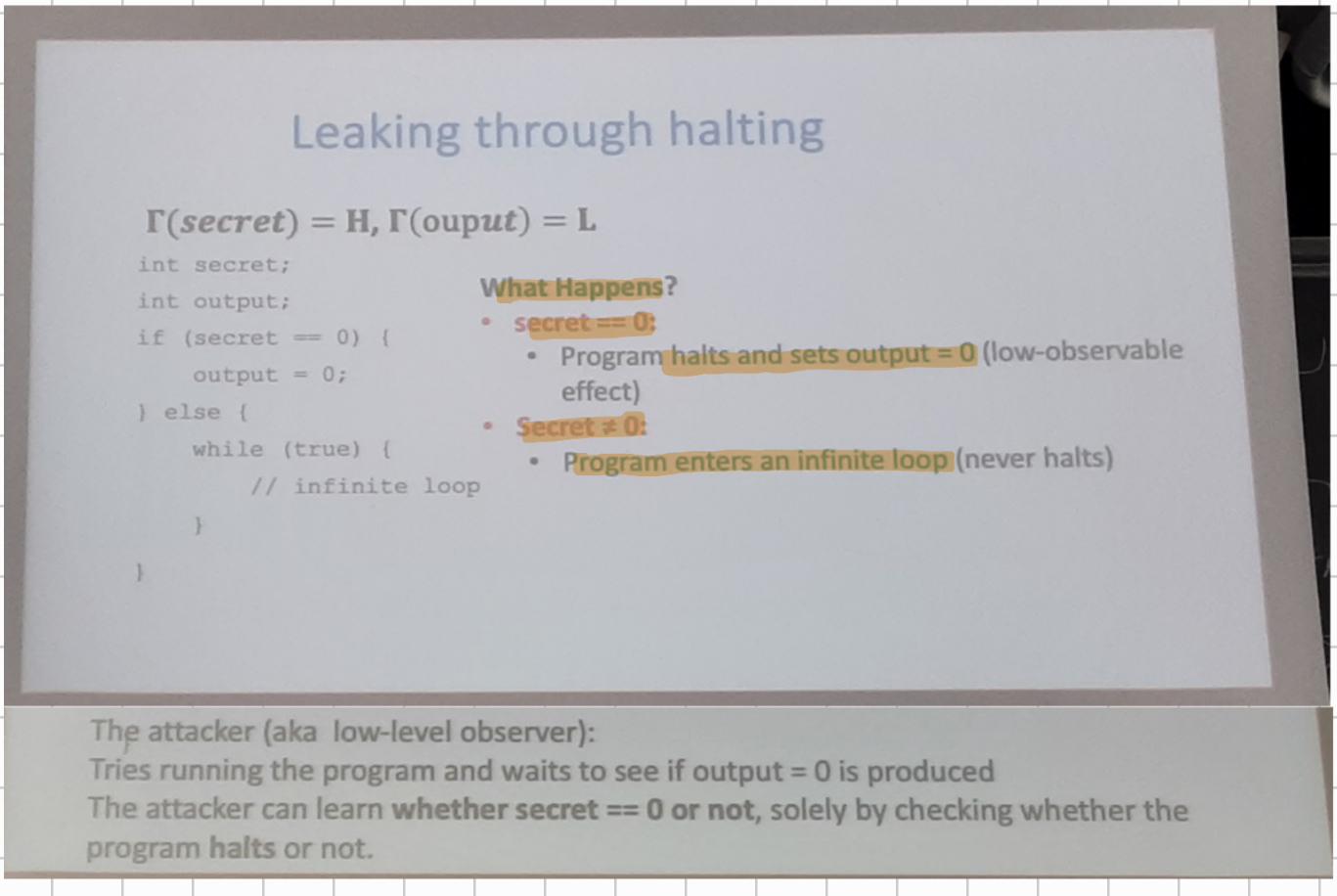
A dynamic mechanism can be more permissive than a static mechanism.

But, there is a caveat…

A dynamic mechanism may leak information
when deciding to halt an execution due to a failed check



else branch will not terminate. This program terminates when secret is = 0.



If it halts for the check, you see output unchanged.

Main problem of dynamic mechanisms (fixed policy / associated)

Leaking through halting

- Assume fixed Γ: $\Gamma(\mathbf{x}) = \mathrm{L}$, $\Gamma(\mathbf{y}) = \mathrm{L}$, $\Gamma(\mathbf{h}) = \mathrm{H}$
- Consider program:

```
x:=0;
if h>0 then x:=1 else skip
y:=x
```

- **Problem**: `h>0` is leaked. Why?

Program has a problem:

Leaking through halting

- Assume fixed Γ: $\Gamma(\mathbf{x}) = \mathrm{L}$, $\Gamma(\mathbf{y}) = \mathrm{L}$, $\Gamma(\mathbf{h}) = \mathrm{H}$
- Consider program:

```
x:=0;
if h>0 then x:=1 else skip
y:=x
```

At termination when !(**h>0**) is *true*, $\Gamma(\mathbf{y}) = \Gamma(\mathbf{x}) = \mathrm{L}$.

  - Two public outputs x = 0 and y = 0

- At termination when **h>0** is *true*, then execution terminates normally with two public outputs x= 1, y = 1
- **Problem**: **h>0** is leaked

Attacker can look at public values and attacker sees $y=0, x>0$ with $h \leq 0$.
If $h>0$, $x=1$, $y=1$.

OCAML SIMULATION

```
type label = Low | High

let join l1 l2 =
  match (l1, l2) with
  | High, _ -> High
  | _, High -> High
  | Low, Low -> Low
```

Pattern matching

```
type 'a labeled = {
  label : label;
  value : 'a;
}
```

We assign security level to data: generic value + label

```
let label_value lbl v = { label = lbl; value = v }
```

Struct interpretation

```
type context = {
  mutable pc : label;
}

let new_context () = { pc = Low }

let elevate ctx l =
  ctx.pc <- join ctx.pc l
```

struct

Record of a mutable entity (modifiable)

Beginning has low value

Join pc and value we pass with this elevate

We only manage assignment so no restore of ctx

```
let assign lhs_label rhs ctx =
  let actual_label = join rhs.label ctx.pc in
  if lhs_label < actual_label then
    failwith ("Security violation: illegal flow")
 else {label = lhs_label; value = rhs.value }
```

right hand side

From fixed labels to flow-sensitive labels

A flow-sensitive label on a variable can change during the analysis of the program.

Flow-sensitive labels can be used both in a static or dynamic mechanism.

Flow sensitive labels: discussion

Assume **Γ(h) =H and Γ(l) =L.**
Consider the program

x:=h; x:=0; l:=x

Is this program safe?

If Γ(x) is fixed to H, then the program is rejected, because the type analysis of l:=x fails.

Assume x does not have a fixed security label

Flow sensitive labels:discussion

Assume **Γ(h) =H and Γ(l) =L.**

Consider the program

x:=h; x:=0; l:=x

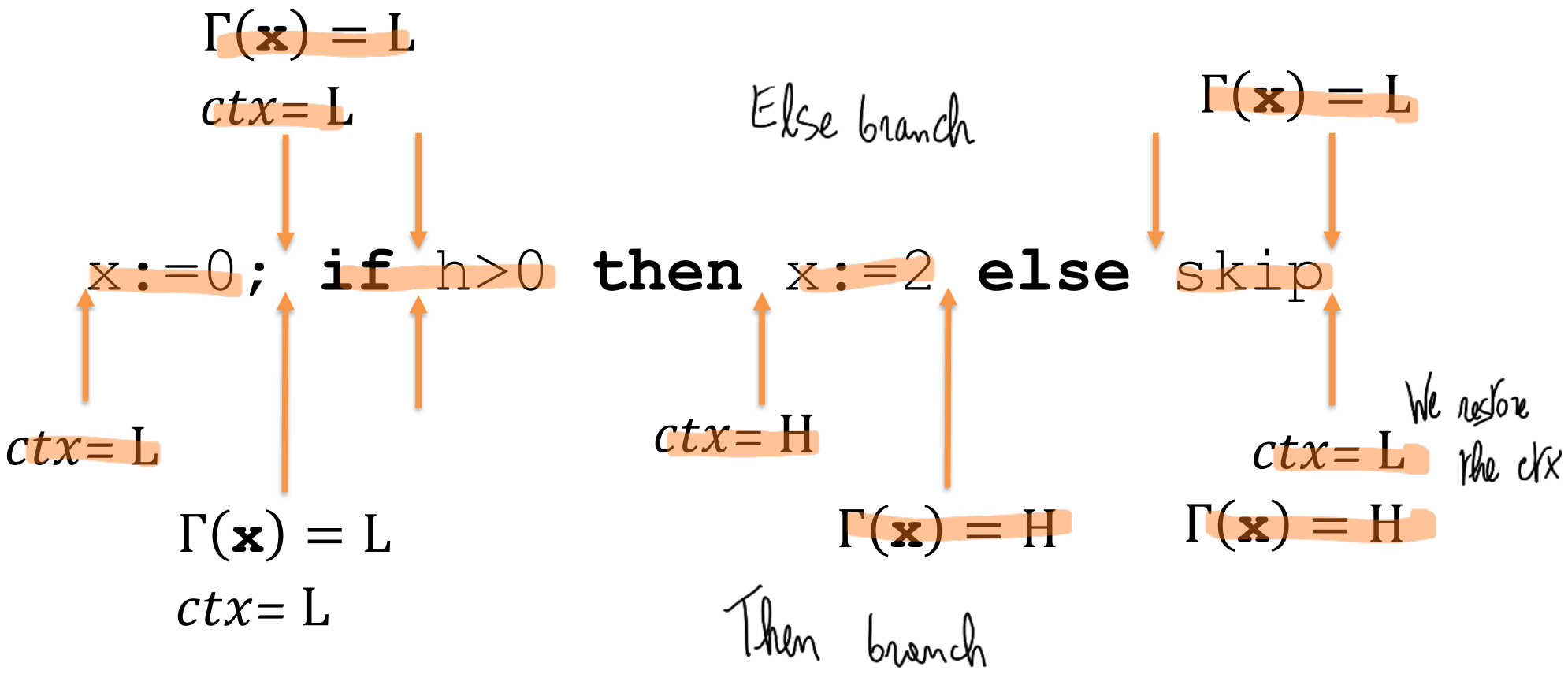
Is this program safe?

If Γ(x) is flow-sensitive, then

- Γ(x) becomes H **after x:=h**,
- Γ(x)= becomes L **after x:=0**,
- the analysis of l:=x succeeds.

Flow-sensitive labels can enhance permissiveness even further.

Assume fixed $\Gamma(\mathbf{h}) = \mathrm{H}$ and flow-sensitive $\Gamma(\mathbf{x})$.



```
x:=0; if h>0 then x:=2 else skip
```

- If **h>0** holds, then after **x:=2**, Γ(**x**) becomes H.
- If **h>0** does not hold, then Γ(**x**) remains L.

This label is not sound!

You need compositionality: split and

Problem: Even though **h** flows to **x**,

1.x is tagged with H only when **h>0**;

2.x is tagged with L otherwise

Imagine calling a function and you expect result of a function in terms of type: you do not say you expect both high level and low level of security.

- This is not ideal: think of function. You don't want a structure in which a function can return both a high security level or a low one.

- Guarantee composability: when you put together two objects (class A and B) that are in 2 different files. To compose them it is enough to have type of A and B.
  Composition: possibility of splitting things and being able to put them back together.
  Without composability, things are much harder.

Solution #1

- Make purely dynamic flow-sensitive mechanism more conservative:
- **Block execution before entering conditional commands with high guards.**

One solution: be conservative: do not execute high level guards.

Back to our example:

But, you can have same problem with

h=1

x=0

x=1+

x:=0; **if** h>0 **then** x:=2 **else** skip

Block executions that would give you this problem

All execution would stop after x:=0.

Solution #2

- **Exploit on-the-fly static analysis to update the labels of target variables in untaken branches to capture implicit flow.**
- **The mechanism is no longer purely dynamic.**

Use on-the-fly static analysis

Problem: **x** was tagged with H only when **h>0** was true, even though **h** always flow to **x**.
Goal: **x** should be tagged with H at every execution.

```
x:=0;
if h>0 then x:=1 else skip
```

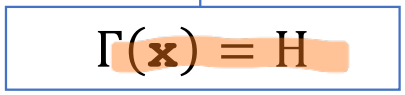
On-the-fly static analysis:
$\Gamma(\mathbf{x}) = \Gamma(\mathbf{1}) \sqcup \Gamma(\mathbf{h>0}) = \mathrm{H}$

Apply on-the-fly static analysis to the untaken branch.

Use on-the-fly static analysis

Problem: **x** was tagged with H only when **h>0** was true, even though **h** always flow to **x**.
Goal: **x** should be tagged with H at every execution.

```
x:=0;
if h>0 then x:=1 else skip
```



Static vs Dynamic

- Static:
  - Low run time overhead.
  - No new covert channels.
  - More conservative.
- Dynamic
  - Increased run time overhead.
  - Possible new covert channels.
  - Less conservative.
- Ongoing research for both static and dynamic.
  - Different expressiveness of policies, different NI versions, different mechanisms.

Dynamic Information Flow in Practice

JSFlow (Sabelfeld)

http://www.jsflow.net

JSFlow

- JSFlow adopts **dynamic information flow control:**
  - all runtime values are augmented with a security label taken form L-H lattice.
  - the concept of *no sensitive upgrade (NSU) is considered*
  - NSU states that security labels are not allowed to change under secret control.
- Tracking the information flow at run-time exploits a *program counter (pc)* abstraction reflecting the current *security context*, i.e. the security level, of the current computation.

Related work

- Bell-La Padula system access control combines
  - Mandatory Access control (MAC)
  - Multi-LevelSecurity(MLS)
- *Similarity with the typing rules we presented b*ut for processes accessing files, instead of a programs accessing variables, and enforced at runtime instead of compile time
- Bell-LaPaluda was developed in the 70s for access control in military applications

Information flow: summary

- Two kinds of flow
  - data-flow (direct/explicit) through assigments
  - control-flow (indirect/implicit) through if, while, . . .
- Classical code analysis technique:
  - compilation/optimization, verification
- Information flow in practice
  - static analysis via type systems not decidable, (over-)-approximation, not complete
  - runtime instrumentation/monitoring techniques (tags, extra checks), not sound (may miss existing flows)

Semantic model: Non Interference

- Check information flow provides partitions inside a program
  - no *influence* among partitions (e,g the partition of the attacker)
- Several applications in security:
  - confidentiality/integrity (e.g., isolation, enclaves)
  - side-channels through shared resources (execution time, cache, . . .)

Readings on TEAMS

- Notes on Information Flow and Non Interference
- Notes on information flow and dynamic information flow
- Andrei Sabelfeld, Andrew C. Myers:Language-based information-flow security. IEEE J. Sel. Areas Commun. 21(1): 5-19 (2003)
- Andrew C. Myers:JFlow: Practical Mostly-Static Information Flow Control. POPL 1999
- François Pottier, Vincent Simonet:
  Information flow inference for ML. ACM Trans. Program. Lang. Syst. 25(1): 117-158 (2003)
  - READING L19