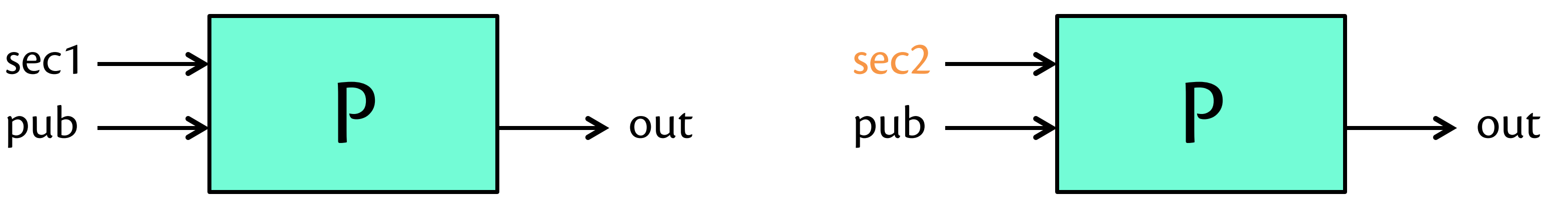
CS 5430

An Information-Flow Type System

Prof. Clarkson
Spring 2016

Review: Noninterference

- **Noninterference** [Goguen and Meseguer 1982]: *actions of high-security users do not affect observations of low-security users*
- Intuition, as commonly adapted to programs: *changes to secret inputs do not cause observable change in public output*



Review: Leakage

- Example of explicit flow:
```
p := p + s
```
- Example of implicit flow:
```
if (s mod 2) = 0
then p := 0 else p := 1
```
- Example of covert channel (termination):
```
while s != 0 do { //nothing }
```

Review: VSI type system

[Volpano, Smith, and Irvine 1996; Smith 2006]

Type system:

- set of rules for deriving facts about types of program expressions and commands
- typing judgment: $\Gamma \vdash \mathbf{c} : \tau\ \mathbf{cmd}$
  - Γ is a typing context: maps names of variables to their types
  - τ is a type: here will be H (high, secret) or L (low, public)
  - $\mathbf{c}$ is a command: assignment, if, while, etc.
  - $\Gamma \vdash \mathbf{c} : \tau\ \mathbf{cmd}$ means, in part, that $\mathbf{c}$ is a well-typed command
- **Theorem.** *If* $\Gamma \vdash \mathbf{c} : \tau\ \mathbf{cmd}$ *then* $\mathbf{c}$ *satisfies noninterference.*

Typing rules

- Example of typing rule from Java or OCaml (not VSI):

```
x + y : int
  if x : int
  and y : int
```

- Inference rule: infer a conclusion from some premises
  - Conclusion: **`x+y : int`**
  - Premise: **`x : int`**
  - Premise: **`y : int`**
- Syntax-directed: which rule to apply is determined by syntax of program

Program syntax

Core imperative language in Backus-Naur form (BNF):

```
e ::= x | n | e1+e2 | ...

c ::= x := e
    | if e then c1 else c2
    | while e do c
    | c1; c2
```

Types

Security types:

- τ ::= `H`|`L`
- H represents high security (secret) information
- L represents low security (public) information
- May flow relation: `L` → `H`, `L` → `L`, `H` → `H`
  - **Update to A6: notation for this relation corrected in problem 4**
- In general, could have a lattice of types

Types

Typing context: types of variables

- historically, context written as function Γ
  - i.e, $\Gamma(\mathbf{x}) = \tau$
- for sake of examples, assume:
  - $\Gamma(\mathbf{h}) = \mathrm{H}$
  - $\Gamma(\mathbf{l}) = \mathrm{L}$
- let's write that Γ as **`[h -> H, l -> L]`**
- and in general, **`[x1 -> τ1, x2 -> τ2, ...]`**

Typing principles

Three key ideas of VSI type system:

1. **Classify expressions:**
   - Expression is H if it contains any H variables
   - Otherwise is L
   - e.g., **`2*h+1`** is H, but **`42+l`** is L

Typing principles

Three key ideas of VSI type system:

2. **Prevent explicit flows:**
    – Forbid H expression being assigned to L variable
    – e.g., forbid `l := h`

Typing principles

Three key ideas of VSI type system:

3. **Prevent implicit flows:**
   - Forbid command with H guard from assigning to L variable
   - e.g., forbid

```
if (h mod 2) = 0
then l := 1 else l := 0
```

Typing judgment

Expressions: $\Gamma \vdash$ **`e`** $: \tau$ **`exp`**

- Means **`e`** is a well-typed expression that does not contain variables of type higher than τ
- But may contain variables of type τ or lower
- So a **`L exp`** contains only L variables
- But a **`H exp`** may contain L or H variables
- Intuition: **`e`** does not "read up" past τ

Variable rule

$\Gamma \vdash$ **`x`** $: \tau$ **`exp`**

  `if` $\Gamma($**`x`**$) = \tau$

Because the expression **`x`** contains variables only of type τ

e.g.

- **`[h -> H, l -> L]`** $\vdash$ **`h`** : **`H exp`**
- **`[x -> L, y -> L, z -> H]`** $\vdash$ **`y`** : **`L exp`**
- but not **`[h -> H]`** $\vdash$ **`z`** : **`??? exp`**

  (there is no way to fill in the ??? to make the judgment hold)

Constant rule

$\Gamma \vdash$ **`n`** : **`L exp`**

e.g.

- **`[h -> H, l -> L]`** $\vdash$ **`42`** : **`L exp`**
- **`[]`** $\vdash$ **`7`** : **`L exp`**

Since **`n`** contains no variables, not clear why **`L exp`** is the right type to give...

Constant rule

- Broaden our understanding:
  - from "$\Gamma \vdash$ **e** : **τ exp** means **e** is a well-typed expression that does not contain variables of type higher than τ"
  - to "$\Gamma \vdash$ **e** : **τ exp** means **e** is a well-typed expression that does not contain information of type higher than τ"
- A constant contains only public information (attacker knows source code)
  - Hence **n** does not contain information of type higher than L
  - Nor does **n** contain information of type higher than H
  - So we could go with $\Gamma \vdash$ **n : L exp** or $\Gamma \vdash$ **n : H exp**
  - An expression that could have two types...?

Subtyping

Java:

```
String s1 = new String("hello");
Object o1 = s1;
```

Constructed object has multiple types:

`String, Object`

Subtyping

- Behavioral subtyping: if S is a **subtype** of T, then objects of type T may be replaced by objects of type S without negative consequences to the behavior of the program
  - not "without any changes to behavior": maybe the subtype provides a more efficient implementation of an interface
  - but "without negative consequences": e.g., no new run-time errors
    - anywhere an `Object` is expected, can use a `String`
    - but if `String` is expected, can't use any `Object`: an `Integer`, e.g., couldn't respond to the `substring` method call
- So `String` is a subtype of `Object`, but not v.v.
- Notation: `S` ≤ `T` means `S` is a subtype of `T`
  - e.g., `String` ≤ `Object`

Subtyping

- Consider replacing `l1 := l2` with `l1 := h1`
  - Can't replace L expression with H expression: might cause negative consequence of leaking information
- Versus replacing `h1 := h2` with `h1 := l1`
  - Can replace H expression with L expression: won't create new information leak
- So `L exp` is a subtype of `H exp`, i.e., `L exp` ≤ `H exp`
- Anywhere a `H exp` is expected can replace with a `L exp`
- So let's make constants have type `L exp`
  - We can use constants as low expressions
  - Then use subtyping to make them be high expressions if ever we needed to

Subtyping rules

`L exp` $\leq$ **`H exp`**

$\Gamma \vdash$ **`e`** $: \tau 2$ **`exp`**
  `if` $\Gamma \vdash$ **`e`** $: \tau 1$ **`exp`**
  `and` $\tau 1 \leq \tau 2$

Subsumption rule

$\tau \leq \tau$

$\tau 1 \leq \tau 3$
  `if` $\tau 1 \leq \tau 2$
  `and` $\tau 2 \leq \tau 3$

Operation rule

$\Gamma \vdash$ `e1+e2` $: \tau$ `exp`

  `if` $\Gamma \vdash$ `e1` $: \tau$ `exp`

  `and` $\Gamma \vdash$ `e2` $: \tau$ `exp`

Because adding two expressions at the same level produces a result at that level

Operation rule

e.g.,

- **`[l1 -> L, l2 -> L]`** ⊢ **`l1 + l2 : L exp`**
  - **`because [l1 -> L, l2 -> L]`** ⊢ **`l1 : L exp`**
    - **`because [l1 -> L, l2 -> L](l1) = L`**
  - **`and [l1 -> L, l2 -> L]`** ⊢ **`l2 : L exp`**
    - **`because [l1 -> L, l2 -> L](l2) = L`**

Proof tree: hierarchical application of rules

Proof tree

```
let Γ = [l1 -> L, l2 -> L]
```

$$
\dfrac{\dfrac{\Gamma(\texttt{l1}) = \texttt{L}}{\Gamma \vdash \texttt{l1} : \texttt{L exp}} \qquad \dfrac{\Gamma(\texttt{l2}) = \texttt{L}}{\Gamma \vdash \texttt{l2} : \texttt{L exp}}}{\Gamma \vdash \texttt{l1 + l2} : \texttt{L exp}}
$$

Proof tree

```
let Γ = [l1 -> L, l2 -> L]
```

$$\dfrac{\dfrac{\Gamma(\texttt{l1}) = \texttt{L}}{\Gamma \vdash \texttt{l1 : L exp}} \qquad \dfrac{\Gamma(\texttt{l2}) = \texttt{L}}{\Gamma \vdash \texttt{l2 : L exp}}}{\Gamma \vdash \texttt{l1 + l2 : L exp}}$$

Operation rule

more examples:

- `[x -> H, y -> H]` ⊢ `x + y : H exp`
  - *proof tree omitted*
- what about
  `[l -> L, h -> H]` ⊢ `l+h : ??? exp`
- `[l -> L, h -> H]` ⊢ `l+h : H exp`
  - `because [l -> L, h -> H]` ⊢ `l : H exp`
    - `because [l -> L, h -> H]` ⊢ `l : L exp`
      - » `because [l -> L, h -> H](l) = L`
    - `and L exp` ≤ `H exp`
  - `and [l -> L, h -> H]` ⊢ `h : H exp`
    - *proof tree omitted*

Typing judgment

Commands: $\Gamma \vdash$ **`c`** $: \tau$ **`cmd`**

- Means **`c`** is a well-typed command that assigns only to variables of type τ or higher
- So a **`L cmd`** may assign to L or H variables
- But a **`H cmd`** assigns only to H variables
- Another intuition: **`c`** does not "write down" past τ

Assignment rule

$\Gamma \vdash$ **x:=e** $: \tau$ **cmd**
  if $\Gamma \vdash$ **e** $: \tau$ **exp**
  and Γ**(x)=**τ

Because it assigns to a variable of type τ, and putting information at level τ in that variable will not cause an insecure explicit flow

e.g.,

- **[l -> L]** $\vdash$ **l := 42 : L cmd**
  - **because [l -> L]** $\vdash$ **42 : L exp**
  - **and [l -> L](l) = L**

Assignment rule

$\Gamma \vdash$ `x:=e` $: \tau$ `cmd`
  `if` $\Gamma \vdash$ `e` $: \tau$ `exp`
  `and` Γ`(x)=`τ

another example:

- `[l -> L, h -> H]` $\vdash$ `h := l : H cmd`
  - because `[l -> L, h -> H]` $\vdash$ `l : H exp`
    - because `[l -> L, h -> H]` $\vdash$ `l : L exp`
      - because `[l -> L, h -> H](l) = L`
    - and `L exp` $\leq$ `H exp`
  - and `[l -> L, h -> H](h) = H`

Assignment rule

$\Gamma \vdash$ `x:=e` $: \tau$ `cmd`

  `if` $\Gamma \vdash$ `e` $: \tau$ `exp`

  `and` Γ`(x)`$= \tau$

Would have to be L but is H

but this proof doesn't succeed:

- `[l -> L, h -> H]` $\vdash$ `l := h : ??? cmd`
  - because `[l -> L, h -> H]` $\vdash$ `h : H exp`
    - because `[l -> L, h -> H](h) = H`
  - and `[l -> L, h -> H](l) = L`

If rule

$\Gamma \vdash$ `if e then c1 else c2` $: \tau$ cmd

  `if` $\Gamma \vdash$ `e` $: \tau$ exp

  `and` $\Gamma \vdash$ `c1` $: \tau$ cmd

  `and` $\Gamma \vdash$ `c2` $: \tau$ cmd

Because guard reads information at level τ, so must not write to variables below that level; ensuring that write to variables at that level or above prohibits insecure implicit flows

If rule

e.g.

let Γ = **[l1 -> L, l2 -> L]**

- $\Gamma \vdash$ **if l1 then l2:=0 else l2:=1** : **L cmd**
  - because $\Gamma \vdash$ **l1** : **L exp**
    - because Γ **(l1) = L**
  - and $\Gamma \vdash$ **l2:=0** : **L cmd**
    - because $\Gamma \vdash$ **0** : **L exp**
    - and Γ **(l2) = L**
  - and $\Gamma \vdash$ **l2:=1** : **L cmd**
    - *proof tree omitted*

If rule

another example:

let Γ `= [l -> L, h -> H]`

- $\Gamma \vdash$ **`if l then h:=0 else h:=1`** : **`H cmd`**
  - **because** $\Gamma \vdash$ **`l`** : **`H exp`**
    - **because** $\Gamma \vdash$ **`l`** : **`L exp`**
      - **because** Γ **`(l) = L`**
    - **and `L exp`** $\leq$ **`H exp`**
  - **and** $\Gamma \vdash$ **`h:=0`** : **`H cmd`**
    - **because** $\Gamma \vdash$ **`0`** : **`H exp`**
      - **because** $\Gamma \vdash$ **`0`** : **`L exp`**
      - **and `L exp`** $\leq$ **`H exp`**
    - **and** Γ **`(h) = H`**
  - **and** $\Gamma \vdash$ **`h:=1`** : **`H cmd`**
    - *proof tree omitted*

If rule

This proof happily gets stuck...

let Γ `= [h -> H, l2 -> L]`

- $\Gamma \vdash$ `if h then l2:=0 else l2:=1 : ??? cmd`
  - because $\Gamma \vdash$ `h : H exp`
    - because Γ `(h) = H`
  - and $\Gamma \vdash$ `l2:=0 : L cmd`
    - because $\Gamma \vdash$ `0 : L exp`
    - and Γ `(l2) = L`
  - and $\Gamma \vdash$ `l2:=1 : L cmd`
    - *proof tree omitted*

While rule

$\Gamma \vdash$ `while e do c :` τ `cmd`

  `if` $\Gamma \vdash$ `e :` τ `exp`

  `and` $\Gamma \vdash$ `c :` τ `cmd`

Just like an if statement but with a single branch

Sequence rule

$\Gamma \vdash$ `c1; c2` $: \tau$ `cmd`

  `if` $\Gamma \vdash$ `c1` $: \tau$ `cmd`

  `and` $\Gamma \vdash$ `c2` $: \tau$ `cmd`

Because if both subcommands assign to τ or higher, then so does whole command

e.g.,

- `[l->L]` $\vdash$ `l:=1; l:=0` : `L cmd`
  - *proof tree omitted*

Sequence rule

This proof unhappily gets stuck...

let Γ `= [h -> H, l1 -> L, l2 -> L]`

- $\Gamma \vdash$ `if l1 then h:=0; l2:=0 else l2:=1` : `??? cmd`
  - can't give it type `H cmd`, because it assigns to `L` variables
  - and can't yet give it type `L cmd`, because `h := 0 : L cmd` would require type of `h` to be exactly `L`
  - but it doesn't leak any information :(
- Recall our intended meaning of τ `cmd` was that it assigns to τ or higher
- So ought to be able to conclude `h := 0 : L cmd`

Command subtyping

$\Gamma \vdash$ **`e`** $: \tau 1$ **`cmd`**

  `if` $\Gamma \vdash$ **`e`** $: \tau 2$ **`cmd`**

  `and` $\tau 1 \leq \tau 2$

Note: backwards from expression subtyping rule!

- Can replace **`H exp`** with **`L exp`** without causing an insecure read up
- Can replace **`L cmd`** with **`H cmd`** without causing an insecure write down

Command subtyping

Now proof succeeds...

let Γ `= [h -> H, l1 -> L, l2 -> L]`

- $\Gamma \vdash$ `if l1 then h:=0; l2:=0 else l2:=1` : `L cmd`
  - because $\Gamma \vdash$ `l1` : `L exp`
    - because Γ`(l1)=L`
  - and $\Gamma \vdash$ `h:=0; l2:=0` : `L cmd`
    - because $\Gamma \vdash$ `h:=0` : `L cmd`
      - because $\Gamma \vdash$ `h:=0` : `H cmd`
        - » *proof tree omitted*
      - and `H cmd` $\leq$ `L cmd`
    - and $\Gamma \vdash$ `l2:=0` : `L cmd`
      - *proof tree omitted*
  - and $\Gamma \vdash$ `l2:=1` : `L cmd`
    - *proof tree omitted*

Noninterference

Theorem. If $\Gamma \vdash \mathtt{e} : \boldsymbol{\tau}\ \mathtt{cmd}$ then $\mathtt{c}$ satisfies noninterference.

Doesn't matter what $\boldsymbol{\tau}$ or Γ are, as long as there is some type and context for which command is typeable

Type systems are imperfect

	Violates noninterference	Satisfies noninterference
Type system rejects	True positive	False positive
Type system accepts	False negative	True negative

Imprecision

Covert channels

Type systems are imperfect

Example of covert channel:

`while s != 0 do { //nothing }`

- how to represent "do nothing" in our little imperative language?
  - `skip` command
  - i.e., `while s != 0 do skip`
  - Typing rule: $\Gamma \vdash \texttt{skip} : \texttt{H cmd}$
- program is typeable even though it leaks over covert channel
- doesn't violate noninterference theorem because noninterference definition itself ignores that channel

Type systems are imperfect

Example of imprecision:

`if 0=1 then l := h else skip`

- program is not typeable even though it does not violate noninterference
- nearly all type systems are conservative in this way

VSI notation vs. this lecture

In case you want to go read the original papers...

VSI...	This lecture...
Γ maps variable to τ **`var`**	Γ maps variable to τ
Expressions have type τ	Expressions have type τ **`exp`**
Subtyping written with $\subseteq$	Subtyping written with $\leq$
Proof trees written with conclusion below premises	Proof trees written with conclusion above premises

Upcoming events

- [Sunday] A6 due
- [May 16] Final exam

A type system is the most cost effective unit test you'll ever have. – Peter Hallam