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:
  \[
  \text{if } (s \mod 2) = 0 \\
  \text{then } p := 0 \text{ else } p := 1
  \]

• Example of covert channel (termination):
  \[
  \text{while } s \neq 0 \text{ do } \{ \text{ //nothing } \}
  \]
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:** $\Gamma \vdash c : \tau_{cmd}$
  - $\Gamma$ is a **typing context**: maps names of variables to their types
  - $\tau$ is a **type**: here will be $H$ (high, secret) or $L$ (low, public)
  - $c$ is a **command**: assignment, if, while, etc.
  - $\Gamma \vdash c : \tau_{cmd}$ means, in part, that $c$ is a well-typed command

**Theorem.** *If* $\Gamma \vdash c : \tau_{cmd}$ *then* $c$ *satisfies noninterference.*
Typing rules

• Example of typing rule from Java or OCaml (not VSI):
  \[
  x + y : \text{int} \\
  \quad \text{if } x : \text{int} \\
  \quad \text{and } y : \text{int}
  \]

• Inference rule: infer a conclusion from some premises
  – Conclusion: \(x + y : \text{int}\)
  – Premise: \(x : \text{int}\)
  – Premise: \(y : \text{int}\)

• Syntax-directed: which rule to apply is determined by syntax of program
Core imperative language in Backus-Naur form (BNF):

$$e ::= x \mid n \mid e_1 + e_2 \mid \ldots$$

$$c ::= x := e$$

$$\mid \text{if } e \text{ then } c_1 \text{ else } c_2$$

$$\mid \text{while } e \text{ do } c$$

$$\mid c_1; c_2$$
Types

Security types:
• $\tau ::= H | L$
• $H$ represents high security (secret) information
• $L$ represents low security (public) information
• May flow relation: $L \rightarrow H$, $L \rightarrow L$, $H \rightarrow H$
  
  – 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 $\Gamma$
  – i.e, $\Gamma(x) = \tau$

• for sake of examples, assume:
  – $\Gamma(h) = H$
  – $\Gamma(l) = L$

• let's write that $\Gamma$ as $[h \rightarrow H, l \rightarrow L]$

• and in general, $[x_1 \rightarrow \tau_1, x_2 \rightarrow \tau_2, \ldots]$
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 $42+1$ 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
     \[
     \text{if } (h \mod 2) = 0 \\
     \text{then } l := 1 \text{ else } l := 0
     \]
Typing judgment

Expressions: \( \Gamma \vdash e : \tau \ exp \)

• Means \( e \) is a well-typed expression that does not contain variables of type higher than \( \tau \)
• But may contain variables of type \( \tau \) 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 \( \tau \)
Variable rule

\[ \Gamma \vdash x : \tau \text{ exp} \]
\[ \text{if } \Gamma(x) = \tau \]

Because the expression \( x \) contains variables only of type \( \tau \)

e.g.

- \([h \rightarrow H, l \rightarrow L] \vdash h : H \text{ exp}\)
- \([x \rightarrow L, y \rightarrow L, z \rightarrow H] \vdash y : L \text{ exp}\)
- but not \([h \rightarrow H] \vdash z : ??? \text{ exp}\)
  (there is no way to fill in the ??? to make the judgment hold)
**Constant rule**

$$\Gamma \vdash n : L \text{ exp}$$

e.g.

- $$[h \rightarrow H, l \rightarrow L] \vdash 42 : L \text{ exp}$$
- $$[] \vdash 7 : L \text{ exp}$$

Since \( n \) contains no variables, not clear why \( L \text{ exp} \) is the right type to give...
Constant rule

• Broaden our understanding:
  – from "\( \Gamma \vdash e : \tau \exp \) means \( e \) is a well-typed expression that does not contain variables of type higher than \( \tau \)"
  – to "\( \Gamma \vdash e : \tau \exp \) means \( e \) is a well-typed expression that does not contain information of type higher than \( \tau \)"

• A constant contains only public information (attacker knows source code)
  – Hence \( n \) does not contain information of type higher than \( \text{L} \)
  – Nor does \( n \) contain information of type higher than \( \text{H} \)
  – So we could go with \( \Gamma \vdash n : \text{L} \exp \) or \( \Gamma \vdash n : \text{H} \exp \)
  – 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 \texttt{Object} is expected, can use a \texttt{String}
    - but if \texttt{String} is expected, can't use any \texttt{Object}: an \texttt{Integer}, e.g., couldn't respond to the \texttt{substring} method call
- **So \texttt{String} is a subtype of \texttt{Object}, but not v.v.**
- **Notation**: \( S \leq T \) means \( S \) is a subtype of \( T \)
  - e.g., \texttt{String} \( \leq \) \texttt{Object}
Subtyping

• Consider replacing \texttt{l1 := l2} with \texttt{l1 := h1}
  – Can’t replace L expression with H expression: might cause negative consequence of leaking information
• Versus replacing \texttt{h1 := h2} with \texttt{h1 := l1}
  – Can replace H expression with L expression: won’t create new information leak
• So \texttt{L exp} is a subtype of \texttt{H exp}, i.e., \texttt{L exp} \leq \texttt{H exp}
• Anywhere a \texttt{H exp} is expected can replace with a \texttt{L exp}
• So let’s make constants have type \texttt{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

\[ L \exp \leq H \exp \]

\[ \Gamma \vdash e : \tau_2 \exp \]
  \[ \text{if } \Gamma \vdash e : \tau_1 \exp \]
  \[ \text{and } \tau_1 \leq \tau_2 \]

\[ \tau \leq \tau \]

\[ \tau_1 \leq \tau_3 \]
  \[ \text{if } \tau_1 \leq \tau_2 \]
  \[ \text{and } \tau_2 \leq \tau_3 \]

Subsumption rule
Operation rule

\( \Gamma \vdash e_1 + e_2 : \tau \text{ exp} \)

if \( \Gamma \vdash e_1 : \tau \text{ exp} \)

and \( \Gamma \vdash e_2 : \tau \text{ exp} \)

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

e.g.,

• \([l_1 \rightarrow L, l_2 \rightarrow L] \vdash l_1 + l_2 : L \exp\)
  • because \([l_1 \rightarrow L, l_2 \rightarrow L] \vdash l_1 : L \exp\)
    – because \([l_1 \rightarrow L, l_2 \rightarrow L](l_1) = L\)
  • and \([l_1 \rightarrow L, l_2 \rightarrow L] \vdash l_2 : L \exp\)
    – because \([l_1 \rightarrow L, l_2 \rightarrow L](l_2) = L\)

Proof tree: hierarchical application of rules
Proof tree

let $\Gamma = [l_1 \rightarrow L, l_2 \rightarrow L]$

\[
\begin{align*}
\Gamma &\vdash l_1 : L \text{ exp} \\
\Gamma &\vdash l_2 : L \text{ exp}
\end{align*}
\]

\[
\Gamma \vdash l_1 + l_2 : L \text{ exp}
\]
Proof tree

let \( \Gamma = [l_1 \rightarrow L, l_2 \rightarrow L] \)

\[
\begin{align*}
\Gamma(l_1) &= L \\
\Gamma(l_2) &= L \\
\Gamma \vdash l_1 : L \text{ exp} \\
\Gamma \vdash l_2 : L \text{ exp} \\
\Gamma \vdash l_1 + l_2 : L \text{ exp}
\end{align*}
\]
Operation rule

more examples:

• $[x \rightarrow H, y \rightarrow H] \vdash x + y : H \text{ exp}$
  • *proof tree omitted*

• what about $[l \rightarrow L, h \rightarrow H] \vdash l+h : ??? \text{ exp}$

• $[l \rightarrow L, h \rightarrow H] \vdash l+h : H \text{ exp}$
  • because $[l \rightarrow L, h \rightarrow H] \vdash l : H \text{ exp}$
    – because $[l \rightarrow L, h \rightarrow H] \vdash l : L \text{ exp}$
      » because $[l \rightarrow L, h \rightarrow H](l) = L$
    – and $L \text{ exp} \leq H \text{ exp}$

• and $[l \rightarrow L, h \rightarrow H] \vdash h : H \text{ exp}$
  – *proof tree omitted*
Typing judgment

Commands: $\Gamma \vdash c : \tau \text{ cmd}$

- Means $c$ is a well-typed command that assigns only to variables of type $\tau$ or higher
- So a $\text{L cmd}$ may assign to L or H variables
- But a $\text{H cmd}$ assigns only to H variables
- Another intuition: $c$ does not "write down" past $\tau$
Assignment rule

\[ \Gamma \vdash x := e : \tau \text{ cmd} \]

\[ \text{if } \Gamma \vdash e : \tau \text{ exp} \]

\[ \text{and } \Gamma(x) = \tau \]

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

e.g.,

- \([l \rightarrow L] \vdash l := 42 : L \text{ cmd} \]
  - because \([l \rightarrow L] \vdash 42 : L \text{ exp} \]
  - and \([l \rightarrow L](l) = L \]
Assignment rule

\[ \Gamma \vdash x := e : \tau \text{ cmd} \]

if \( \Gamma \vdash e : \tau \text{ exp} \)
and \( \Gamma(x) = \tau \)

another example:

- \([l \rightarrow L, h \rightarrow H] \vdash h := l : H \text{ cmd}\)
  - because \([l \rightarrow L, h \rightarrow H] \vdash l : H \text{ exp}\)
    - because \([l \rightarrow L, h \rightarrow H] \vdash l : L \text{ exp}\)
      - because \([l \rightarrow L, h \rightarrow H](l) = L\)
    - and \(L \text{ exp} \leq H \text{ exp}\)
      - and \([l \rightarrow L, h \rightarrow H](h) = H\)
Assignment rule

\[ \Gamma \vdash x := e : \tau \text{ cmd} \]
if \( \Gamma \vdash e : \tau \text{ exp} \)
and \( \Gamma(x) = \tau \)

but this proof doesn't succeed:

- \([l \rightarrow L, h \rightarrow H] \vdash l := h : \text{ ??? cmd}\)
  - because \([l \rightarrow L, h \rightarrow H] \vdash h : \text{ H exp}\)
    - because \([l \rightarrow L, h \rightarrow H](h) = H\)
    - and \([l \rightarrow L, h \rightarrow H](l) = L\)
If rule

\[ \Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \tau \text{cmd} \]

\[ \text{if } \Gamma \vdash e : \tau \text{exp} \]

\[ \text{and } \Gamma \vdash c_1 : \tau \text{cmd} \]

\[ \text{and } \Gamma \vdash c_2 : \tau \text{cmd} \]

Because guard reads information at level \( \tau \), 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 $\Gamma = [l_1 \rightarrow L, l_2 \rightarrow L]$

• $\Gamma \vdash \text{if } l_1 \text{ then } l_2 := 0 \text{ else } l_2 := 1 : L$
  cmd
    – because $\Gamma \vdash l_1 : L$ exp
      • because $\Gamma(l_1) = L$
    – and $\Gamma \vdash l_2 := 0 : L$ cmd
      • because $\Gamma \vdash 0 : L$ exp
      • and $\Gamma(l_2) = L$
    – and $\Gamma \vdash l_2 := 1 : L$ cmd
      • proof tree omitted
If rule

another example:
let $\Gamma = [l \rightarrow L, h \rightarrow H]$

- $\Gamma \vdash \text{if } l \text{ then } h := 0 \text{ else } h := 1 : H \text{ cmd}$
  - because $\Gamma \vdash l : H \text{ exp}$
    - because $\Gamma \vdash l : L \text{ exp}$
      - because $\Gamma (l) = L$
    - and $L \text{ exp} \leq H \text{ exp}$
  - and $\Gamma \vdash h := 0 : H \text{ cmd}$
    - because $\Gamma \vdash 0 : H \text{ exp}$
      - because $\Gamma \vdash 0 : L \text{ exp}$
        - and $L \text{ exp} \leq H \text{ exp}$
      - and $\Gamma (h) = H$
  - and $\Gamma \vdash h := 1 : H \text{ cmd}$
    - proof tree omitted
If rule

This proof happily gets stuck...

let $\Gamma = [h \rightarrow H, l2 \rightarrow L]$

• $\Gamma \vdash \text{if } h \text{ then } l2:=0 \text{ else } l2:=1 : \text{ ??? cmd}$
  
  – because $\Gamma \vdash h:H \exp$
    • because $\Gamma \vdash (h) = H$
  
  – and $\Gamma \vdash l2:=0:L \exp$
    • because $\Gamma \vdash 0:L \exp$
    • and $\Gamma \vdash (l2) = L$

  – and $\Gamma \vdash l2:=1:L \exp$
    • $\text{proof tree omitted}$
While rule

\[ \Gamma \vdash \text{while } e \, \text{do } c : \tau \text{cmd} \]

\[ \text{if } \Gamma \vdash e : \tau \text{exp} \]

\[ \text{and } \Gamma \vdash c : \tau \text{cmd} \]

Just like an if statement but with a single branch
Sequence rule

$\Gamma \vdash c_1; c_2 : \tau \text{ cmd}$

if $\Gamma \vdash c_1 : \tau \text{ cmd}$
and $\Gamma \vdash c_2 : \tau \text{ cmd}$

Because if both subcommands assign to $\tau$ or higher, then so does whole command

e.g.,

• $[l \rightarrow L] \vdash l := 1; \ l := 0 : L \text{ cmd}$
  – proof tree omitted
Sequence rule

This proof unhappily gets stuck...

let $\Gamma = [h \to H, l1 \to L, l2 \to L]$

- $\Gamma \vdash \text{if } l1 \text{ then } h := 0; l2 := 0 \text{ else } l2 := 1 : ??? \text{ cmd}$
  - can't give it type $H \text{ cmd}$, because it assigns to $L$ variables
  - and can't yet give it type $L \text{ cmd}$, because $h := 0 : L \text{ cmd}$ would require type of $h$ to be exactly $L$
  - but it doesn't leak any information :(

- Recall our intended meaning of $\tau \text{ cmd}$ was that it assigns to $\tau$ or higher

- So ought to be able to conclude $h := 0 : L \text{ cmd}$
Command subtyping

\[ \Gamma \vdash e : \tau_1 \text{ cmd} \]
\[ \text{ if } \Gamma \vdash e : \tau_2 \text{ cmd} \]
\[ \text{ and } \tau_1 \leq \tau_2 \]

Note: backwards from expression subtyping rule!

• Can replace \texttt{H \ exp} with \texttt{L \ exp} without causing an insecure read up
• Can replace \texttt{L \ cmd} with \texttt{H \ cmd} without causing an insecure write down
Command subtyping

Now proof succeeds...

\[ \Gamma = [h \rightarrow H, \ l1 \rightarrow L, \ l2 \rightarrow L] \]

- \( \Gamma \vdash \text{if} \ l1 \ \text{then} \ h:=0; \ l2:=0 \ \text{else} \ l2:=1 : L \ cmd \)
  - because \( \Gamma \vdash \text{if} \ l1 \ \text{then} \ 0; \ 0 : L \ exp \)
    - because \( \Gamma(l1)=L \)
  - 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 $\Gamma \vdash e : \tau \text{ cmd}$ then $c$ satisfies noninterference.

Doesn't matter what $\tau$ or $\Gamma$ are, as long as there is some type and context for which command is typeable.
Type systems are imperfect

<table>
<thead>
<tr>
<th></th>
<th>Violates noninterference</th>
<th>Satisfies noninterference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type system rejects</strong></td>
<td>True positive</td>
<td>False positive</td>
</tr>
<tr>
<td><strong>Type system accepts</strong></td>
<td>False negative</td>
<td>True negative</td>
</tr>
</tbody>
</table>

Imprecision

Covert channels
Type systems are imperfect

Example of covert channel:

```plaintext
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: $\Gamma \vdash \text{skip} : H \text{ cmd}$

- 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:
\[
\text{if } 0=1 \text{ then } l := h \text{ else } \text{skip}
\]

• 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...

<table>
<thead>
<tr>
<th>VSI...</th>
<th>This lecture...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$ maps variable to $\tau$ $\text{var}$</td>
<td>$\Gamma$ maps variable to $\tau$</td>
</tr>
<tr>
<td>Expressions have type $\tau$</td>
<td>Expressions have type $\tau$ $\text{exp}$</td>
</tr>
<tr>
<td>Subtyping written with $\subseteq$</td>
<td>Subtyping written with $\leq$</td>
</tr>
<tr>
<td>Proof trees written with conclusion below premises</td>
<td>Proof trees written with conclusion above premises</td>
</tr>
</tbody>
</table>
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