Noninterference with Dynamic Security Domains and Policies

Robert Grabowski

LMU Munich, Germany

Lennart Beringer

Princeton University, USA

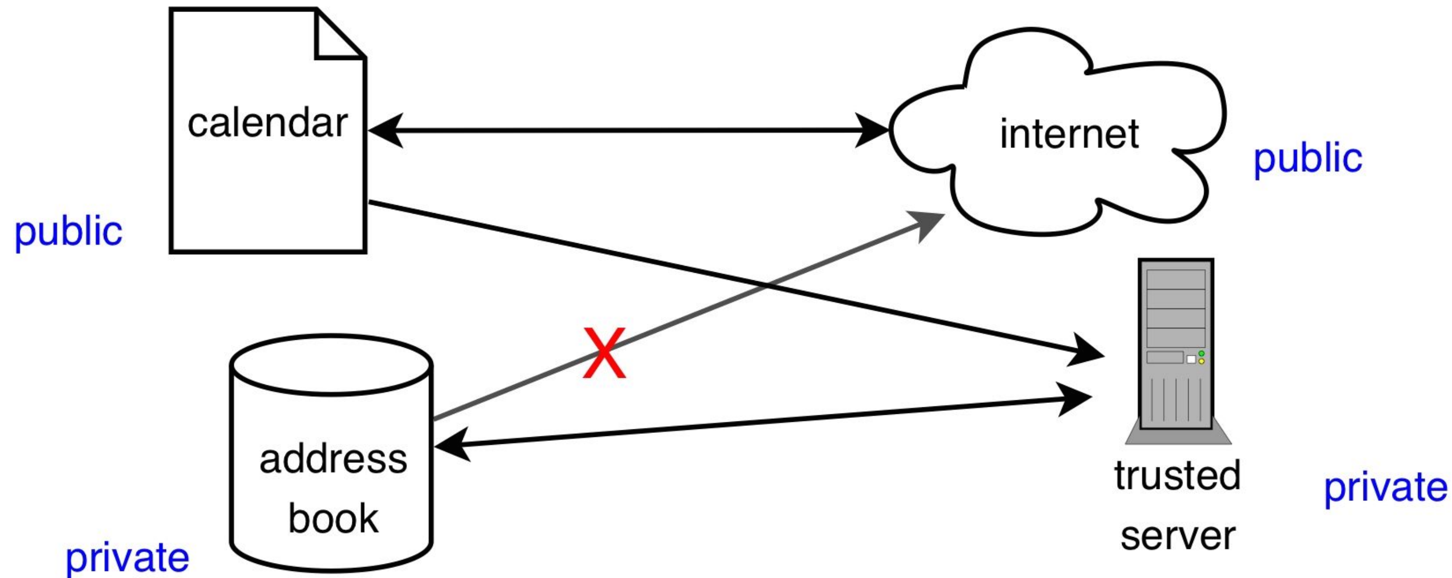
13th Asian Computing Science Conference
Seoul, 16/12/2009

InfoZert

LMU

Information flow security

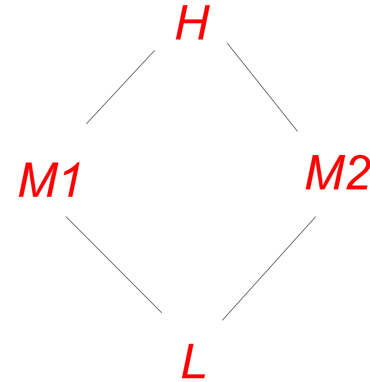
- example: personal organizer application



- control the flow of information between data resources/objects
- assign confidentiality *domains* (security levels) to objects
- specify secure information flows using a *flow policy*
    - e.g. allow flows from public to private domain, but not vice versa

Language-based information flow security

- Flow policy: modelled as lattice over domains
  ($\mathcal{D}$, $\leq$, $\vee$, $\wedge$, H, L)



- security expressed as *noninterference* property
  - assign domains to object fields, method parameters, local variables
  - data of domain *A* may not influence computation of values of domain *B* unless $A \leq B$
- examples for insecure flows:

  $x_L := y_L.f_H * 2$ **if** $z_H > 5$ **then** $x_L := 1$ **else** $x_L := 0$

- static program analysis shows the absence of insecure flows

Dynamic domains

- statically assigned domains and fixed policy: not realistic for mobile code
- example: a class that wraps file operations:

```
class FileLow {
    contents : String_L;
    readInt() : Int_L;
    writeInt(v: Int_L);
}
```

```
class FileHigh {
    contents : String_H;
    readInt() : Int_H;
    writeInt(v: Int_H);
}
```

  - security domain of file contents given by environment
  - might not even know the available domains or the security policy that is effective when program executes

- need program that dynamically adapts behaviour to the security environment

Dynamic domains

- solution: model domain explicitly and use (weak) dependent types

```
class File {
    δ : Domain⊥;
    contents : Stringδ ;
    readInt() : Intδ ;
    writeInt(v: Intδ);
}
```

- check whether flow is permitted before performing action that induces flow

```
out := new File(...);
...
if ⊤ ⊑ out.δ then
    out.writeInt(secret⊤);
```

- secure, regardless of...
  - domain of file accessed with `out`
  - concrete policy $\leq$
  - actual top-most domain ($\top$)

Previous work

- type systems for static analysis of information flows
  - WHILE language [Volpano et al, 1996]
  - object-oriented language [Banerjee/Naumann, 2003]
  - bytecode language [Barthe et al, 2005]
- static analysis of dynamic security domains, policies, or both
  - JIF: Java with Information Flow [Myers 1999, Zheng/Myers 2004]
  - RTI: dynamic roles [Bandhakavi et al, 2008]
  - λ^{deps^+}: dynamic dependency monitoring [Shroff et al, 2007]

Contribution

- object-oriented language...

  ...with support for **dynamic security domains and policies**

- static analysis in form of a type system...

  ...proves security **for any execution environment**

- transferred results to JVM-like bytecode...

  ...to verify secure flows in **mobile code**

Overview

1. Noninterference with dynamic security domains and policies

2. High-level type system

3. Bytecode language

Syntax and semantics of DSD language

numbers $n \in \mathbb{N}$, domains $k \in \mathcal{D}$, heap addresses $a \in \mathcal{A}$, variables $x \in \mathcal{X}$, fields $f \in \mathcal{F}$

$$
\begin{array}{rcl}
\mathcal{V} \ni v & ::= & n \mid a \mid k \\
\mathcal{E} \ni e & ::= & n \mid x \mid e.f \mid e + e \mid e < e \mid \top \mid \bot \mid e \sqcup e \mid e \sqsubseteq e \\
\mathcal{P} \ni P & ::= & \textbf{skip} \mid P\ ;\ P \mid \textbf{if}\ e\ \textbf{then}\ P\ \textbf{else}\ P \mid \textbf{while}\ e\ \textbf{do}\ P \mid \\
 & & x := e \mid e.f := e \mid x := \textbf{new}\ C \mid x := e.m(\overline{e})
\end{array}
$$

each method has a special argument variable $x_\delta \in \mathcal{X}$

each object has a special field $f_\delta \in \mathcal{F}$

- standard semantics
  - state model: σ (store and heap)
  - denotational semantics for expressions: $[\![e]\!]_\sigma$
  - big-step operational semantics for programs: $\sigma_1 \xrightarrow{P} \sigma_2$

Visibility of data

- assign abstract domains using type environments

$$\Gamma_m : \mathcal{X} \rightharpoonup \{\top, \bot, x_\delta\}$$ – current variable type environment

$$\Phi : \mathcal{F} \rightarrow \{\top, \bot, f_\delta\}$$ – global class-insensitive field typing

- a variable x in a state σ is *visible* at a domain k if $[\![\Gamma(x)]\!]_\sigma \leq k$

$[\![\bot]\!]_\sigma = L$ – bottom-most domain in concrete policy

$[\![\top]\!]_\sigma = H$ – top-most domain in concrete policy

$[\![x_\delta]\!]_\sigma = \sigma(x_\delta)$ – domain that is stored in x_δ

- similarly for fields

- f_δ and x_δ are themselves public, formally: $\Gamma_m(x_\delta) = \bot$ and $\Phi(f_\delta) = \bot$

Noninterference

- data visible at *k* may only be influenced by data visible at *k*
- *k*-indistiguishable states: $\sigma_1 \sim_\beta^k \sigma_2$
    - all variables and fields visible at *k* have the same value
    - relative to different address allocations: bijection *β* [Banerjee/Naumann 2003]

A program *P* is secure (noninterferent) if for all domains *k*, for all states σ_1, σ_2, σ_1', σ_2' and all partial bijections *β,*

if $\sigma_1 \sim_\beta^k \sigma_2$ and $\sigma_1 \xrightarrow{P} \sigma_1'$ and $\sigma_2 \xrightarrow{P} \sigma_2'$, then $\sigma_1' \sim_\gamma^k \sigma_2'$

for some $\gamma \supseteq \beta$.

Overview

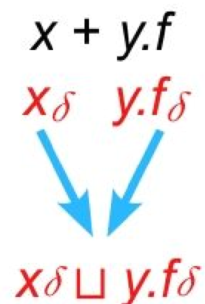
1. Noninterference with dynamic security domains and policies
2. High-level type system
3. Bytecode language

Information flow type system

- type system verifies noninterference statically [Volpano et al, 1996]
  1. compute security domains of expressions
  2. verify that statements respect security policy
- soundness: if program is typable, then it is secure, i.e. noninterferent
- here: security domains and policy depend on the execution environment
  - extend type system to reason over these abstractly

Abstract information flow type system

1. compute abstract domains of expressions



abstract domains = types = **labels**:

$$\mathcal{L} \ni \ell \ ::= \ \top \mid \bot \mid x_\delta \mid e.f_\delta \mid \ell \sqcup \ell$$

- subclass of expressions that evaluate to a domain
- typing judgement: $\Gamma \vdash e : \ell$

"e depends at most on information of domain $[\![\ell]\!]_\sigma$"

2. verify that statements respect abstract security policy

- abstract reasoning using lattice facts like $\bot \sqsubseteq x_\delta$
- not enough, need runtime queries of policy in program

Constraint sets

- example:

$$\texttt{\textbf{if}}\ \top \sqsubseteq \texttt{out.f}_\delta\ \texttt{\textbf{then}}\ \texttt{out.contents := secret}\ \texttt{\textbf{else}}\ \texttt{\textbf{skip}}$$

  - if Φ(contents)=f_δ: “then” branch is secure because it has been tested whether flow is permitted

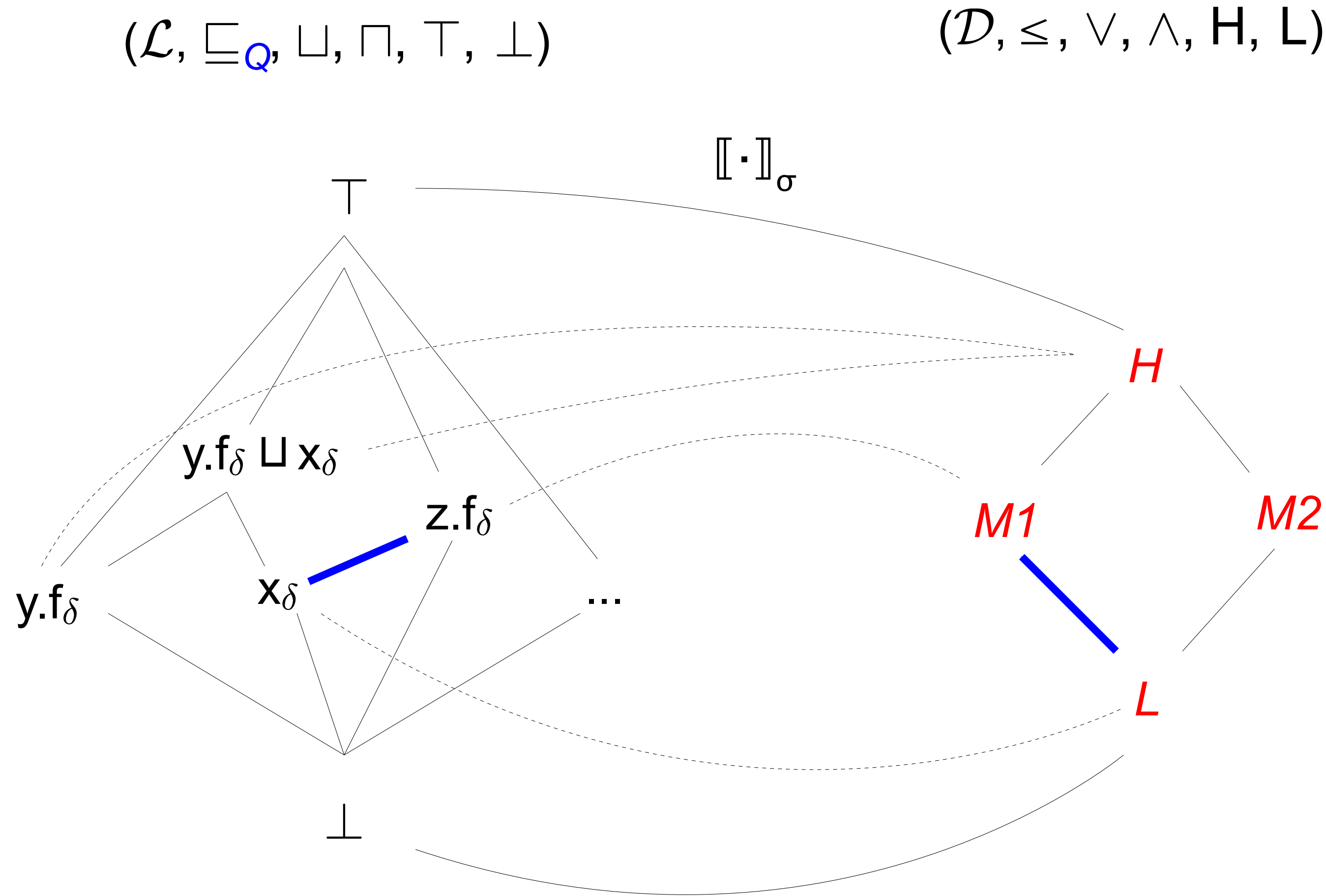
- in general:

$$\texttt{\textbf{if}}\ \ell_1 \sqsubseteq \ell_2\ \texttt{\textbf{then}}\ \texttt{P}_1\ \texttt{\textbf{else}}\ \texttt{P}_2$$

  - we use information “$\ell_1 \sqsubseteq \ell_2$” for analysis of P_1
  - collect label order information in a constraint set *Q*

- program typing judgement: $\Gamma, pc \vdash Q\,\{P\}\,Q'$
  - “if pre-state satisfies *Q*, then P is noninterferent and post-state satisfies *Q*”
  - Hoare-style reasoning using pre- and post-sets

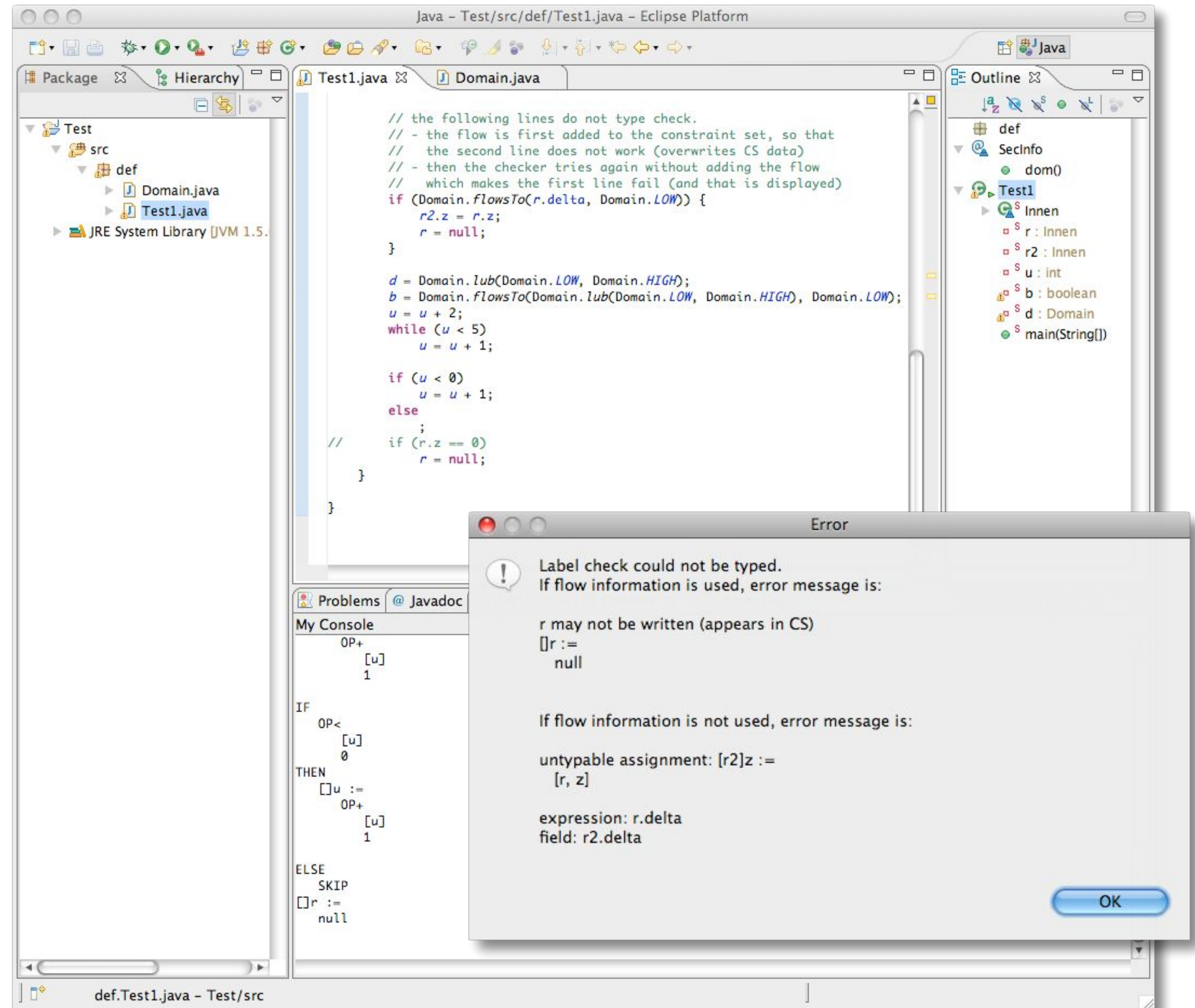
Label lattice and domain lattice



- basic label lattice describes properties of any domain lattice (policy)
- Q stores information about specific state and policy
- $[\![\cdot]\!]_\sigma$ is a lattice homomorphism (if σ satisfies Q)

Implementation of the type system

- type-checking plug-in for Eclipse (with Florian Lasinger)
- language encoded into subset of Java
- security types as Java annotations



Overview

1. Noninterference with dynamic security domains and policies
2. High-level type system
3. Bytecode language

Bytecode semantics by example

```
while x.f > 10 do
    z := y.m(x)
```

i	P(i)	ρ(i)
1	**load** *x*	[a_2]
2	**getfield** *f*	[4]
3	**push** 10	[10,4]
4	**prim** >	[1]
5	**ifeq** 7	[]
6	**goto** 12	[]
7	**load** *x*	[15]
8	**load** *y*	[13]
9	**call** *m*	[2]
10	**store** *z*	[]
11	**goto** 1	[]
12	...	

- stack-based, unstructured language
- conditional and unconditional jumps instead of structured *if* and *while*
- sequence of stack operations instead of expressions

Bytecode type system

```
while x.f > 10 do
    z := y.m(x)
```

i	P(i)	S(i)
1	**load** *x*	[⊥]
2	**getfield** *f*	$[x.f_\delta]$
3	**push** 10	$[\bot, x.f_\delta]$
4	**prim** >	$[\bot \sqcup x.f_\delta]$
5	**ifeq** 7	[]
6	**goto** 12	[]
7	**load** *x*	$[\bot \sqcup x.f_\delta]$
8	**load** *y*	$[\top, \bot \sqcup x.f_\delta]$
9	**call** *m*	$[\top]$
10	**store** *z*	[]
11	**goto** 1	[]
12	...	

- typing judgements: abstraction of actual semantics
- assign labels to values on stack at each program point
- compute control dependence regions

Constraint sets

```
if ⊤ ⊑ out.fδ then
   (P1)
else
   (P2)
```

i	P(i)	E(i)	Q(i)
1	**load** *out*	["out"]	{}
2	**getfield** f_δ	["out.f_δ"]	{}
3	**push** $\top$	["$\top$", "out.f_δ"]	{}
4	**prim** $\sqsubseteq$	["$\top \sqsubseteq$ out.f_δ"]	{}
5	**ifeq** 17	[]	{}
	...		
16	**goto** 34		
17	...(P_1)...		{ $\top \sqsubseteq$ out.f_δ }
	...		{ $\top \sqsubseteq$ out.f_δ }
34	...		

- for constraint sets: need to know which expression is actually on stack
- store symbolic expression information in stack typing

Soundness

- typing judgements: abstract small-step semantics

$$\Gamma, Q, E, pc, i \vdash S \Rightarrow S'$$

- for two parallel small steps from program point *i*:
    - if initial stacks are *k*-indistinguishable with respect to *S*
      and Q(i) holds
      and E(i) evaluates to the stacks,
      then final stacks are *k*-indistinguishable with respect to *S'*
    - similar for stores and heaps

- soundness:
    - if there is a typing judgement for each point *i* of a method *m*, then *m* is secure

Summary

Results

- information flow analysis with dynamic security domains and policies
- type system with symbolic flow analysis using abstract label lattice, independent of actual domains and policy
- translated analysis to bytecode level
- soundness results

Future work

- proof that compilation to bytecode preserves types
- embed into PCC architecture
- polymorphic information flow security
- larger case study

Thank you for your attention!

- state model:

$$\begin{array}{rrcl} \text{states:} & \sigma \in \Sigma & = & \mathcal{S} \times \mathcal{H} \\ \text{stores:} & s \in \mathcal{S} & = & \mathcal{X} \rightharpoonup \mathcal{V} \\ \text{heaps:} & h \in \mathcal{H} & = & \mathcal{A} \rightharpoonup \mathcal{O} \\ \text{objects:} & o \in \mathcal{O} & = & \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}) \end{array}$$

includes f_δ

- methods:

$$\text{mtable} : \mathcal{C} \rightarrow (\mathcal{M} \rightharpoonup \overline{\mathcal{X}} \times \mathcal{P})$$

includes x_δ

dynamic dispatch

- use special variables *this* and *ret* for self reference and return value

s	h
$x_\delta \rightarrow M$ $x_1 \rightarrow 4$ $x_2 \rightarrow a_2$ $this \rightarrow a_2$ $ret \rightarrow 6$	$a_1 \rightarrow$ C_1: $f_\delta \rightarrow L$, $f_1 \rightarrow 5$, $f_2 \rightarrow a_2$ $a_2 \rightarrow$ C_2: $f_\delta \rightarrow M$, $f_6 \rightarrow a_2$

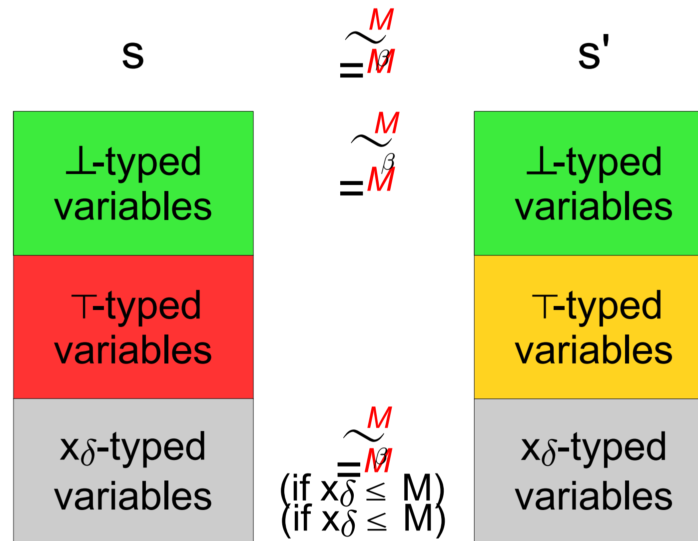
$C_1 \rightarrow m(x_\delta, x_6, x_7)\ \{P_1\}$
$C_2 \rightarrow m(x_\delta, x_4)\ \{P_2\}$

$x_3 := x_2.m(H, 42)$

s':	h: (unchanged)
$x_\delta \rightarrow H$ $x_4 \rightarrow 42$ $this \rightarrow a_2$ $ret \rightarrow 0$	

Indistinguishable states with dynamic domains

- two states are indistinguishable for domain *k* if all their data visible at *k* has the same values
- example for *M*-indistinguishable stores (objects are similar):

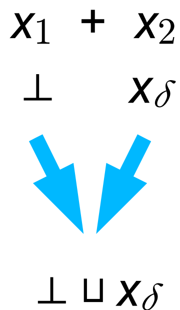


- “same value” is relative to different address allocations
  - relate locations by bijection β (Banerjee/Naumann 2003)
  - $\sigma_1 \sim_\beta^M \sigma_2$

Expression typing

$$\Gamma \vdash e : \ell$$

- "expression *e* in method *m* depends on data of security domain *k*"
- typing rules capture flows without side effects, for example:
  - from operands to result:



$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \ell_1 \qquad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \mathbf{op}\, e_2 : \ell_1 \sqcup \ell_2}$$

soundness: if $\Gamma \vdash e : \ell$ and $\sigma \sim_\beta^k \sigma'$ and $[\![\ell]\!]_\sigma \leq k$, then $[\![e]\!]_\sigma \sim_\beta [\![e]\!]_{\sigma'}$

Program typing

$\Gamma, pc \vdash Q\ \{P\}\ Q'$

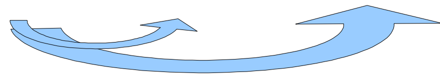
- "program *P* in method *m* does not contain insecure flows"
  - assignments: from assigned expression to variable/field
    field write access: from reference to field (aliasing)

x . f := *e*

$$\frac{\begin{array}{c}\Gamma \vdash \pi : \ell_1 \qquad \Gamma \vdash e : \ell_2 \\ \ell_1 \sqcup \ell_2 \sqcup pc_h \leq_Q \mathrm{ft}_\pi(f) \\ f = f_\delta \Rightarrow \pi.f \leq_Q e \\ f \notin pc \qquad f \notin Q'[e/\pi.f]\end{array}}{\Gamma, pc \vdash Q'[e/\pi.f] \cup Q\ \{\pi.f := e\}\ Q'}$$

  - if/while: from branched expression to assigned variables/fields (nonlocal)

if x **then** y := 1 **else** y := 2



$$\frac{\begin{array}{c}\Gamma \vdash e : \ell \\ \Gamma, pc \sqcup \ell \vdash Q\ \{P_1\}\ Q' \\ \Gamma, pc \sqcup \ell \vdash Q\ \{P_2\}\ Q'\end{array}}{\Gamma, pc \vdash Q\ \{\textbf{if}\ e\ \textbf{then}\ P_1\ \textbf{else}\ P_2\}\ Q'}$$

  - method call: various flows

$x_0 := e_0.m(e_\delta, e_1)$

local variables: *ret* *this* x_δ x_1

Type system

$$\frac{\Gamma, pc \vdash Q_0\ \{P\}\ Q_0' \qquad Q \Rightarrow Q_0 \qquad Q_0' \Rightarrow Q'}{\Gamma, pc \vdash Q\ \{P\}\ Q'} \qquad \frac{}{\Gamma, pc \vdash Q\ \{\textbf{skip}\}\ Q}$$

$$\frac{\Gamma, pc \vdash Q\ \{P_1\}\ Q' \quad \Gamma, pc \vdash Q'\ \{P_2\}\ Q''}{\Gamma, pc \vdash Q\ \{P_1\ ;\ P_2\}\ Q''} \qquad \frac{\Gamma \vdash e : \ell \quad \Gamma, pc \sqcup \ell \vdash Q\ \{P\}\ Q}{\Gamma, pc \vdash Q\ \{\textbf{while}\ e\ \textbf{do}\ P\}\ Q}$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, pc \sqcup \ell \vdash Q\ \{P_1\}\ Q' \quad \Gamma, pc \sqcup \ell \vdash Q\ \{P_2\}\ Q'}{\Gamma, pc \vdash Q\ \{\textbf{if}\ e\ \textbf{then}\ P_1\ \textbf{else}\ P_2\}\ Q'}$$

$$\frac{\Gamma \vdash \ell_1 \sqsubseteq \ell_2 : \ell \quad \Gamma, pc \sqcup \ell \vdash Q, \ell_1 \sqsubseteq \ell_2\ \{P_1\}\ Q' \quad \Gamma, pc \sqcup \ell \vdash Q\ \{P_2\}\ Q'}{\Gamma, pc \vdash Q\ \{\textbf{if}\ \ell_1 \sqsubseteq \ell_2\ \textbf{then}\ P_1\ \textbf{else}\ P_2\}\ Q'}$$

$$\frac{\Gamma \vdash e : \ell \quad \ell \sqcup pc_s \leq_Q \Gamma(x) \quad x = x_\delta \Rightarrow x \leq_Q e \quad x \notin pc}{\Gamma, pc \vdash Q'[e/x] \cup Q\ \{x := e\}\ Q'} \qquad \frac{\Gamma \vdash \pi : \ell_1 \quad \Gamma \vdash e : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqcup pc_h \leq_Q \mathrm{ft}_\pi(f) \quad f = f_\delta \Rightarrow \pi.f \leq_Q e \quad f \notin pc \quad f \notin Q'[e/\pi.f]}{\Gamma, pc \vdash Q'[e/\pi.f] \cup Q\ \{\pi.f := e\}\ Q'}$$

$$\frac{pc_s \leq_Q \Gamma(x) \qquad x \notin pc \qquad x \neq x_\delta \qquad x \notin Q'}{\Gamma, pc \vdash Q' \cup Q\ \{x := \textbf{new}\ C\}\ Q', x.f_\delta \leq \bot}$$

$$\frac{\begin{array}{c}\Gamma \vdash \pi : \ell_{this} \qquad \Gamma \vdash \overline{e} : \overline{\ell} \qquad \mathrm{mtype}(m)[\overline{e}_{\#1}/x_\delta] = \ell'_{this}, \overline{\ell}' \xrightarrow{pc_h'} \ell_{ret} \\ \ell_{this} \leq_Q \ell'_{this} \qquad \overline{\ell} \leq_Q \overline{\ell}' \qquad pc_h \sqcup \ell_{this} \leq_Q pc_h' \qquad \ell_{ret} \sqcup \ell_{this} \sqcup pc_s \leq_\emptyset \Gamma(x) \\ \mathrm{mreq}(m)[\overline{e}/\ \mathrm{margs}(m)][\pi/this] = Q \qquad \mathrm{mens}(m)[x/ret] = Q' \\ x \notin pc \qquad x \neq x_\delta\end{array}}{\Gamma, pc \vdash Q\ \{x := \pi.m(\overline{e})\}\ Q'}$$

Meta-label monotonicity

- consider the following case:
  - two indistinguishable states $\sigma_1 \sim_{\beta}^{M} \sigma_2$
  - expression *e* typed with label ℓ
  - ℓ evaluates to *L* in σ_1, to *H* in σ_2
- then ℓ contains confidential data
  - otherwise it would have the same value in both states
  - "the fact that *e* is public in σ_1 is confidential"
- it can be shown: such labels do not occur in the type system
  - the value of a label ℓ is always a least as confidential as the value of the label of ℓ

Bytecode language

- stack-based, unstructured language
- instruction set:

  B ∋ b ::= **push** *v* | **prim** *op* | **store** *x* | **ifeq** *j* | **goto** *j* |
  putfield *f* | **getfield** *f* | **new** *C* | **call** *m* | **return**

- bytecode program: P = [b_1, b_2, b_3, b_4....]

- small-step semantics

  $(i, s, h, \rho) \rightarrow (i', s', h', \rho')$

  - *i* : instruction pointer
  - s, h: store and heap (as before)
  - ρ: operand stack

Control dependence regions

- syntax provides
  no structural information

  while $x.f$ > 10 **do**
  $z := y.m(x)$

- need to know program blocks
  to prevent indirect flows
- solution: assume given
  control dependency analysis
  - region between branch
    and junction point
- do not give computation, but only
  safe over-approximation properties
- label of branching expression affects at least blue instructions

i	P(i)	S(i)	pc(i)
1	**load** x	$[\bot]$	$\bot$
2	**getfield** f	$[x.f_\delta]$	$\bot$
3	**push** 10	$[\bot, x.f_\delta]$	$\bot$
4	**prim** >	$[\bot \sqcup x.f_\delta]$	$\bot$
5	**ifeq** 11	$[]$	$\bot$
6	**load** x	$[\bot \sqcup x.f_\delta]$	$\bot \sqcup x.f_\delta$
7	**load** y	$[\top, \bot \sqcup x.f_\delta]$	$\bot \sqcup x.f_\delta$
8	**call** m	$[\top]$	$\bot \sqcup x.f_\delta$
9	**store** z	$[]$	$\bot \sqcup x.f_\delta$
10	**goto** 1	$[]$	$\bot \sqcup x.f_\delta$
11			$\bot$

Type system features that were not shown

- *pc* label is pair/triple
  - pc_s/se_s for store side effects
  - pc_h/se_h for heap side effects
  - se_ρ for stack side effects
- methods have a signature that includes:
  - types of formal arguments, including *this* and *ret*
  - pc_h
  - required and ensured constraint sets, which may refer to local variables/fields
- additional challenges addressed by type system:
  - f_δ field may only be overwritten with higher domain to prevent declassification of f_δ-typed fields; same with x_δ
  - overwritten fields may not conflict with *pc* label (invariant)