

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

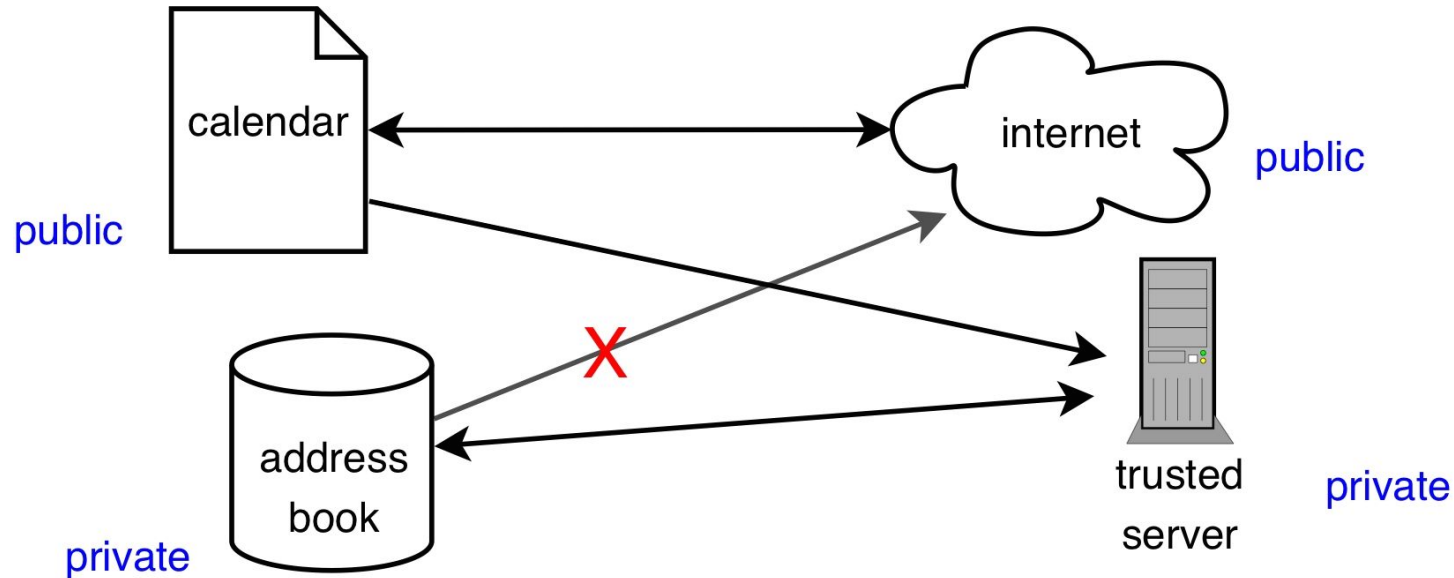


InfoZert

LMU

Information flow security

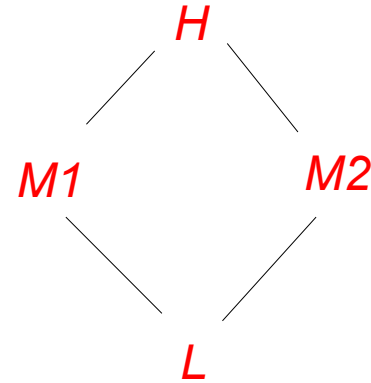
- example: personal organizer application



- 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:
$$x_L := y_L.f_H * 2 \quad \text{if } z_H > 5 \text{ then } x_L := 1 \text{ else } x_L := 0$$
- 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 : StringL;  
    readInt() : IntL;  
    writeInt(v: IntL);  
}
```

```
class FileHigh {  
    contents : StringH;  
    readInt() : IntH;  
    writeInt(v: IntH);  
}
```

- 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 {  
   $\delta$ : Domain $\perp$ ;  
  contents : String $\delta$ ;  
  readInt() : Int $\delta$ ;  
  writeInt(v: Int $\delta$ );  
}
```

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

```
out := new File(...);  
...  
if  $\top \sqsubseteq$  out. $\delta$  then  
  out.writeInt(secret $\top$ );
```

- 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 $f \in \mathcal{F}$

$$\mathcal{V} \ni v ::= n \mid a \mid k$$
$$\mathcal{E} \ni e ::= n \mid x \mid e.f \mid e + e \mid e < e \mid \top \mid \perp \mid e \sqcup e \mid e \sqsubseteq e$$
$$\mathcal{P} \ni P ::= \mathbf{skip} \mid P ; P \mid \mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ P \mid \mathbf{while} \ e \ \mathbf{do} \ P \mid \\ x := e \mid e.f := e \mid x := \mathbf{new} \ C \mid x := e.m(\bar{e})$$

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: $\llbracket e \rrbracket_\sigma$
 - big-step operational semantics for programs: $\sigma_1 \xrightarrow{P} \sigma_2$

Visibility of data

- assign **abstract** domains using type environments

$$\begin{aligned}\Gamma_m &: \mathcal{X} \rightarrow \{\top, \perp, x_\delta\} && \text{– current variable type environment} \\ \Phi &: \mathcal{F} \rightarrow \{\top, \perp, f_\delta\} && \text{– global class-insensitive field typing}\end{aligned}$$

- a variable x in a state σ is *visible* at a domain k if $\llbracket \Gamma(x) \rrbracket_\sigma \leq k$

$$\llbracket \perp \rrbracket_\sigma = L \quad \text{– bottom-most domain in concrete policy}$$

$$\llbracket \top \rrbracket_\sigma = H \quad \text{– top-most domain in concrete policy}$$

$$\llbracket x_\delta \rrbracket_\sigma = \sigma(x_\delta) \quad \text{– domain that is stored in } x_\delta$$

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

Noninterference

- data visible at k may only be influenced by data visible at k
- k -indistinguishable 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 $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ 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

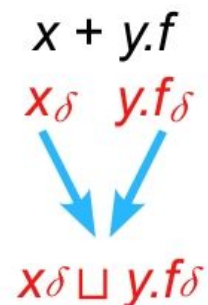
1. Noninterference with dynamic security domains and policies
2. **High-level type system**
3. Bytecode language

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**:

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

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

“e depends at most on information of domain $\llbracket \ell \rrbracket_\sigma$ ”

2. verify that statements respect abstract security policy

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

Constraint sets

- example:

```
if  $T \sqsubseteq \text{out}.f_\delta$  then  $\text{out}.contents := \text{secret}$  else skip
```

- if $\Phi(\text{contents})=f_\delta$: “then” branch is secure because it has been tested whether flow is permitted

- in general:

```
if  $l_1 \sqsubseteq l_2$  then  $P_1$  else  $P_2$ 
```

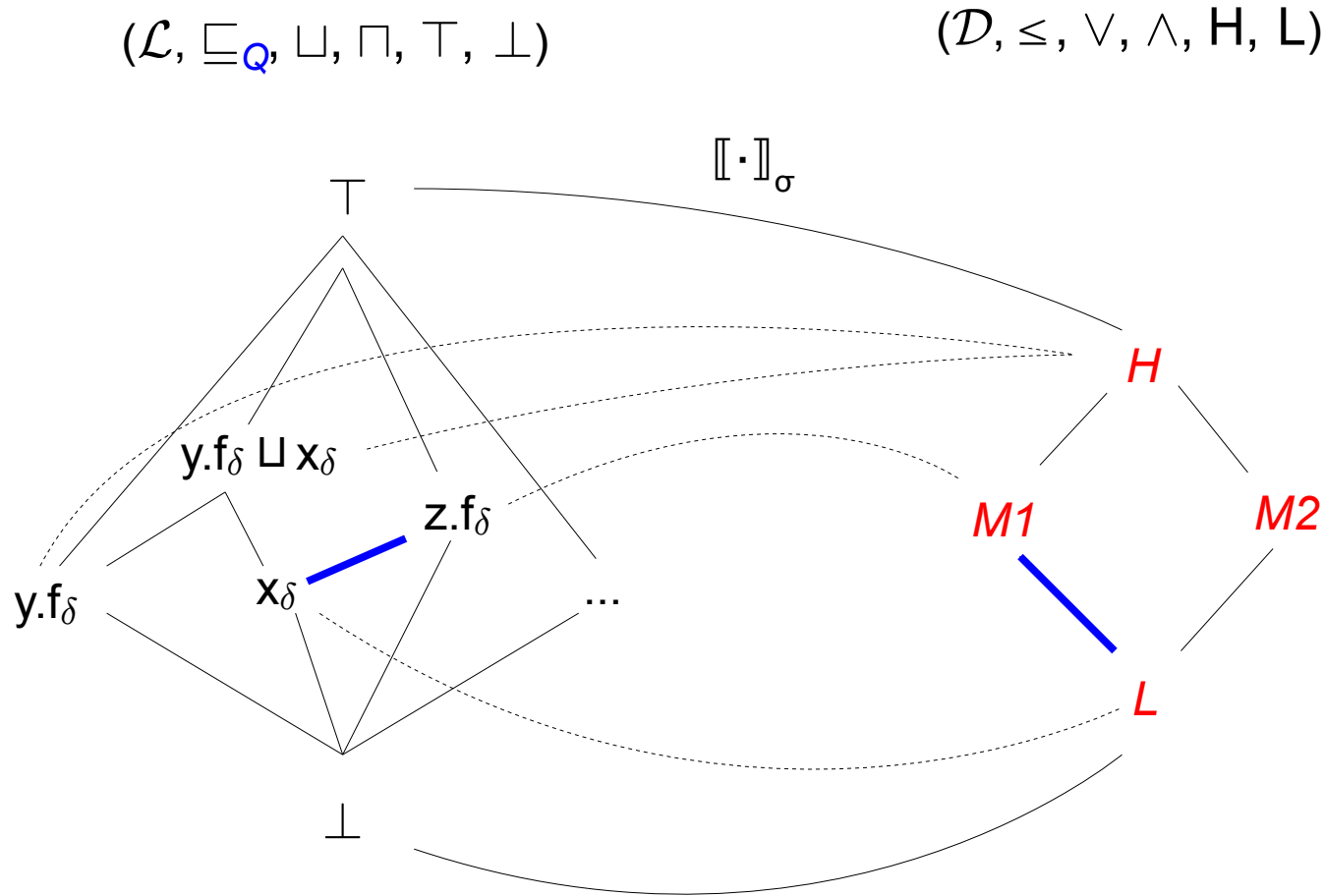
- we use information “ $l_1 \sqsubseteq l_2$ ” for analysis of P_1
- collect label order information in a constraint set Q

- 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

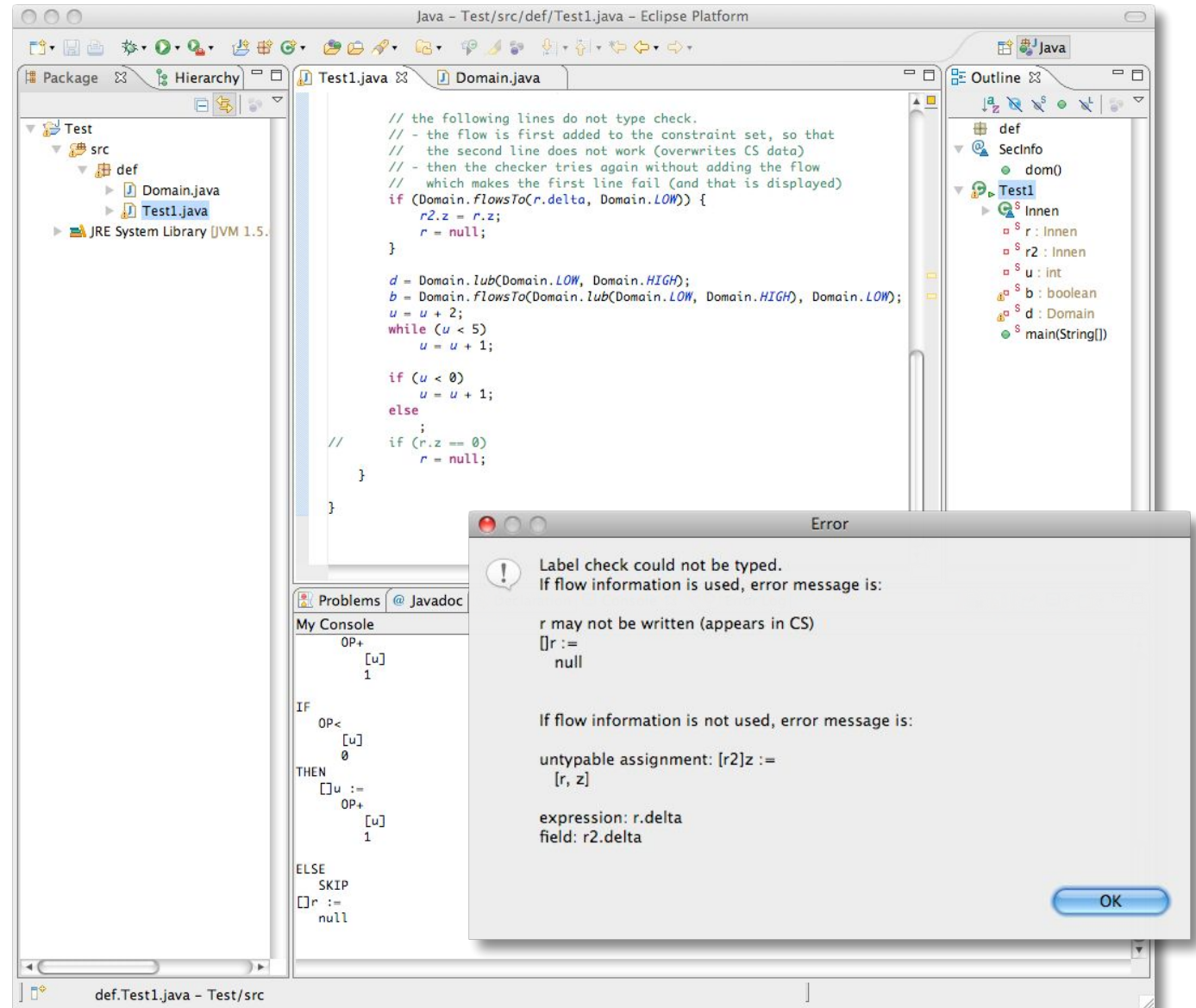
Label lattice and domain lattice



- basic label lattice describes properties of any domain lattice (policy)
- Q stores information about specific state and policy
- $\llbracket \cdot \rrbracket_\sigma$ 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	$P(i)$	$\rho(i)$
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	$[\perp]$
2	getfield f	$[x.f_\delta]$
3	push 10	$[\perp, x.f_\delta]$
4	prim $>$	$[\perp \sqcup x.f_\delta]$
5	ifeq 7	$[\]$
6	goto 12	$[\]$
7	load x	$[\perp \sqcup x.f_\delta]$
8	load y	$[\top, \perp \sqcup x.f_\delta]$
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 $T \sqsubseteq \text{out}.f_\delta$ **then**

(P₁)

else

(P₂)



i	P(i)	E(i)	Q(i)
1	load out	["out"]	{}
2	getfield f_δ	["out.f _δ "]	{}
3	push T	["T", "out.f _δ "]	{}
4	prim \sqsubseteq	["T \sqsubseteq out.f _δ "]	{}
5	ifeq 17	[]	{}
...	...		
16	goto 34		
17	...(P ₁)...		{ T \sqsubseteq out.f _δ }
	...		{ T \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!

Language semantics

- state model:

states: $\sigma \in \Sigma = \mathcal{S} \times \mathcal{H}$
 stores: $s \in \mathcal{S} = \mathcal{X} \rightarrow \mathcal{V}$
 heaps: $h \in \mathcal{H} = \mathcal{A} \rightarrow \mathcal{O}$
 objects: $o \in \mathcal{O} = \mathcal{C} \times (\mathcal{F} \rightarrow \mathcal{V})$

includes f_δ

- methods:

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

includes x_δ

dynamic dispatch

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

s		h						
$x_\delta \rightarrow M$		$a_1 \rightarrow$ <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>C₁</td> <td>$f_\delta \rightarrow L$</td> </tr> <tr> <td></td> <td>$f_1 \rightarrow 5$</td> </tr> <tr> <td></td> <td>$f_2 \rightarrow a_2$</td> </tr> </table>	C₁	$f_\delta \rightarrow L$		$f_1 \rightarrow 5$		$f_2 \rightarrow a_2$
C₁	$f_\delta \rightarrow L$							
	$f_1 \rightarrow 5$							
	$f_2 \rightarrow a_2$							
$x_1 \rightarrow 4$								
$x_2 \rightarrow a_2$		$a_2 \rightarrow$ <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>C₂</td> <td>$f_\delta \rightarrow M$</td> </tr> <tr> <td></td> <td>$f_6 \rightarrow a_2$</td> </tr> </table>	C₂	$f_\delta \rightarrow M$		$f_6 \rightarrow a_2$		
C₂	$f_\delta \rightarrow M$							
	$f_6 \rightarrow a_2$							
<i>this</i> $\rightarrow a_2$								
<i>ret</i> $\rightarrow 6$								

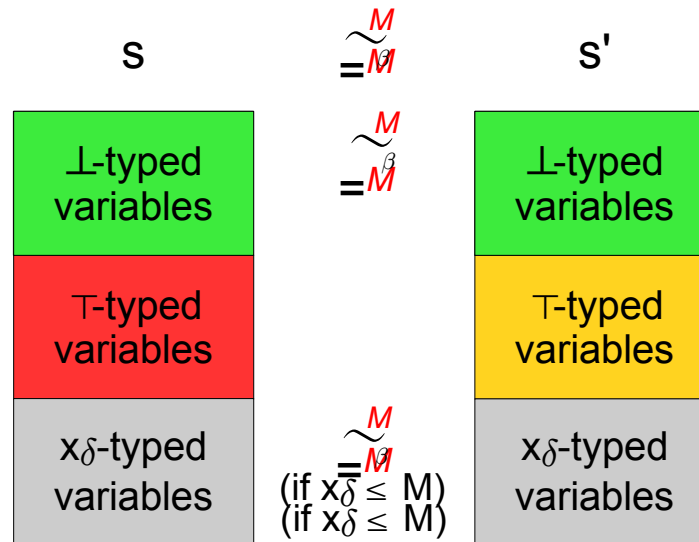
$C_1 \rightarrow m(x_\delta, x_6, x_7) \{P_1\}$
 $C_2 \rightarrow m(x_\delta, x_4) \{P_2\}$

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

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

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)
 - $\sigma_1 \overset{M}{\sim}_\beta \sigma_2$

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:

$x_1 + x_2$
 $\perp \quad x_\delta$

 $\perp \sqcup x_\delta$

$\frac{}{\Gamma \vdash x : \Gamma(x)}$

$\frac{\Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \text{ op } e_2 : \ell_1 \sqcup \ell_2}$

soundness: if $\Gamma \vdash e : \ell$ and $\sigma \sim_\beta^k \sigma'$ and $\llbracket \ell \rrbracket_\sigma \leq k$, then $\llbracket e \rrbracket_\sigma \sim_\beta \llbracket e \rrbracket_{\sigma'}$

Program typing

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

- “program P in method m does not contain insecure flows”

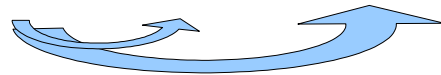
- assignments: from assigned expression to variable/field
- field write access: from reference to field (aliasing)

$x.f := e$

$$\frac{\begin{array}{l} \Gamma \vdash \pi : l_1 \quad \Gamma \vdash e : l_2 \\ l_1 \sqcup l_2 \sqcup pc_h \leq_Q ft_\pi(f) \\ f = f_\delta \Rightarrow \pi.f \leq_Q e \\ f \notin pc \quad f \notin Q'[e/\pi.f] \end{array}}{\Gamma, pc \vdash Q'[e/\pi.f] \cup Q \{ \pi.f := e \} Q'}$$

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

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



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

- method call: various flows

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

local variables: ret $this$ x_δ x_1

Type system

$$\frac{\Gamma, pc \vdash Q_0 \{P\} Q'_0 \quad Q \Rightarrow Q_0 \quad Q'_0 \Rightarrow Q'}{\Gamma, pc \vdash Q \{P\} Q'} \quad \frac{}{\Gamma, pc \vdash Q \{\text{skip}\} Q}$$

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

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

$$\frac{\Gamma \vdash l_1 \sqsubseteq l_2 : l \quad \Gamma, pc \sqcup l \vdash Q, l_1 \sqsubseteq l_2 \{P_1\} Q' \quad \Gamma, pc \sqcup l \vdash Q \{P_2\} Q'}{\Gamma, pc \vdash Q \{\text{if } l_1 \sqsubseteq l_2 \text{ then } P_1 \text{ else } P_2\} Q'}$$

$$\frac{\Gamma \vdash e : l \quad l \sqcup pc_s \leq_Q \Gamma(x) \quad x = x_\delta \Rightarrow x \leq_Q e \quad x \notin pc}{\Gamma, pc \vdash Q'[e/x] \cup Q \{x := e\} Q'} \quad \frac{\Gamma \vdash \pi : l_1 \quad \Gamma \vdash e : l_2 \quad l_1 \sqcup l_2 \sqcup pc_h \leq_Q ft_\pi(f) \quad f = f_\delta \Rightarrow \pi.f \leq_Q e \quad f \notin pc \quad f \notin Q'[e/\pi.f]}{\Gamma, pc \vdash Q'[e/\pi.f] \cup Q \{\pi.f := e\} Q'}$$

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

$$\frac{\Gamma \vdash \pi : l_{this} \quad \Gamma \vdash \bar{e} : \bar{l} \quad \text{mtype}(m)[\bar{e}_{\#1}/x_\delta] = l'_{this}, \bar{l}' \xrightarrow{pc'_h} l_{ret} \quad l_{this} \leq_Q l'_{this} \quad \bar{l} \leq_Q \bar{l}' \quad pc_h \sqcup l_{this} \leq_Q pc'_h \quad l_{ret} \sqcup l_{this} \sqcup pc_s \leq_\emptyset \Gamma(x) \quad \text{mreq}(m)[\bar{e}/\text{margs}(m)][\pi/this] = Q \quad \text{mens}(m)[x/ret] = Q' \quad x \notin pc \quad x \neq x_\delta}{\Gamma, pc \vdash Q \{x := \pi.m(\bar{e})\} Q'}$$

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 at least as confidential as the value of the label of ℓ

Bytecode language

- stack-based, unstructured language

- instruction set:

$B \ni b ::= \mathbf{push} \ v \mid \mathbf{prim} \ op \mid \mathbf{store} \ x \mid \mathbf{ifeq} \ j \mid \mathbf{goto} \ j \mid$
 $\mathbf{putfield} \ f \mid \mathbf{getfield} \ f \mid \mathbf{new} \ C \mid \mathbf{call} \ m \mid \mathbf{return}$

- bytecode program: $P = [b_1, b_2, b_3, b_4, \dots]$

- 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
   $z := y.m(x)$ 
```

indirect flow



- 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	$[\perp]$	\perp
2	getfield f	$[x.f_\delta]$	\perp
3	push 10	$[\perp, x.f_\delta]$	\perp
4	prim >	$[\perp \sqcup x.f_\delta]$	\perp
5	ifeq 11	$[\]$	\perp
6	load x	$[\perp \sqcup x.f_\delta]$	$\perp \sqcup x.f_\delta$
7	load y	$[\top, \perp \sqcup x.f_\delta]$	$\perp \sqcup x.f_\delta$
8	call m	$[\top]$	$\perp \sqcup x.f_\delta$
9	store z	$[\]$	$\perp \sqcup x.f_\delta$
10	goto 1	$[\]$	$\perp \sqcup x.f_\delta$
11			\perp

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)