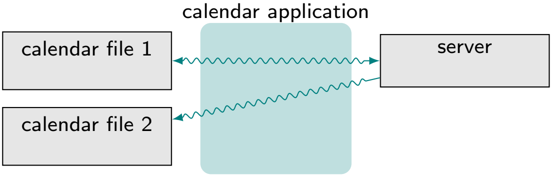
Noninterference for Dynamic Security Environments

Robert Grabowski

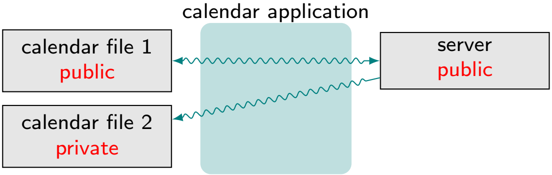
Princeton University, Princeton, NJ

Programming Languages Day, July 29, 2010
IBM Research, Hawthorne, NY

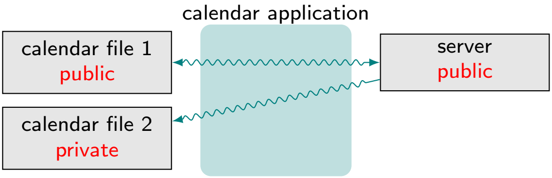
Noninterference

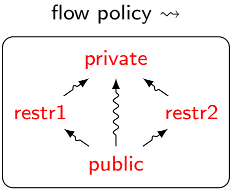


Noninterference

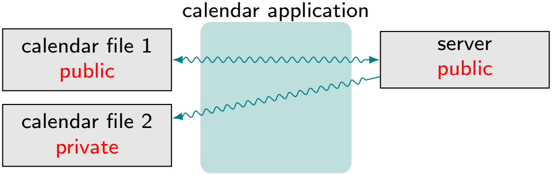


Noninterference

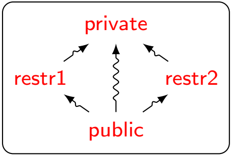




Noninterference

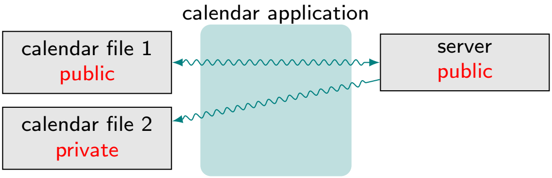


flow policy $\rightsquigarrow$

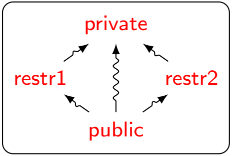


application is *noninterferent* with respect to $\rightsquigarrow$:
data of domain A does not influence computation of data of domain B unless A $\rightsquigarrow$ B

Noninterference



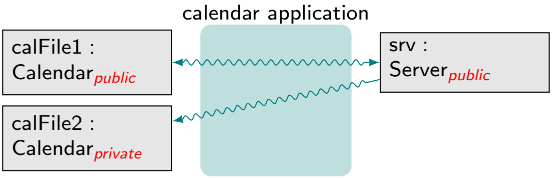
flow policy $\rightsquigarrow$

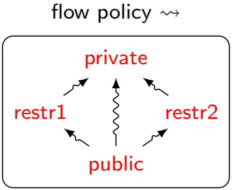


information flow analysis for $\rightsquigarrow$

application is *noninterferent* with respect to $\rightsquigarrow$:
data of domain A does not influence computation of data of domain B unless A $\rightsquigarrow$ B

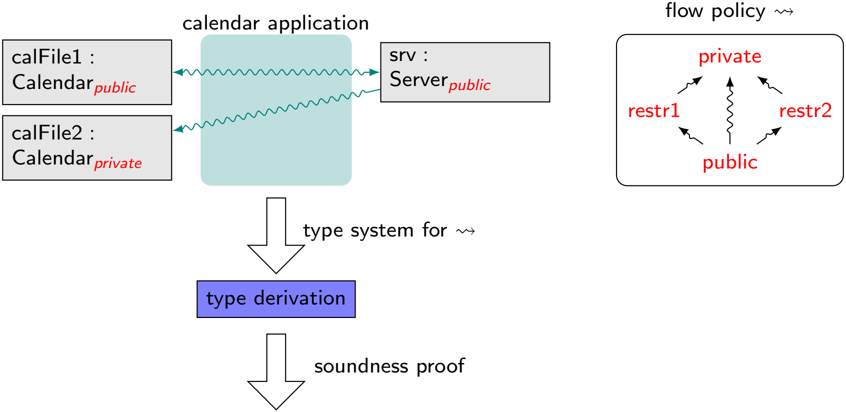
Type-based information flow analysis





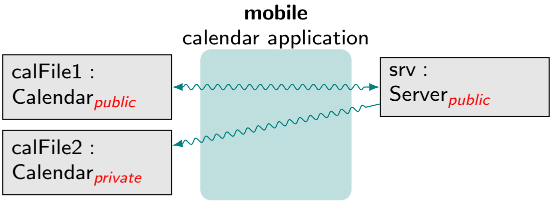
application is *noninterferent* with respect to $\rightsquigarrow$:
data of domain A does not influence computation of data of domain B unless A $\rightsquigarrow$ B

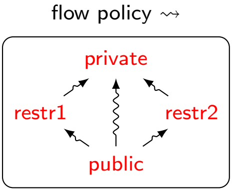
Type-based information flow analysis



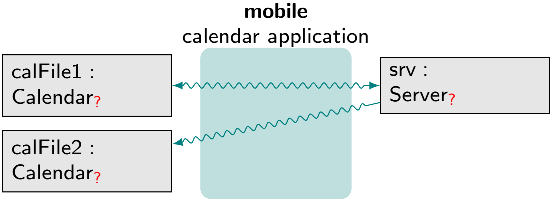
application is *noninterferent* with respect to $\rightsquigarrow$:
data of domain A does not influence computation of data of domain B unless A $\rightsquigarrow$ B

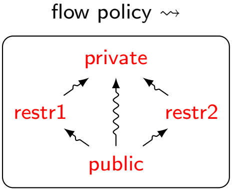
Dynamic security environments



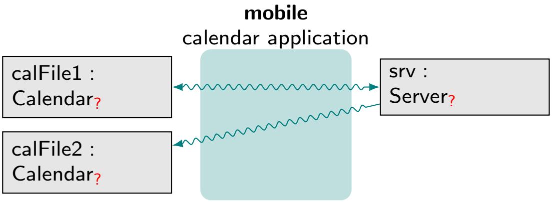


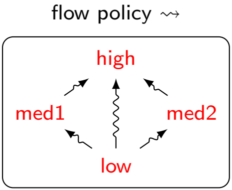
Dynamic security environments



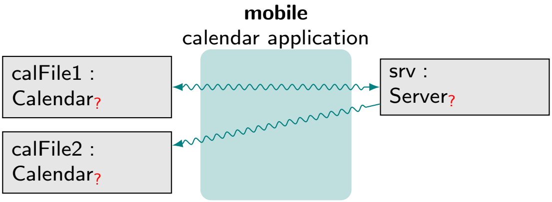


Dynamic security environments





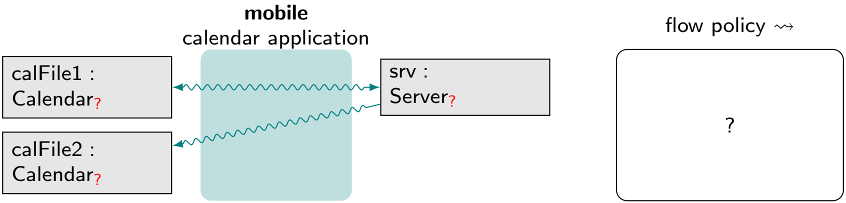
Dynamic security environments



flow policy $\rightsquigarrow$

?

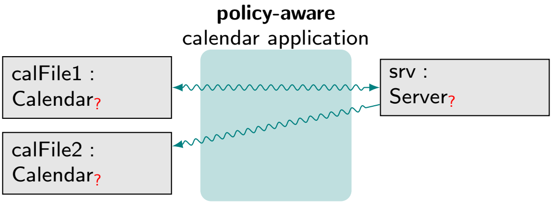
Dynamic security environments



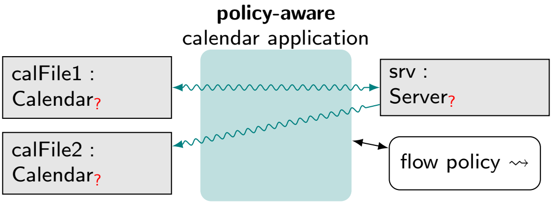
goal:

universal noninterference:
application is secure *for any security environment (domains and policies)*

Approach

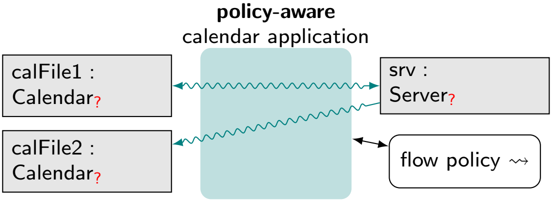

policy-aware
calendar application
calFile1 :
Calendar?
calFile2 :
Calendar?
srv :
Server?

Approach



application can query domains and policy before each flow-inducing action

Approach



application can query domains and policy before each flow-inducing action

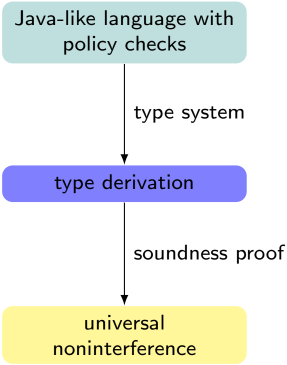
type system
checks that all critical flows have been guarded

type derivation

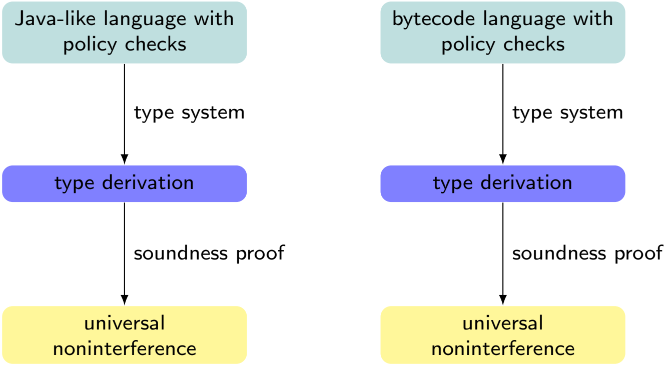
soundness proof

application is noninterferent for any security environment

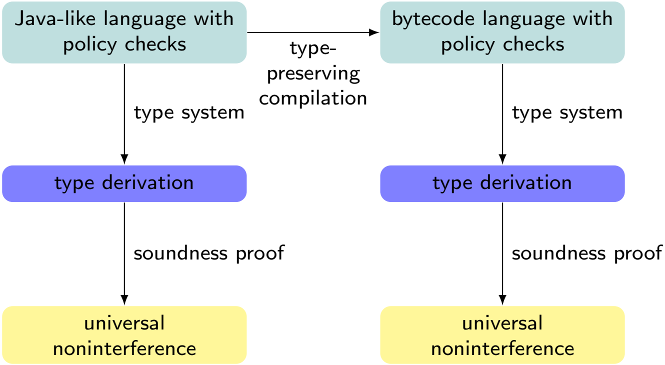
Goal: framework for universal noninterference



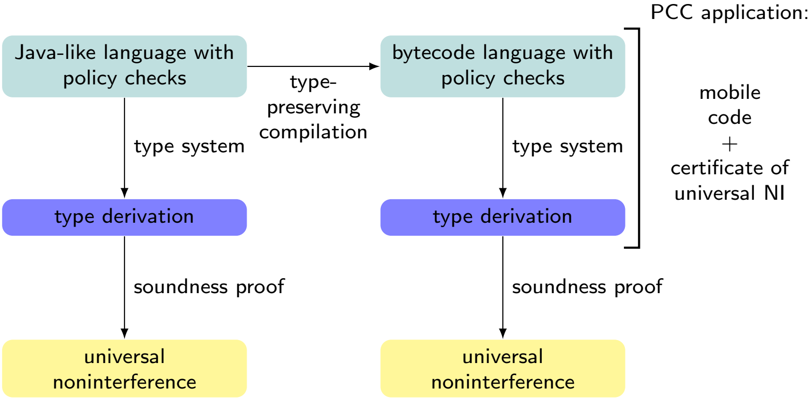
Goal: framework for universal noninterference



Goal: framework for universal noninterference



Goal: framework for universal noninterference



Related work

Information flow type systems for static security environments

- WHILE language [Volpano et al, 1996]
- object-oriented languages [Banerjee/Naumann, 2003]
- bytecode language [Barthe et al, 2005]

Analysis for dynamic security environments

- JIF: Java with Information Flows [Myers, 1999; Zheng/Myers, 2004]
- RTI: dynamic roles [Bandhakavi et al, 2008]
- λ^{deps^+}: dynamic dependency monitoring [Shroff et al, 2007]

Novelty here:

- bytecode with dynamic security environments
- framework for information-flow certification of mobile code

High-level language

- Java-like language with few additions
- dynamic domains encoded in f_δ fields

```
class Calendar {
  fδ: Domain⊥;
  contents : Data_fδ;
}
```

```
class Server {
  fδ: Domain⊥;
  write(d : Data_fδ);
}
```

- domains are first-class values, operator $\sqsubseteq$ for policy queries:

$$\textbf{if } cal.f_\delta \sqsubseteq srv.f_\delta \textbf{ then}$$
$$\quad srv.write(cal.contents);$$

  - flow from *cal.contents* to formal parameter *d* of *srv.write*
  - is only called if this flow is permitted
  - program is universally noninterferent

High-level type system

```
class Calendar {
  f_δ: Domain_⊥;
  contents : Data_{f_δ};
}
```

```
class Server {
  f_δ: Domain_⊥;
  write(d : Data_{f_δ});
}
```

if $cal.f_\delta \sqsubseteq srv.f_\delta$ **then**

$srv.write(cal.contents);$

High-level type system

class Calendar {
  f_δ: Domain$_{\bot}$;
  contents : Data$_{f_\delta}$;
}

class Server {
  f_δ: Domain$_{\bot}$;
  write(d : Data$_{f_\delta}$);
}

if $cal.f_\delta \sqsubseteq srv.f_\delta$ **then**

$srv.write(cal.contents)$;

expected type: $srv.f_\delta$

type of expression: $cal.f_\delta$

1. types are symbolic expressions that refer to a security domain

High-level type system

class Calendar {
  f_δ: Domain$_{\bot}$;
  contents : Data$_{f_\delta}$;
}

class Server {
  f_δ: Domain$_{\bot}$;
  write(d : Data$_{f_\delta}$);
}

if $cal.f_\delta \sqsubseteq srv.f_\delta$ **then**

$srv.write(cal.contents)$;

flow information for **then** branch:
$Q = \{cal.f_\delta \sqsubseteq srv.f_\delta\}$

expected type: $srv.f_\delta$

type of expression: $cal.f_\delta$

1. types are symbolic expressions that refer to a security domain
2. collect information about allowed flows: $\Gamma \vdash \{Q\}\ P\ \{Q'\}$

Bytecode language and type system

if $\mathsf{cal}.f_\delta \sqsubseteq \mathsf{srv}.f_\delta$ **then** $\quad$ *srv.write(cal.contents)*;

#	instruction
1	`load` *cal*
2	`getf` f_δ
3	`load` *srv*
4	`getf` f_δ
5	`prim` $\sqsubseteq$
6	`bnz` 12
⋮	
12	`load` *cal*
13	`getf` *contents*
14	`load` *srv*
15	`call` *write*

Bytecode language and type system

if cal.$f_\delta \sqsubseteq$ srv.f_δ **then** $srv.write(cal.contents)$;

#	instruction	abstract operand stack
1	`load` *cal*	$[\ a_{19}\ ]$
2	`getf` f_δ	$[\ a_3\ ]$
3	`load` *srv*	$[\ a_{17}\ ,\ a_3\ ]$
4	`getf` f_δ	$[\ a_4\ ,\ a_3\ ]$
5	`prim` $\sqsubseteq$	$[a_4 \sqsubseteq a_3]$
6	`bnz` 12	$[]$
$\vdots$		
12	`load` *cal*	$[a_{19}]$
13	`getf` *contents*	$[a_{53}]$
14	`load` *srv*	$[a_{17}, a_{53}]$
15	`call` *write*	$[]$

Bytecode language and type system

if cal.$f_\delta \sqsubseteq$ srv.f_δ **then** $srv.write(cal.contents)$;

#	instruction	abstract operand stack
1	`load` *cal*	$[\ a_{19}\ ]$
2	`getf` f_δ	$[a_3]$
3	`load` *srv*	$[\ a_{17}\ ,\ a_3]$
4	`getf` f_δ	$[a_4,\ a_3]$
5	`prim` $\sqsubseteq$	$[a_4 \sqsubseteq a_3]$
6	`bnz` 12	$[]$
⋮		
12	`load` *cal*	$[a_{19}]$
13	`getf` *contents*	$[a_{53}]$
14	`load` *srv*	$[a_{17}, a_{53}]$
15	`call` *write*	$[]$

Bytecode language and type system

if cal.f_δ $\sqsubseteq$ srv.f_δ **then** $srv.write(cal.contents)$;

#	instruction	abstract operand stack	stack types
1	`load` *cal*	$[\ a_{19}\ ]$	$[\bot]$
2	`getf` f_δ	$[a_3]$	$[\bot]$
3	`load` *srv*	$[\ a_{17}\ ,\ a_3]$	$[\bot,\bot]$
4	`getf` f_δ	$[a_4,\ a_3]$	$[\bot,\bot]$
5	`prim` $\sqsubseteq$	$[a_4 \sqsubseteq a_3]$	$[\bot]$
6	`bnz` 12	$[]$	$[]$
$\vdots$			
12	`load` *cal*	$[a_{19}]$	$[\bot]$
13	`getf` *contents*	$[a_{53}]$	$[\ a_3\ ]$
14	`load` *srv*	$[a_{17},\ a_{53}]$	$[\bot,\ a_3\ ]$
15	`call` *write*	$[]$	$[]$

Bytecode language and type system

if $\text{cal}.f_\delta \sqsubseteq \text{srv}.f_\delta$ **then** $\quad srv.write(cal.contents);$

type: $cal.f_\delta$

#	instruction	abstract operand stack	stack types
1	`load` *cal*	$[\,a_{19}\,]$	$[\bot]$
2	`getf` f_δ	$[\,a_3\,]$	$[\bot]$
3	`load` *srv*	$[\,a_{17}\,,\,a_3\,]$	$[\bot,\bot]$
4	`getf` f_δ	$[\,a_4\,,\,a_3\,]$	$[\bot,\bot]$
5	`prim` $\sqsubseteq$	$[a_4 \sqsubseteq a_3]$	$[\bot]$
6	`bnz` 12	$[]$	$[]$
⋮			
12	`load` *cal*	$[a_{19}]$	$[\bot]$
13	`getf` *contents*	$[a_{53}]$	$[\,a_3\,]$
14	`load` *srv*	$[a_{17},\,a_{53}]$	$[\bot,\,a_3\,]$
15	`call` *write*	$[]$	$[]$

Bytecode language and type system

if $\boxed{cal.f_\delta} \sqsubseteq \boxed{srv.f_\delta}$ **then** $srv.write(cal.contents);$

type: $cal.f_\delta$

#	**instruction**	**abstract operand stack**	**stack types**	**flow information**
1	`load` *cal*	$[a_{19}]$	$[\bot]$	$\emptyset$
2	`getf` f_δ	$[a_3]$	$[\bot]$	$\emptyset$
3	`load` *srv*	$[a_{17}, a_3]$	$[\bot, \bot]$	$\emptyset$
4	`getf` f_δ	$[a_4, a_3]$	$[\bot, \bot]$	$\emptyset$
5	`prim` $\sqsubseteq$	$[a_4 \sqsubseteq a_3]$	$[\bot]$	$\emptyset$
6	`bnz` 12	$[]$	$[]$	$\{a_4 \sqsubseteq a_3\}$
$\vdots$				
12	`load` *cal*	$[a_{19}]$	$[\bot]$	$\{a_4 \sqsubseteq a_3\}$
13	`getf` *contents*	$[a_{53}]$	$[a_3]$	$\{a_4 \sqsubseteq a_3\}$
14	`load` *srv*	$[a_{17}, a_{53}]$	$[\bot, a_3]$	$\{a_4 \sqsubseteq a_3\}$
15	`call` *write*	$[]$	$[]$	$\{a_4 \sqsubseteq a_3\}$

Bytecode language and type system

if cal.f_δ $\sqsubseteq$ srv.f_δ **then** $srv.write(cal.contents)$;

$Q = \{cal.f_\delta \sqsubseteq srv.f_\delta\}$

type: $cal.f_\delta$

#	instruction	abstract operand stack	stack types	flow information
1	`load` *cal*	$[\ a_{19}\ ]$	$[\bot]$	$\emptyset$
2	`getf` f_δ	$[\ a_3\ ]$	$[\bot]$	$\emptyset$
3	`load` *srv*	$[\ a_{17}\ ,\ a_3\ ]$	$[\bot, \bot]$	$\emptyset$
4	`getf` f_δ	$[\ a_4\ ,\ a_3\ ]$	$[\bot, \bot]$	$\emptyset$
5	`prim` $\sqsubseteq$	$[a_4 \sqsubseteq a_3]$	$[\bot]$	$\emptyset$
6	`bnz` 12	$[]$	$[]$	$\{a_4 \sqsubseteq a_3\}$
⋮				
12	`load` *cal*	$[a_{19}]$	$[\bot]$	$\{a_4 \sqsubseteq a_3\}$
13	`getf` *contents*	$[a_{53}]$	$[\ a_3\ ]$	$\{a_4 \sqsubseteq a_3\}$
14	`load` *srv*	$[a_{17},\ a_{53}]$	$[\bot,\ a_3\ ]$	$\{a_4 \sqsubseteq a_3\}$
15	`call` *write*	$[]$	$[]$	$\{a_4 \sqsubseteq a_3\}$

Certifying compiler (work in progress)

high-level program with policy checks

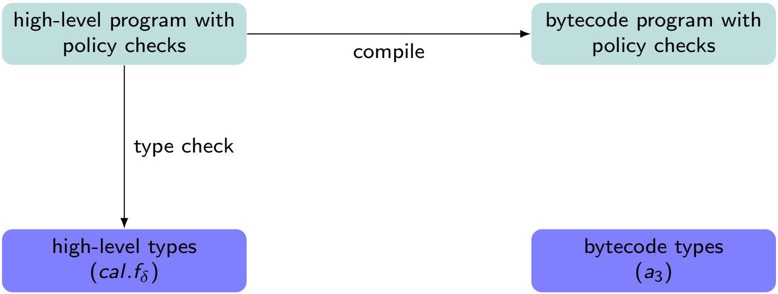
bytecode program with policy checks

high-level types ($cal.f_\delta$)

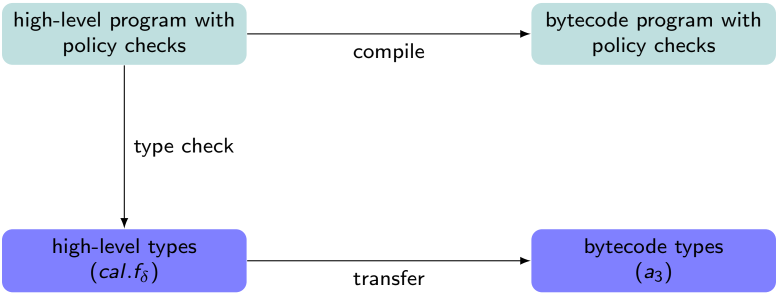
bytecode types (a_3)

Certifying compiler (work in progress)

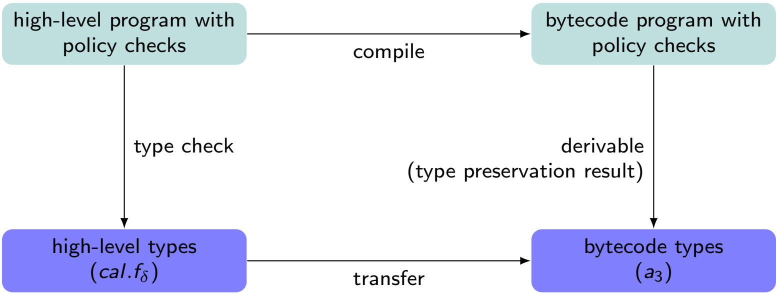
Certifying compiler (work in progress)



Certifying compiler (work in progress)

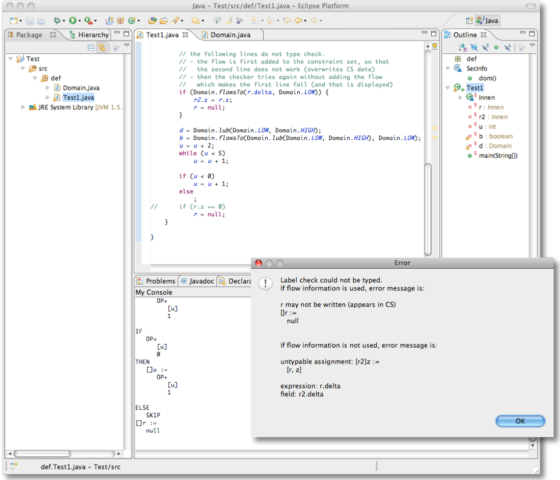


Certifying compiler (work in progress)



Implementation

- high-level language encoded in subset of Java
- type checker as Eclipse plug-in



Summary

Results

- information flow analysis with dynamic security domains and policies
- languages to inspect environment at runtime
- type systems to check proper guarding of flow-inducing actions

Current work

- type preservation result
- implementation of certifying compilation

Future work

- larger application scenario
- polymorphic information flow security

Backup slide: More features of analysis

Indirect information flows: **if** $x_{private} > 0$ **then** $y_{public} := 1$

- maintain *pc* label on high-level, confluence point stack in bytecode

Domain update: $cal.f_\delta := srv.f_\delta$

- ensure f_δ is updated with stricter confidentiality level to avoid leaks
- future work: declassification by downgrading f_δ

Meta-label monotonicity

- if domain expression e is used as a type, e is always at least as confidential as type of e itself

  "the fact that something is public cannot be private"