Language-based Security

FOSAD 2008

Steve Zdancewic

University of Pennsylvania

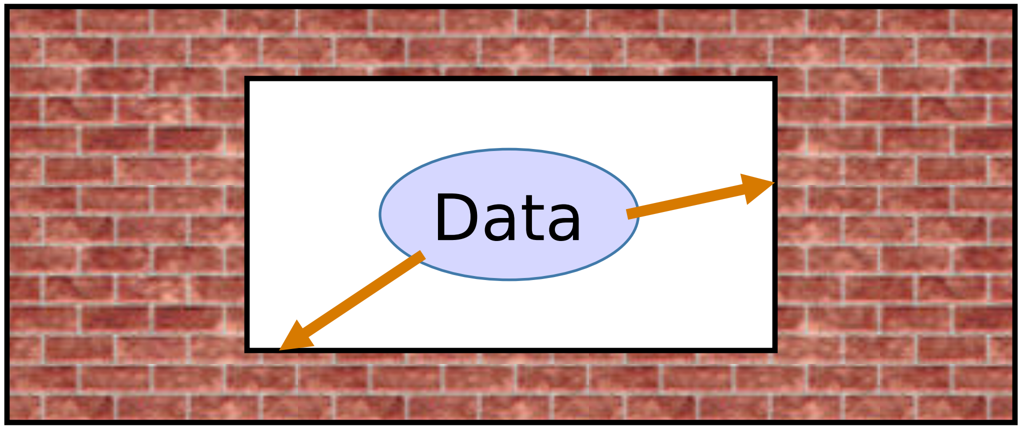
Confidential Data

- Networked information systems:
  - PCs store passwords, e-mail, finances,...
  - Businesses rely on computing infrastructure
  - Military & government communications
- Security of data and infrastructure is critical [Trust in Cyberspace, Schneider et al. '99]
- How to protect confidential data?

Technical Challenges

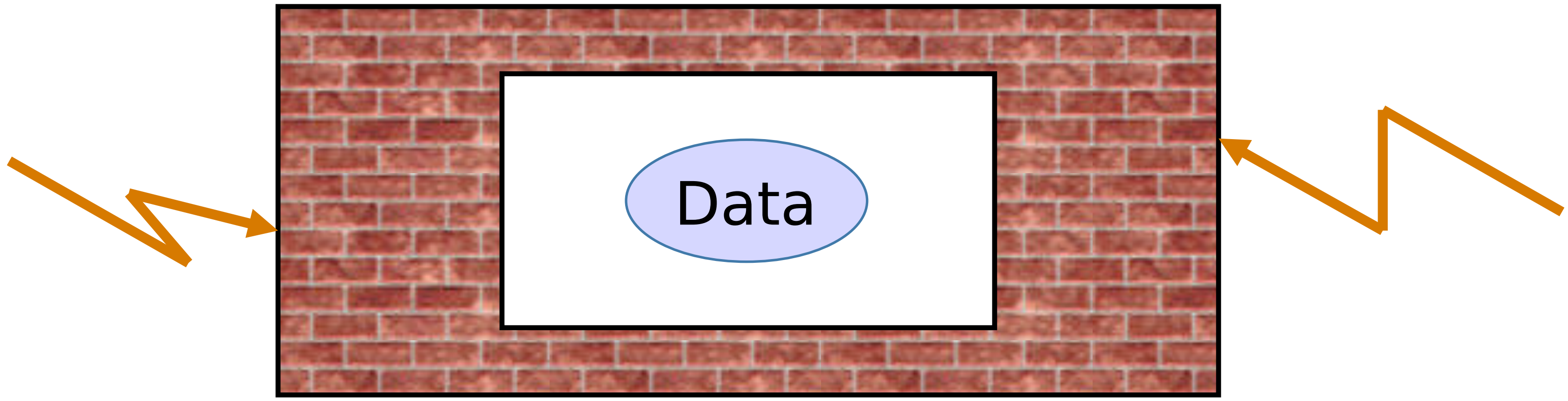
- Software is large and complex
  - Famous bugs: e.g. MS HotMail
  - Buffer overflows
- Security policies become complex
  - Mutually distrusting parties
- Requires tools & automation
- Look at traditional security concerns to set the context…
  - Confidentiality
  - Integrity
  - Availability

Quality 1: *Confidentiality*



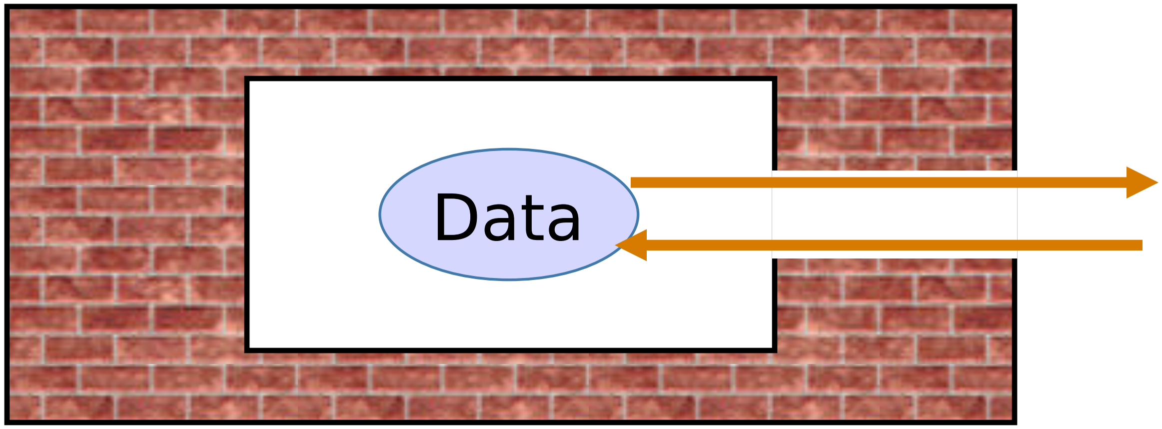
- Keep data or actions *secret.*
- Related to: Privacy, Anonymity, Secrecy
- Examples:
  - Pepsi secret formula
  - Medical information
  - Personal records (e.g. credit card information)
  - Military secrets

Quality 2: *Integrity*



- Protect the *reliability* of data against unauthorized tampering
- Related to: Corruption, Forgery, Consistency
- Example:
  - Bank statement agrees with ATM transactions
  - The mail you send is what arrives

Quality 3: *Availability*



- Resources usable in timely fashion by authorized principals
- Related to: Reliability, Fault Tolerance, Denial of Service
- Example:
  - You want the web-server to reply to your requests
  - The military communication devices must work

Information-flow Policy

- Downloadable financial planner:

- Access control insufficient
- Encryption necessary,but not end-to-end

Access Control

- Access control
  - e.g. File permissions
  - Access control lists or capabilities
  - Modern variants: Stack inspection

- Drawback:
  - Does not regulate propagation of information after permission has been granted.

Cryptography

- Essential for:
  - Protecting confidentiality & integrity of data transmitted via untrusted media
  - Authentication protocols

- Drawbacks:
  - Impractical to compute with encrypted data!
    - There are secret sharing techniques.
  - Doesn’t prevent information propagation once decrypted

Requirements

- Need a way to distinguish *confidential* information from *public* information.
  - Some simple policy language
- Need a way to *track the effects* of computation with respect to secrets
  - When is a secret leaked?
- Need a way to securely and efficiently enforce the policy.

One idea: Dynamic Tags

- Add a “tag” to each piece of data
  - Tags: hi (secret) or low (public)

- Modify every operation of the program to propagate tags
  - e.g.: (1:hi) + (2:low) → (3:hi)

- Assign “policy” to communication channels
  - e.g.: all data sent over network must have low tag
  - Check at run time whether policy is met

Example of Dynamic Tagging

```
int a = input_int(hi);
int b = input_int(low);
int c = (a + b) / 2;
output_int(low, c);
```

Variable c will have tag hi, and the output check will fail.   Great!

Problem with Dynamic Tags

```
int a = input_int(hi);
int c = 1;
if (a > 17) {
  c = 0;
}
output_int(low, c);
```

What is variable c’s tag at the output?

It gets worse

```
int a = input_int(hi);
int c = 1;
if (a > 17) {
  c = f();      // function f may affect state
}
output_int(low, c);
```

What if function f itself does output?

Sound Dynamic Enforcement

- To soundly enforce information-flow with dynamic tags:
  - Must track *all* memory locations that could have been affected in either branch of a conditional expression.
  - Update the tags of those memory locations on every branch.

- Extremely expensive
  - Worse: “efficient” implementations are conservative: tag propagation makes too many locations hi.

Static Analysis

- Uses static analysis (e.g. type systems) rather than dynamic enforcement
- Benefits:
  - No run-time cost
  - Have access to the program's control-flow graph, so they can approximate *all* runs of the program
  - Determine whether the program is secure *before* running it.
- Drawbacks:
  - No run-time information means approximation (we'll see)

End-to-end Solution

- Rely on access control & encryption
  - Essential (authentication, untrusted networks, etc.)
- But… also use language techniques:
  - verify programs to validate information flows that they contain.

Benefits (of PL-based mechanisms)

- Explicit, fine-grained policies
  - Level of single variable if necessary
  - Bytecode or assembly level
- Program abstractions
  - Programmers can design custom policies
- Regulate end-to-end behavior
  - Information Flow vs. Access Control
- Tools: increase confidence in security

Outline

- Defining information flow formally
- A simple language for information-flow security
  - One proof of *noninterference*
- Scaling up the language: features
- Language-based security in practice

- If there's time and interest:
  - Authorization and access control
  - Stack inspection
  - Secure program partitioning

Lattice Model of Policies

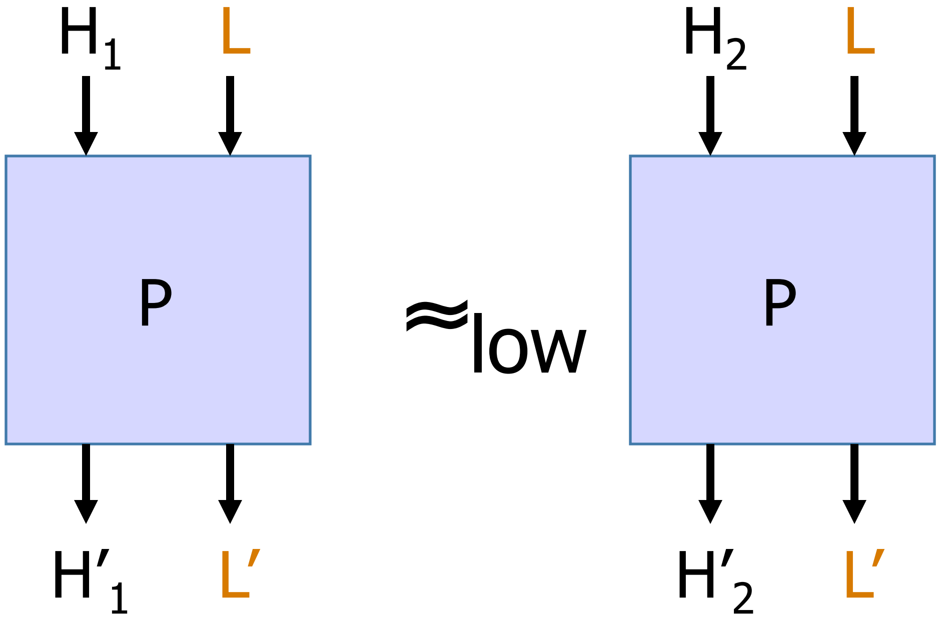
- Proposed by Denning '76
- Use a lattice $\mathcal{L}$ of *security labels*
  - Higher in lattice is more "confidential" or "secret"
  - Use $\sqsubseteq$ for order relation
  - Use $\sqcup$ for join (l.u.b.)
  - Use $\sqcap$ for meet (g.l.b.)
- Prototypical example: low $\sqsubseteq$ hi

Noninterference

[Reynolds '78, Goguen&Meseguer '82,'84]

- Private data does not *interfere* with network communication
- Baseline confidentiality policy

Noninterference



- Proved by:
  - Logical relations
  - Simulation techniques
  - Self composition techniques

Formalizing Noninterference

- Original formulation: Trace-based models of computation
  - Goguen & Meseguer 1982
  - McClean – late 1980’s early 1990’s
- Dorothy Denning proposed program analysis techniques
  - Mid-late 1970’s (but no proofs of correctness)
- Experiments with Multics
- Volpano & Smith 1996
  - Type system for noninterference
- See Sabelfeld & Myers 2003 for survey.

External Observation

- *External behavior*
  - Observations seen by someone “outside” the system
  - Outputs (i.e. strings printed to terminal)
  - Running time
  - Power or memory consumption
  - Comments
  - Variable names
- Very hard to regulate!
  - There is always some attack below the level of abstraction you choose.
  - But… attacks against external behavior tend to be difficult to carry out and/or have low bandwidth

Internal Observation

- *Internal behavior*
  - At the programming language level of abstraction
  - Note that many “external observations” can be internalized by enriching the language (e.g. add a clock)
- Observational equivalence (roughly):
  - $e_1 \approx e_2$ iff for all C[]

$$C[e_1] \rightarrow^* v \quad \text{iff} \quad C[e_2] \rightarrow^* v$$

Observations

- Final output of the program.
  - Pure, functional language
- Aliasing of pointers
  - Lambda calculus with state
- Thread scheduling decisions
  - Multithreaded languages with state/ message passing
- Timing behavior

Low Equivalence

- Captures what a "low-security" observer can "see"
- Example: Suppose program states consist of pairs: (hi , low)

$$(\text{"attack at dawn"}, 3) \approx_{low} (\text{"stay put"}, 3)$$

$$(\text{"attack at dawn"}, 3) \not\approx_{low} (\text{"stay put"}, 4)$$

Attack models

- Low equivalence captures the powers of the attacker.
  - e.g. If the attacker can see all intermediate states of the computation, then the observational model must distinguish programs that generate different traces.
- It's convenient to take the attacker to be a program context
  - the attacker operates at the same level of abstraction as the program.
  - Any abstraction violation may lead to security holes…

Outline

- Defining information flow formally
- A simple language for information-flow security
  - One proof of *noninterference*
- Scaling up the language: features
- Language-based security in practice

- If there's time and interest:
  - Authorization and access control
  - Stack inspection
  - Secure program partitioning

Lambda Calculus

λ-calculus with booleans

v ::= x | true | false	values
| λx:s.e	
e ::= v	values
| (e e)	application
| if e then e else e	conditional

Operational Semantics (1)

Note: Capture-avoiding substitution

$$(\lambda x{:}s.e)\ v \quad \rightarrow \quad e\{v/x\}$$

$$\text{if true then } e_1 \text{ else } e_2 \quad \rightarrow \quad e_1$$

$$\text{if false then } e_1 \text{ else } e_2 \quad \rightarrow \quad e_2$$

Operational Semantics (2)

$$\frac{e_1 \rightarrow e_1'}{(e_1\ e_2) \rightarrow (e_1'\ e_2)}$$

$$\frac{e_2 \rightarrow e_2'}{(v\ e_2) \rightarrow (v\ e_2')}$$

$$\frac{e \rightarrow e'}{\text{if } e \text{ then } e_1 \text{ else } e_2 \rightarrow \text{if } e' \text{ then } e_1 \text{ else } e_2}$$

Write $\rightarrow^*$ for the reflexive, transitive closure.

Modeling I/O

- λ-calculus does not have input/output
  - Only observable behavior is the output of the program.
  - Inputs to the program are its free variables.

- A *substitution* γ maps variables to values
- Given e, write γ(e) for the term obtained by substituting γ(x) for free occurences of x in e, for each x in the dom(γ).

How can information leak?

- Substitution $\gamma 1$(x) = true $\gamma 2$(x) = false
- Explicit flow (trivial):
  - Program e = x
  - So: $\gamma 1$(e) = $\gamma 1$(x) = true
  - And: $\gamma 2$(e) = $\gamma 2$(x) = false
- Implicit flow (slightly less trivial):
  - Program e = if x then false else true
  - So: $\gamma 1$(e) = if true then false else true $\rightarrow$ false
  - And: $\gamma 2$(e) = if false then false else true $\rightarrow$ true

Static Semantics

- Static semantics
  - Lattice lifted to a subtyping relation
  - “Standard” information-flow type system
  - Heintze & Riecke’s SLam calculus POPL’98
  - Pottier & Conchon ICFP’0
- Many variants
  - E.g. DCC

Types for Information Flow

- Basic idea: assign types that include security labels.
- Use the type system to track the flow of information.
- Prove that the type system is sound with respect to the model of I/O we just saw.

Simply-typed secure language

λ_{sec}

$L \in \mathcal{L}$	labels
t ::= bool | s → s	types
s ::= t{L}	secure types
v ::= x | true | false	values
| λx:s.e	
e ::= v	values
| (e e)	application
| if e then e else e	conditional

Type System (1)

$\Gamma ::= \; . \; \mid \; \Gamma, x{:}s$ Type environments

$\Gamma \vdash e : s$ Type judgments: “e has security type s”

$$\frac{x{:}s \in \Gamma}{\Gamma \vdash x : s}$$

$$\frac{}{\Gamma \vdash \mathrm{true} : \mathrm{bool}\{L\}} \qquad \frac{}{\Gamma \vdash \mathrm{false} : \mathrm{bool}\{L\}}$$

Type System (2)

$$\frac{\Gamma, x{:}s_1 \vdash e : s_2}{\Gamma \vdash \lambda x{:}s_1.\, e \;:\; (s_1 \rightarrow s_2)\{L\}}$$

$$\frac{\Gamma \vdash e_1 : (s_2 \rightarrow s)\{L\} \qquad \Gamma \vdash e_2 : s_2}{\Gamma \vdash (e_1\, e_2) : s \sqcup L}$$

Note: $t\{L_1\} \sqcup L_2 = t\{L_1 \sqcup L_2\}$

Type System (3)

$$\frac{\Gamma \vdash e : \text{bool}\{L\} \qquad \Gamma \vdash e_1, e_2 : t\{L\}}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t\{L\}}$$

$$\frac{\Gamma \vdash e : s_1 \qquad s_1 \leq s_2}{\Gamma \vdash e : s_2}$$

Subtyping Relations

$$\frac{}{t \leq t} \qquad \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}$$

$$\frac{s_1' \leq s_1 \quad s_2 \leq s_2'}{s_1 \to s_2 \;\leq\; s_1' \to s_2'}$$

$$\frac{t_1 \leq t_2 \quad L_1 \sqsubseteq L_2}{t_1\{L_1\} \;\leq\; t_2\{L_2\}}$$

Type safety properties

- Preservation:
  If $\Gamma \vdash e : s$ and $e \rightarrow e'$ then $\Gamma \vdash e' : s$.

- Progress:
  If $. \vdash e : s$ then either:
  - e is a value, or
  - There exists e' such that $e \rightarrow e'$

Basic Lemmas

- Substitution:
  If $\Gamma_1, x{:}s_1, \Gamma_2 \vdash e_2 : s_2$ and $\Gamma_1 \vdash e_1 : s_1$
  then $\Gamma_1, \Gamma_2 \vdash e_2\{e_1/x\} : s_2$.

- Canonical forms:
  - If $. \vdash v : bool\{L\}$ then $v = true$ or $v = false$
  - If $. \vdash v : (s_1 \to s_2)\{L\}$ then
    $v = \lambda x{:}s_3.\ e$ where $s_1 \leq s_3$

Noninterference Theorem

If $x{:}t\{hi\} \vdash e : bool\{low\}$

$\vdash v_1, v_2 : t\ \{hi\}$

$hi \not\sqsubseteq low$

then

$e\{v_1/x\} \longrightarrow^* v$

iff

$e\{v_2/x\} \longrightarrow^* v$

Proof

- Uses a logical relations argument
  - Relations defined inductively over the structure of types
- Two terms are related at a security level L if they “look the same” to observer at level L

- Define logical relations
- Subtyping lemma
- Substitution lemma

Logical Relations (1)

- Recall the structure of types:

$$t ::= \text{bool} \mid s \rightarrow s \qquad \text{types}$$
$$s ::= t\{L\} \qquad \text{secure types}$$

- Note: assume all terms mentioned are well typed
- Define 3 relations on this structure:
- $v_1 \sim_L v_2 : \text{bool}$ iff $v1 = v2 = \text{true}$ or $v1 = v2 = \text{false}$
- $v_1 \sim_L v_2 : s_1 \rightarrow s_2$ iff forall $u_1 \sim_L u_2 : s_1$, $(v_1\ u_1) \approx_L (v_2\ u_2) : s_2$

Logical Relations (2)

- $v_1 \sim_L v_2 : t\{L'\}$ iff
  - $L' \sqsubseteq L$ implies $v_1 \sim_L v_2 : t$

- $e_1 \approx_L e_2 : s$ iff
  - $e_1 \longrightarrow^* v_1$
  - $e_2 \longrightarrow^* v_2$
  - $v_1 \sim_L v_2 : s$

Examples

- $\text{true} \sim_{low} \text{true} : \text{bool}\{\text{low}\}$
- $\text{true} \not\sim_{low} \text{false} : \text{bool}\{\text{low}\}$
- $\text{true} \sim_{low} \text{false} : \text{bool}\{\text{hi}\}$

- $\lambda x{:}\text{bool}\{\text{low}\}.\ x \sim_{low} \lambda x{:}\text{bool}\{\text{low}\}.\ \text{not}(x)$

Are low-related at the types

$: (\text{bool}\{\text{low}\} \to \text{bool}\{\text{hi}\})\{\text{low}\}$

$: (\text{bool}\{\text{low}\} \to \text{bool}\{\text{low}\})\{\text{hi}\}$

But not at the type

$: (\text{bool}\{\text{low}\} \to \text{bool}\{\text{low}\})\{\text{low}\}$

Subtyping Lemma

- If $v_1 \sim_L v_2 : t$ and $t \leq t'$ then $v_1 \sim_L v_2 : t'$.
- If $v_1 \sim_L v_2 : s$ and $s \leq s'$ then $v_1 \sim_L v_2 : s'$.
- If $e_1 \approx_L e_2 : s$ and $s \leq s'$ then $e_1 \approx_L e_2 : s'$.

- Proof: By mutual induction on structure of types t and s, with an auxiliary induction to handle transitivity.

Related Substitutions

- Need to extend the logical relation to programs with free variables.

- Write $\gamma_1 \sim_L \gamma_2 \ : \Gamma$ to mean:
  - $\text{dom}(\gamma_1) = \text{dom}(\gamma_2) = \text{dom}(\Gamma)$
  - For all $x \in \text{dom}(\Gamma)$, $\gamma_1(x) \sim_L \gamma_2(x) \ : \Gamma(x)$

Fundamental Lemma

- If $\Gamma \vdash e : s$ and $\gamma_1 \sim_L \gamma_2 : \Gamma$ then $\gamma_1(e) \approx_L \gamma_2(e) : s$.

- Proof: By induction on the derivation that $\Gamma \vdash e : s$.

Back to Noninterference

If $x{:}t\{hi\} \vdash e : bool\{low\}$

$\vdash v_1, v_2 : t\ \{hi\}$

$hi \not\sqsubseteq low$

then

$e\{v_1/x\} \rightarrow^* v$

iff

$e\{v_2/x\} \rightarrow^* v$

Back to Noninterference

If $x{:}t\{hi\} \vdash e : bool\{low\}$

$\vdash v_1, v_2 : t\ \{hi\}$

$hi \not\sqsubseteq low$

then let $\gamma_1(x) = v_1, \gamma_2(x) = v_2$

and observe that $\gamma_1 \sim_{low} \gamma_2 : x{:}t\{hi\}$

So, $\gamma_1(e) \approx_{low} \gamma_2(e) : bool\{low\}$

Other Proof Techniques

- Information-flow is a property of *two* runs of the program.
  - It talks about correlating two different possible runs

- Proof techniques relate two runs:
  - Nonstandard operational semantics [Pottier & Simonet]
  - Bisimulation techniques
  - Self composition – reduce the problem to a property on a single execution, but run the program twice.

Outline

- Defining information flow formally
- A simple language for information-flow security
  - One proof of *noninterference*
- Scaling up the language: features
- Language-based security in practice
- Secure program partitioning

Scaling Up

- Polymorphism & Inference
- Sums
- State and effects
  - Simple state
  - References
- Termination & Timing

Polymorphism & Inference

- Add quantification over security levels
  - $\forall L::\text{label}.\ (\text{bool}\{L\} \rightarrow \text{bool}\{L\})\{L\}$
  - Reuse code at multiple security levels.

- Inference of security labels
  - Type system generates a set of lattice inequalities
  - Equations have the form $l \sqsubseteq l_1 \sqcup \ldots \sqcup l_2$
  - Constraint of this form can be solved efficiently

Polymorphism in Flow Caml

- Lists in Flow Caml
  [Vincent Simonet & François Pottier ’02,’03]
- Base types parameterized by security level bool{low} is written low bool
- Type of lists also parameterized:
  ∀’a::type. ∀’L::label. (‘a, ‘L) list

```
x1 : hi int
[1;2;3;4] : ('L int, 'M) list
[x1; x1]  : (hi int, 'L) list
```

Example: List Length

- Length does not depend on contents of list:

```
let rec length l =
match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
:
 (‘a, 'M) list -> 'M int
```

Example: has0

- Lookup depends on both contents and structure of the list:

```
let rec has0 l =
match l with
  | [] -> false
  | hd :: tl -> hd = 0 || has0 tl
:
  ('L int, 'L) list -> 'L bool
```

Sums & Datatypes

- In general: destructors reveal information
- Accuracy of information-flow analysis is important [Vincent Simonet CSFW'02]
- Suppose x:bool{L_1}, y:bool{L_2}, z:bool{L_3}

```
type t = A | B | C
let v = if x then (if y then A else B)
               else (if z then A else C)
let i = match v with
  | A | B -> 1
  | C     -> 0
```

- What is label of i?

Simple State & Implicit Flows

PC Label

```
int{high} a;
int{low} b;
...
```

{low} →

```
if (a>0) {
```

{low}⊔{high}={high} →

```
   b := 4;
}
```

{low} →

Simple State & Implicit Flows

PC Label

```
int{high} a;
int{low} b;
...

if (a>0) {
    b := 4;
}
```

{low} → (PC label before `if (a>0) {`)

{low}⊔{high}={high} → (PC label at `b := 4;`)

To assign to variable with label L, must have $PC \sqsubseteq L$.

Full References: Aliasing

```
h:int{high}

let lr = ref 3 in
let hr = lr in
  hr := h
```

Information leaks through aliasing:
Both the pointer *and* data pointed to can cause leaks.

Two more leaks

```
h:int{high}

let lr1 = ref 3 in
let lr2 = ref 4 in
let lr = if h then lr1 else lr2 in
  l := !lr

let lr1 = ref 3 in
let lr2 = ref 4 in
let lr = if h then lr1 else lr2 in
  lr := 2
```

Secure References

t ::= … | s ref	types
s ::= t{L}	secure types
v ::= … | r	heap pointers
e ::= …	
| ref e	reference alloc.
| !e	dereference
| e := e	assignment

Type System for State

- Modified type system for *effects*

[Jouvelot & Gifford '91]

- pc label approximates control-flow info.

$$\Gamma\ [pc] \vdash e : s$$

- Notation: lblof(t{L}) = L
- Invariant of this type system:

$$\Gamma\ [pc] \vdash e : s \quad \Rightarrow \quad pc \sqsubseteq \text{lblof}(s)$$

Typing Rules for State (1)

$$\Gamma\ [pc] \vdash \text{true} : \text{bool}\{pc\}$$

$$\frac{\Gamma\ [pc] \vdash e : \text{bool}\{L\} \qquad \Gamma\ [pc \sqcup L] \vdash e_1, e_2 : s}{\Gamma\ [pc] \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : s}$$

Typing Rules for State (2)

- Prevent information leaks through assignment.
- Recall that pc ⊑ L

$$\frac{\Gamma\ [pc] \vdash e_1 : s\ \mathrm{ref}\{L\} \qquad \Gamma\ [pc] \vdash e_2 : s \qquad L \sqsubseteq \mathrm{lblof}(s)}{\Gamma\ [pc] \vdash e_1 := e_2\ :\ \mathrm{unit}\{pc\}}$$

Typing Rules for State (3)

$$\frac{\Gamma\ [pc] \vdash e : s\ \text{ref}\{L\}}{\Gamma\ [pc] \vdash\ !e\ :\ s \sqcup L}$$

$$\frac{\Gamma\ [pc] \vdash e : s}{\Gamma\ [pc] \vdash \text{ref}\ e\ :\ s\ \text{ref}\{pc\}}$$

Function Calls

PC Label

```
int{high} a;
int{low} b;
...

if (a>0) {
    f(4);
}
```

{low} → (before `if`)

{low}⊔{high}={high} → `f(4);`

{low} → (after `}`)

Function Calls

PC Label

```
int{high} a;
int{low} b;
...
if (a>0) {
    f(4);
}
```

{low} → (before `if (a>0) {`)

{low}⊔{high}={low} → `f(4);`

To call a function with effects bounded by L must have PC ⊑ L.

Effect Types for Functions

`t ::= …   |   [pc]s → s     types`

$$\frac{\Gamma, x{:}s_1\ [pc'] \vdash e : s_2}{\Gamma\ [pc] \vdash \lambda x{:}s_1.e :\ ([pc']s_1 \to\ s_2)\{pc\}}$$

Typing Application

$$\frac{\Gamma\,[pc] \vdash e_2 : s_1 \qquad \Gamma\,[pc] \vdash e_1 : ([pc']s_1 \rightarrow s_2)\{L\} \qquad L \sqsubseteq pc'}{\Gamma\,[pc] \vdash e_1\ e_2 : s_2 \sqcup L}$$

More Effects

- Exceptions
  - Very important to track accurately
  - Related to sums

- Termination & Timing
  - Is termination observable?
  - For practicality, we sometimes want to allow termination channels.
  - Timing behavior can be regulated by padding (but is expensive!)

[Agat'00]

Outline

- Defining information flow formally
- A simple language for information-flow security
  - One proof of *noninterference*
- Scaling up the language: features
- Language-based security in practice
- Secure program partitioning

Practicality

- Expressiveness
- Full implementations: Flow Caml & Jif

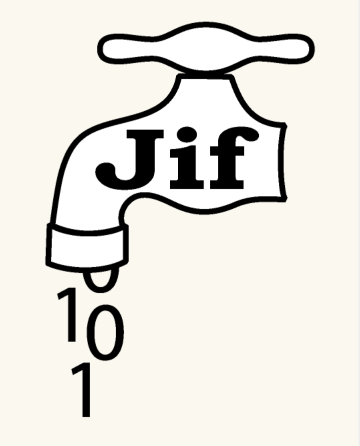
- Decentralized label model
- Downgrading & Declassification

Expressiveness

- Languages are still Turing complete
  - Just program at one level of security
- How to formalize expressiveness?
- … I don't know! (Try to write programs…)
- Agat & Sands '01:
  - Considered strong noninterference with timing constraints
  - Algorithms take worst-case running time
  - Heapsort more efficient than quicksort!
  - Relax to probabilistic noninterference to allow use of randomized algorithms

Jif: Java+Information Flow

[Myers, Nystrom, Zdancewic, Zheng]



- Java
  - With some restrictions
- Policy Language:
  - Principals, Labels, Authority
  - Principal Hierarchy (delegation)
  - Confidentiality & Integrity constraints
  - Robust Declassification & Endorsement
  - Language features (i.e. polymorphism)
- http://www.cs.cornell.edu/jif

Parameterized Classes

- Jif allows classes to be parameterized by labels and principals
  - Code reuse
  - e.g. Containers parameterized by labels

-
```
class MyClass[label L] {
    int{L} x;
}
```

Decentralized Labels

[Myers & Liskov '97, '00]

- Simple Component {owner: readers}
  - {Alice: Bob, Eve}

    "Alice owns this data and she permits Bob & Eve to read it."

- Compound Labels
  - {Alice: Charles; Bob: Charles}

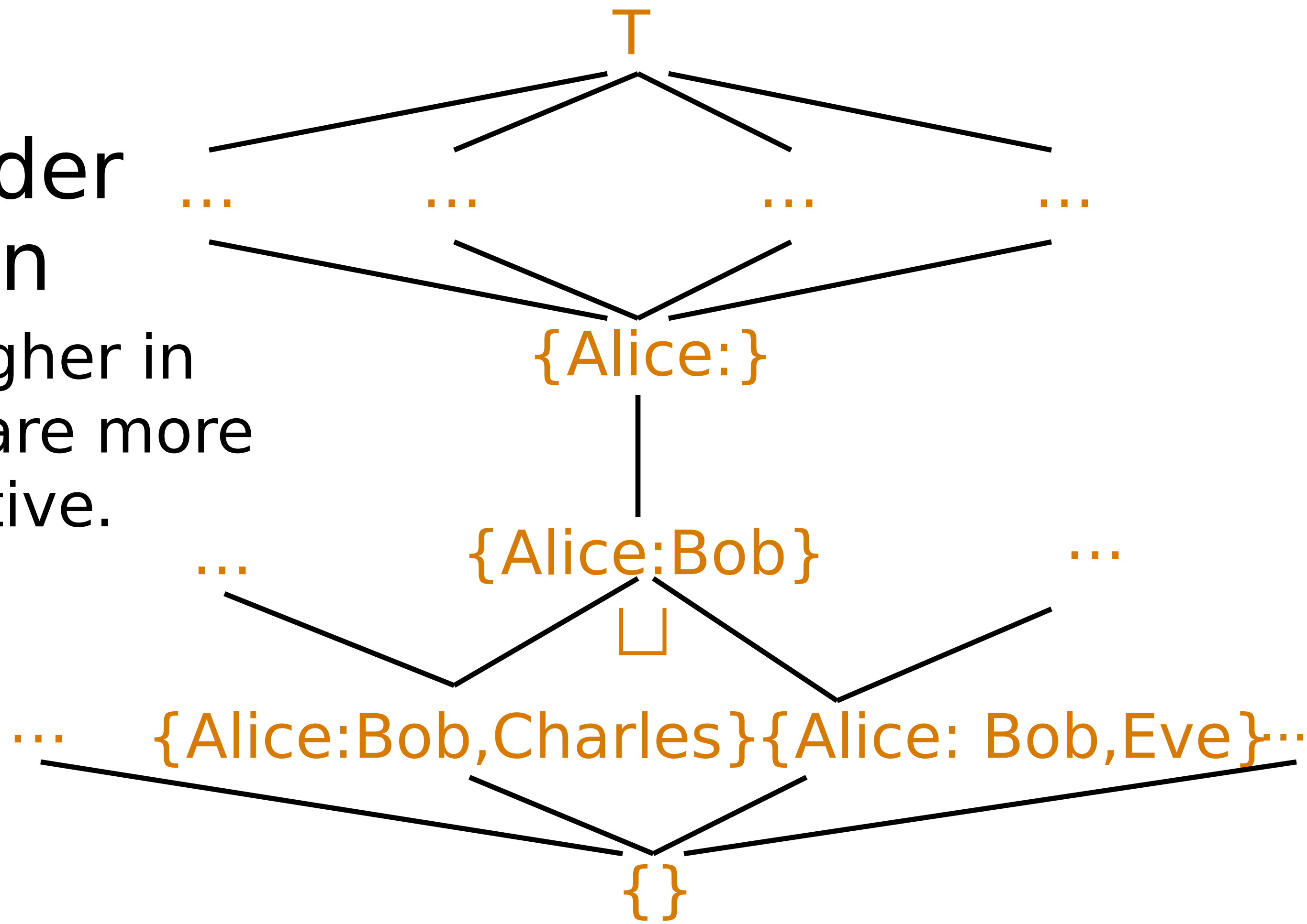
    "Alice & Bob own this data but only Charles can read it."

Decentralized Label Lattice

⊑ Order

⊔ Join

Labels higher in the lattice are more restrictive.



Integrity Constraints

- Specify who can write to a piece of data
  - {Alice? Bob}

"Alice owns this data and
she permits Bob to change it."

- Both kinds of constraints
  - {Alice: Bob; Alice?}

Integrity/Confidentiality Duality

- Confidentiality policies constrain where data can flow *to*.
- Integrity policies constrain where data can flow *from*.

- Confidentiality: Public $\sqsubseteq$ Secret
- Integrity: Untainted $\sqsubseteq$ Tainted

Weak Integrity

- Integrity, if treated dually to confidentiality is *weak*.
  - Guarantee about the source of the data
  - No guarantee about the quality of the data

- In practice, probably want stronger policies on data:
  - Data satisfies an invariant
  - Data only modified in appropriate ways by permitted principals

Richer Security Policies

- More complex policies:

"Alice will release her data to Bob, but only after he has paid $10."

- Noninterference too restrictive
  - In practice programs do leak some information
  - Rate of info. leakage too slow to matter
  - Justification lies outside the model (i.e. cryptography)

Declassification

```
int{Alice:} a;
int Paid;
...  // compute Paid
if (Paid==10) {
  int{Alice:Bob} b =
    declassify(a, {Alice:Bob});
  ...
}
```

"down-cast"
int{Alice:} to
int{Alice:Bob}

Declassification Problem

- Declassification is necessary & useful
- ...but, it breaks the noninterference theorem
  - Like a downcast mechanism
- So, must constrain its use. How?
  - Arbitrary specifications too hard to check.
    (though see recent work by Banerjee & Naumann)
  - Decentralized label model: *Authority*
  - Robust declassification
  - Subject of many, many research papers

Robust Declassification

```
int{Alice:} a;
int{Alice?} Paid;
... // compute Paid
if (Paid==10) {
  int{Alice:Bob} b =
    declassify(a, {Alice:Bob});
  ...
}
```

Alice needs to trust the contents of paid.

Introduces constraint
$PC \sqsubseteq \{Alice?\}$

[Zdancewic & Myers'01,Zdancewic'03,Myers, Sabelfeld & Zdancewic'06]

Typing Rule for Declassify

$$\frac{\Gamma\ [pc] \vdash e : t\{L'\} \qquad PC \sqsubseteq auth(L',L)}{\Gamma\ [pc] \vdash declassify(e,\{L\}) : t\{L\}}$$

auth(L',L) - returns integrity label that authorizes the downgrading

Does it Help?

- Intuitively appealing for programmers
  - But programmers are still trusted
  - Easy to implement
- Declassification doesn't change the integrity level of a piece of data
  - Noninterference for integrity sublattice still holds
  - Weaker guarantee than needed?
- Could further refine auth(L',L)
  - Restrict declassification to data with particular integrity labels

Endorsement

- The integrity dual of declassification is called *endorsement*.
  - Increases the integrity level of a value
  - Also an unsafe “downcast”
- Jif syntax: endorse(x,{Alice?})
- Decentralized Label Model:
  - Endorsing requires authority of the owner

Dynamic Policies

- Dynamic Principals
  - Identity of principals may change at run time
  - Policy may depend on identity
  - Requires authentication
  - Add a new primitive type principal
- Dynamic Labels
  - Policies for dynamic principals
  - May need to examine label dynamically
  - Add a new primitive type label

Interface to Outside World

- Should reflect OS file permissions into security types
  - Requires dynamic test of access control
- Legacy code is a problem
  - Interfaces need to be annotated with labels that soundly approximate information flow.

Unix cat in Jif

```
public static void main{}(String{}[]{} args) {
  String filename = args[0];
  final principal p = Runtime.user();
  final label lb;
  lb = new label{p:};
  Runtime[p] runtime = Runtime.getRuntime(p);
  FileInputStream{*lb} fis =
      runtime.openFileRead(filename, lb);
  InputStreamReader{*lb} reader =
      new InputStreamReader{*lb}(fis);
  BufferedReader{*lb} br = new BufferedReader{*lb}(reader);
  PrintStream{*lb} out = runtime.out();
  String line = br.readLine();
  while (line != null) {
    out.println(line);
    line = br.readLine();
  }
}
```

Jif Applications

- Many small examples
- Jif/split – distributed system extraction
  - Myers, Zdancewic, Zheng, Chong
- Jif Web Servlet – web applications in Jif
  - Myers, Chong
- Civitas – voting software
  - Myers, Clarkson
- Distributed Poker
  - Sabelfeld et al.
- JPMail
  - McDaniel, Hicks, et al.
- More in progress… There is a Jif users mailing list.

Outline

- Defining information flow formally
- A simple language for information-flow security
  - One proof of *noninterference*
- Scaling up the language: features
- Language-based security in practice
- Secure program partitioning

[Jump to other slides]

Challenges

- Integrating information flow with other kinds of security
  - Access control
  - Encryption

- Concurrency and distributed programs
  - Threads can “observer” each other’s behavior
  - Information can leak through scheduler and through synchronization mechanisms.
  - Application of bisimulation & observational equivalence
  - Application of information-flow technology to distributed systems

Other Recent work

- Concurrency
  - Sabelfeld et al; Smith; Barthe et al; …
- Declassification
  - Zdancewic & Myers; Sabelfeld, Sands; Banerjee & Nauman
- Connections to cryptography
  - Sabelfeld et al.; Vaughan & Zdancewic; Fournet & Rezk; Laud

Low-level Info.-flow Security

- Java Bytecode
  - Barthe & Rezk; Naumann; …
- Assembly language level
  - Medel, Compagnoni, Bonelli; Yu & Islam
- See Gilles Barthe's talks later this week…

Summary

- Information-flow security is a promising application domain for language technology.
- There are a lot of good results:
  - Basic theory
  - Polymorphism & Inference
  - State & Effects
  - Implementations
- but more are needed!
- There is an excellent survey paper by Sabelfeld and Myers:
  - Language-based Information-flow Security
  - JSAC 21(1) 2003
  - 147 references to other work

Thanks!