Downgrading Policies for Information Flow Control

Peng Li  Steve Zdancewic
University of Pennsylvania

TIME-DC Project
Language-based Information-flow Security

Does this program leak secret information to public channels?
Language-based Information-flow Control

- **Security Mechanism: Static Checking**
  - Type systems [Volpano et.al. POPL’96]
  - Languages: Jif [Myers et.al. POPL’99], FlowCaml [Pottier et.al. ICFP’00]

- **Security Levels: Ordered**
  - Two-point lattice: $L \sqsubseteq H$ (public $\sqsubseteq$ secret)
Language-based Information-flow Control

- **Security Mechanism: Static Checking**
  - Type systems [Volpano et.al. POPL’96]
  - Languages: Jif [Myers et.al. POPL’99], FlowCaml [Pottier et.al. ICFP’00]

- **Security Levels: Ordered**
  - Two-point lattice: $L \sqsubseteq H$ (public $\sqsubseteq$ secret)

- **Security Goal: Noninterference** [Goguen et.al. SP’82]
  - No information flow from high levels to low levels
Language-based Information-flow Control

- **Security Mechanism: Static Checking**
  - Type systems [Volpano et.al. POPL’96]
  - Languages: Jif [Myers et.al. POPL’99], FlowCaml [Pottier et.al. ICFP’00]

- **Security Levels: Ordered**
  - Two-point lattice: \( L \sqsubseteq H \) (public \( \sqsubseteq \) secret)

- **Security Goal: Noninterference** [Goguen et.al. SP’82]
  - No information flow from high levels to low levels
Noninterference

\[
\text{Program } f = f_H + f_L
\]
Noninterference

\[ f \text{ is noninterfering} \iff \forall h_1 \forall h_2 \forall x. f_L(h_1, x) = f_L(h_2, x) \]

- Extensional, end-to-end property on program input and output
- Enforcable by type systems: if \( \vdash f : \tau \) then \( f \) is noninterfering
  ...

...
Is noninterference what we really need?

A useful program in practice:

```plaintext
string {secret} password = read_password();
string {public} input = read_user_input();
string {public} message;
....
>>> if ( password == input ) then
    message := 'Login OK!';
else
    message := 'Login Failed!';
....
print_to_public(message);
```

Not noninterfering — cannot pass type-checking. Implicit information flow from `password` to `message`
Is noninterference what we really need?

A useful program in practice:

```plaintext
string {secret} password = read_password();
string {public} input = read_user_input();
string {public} message;
....
>>> if ( password == input ) then
   message := 'Login OK!';
else
   message := 'Login Failed!';
....
print_to_public(message);
```

Not noninterfering — cannot pass type-checking.

- Implicit information flow from password to message
Downgrading

In reality: allow information flow from high levels to low levels

- Often called **downgrading** or declassification (endorsement)
- It works like type down-casts in the language
Downgrading

In reality: allow information flow from high levels to low levels

- Often called **downgrading** or declassification (endorsement)
- It works like type down-casts in the language

```plaintext
string {secret} password = read_password();
string {public} input      = read_user_input();
string {public} message;

>>> if ( declassify(password==input) ) then
    message := 'Login OK!';
else
    message := 'Login Failed!';

print_to_public(message);
```

Now it passes type-checking! But wait...
Downgrading

In reality: allow information flow from high levels to low levels

- Often called **downgrading** or declassification (endorsement)
- It works like type down-casts in the language

```c
string {secret} password = read_password();
string {public} input = read_user_input();
string {public} message;

....

>>> if ( declassify(password==input) ) then
    message := 'Login OK!';
else
    message := 'Login Failed!';

....

print_to_public(message);
```

Now it passes type-checking! But wait...

Theorem (Security goal)

\[ \text{If } \vdash f : \tau, \text{ then } ? \quad [\text{Downgrading breaks noninterference}!!!] \]
A very bad (yet well-typed) program:

```c
string {secret} password = read_password();
string {public} message;
....
>>> message := declassify(password);
print_to_public(message);
```
Downgrading is dangerous

A very bad (yet well-typed) program:

```c
string {secret} password = read_password();
string {public} message;
....
>>> message := declassify(password);
print_to_public(message);
```

Practical solution: the *Decentralized Label Model*

- Control downgrading using code privileges
- Intensional mechanism
Downgrading is dangerous

- A very bad (yet well-typed) program:

  ```
  string {secret} password = read_password();
  string {public} message;
  ....
  >>> message := declassify(password);
  print_to_public(message);
  ```

- Practical solution: the *Decentralized Label Model*
  - Control downgrading using code privileges
  - Intensional mechanism

- Does `noninterference` still work?
Another view of noninterference

Recall: \( f \) is noninterfering \( \iff \forall h_1 \forall h_2 \forall x. f_L(h_1, x) = f_L(h_2, x) \)

What does this tell us?

- \( f_L \) does not care about its \( H \) input.
- So we can rewrite \( f_L \) as \( g_L(x) \triangleq f_L(0, x) \) without using its \( H \) input at all!

Idea: \( f \) is noninterfering \( \iff f_L(x_H, x_L) \equiv g_L(x_L) \)
Alternative definition of noninterference

\[ f \] is noninterfering whenever the following is true:

\[ f = f_H + f_L \]
Example

1 int {secret} x,y,z;
2 int {public} s,t,u;
3 if (x==y) then z := 3; else z := s+t;
4 s := t + u;
5 print_to_public( s );
Example

1 int {secret} x,y,z;
2 int {public} s,t,u;
3 if (x==y) then z := 3; else z := s+t;
4 s := t + u;
5 print_to_public( s );

If we only care about the public output of the program, it is equivalent to:

1 /////////////
2 int {public} s,t,u;
3 /////////////////
4 s := t + u;
5 print_to_public( s );

Idea: If $f$ is noninterfering then we can probably rewrite $f_L$ to a form that does not (syntactically) use any high-level data at all!
Our ideas

- **Relaxed Noninterference**
  - Generalize noninterference using program equivalences
  - Precisely describes **how** the data is downgraded according to the *downgrading policies*
  - Formal, extensional security guarantee

- **Downgrading Policies**
  - Let the programmer specify how secrets can be downgraded
  - A security lattice of policies and labels
Idea: Downgrading Policy

Let the programmer specify **how** the data can be downgraded in the type annotations

- What can be a **downgrading policy**? (informal)

<table>
<thead>
<tr>
<th>Policy</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>password = x</td>
<td>The password can be downgraded by comparing it with another value</td>
</tr>
<tr>
<td>hash(text)</td>
<td>The hash of the secret variable text can be released to public places</td>
</tr>
<tr>
<td>rightstr(SSN, 3)</td>
<td>The last three digits of the SSN can be released</td>
</tr>
</tbody>
</table>

- How to **formalize** them?
**Policy**: a function that specifies how the data can be downgraded

- Recall: policy for the password?
  - Downgrading through equality tests
- How to specify such a policy?
  - A $\lambda$-calculus term: $(\lambda p. \lambda x. p=x)$
- What does it mean?
  - When we **apply** this function to `password`, the result is considered as public:

    $$(\lambda p. \lambda x. p=x) \text{password} \rightarrow (\lambda x. \text{password}=x)$$

  - Any use of the result $(\lambda x. \text{password}=x)$ only leaks information about the password through equality tests.
Summary so far...

**Downgrading Policy**: a function (a \(\lambda\)-calculus term)
- Label
- Label Ordering
- Label Downgrading
**Label**: a non-empty set of downgrading policies.

- Each piece of data is annotated by a label
- A label specifies all possible ways to downgrade the data it annotates

**Example:**

- \( \text{int} \ \{ (\lambda p. \ \lambda x. \ p=x), (\lambda p. \ p\%2) \} \) password;
- Two downgrading policies for password
Label

Label: a non-empty set of downgrading policies.

- Each piece of data is annotated by a label
- A label specifies all possible ways to downgrade the data it annotates

Example:
- \text{int} \ \{(\lambda \ p. \ \lambda \ x. \ p=x), (\lambda \ p. \ p\%2)\} \text{ password};
  - Two downgrading policies for password

Each label corresponds to a security level:
- Labels are ordered; they form a lattice
- The lattice generalizes the two-point lattice $L \sqsubseteq H$
Summary so far...

*Downgrader Policy* : function (\(\lambda\)-calculus term)

*Label* : set of functions (set of policies)

Label Ordering
Label Downgrading
The Meaning of Labels

- The highest label (secret): \{\lambda x. c\}
  - \((\lambda x. c)\) data \rightarrow c
  - The policy permits no ways to release the data
- Labels in the middle: \{\lambda p\. \lambda x. p=x\}, \{\lambda x. x\%2\}, ..., etc
  - \((\lambda p\. \lambda x. p=x)\) data \rightarrow (\lambda x. data=x)
  - The policies permits some ways to release the data
- The lowest label (public): \{\lambda x. x\}
  - \((\lambda x. x)\) data \rightarrow data
  - The policy permits all ways release the data

To generalize the two-point lattice public \(\sqsubseteq\) secret, we would like to arrange the labels in this order:

\[
\{\lambda x. x\} \sqsubseteq \{\lambda p\. \lambda x. p=x\} \sqsubseteq \{\lambda x. c\}
\]

Question: how to formalize this ordering?
Label Interpretation

**Intuition:** If $\text{hash}(x)$ is public, then $\text{hash}(x) \mod 2$ is also public.

**Definition (Label interpretation)**

Let $\mathcal{S}(l)$ denote the semantic interpretation of the label $l$:

$$\mathcal{S}(l) \triangleq \{ n' | n' \equiv m \circ n, \ n \in l \}$$

- What is $\mathcal{S}({\lambda x. c})$? *all constant functions*
**Label Interpretation**

*Intuition:* If $\text{hash}(x)$ is public, then $\text{hash}(x) \% 2$ is also public.

**Definition (Label interpretation)**

Let $\mathcal{S}(l)$ denote the semantic interpretation of the label $l$:

$$\mathcal{S}(l) \triangleq \{ n' \mid n' \equiv m \circ n, \ n \in l \}$$

- What is $\mathcal{S}(\{\lambda x. \ c\})$? *all constant functions*
- What is $\mathcal{S}(\{\lambda p. \ \lambda x. \ p=x\})$? *more functions..*
Label Interpretation

Intuition: If \( \text{hash}(x) \) is public, then \( \text{hash}(x) \% 2 \) is also public.

Definition (Label interpretation)

Let \( S(l) \) denote the semantic interpretation of the label \( l \):

\[
S(l) \triangleq \{ n' \mid n' \equiv m \circ n, \ n \in l \}
\]

- What is \( S(\{\lambda x. c\}) \)? \( \text{all constant functions} \)
- What is \( S(\{\lambda p. \lambda x. p=x\}) \)? \( \text{more functions} \)
- What is \( S(\{\lambda x. x\}) \)? \( \text{all functions} \)
**Label Interpretation**

*Intuition:* If \( \text{hash}(x) \) is public, then \( \text{hash}(x) \% 2 \) is also public.

**Definition (Label interpretation)**

Let \( \mathbb{S}(l) \) denote the semantic interpretation of the label \( l \):

\[
\mathbb{S}(l) \overset{\Delta}{=} \{ n' \mid n' \equiv m \circ n, \ n \in l \}
\]

- What is \( \mathbb{S}(\{\lambda x. \ c\}) \)? *all constant functions*
- What is \( \mathbb{S}(\{\lambda p. \ \lambda x. \ p=x\}) \)? *more functions..*
- What is \( \mathbb{S}(\{\lambda x. \ x\}) \)? *all functions*

*Now we have some idea about the ordering:*

\[
\mathbb{S}(\{\lambda x. \ x\}) \supseteq \mathbb{S}(\{\lambda p. \ \lambda x. \ p=x\}) \supseteq \mathbb{S}(\{\lambda x. \ c\})
\]
**Label Interpretation**

*Intuition:* If \( \text{hash}(x) \) is public, then \( \text{hash}(x) \mod 2 \) is also public.

**Definition (Label interpretation)**

Let \( \mathcal{S}(l) \) denote the semantic interpretation of the label \( l \):

\[
\mathcal{S}(l) \triangleq \{ n' \mid n' \equiv m \circ n, \ n \in l \}
\]

- What is \( \mathcal{S}(\{\lambda x. c\}) \)? *all constant functions*
- What is \( \mathcal{S}(\{\lambda p. \lambda x. p=x\}) \)? *more functions..*
- What is \( \mathcal{S}(\{\lambda x. x\}) \)? *all functions*

*Now we have some idea about the ordering:*

\[
\mathcal{S}(\{\lambda x. x\}) \supseteq \mathcal{S}(\{\lambda p. \lambda x. p=x\}) \supseteq \mathcal{S}(\{\lambda x. c\}) \\
\{\lambda x. x\} \subseteq \{\lambda p. \lambda x. p=x\} \subseteq \{\lambda x. c\}
\]
Label Ordering

Definition (Label ordering)
Let $\sqsubseteq$ be a binary relation on labels s.t.

$l_1 \sqsubseteq l_2 \iff S(l_2) \subseteq S(l_1)$
Summary so far...

*Downgrading Policy* : function (\(\lambda\)-term)

*Label* : set of functions (set of policies)

*Label Ordering* : set inclusion of label interpretation

*Label Downgrading*
How about downgrading?

Downgrading happens as data are involved in computation:

\[ p_1 : \text{int}_{l_1}; \quad l_1 \triangleq \{ \lambda x. \lambda y. \text{hash}(x) = y \} \]

\[ p_2 : \text{int}_{l_2} = \text{hash}(p_1); \quad l_2 \triangleq \{ \lambda x. \lambda y. x = y \} \]

\[ p_3 : \text{int}_{l_3} = (p_2 = q); \quad l_3 \triangleq \{ \lambda x. x \} \quad \text{(Public)} \]

\[ p_4 : \text{int}_{l_4} = (p_1 \ast p_2); \quad l_4 \triangleq \{ \lambda x. c \} \quad \text{(Secret)} \]

Intuitively speaking:

\( l_1 \) can be downgraded to \( l_2 \) by a "hash".

\( l_2 \) can be downgraded to \( l_3 \) by an equality test \( = \).
How about downgrading?

Downgrading happens as data are involved in computation:

\[ p_1 : \text{int}_{l_1}; \quad l_1 \triangleright \{ \lambda x. \lambda y. \text{hash}(x) = y \} \]

\[ p_2 : \text{int}_{l_2} = \text{hash}(p_1); \quad l_2 \triangleright \{ \lambda x. \lambda y. x = y \} \]

\[ p_3 : \text{int}_{l_3} = (p_2 = q); \quad l_3 \triangleright \{ \lambda x. x \} \text{ (Public)} \]

\[ p_4 : \text{int}_{l_4} = (p_1 \ast p_2); \quad l_4 \triangleright \{ \lambda x. c \} \text{ (Secret)} \]

Intuitively speaking:

- \( l_1 \) can be downgraded to \( l_2 \) by a “hash”.
- \( l_2 \) can be downgraded to \( l_3 \) by an equality test “=”. 
How about downgrading?

Downgrading happens as data are involved in computation:

\[ p_1 : \text{int}_{l_1}; \quad l_1 \triangleq \{ \lambda x. \lambda y. \text{hash}(x) = y \} \]

\[ p_2 : \text{int}_{l_2} = \text{hash}(p_1); \quad l_2 \triangleq \{ \lambda x. \lambda y. x = y \} \]

\[ p_3 : \text{int}_{l_3} = (p_2 = q); \quad l_3 \triangleq \{ \lambda x. x \} \text{ (Public)} \]

\[ p_4 : \text{int}_{l_4} = (p_1 * p_2); \quad l_4 \triangleq \{ \lambda x. c \} \text{ (Secret)} \]

Intuitively speaking:

- \( l_1 \) can be downgraded to \( l_2 \) by a “hash”.
- \( l_2 \) can be downgraded to \( l_3 \) by an equality test “=”. 

Formally speaking:

- \( l_1 \overset{a_1}{\sim} l_2, \ a_1 \triangleq \lambda x. \text{hash}(x) \)
- \( l_2 \overset{a_2}{\sim} l_3, \ a_2 \triangleq \lambda x. \lambda y. x = y \)
- \( l_1 \overset{a_3}{\sim} l_4, \ a_3 \triangleq \lambda x. \lambda y. x \ast y \)
Label Downgrading

Definition (Label downgrading)

\[ l_1 \sim a \leftrightarrow l_2 \quad \forall n_2 \in \mathcal{S}(l_2), n_2 \circ a \in \mathcal{S}(l_1) \]

\[ l_1 : \{ \lambda x. \lambda y. \text{hash}(x) = y \} \]

\[ l_2 : \{ \lambda x. \lambda y. x = y \} \]

\[ l_3 : \{ \lambda x. x \} \]

\[ l_4 : \{ \lambda x. c \} \]
Definition (Label downgrading)

\[ l_1 \xrightarrow{a} l_2 \iff \forall n_2 \in S(l_2), n_2 \circ a \in S(l_1) \]

\[ l_1 : \{ \lambda x. \lambda y. \text{hash}(x) = y \} \]

\[ l_2 : \{ \lambda x. \lambda y. x = y \} \]

\[ l_3 : \{ \lambda x. x \} \]

\[ l_4 : \{ \lambda x. c \} \]
Label Downgrading

Definition (Label downgrading)

\[ l_1 \xrightarrow{a} l_2 \iff \forall n_2 \in S(l_2), n_2 \circ a \in S(l_1) \]

\[ l_1 : \{ \lambda x. \lambda y. \, \text{hash}(x) = y \} \]

\[ l_2 : \{ \lambda x. \lambda y. \, x = y \} \]

\[ l_3 : \{ \lambda x. \, x \} \]

\[ l_4 : \{ \lambda x. \, c \} \]
Definition (Label downgrading)

\[ l_1 \xrightarrow{a} l_2 \iff \forall n_2 \in S(l_2), n_2 \circ a \in S(l_1) \]

- \( l_1 : \{ \lambda x. \lambda y. \text{hash}(x) = y \} \)
- \( l_2 : \{ \lambda x. \lambda y. x = y \} \)
- \( l_3 : \{ \lambda x. x \} \)
- \( l_4 : \{ \lambda x. c \} \)

Arrows indicate the reduction relationship between the labels.
Label Downgrading

Definition (Label downgrading)

\[ l_1 \xrightarrow{a} l_2 \iff \forall n_2 \in S(l_2), n_2 \circ a \in S(l_1) \]

\[ l_1 : \{ \lambda x. \lambda y. \text{hash}(x) = y \} \]

\[ l_2 : \{ \lambda x. \lambda y. x = y \} \]

\[ l_3 : \{ \lambda x. x \} \]

\[ l_4 : \{ \lambda x. c \} \]

\[ l_1 \sqsubseteq l_2 \]
Summary of concepts

Downgrading Policy : function (λ-term)
Label : set of functions (set of policies)
Label Ordering : set inclusion of label interpretation
Label Downgrading : function composition
Language and Type System

Syntax
- Full language: simply-typed $\lambda$-calculus with fixpoints
  - Data types are annotated with labels, for example, $\text{int}\{\lambda x. x \% 2\}$
- Label language (type-level): simply-typed $\lambda$-calculus

Type system
- Similar to conventional information-flow type system with simple lattices like $L \sqsubseteq H$ [Volpano et. al., POPL 1996]
- Downgrading happens during computation
- Label operations (ordering, join, downgrading) are undecidable but can be closely approximated.

Term equivalence
- Standard $\beta - \eta$ equivalence

Details are in our POPL 2005 paper
Security Goal of the Type System

Suppose $\sigma_1, \ldots, \sigma_n$ are the inputs to the program and $P(\sigma_i)$ is the label specified for $\sigma_i$. Let $H \triangleq \{\lambda x. c\}$, $L \triangleq \{\lambda x. x\}$.

Theorem (Relaxed noninterference)

If $\vdash e : \text{int}_L$, then $e \equiv f (n_1 \sigma_1) \ldots (n_k \sigma_k)$ where $\forall i. \sigma_i \notin FV(f)$ and $\forall j. n_j \in P(\sigma_j)$.

Example:
Suppose $P(\sigma_{pwd}) = \{\lambda x. \lambda y. x = y\}$ and $\vdash e : \text{int}_L$, then

1. $e \equiv f ((\lambda x. \lambda y. x = y)\sigma_{pwd})$
2. $\forall i. \sigma_i \notin FV(f)$ ($f$ does not contain any secrets)
Suppose $\sigma_1, \ldots, \sigma_n$ are the inputs to the program and $P(\sigma_i)$ is the label specified for $\sigma_i$. Let $H \overset{\Delta}{=} \{\lambda x. \ c\} \land L \overset{\Delta}{=} \{\lambda x. \ x\}$.

**Theorem (Relaxed noninterference)**

If $\vdash e : \text{int}_L$, then $e \equiv f(\sigma_1) \cdots (\sigma_k)$ where $\forall i. \sigma_i \notin \text{FV}(f)$ and $\forall j. \sigma_j \in P(\sigma_j)$.

**Corollary (Pure noninterference)**

If $\vdash e : \text{int}_L \land \forall i. P(\sigma_i) = H$, then $e \equiv f$ where $\forall i. \sigma_i \notin \text{FV}(f)$.
Relaxed vs. Pure Noninterference

Relaxed Noninterference:

Pure Noninterference:
Related Work on Downgrading

Most related:

- **Intransitive Noninterference** [Roscoe et.al. CSFW’99]
  - Downgrading paths in the lattice
- **Delimited Information Release** [Sabelfeld et.al. ISSS’03]
  - Similar to our *relaxed noninterference*

Other work:

- **Who** can downgrade: Decentralized Label Model [Myers et.al. SOSP’97], Robust Declassification [Zdancewic et.al. CSFW’01]
- **When** to downgrade [Chong et.al. CCS’04]
- **How much** information to downgrade [Lowe CSFW’02]
- Relative Secrecy [Volpano et.al. POPL’00]
- Abstract Noninterference [Giacobazzi et.al. POPL’04]
Conclusion so far...

Noninterference and downgrading can be put together in the same framework using program equivalences and function compositions.

**Downgrading policies:**
- Functions that describe how data can be downgraded

**Relaxed noninterference:**
- Generalizes pure noninterference
- Precisely describes how data is downgraded
- Formal, extensional, end-to-end guarantee
More expressive downgrading policies [POPL'05]
Integrity downgrading policies [FAST'03] [FCS'05]
Practical application [CSFW’05]
Local language constructs
More expressive downgrading policies

How about policies like these?

- encrypt($\sigma$, k)
- $(\sigma_1 + \sigma_2)/2$
More expressive downgrading policies

How about policies like these?

- $\text{encrypt}(\sigma, k)$
- $(\sigma_1 + \sigma_2)/2$

— Policies that describe the interaction among secret inputs
More expressive downgrading policies

How about policies like these?

- $\text{encrypt}(\sigma, k)$
- $(\sigma_1 + \sigma_2)/2$

— Policies that describe the interaction among secret inputs

Questions:

- How to specify such policies?
- How does it change the security lattice?
- How does it change the security goal?

Recent paper:

- Downgrading Policies and Relaxed Noninterference [POPL’05]
Integrity policies

Belief: Confidentiality and Integrity are **duals**
- Secret — Tainted; Public — Untainted

Downgrading policies for integrity?
- What does a policy mean?
  - What happened to the value in the **past**
- How does the security lattice look like?
  - \( \lambda x. x: \text{tainted} \)
  - What corresponds to **untainted**?

Reality: confidentiality and integrity are not exactly symmetric

Recent papers:
- Information Integrity Policies [FAST’03]
- Unifying Confidentiality and Integrity in Downgrading Policies [FCS’05]
Question: where are we going to use these technologies?

A secure web scripting language (like PHP)
- Typed interface with databases (data annotated with security policies)
- Static, end-to-end security guarantee

Recent paper:
- Practical Information-flow Control in Web Scripting Languages [CSFW'05]
Better languages constructs

Problem: All secret variables are *global* and *external*
- Good: Intuitive, end-to-end security guarantee
- Bad: not flexible

Idea:
- **Local** language constructs that introduces the secret variables and downgrading policies

Challenges:
- Expressiveness
- Type soundness
- Useful, intuitive, end-to-end security guarantee