

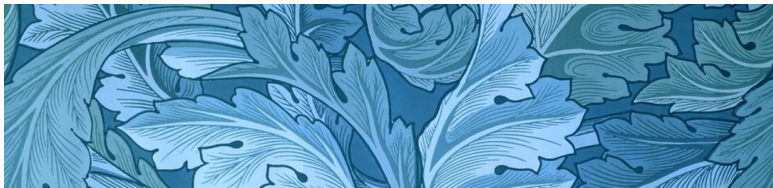


# The Holy Grail of Gradual Security

Final Examination for Doctor of Philosophy in Computer Science

Tianyu Chen

Indiana University



# Tianyu's Thesis Statement

It *is possible* to design a gradual IFC programming language that satisfies both noninterference and the gradual guarantee while supporting type-based reasoning, by excluding the unknown label ★ from runtime security labels and using security coercions to represent casts.

# Road Map

## 👉 Background

- Explicit flow and implicit flow
  - Information flow control (IFC): static, dynamic, and gradual
  - The gradual guarantee and its tension with IFC
- ▶ Source of the tension: including ★ in runtime labels
  - ▶ Comparing  $\lambda_{\text{IFC}}^*$  with  $\text{GSL}_{\text{Ref}}$
  - ▶ Timeline of dissertation writing

# Explicit Information Flow

Can we infer output from input in the following program?

```
let input = private-input () in  
  publish ( $\neg$  input)
```

# Explicit Information Flow

Can we infer output from input in the following program?

```
let input = private-input () in  
  publish ( $\neg$  input)
```

- ✓ Yes!
  - ▶ Witness at least two executions
  - ▶ Output is the negation of input
  - ▶ Explicit flow

# Implicit Information Flow

Can we infer output from input in the following program?

```
let input = private-input () in  
  publish (if input then false else true)
```

# Implicit Information Flow

Can we infer output from input in the following program?

```
let input = private-input () in  
  publish (if input then false else true)
```

- ✓ Also yes
  - ▶ Again, output is the negation of input
  - ▶ **Implicit flow**: input influences output through *branching*

# Information-Flow Control (IFC)

- ▶ Ensures that information transfers adhere to a security policy
- ▶ For example, **high** input must not flow to **low** output
- ▶ Propagate and check the security labels
- ▶ IFC in PL  $\left\{ \begin{array}{l} \text{static} \text{ using a type system} \\ \text{dynamic} \text{ using runtime monitoring} \end{array} \right.$



# Static IFC Accepts Legal Explicit Flow

(Static IFC using a type system)

```
1 let fconst = λ b : Boolhigh. false in
2 let input  = private-input () in
3 let result = fconst input in
4   publish result
```

✓ Well-typed and runs successfully to unit

- ▶ Why? The return value of fconst is  $\begin{cases} \text{always false} \\ \text{of low-security} \end{cases}$
- ▶ Accepted by type-checker. No runtime check

---

<sup>o</sup>private-input : Unit<sub>low</sub> → Bool<sub>high</sub> and publish : Bool<sub>low</sub> → Unit<sub>low</sub>

# Static IFC Rejects **Illegal** Explicit Flow

(Replace fconst with flip)

```
1 let flip    = λ b : Boollow . ¬ b in
2 let input   = private-input () in
3 let result  = flip input in  // compilation error
4   publish result
```

✗ **Ill-typed.** Illegal explicit flow:

- input is **high**
  - flip expects **low** argument
- Rejected by type-checker. Again no runtime check

# Dynamic Enforcement of Explicit Flow

(Revisit flip with dynamic IFC)

```
1 let flip      =  $\lambda b. \neg b$  in
2 let input     = private-input () in
3 let result    = flip input in
4   publish result    // runtime error
```

✗ **Errors** at runtime (regardless of input)

- ▶ **A runtime check** happens before calling publish

In dynamic IFC, runtime values are tagged with their security level.  
The labels can originate from

- ▶ primitive operations
- ▶ annotations on literals
- ▶ the security level of the execution context

# Static Enforcement of Implicit Flow

(Different behavior in different branches)

```
1 let flip : Boolhigh → Boollow =  
2   λ b : Boolhigh. if b then false else true in  
3 let input  = private-input () in  
4 let result = flip input in  
5   publish result
```

## ✗ Ill-typed

- ▶ Security label on the type of if is the join (least upper bound) of its branches (<sub>low</sub>) and the branch condition (<sub>high</sub>).
- ▶ Rejected by type-checker. No runtime check

# Dynamic Enforcement of Implicit Flow

(Enforcing implicit flow with dynamic IFC)

```
1 let flip    = λ b. if b then false else true in
2 let input  = private-input () in
3 let result = flip input in
4   publish result
```

✗ **Errors** at runtime (regardless of input)

- ▶ flip produces a **high** value because of **high** branch condition
- ▶ A runtime check happens before calling publish
- ▶ Illegal implicit flow ruled out **at runtime**

# Gradual Typing Bridges Static and Dynamic IFC

Partially-annotated flip:

```
1 let flip : Bool★ → Boollow =  
2   λ b : Bool★. if b then false else true in  
3 let input = private-input () in  
4 let result = flip input in  
5   publish result
```

- ▶ Well-typed but errors at runtime
- ▶ Checking happens on the boundaries between static and dynamic fragments
- ▶ The information flow violation is detected earlier than the dynamic version, as flip returns

# The Gradual Guarantee

less precise

```
let f : Bool★ → Bool★ =  
  λ b : Bool★. true in  
let i = private-input () in  
let result = f i in  
publish result
```

```
let f : Bool★ → Boollow =  
  λ b : Bool★. true in  
let i = private-input () in  
let result = f i in  
publish result
```

more precise

```
let f : Boolhigh → Boollow =  
  λ b : Boolhigh. true in  
let i = private-input () in  
let result = f i in  
publish result
```

- ▶ In the absense of errors, adding or removing security annotations does not change the result of the program
- ▶ Adding security annotations may trigger errors
- ▶ Removing security annotations may not trigger errors

# Static Enforcement of Flows Through Mutable References

```
1 let a      = ref low true in
2 let input = private_input () in
3 if input then
4     a := false
5 else
6     a := true
7 publish (! a)
```

- ▶ The reference has type `Ref (Boollow)`. It points to a low memory location
- ▶ The type of the branch condition is `Boolhigh`
- ✗ ~~Writing to low memory under a high branch condition~~



# Dynamic Enforcement of Flows Through Mutable References

```
1 let a      = ref low true in
2 let input = private_input () in
3 if input then
4     a := false
5 else
6     a := true
7 publish (! a)
```

The assignments fail at runtime because the no-sensitive-upgrade (NSU) mechanism<sup>1</sup> prevents writing to a **low** security pointer in a **high** security branch.

---

<sup>1</sup> Austin and Flanagan. *Efficient purely-dynamic information flow analysis*. PLAS 2009.

# Counterexample of Gradual Guarantee in $\text{GSL}_{\text{Ref}}$

less precise

```
1 let x = private-input () in
2 let a = ref ★ true★ in
3 if x then (a := falsehigh)
4       else ()
```

more precise

```
let x = private-input () in
let a = ref hightruehigh in
if x then (a := falsehigh)
       else ()
```

- ✓ The **more precise** program (right) runs successfully
- ✗ But the **less precise** version (left) errors in  $\text{GSL}_{\text{Ref}}$ <sup>2</sup>
  - ▶ The assignment fails because it is in a high-security branch and  $\text{GSL}_{\text{Ref}}$  conservatively treats the reference's label (★) as if it were **low**

---

<sup>2</sup>Toro, Garcia, Tanter. *Type-Driven Gradual Security with References*. TOPLAS 2018.

## But wait... $\text{GSL}_{\text{Ref}}$ allows $\star$ labels on values?

The counterexample depends on labeling a reference with unknown security ( $\star$ ):

```
1 let x = private-input () in
2 let a = ref  $\star$  true $\star$  in
3 if x then (a := falsehigh)
4           else ()
```

- ▶ Dynamic IFC languages don't use  $\star$  as a runtime security label.
- ▶ Gradual languages traditionally use  $\star$  for type checking, but not for categorizing runtime values.
- ▶ The inputs to an information flow system are the user's choices regarding what data is high or low security.

# Sources of the Tension with the Gradual Guarantee

Lang.	Noninterference	Gradual Guarantee	Type-guided classification	NSU	Runtime security labels
GSL <sub>Ref</sub>	✓	✗	✓	✓	{low, high, ★}
GLIO	✓	✓	✗	✓	{low, high}
WHILE <sup>G</sup>	✓	✓	✓	✗	{low, high, ★}
$\lambda_{\text{Ifc}}^*$ (ours)	✓	✓	✓	✓	{low, high}

Removing ★ from the runtime labels enables the gradual guarantee.

# $\lambda_{\text{IFC}}^*$ Excludes ★ From Runtime Labels

## less precise

```
1 let x = private-input () in
2 let a : (Ref Bool★)★ =
3   ref high truehigh in
4 if x then (a := falsehigh)
5   else ()
```

## more precise

```
let x = private-input () in
let a : (Ref Boolhigh)high =
  ref high truehigh in
if x then (a := falsehigh)
  else ()
```

- ✓ The more precise program runs successfully to unit
- ✓ The less precise program also runs successfully to unit

# $\lambda_{\text{IFC}}^*$ Excludes $\star$ From Runtime Labels

less precise

```
1 let x = private-input () in
2 let a : (Ref Bool $\star$ ) $\star$  =
3   ref high truehigh in
4 if x then (a := falsehigh)
5   else ()
```

more precise

```
let x = private-input () in
let a : (Ref Boolhigh)high =
  ref high truehigh in
if x then (a := falsehigh)
  else ()
```

- ✓ The more precise program runs successfully to unit
- ✓ The less precise program also runs successfully to unit
- ✓ Problem solved!



## Comparing $\lambda_{\text{IFC}}^*$ With $\text{GSL}_{\text{Ref}}$

- ▶ The default security label in  $\lambda_{\text{IFC}}^*$  is **low**, so the programmer does not have to label constants
- ▶ Remove the labels on constants to obtain the following program, which also reduces successfully to unit:

```
let x = private-input () in
let a : (Ref Bool★)★ = ref high true in
if x then (a := false)
  else ()
```

Comparing with the program in  $\text{GSL}_{\text{Ref}}$ , which errors:

```
let x = private-input () in
let a = ref ★ true★ in
if x then (a := falsehigh)
  else ()
```

# The Syntax of $\lambda_{\text{IFC}}^{\star}$

Highlighted security labels default to **low** if omitted:

$$\begin{aligned}\ell &\in \{\text{low}, \text{high}\} \\ g &\in \{\text{low}, \text{high}, \star\} \\ M &::= x \mid (\$k)_{\ell} \mid (\lambda^g x:A. M)_{\ell} \mid (M M)^p \\ &\quad \mid (\text{if } M \text{ then } M \text{ else } M)^p \\ &\quad \mid (\text{ref } \ell M)^p \mid !^p M \mid (M := M)^p\end{aligned}$$



# Vigilance: Type-Based Reasoning for Explicit Flows

Consider the example from Toro et al. [2018]:

```
1 let mix : Intlow → Inthigh → Intlow =  
2   λ pub priv .  
3     if pub < (priv : Int★ : Intlow) then 1 else 2 in  
4 mix 1low 5low
```

Free theorem: The mix function either ① returns a value that does not depend on priv or ② produces a runtime error

In  $\lambda_{\text{IFC}}^*$ ,  $5 \langle \uparrow; \text{high}!; \text{low}?^p \rangle \longrightarrow \text{blame } p$

# Type-Guided Classification:

## Type-Based Reasoning for Implicit Flows

Consider another example from Toro et al. [2018]:

```
1 let mix : Intlow → Int* → Intlow =  
2   λ pub priv. if pub < priv then 1 else 2 in  
3 let smix : Intlow → Inthigh → Intlow =  
4   λ pub priv. mix pub priv in  
5 smix 1low 5low
```

Free theorem: The `smix` function either ① returns a value that does not depend on `priv` or ② produces a runtime error

# Type-Based Reasoning for Implicit Flows in $\lambda_{\text{IFC}}^*$

```
let mix : Intlow → Int* → Intlow =  
  λ pub priv. if pub < priv then 1 else 2 in  
let smix : Intlow → Inthigh → Intlow =  
  λ pub priv. mix pub priv in  
smix 1low 5low
```

```
let mix = λ pub priv.  
  (if ((pub < low!) < priv)  
    then (1 < low!)  
    else (2 < low!))) < low?p > in  
⇒ let smix = λ pub priv.  
  mix pub (priv < high! >) in  
  smix 1 (5 < ↑ >)
```

→\* (if (1 < low! > < 5 < ↑; high! > >) then 1 < low! > else ...) < low?<sup>p</sup> >

→\* (if (true < ↑; high! > >) then 1 < low! > else ...) < low?<sup>p</sup> >

→\* (prot high (1 < low! >)) < low?<sup>p</sup> >

→\* 1 < ↑; high! > < low?<sup>p</sup> >

→\* blame *p*

# Conclusion

- ▶ Noninterference and the gradual guarantee can co-exist if we keep  $\star$  out of the runtime security labels.
- ▶ The security labels on constants and memory locations should default to **low** or **high** instead of  $\star$ .
- ▶ The Agda mechanization of  $\lambda_{\text{IFC}}^{\star}$  is at <https://github.com/Gradual-Typing/LambdaIFCStar>

Thank you! 😊