The Holy Grail of Gradual Security

Final Examination for Doctor of Philosophy in Computer Science

Tianyu Chen

Indiana University

° Acanthus textiles. William Morris Gallery

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 $\star$ from run-time 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 $\star$ in runtime labels
- Comparing $\lambda^{\star}_{\mathtt{IFC}}$ with $\mathrm{GSL}_{\mathsf{Ref}}$
- Timeline of dissertation writing

Explicit Information Flow

Can we infer output from input in the following program?

```
let input = private-input () in
  publish (¬ input)
```

Explicit Information Flow

Can we infer output from input in the following program?

```
let input = private-input () in
  publish (¬ 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 $\begin{cases} \text{static using a type system} \\ \text{dynamic using runtime monitoring} \end{cases}$

Static IFC Accepts Legal Explicit Flow

(Static IFC using a type system)

```
let fconst = λ b : Bool_high . false in
let input  = private-input () in
let result = fconst input in
  publish result
```

✓ Well-typed and runs successfully to `unit`

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

°`private-input` : $\texttt{Unit}_{\texttt{low}} \to \texttt{Bool}_{\texttt{high}}$ and `publish` : $\texttt{Bool}_{\texttt{low}} \to \texttt{Unit}_{\texttt{low}}$

Static IFC Rejects Illegal Explicit Flow

(Replace `fconst` with `flip`)

```
let flip   = λ b : Bool_low . ¬ b in
let input  = private-input () in
let result = flip input in   // compilation error
  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)

```
let flip    = λ b. ¬ b in
let input   = private-input () in
let result  = flip input in
  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)

```
let flip : Bool_high → Bool_low =
      λ b : Bool_high. if b then false else true in
let input  = private-input () in
let result = flip input in
  publish result
```

✗ Ill-typed

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

Dynamic Enforcement of Implicit Flow

(Enforcing implicit flow with dynamic IFC)

```
let flip   = λ b. if b then false else true in
let input  = private-input () in
let result = flip input in
  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`:

```
let flip : Bool_★ → Bool_low =
    λ b : Bool_★ . if b then false else true in
let input  = private-input () in
let result = flip input in
  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
```

$\sqsubseteq$

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

$\sqsubseteq$

more precise

```
let f : Bool_high → Bool_low =
    λ b : Bool_high . 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

```
let a     = ref low true in
let input = private_input () in
if input then
    a := false
else
    a := true
publish (! a)
```

- The reference has type Ref $(\mathtt{Bool}_{\mathtt{low}})$. It points to a low memory location
- The type of the branch condition is $\mathtt{Bool}_{\mathtt{high}}$

✗ ~~Writing to low memory under a high branch condition~~

Dynamic Enforcement of Flows Through Mutable References

```
let a     = ref low true in
let input = private_input () in
if input then
    a := false
else
    a := true
publish (! a)
```

The assignments fail at runtime because the no-sensitive-upgrade (NSU) mechanism [1] prevents writing to a `low` security pointer in a `high` security branch.

[1] Austin and Flanagan. *Efficient purely-dynamic information flow analysis*. PLAS 2009.

Counterexample of Gradual Guarantee in $\mathsf{GSL_{Ref}}$

less precise

```
let x = private-input () in
let a = ref ⋆ true_⋆ in
if x then (a := false_high)
      else ()
```

more precise

```
let x = private-input () in
let a = ref high true_high in
if x then (a := false_high)
      else ()
```

- ✓ The more precise program (right) runs successfully
- ✗ But the less precise version (left) errors in $\mathsf{GSL_{Ref}}$[2]
  - The assignment fails because it is in a high-security branch and $\mathsf{GSL_{Ref}}$ conservatively treats the reference's label (⋆) as if it were `low`

[2] Toro, Garcia, Tanter. *Type-Driven Gradual Security with References.* TOPLAS 2018.

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

The counterexample depends on labeling a reference with unknown security (⋆):

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

- Dynamic IFC languages don't use ⋆ as a runtime security label.
- Gradual languages traditionally use ⋆ 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.	Noninter-ference	Gradual Guarantee	Type-guided classification	NSU	Runtime security labels
$\mathrm{GSL}_{\mathsf{Ref}}$	✓	✗	✓	✓	{low, high, ⋆}
GLIO	✓	✓	✗	✓	{low, high}
$\mathrm{WHILE}^{\mathrm{G}}$	✓	✓	✓	✗	{low, high, ⋆}
$\lambda^{\star}_{\mathtt{IFC}}$ (ours)	✓	✓	✓	✓	{low, high}

Removing ⋆ from the runtime labels enables the gradual guarantee.

$\lambda^{\star}_{\mathtt{IFC}}$ Excludes ⋆ From Runtime Labels

less precise

```
let x = private-input () in
let a : (Ref Bool_⋆)_⋆ =
    ref high true_high in
if x then (a := false_high)
      else ()
```

more precise

```
let x = private-input () in
let a : (Ref Bool_high)_high =
    ref high true_high in
if x then (a := false_high)
      else ()
```

✓ The more precise program runs successfully to `unit`

✓ The less precise program also runs successfully to `unit`

$\lambda^{\star}_{\mathtt{IFC}}$ Excludes ★ From Runtime Labels

less precise

```
let x = private-input () in
let a : (Ref Bool_★)_★ =
    ref high true_high in
if x then (a := false_high)
      else ()
```

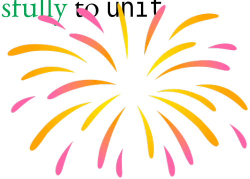
more precise

```
let x = private-input () in
let a : (Ref Bool_high)_high =
    ref high true_high in
if x then (a := false_high)
      else ()
```

✓ The more precise program runs successfully to `unit`

✓ The less precise program also runs successfully to `unit`

✓ **Problem solved!**



Comparing $\lambda^{\star}_{\mathtt{IFC}}$ With $\mathrm{GSL}_{\mathsf{Ref}}$

- The default security label in $\lambda^{\star}_{\mathtt{IFC}}$ 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 $\mathrm{GSL}_{\mathsf{Ref}}$, which errors:

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

The Syntax of $\lambda^{\star}_{\texttt{IFC}}$

We define a gradual IFC calculus $\lambda^{\star}_{\texttt{IFC}}$ with mutable references, first-class functions, and conditionals.

Highlighted security labels default to $\texttt{low}$ if omitted:

$$
\begin{array}{rcl}
\ell & \in & \{\texttt{low}, \texttt{high}\} \\
g & \in & \{\texttt{low}, \texttt{high}, \star\} \\
\iota & ::= & \texttt{Unit} \mid \texttt{Bool} \\
T & ::= & \iota \mid A \xrightarrow{g} A \mid \texttt{Ref}\ (T_g) \\
A & ::= & T_g \\
M & ::= & x \mid (\$\ k)_{\boxed{\ell}} \mid (\lambda^{g} x{:}A.\ M)_{\boxed{\ell}} \mid (M\ M)^{p} \\
 & \mid & (\texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M)^{p} \\
 & \mid & (\texttt{ref}\ \boxed{\ell}\ M)^{p} \mid\ !^{p}\ M \mid (M \mathrel{:=} M)^{p}
\end{array}
$$

Vigilance: Type-Based Reasoning for Explicit Flows

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

```
let mix : Int_low → Int_high → Int_low =
  λ pub priv .
    if pub < (priv : Int_⋆ : Int_low) then 1 else 2 in
mix 1_low 5_low
```

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

In $\lambda^{\star}_{\mathtt{IFC}}$, $\quad 5 \langle \uparrow ; \mathtt{high}\,!\,; \mathtt{low}\,?^{p} \rangle \longrightarrow \mathtt{blame}\ p$

Type-Guided Classification: Type-Based Reasoning for Implicit Flows

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

```
let mix : Int_low → Int_★ → Int_low =
  λ pub priv. if pub < priv then 1 else 2 in
let smix : Int_low → Int_high → Int_low =
  λ pub priv. mix pub priv in
smix 1_low 5_low
```

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 $\lambda^{\star}_{\mathtt{IFC}}$

```
let mix : Int_low → Int_★ → Int_low =
  λ pub priv. if pub < priv then 1 else 2 in
let smix : Int_low → Int_high → Int_low =
  λ pub priv. mix pub priv in
smix 1_low 5_low
```

⇒

```
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 ⟨ ↑ ⟩)
```

$\longrightarrow^{*}$ (if (1 ⟨ low ! ⟩ < 5 ⟨ ↑ ; high ! ⟩) then 1 ⟨ low ! ⟩ else ...) ⟨ low ?^{p} ⟩

$\longrightarrow^{*}$ (if (true ⟨ ↑ ; high ! ⟩) then 1 ⟨ low ! ⟩ else ...) ⟨ low ?^{p} ⟩

$\longrightarrow^{*}$ (prot high (1 ⟨ low ! ⟩)) ⟨ low ?^{p} ⟩

$\longrightarrow^{*}$ 1 ⟨ ↑ ; high ! ⟩ ⟨ low ?^{p} ⟩

$\longrightarrow^{*}$ blame p

The Cast Calculus $\lambda^{c}_{\mathtt{IFC}}$

The semantics of $\lambda^{\star}_{\mathtt{IFC}}$ is by translation $\mathcal{C}$ to a cast calculus $\lambda^{c}_{\mathtt{IFC}}$.

The casts are represented by coercions on types (a la Henglein) and coercions on security labels.

$$
\begin{array}{rcl}
c & ::= & \mathrm{id}(g) \mid \uparrow \mid \ell\,\mathbf{!} \mid \ell\,\mathbf{?}^{p} \mid \bot^{p} \\
\bar{c} & ::= & \mathrm{id}(g) \mid \bot^{p}\ g_1\ g_2 \mid \bar{c}\,\mathbf{;}\,c \\
e & ::= & \ell \mid \mathtt{blame}\ p \mid e\,\langle\,\bar{c}\,\rangle \\
c_r & ::= & \mathrm{id}(\iota) \mid \mathrm{Ref}\ \boldsymbol{c}\ \boldsymbol{c} \mid (\bar{c}\mathbf{,}\ \boldsymbol{c} \rightarrow \boldsymbol{c}) \\
\boldsymbol{c} & ::= & c_r\mathbf{,}\ \bar{c} \\
M & ::= & x \mid \$\ k \mid \mathtt{addr}\ n \mid \lambda x.\ M \mid \mathtt{let}\ x \mathtt{=} M \mathtt{:} A\ \mathtt{in}\ M \\
 & \mid & \mathtt{app}\ M\ M\ A\ B\ \ell \mid \mathtt{app}\star\ M\ M\ A\ T \\
 & \mid & \mathtt{if}\ M\ A\ \ell\ M\ M \mid \mathtt{if}\star\ M\ T\ M\ M \\
 & \mid & \mathtt{ref}\ \ell\ M \mid \mathtt{ref?}^{p}\ \ell\ M \mid \mathtt{!}\ M\ A\ \ell \mid \mathtt{!}\star\ M\ T \\
 & \mid & \mathtt{assign}\ M\ M\ T\ \ell\ \ell \mid \mathtt{assign?}^{p}\ M\ M\ T\ g \\
 & \mid & \mathtt{prot}\ e\ \ell\ M\ A \mid M\,\langle\,\boldsymbol{c}\,\rangle \mid \mathtt{blame}\ p
\end{array}
$$

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 $\lambda^{\star}_{\mathtt{IFC}}$ is at

  `https://github.com/Gradual-Typing/LambdaIFCStar`

Thank you! ☺