The Holy Grail of Gradual Security

Ph.D Thesis Proposal Presentation

Tianyu Chen

Computer Science, 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, by excluding the unknown label $\star$ from runtime security labels.

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 $\mathsf{GSL}_{\mathsf{Ref}}$
- ▸ 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` : $\texttt{Unit}_{\texttt{low}} \rightarrow \texttt{Bool}_{\texttt{high}}$ and `publish` : $\texttt{Bool}_{\texttt{low}} \rightarrow \texttt{Unit}_{\texttt{low}}$

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 : $\mathtt{Bool}_{\star} \to \mathtt{Bool}_{\star}$ =
  λ b : $\mathtt{Bool}_{\star}$. true in
let i = private-input () in
let result = f i in
  publish result

$\sqsubseteq$

let f : $\mathtt{Bool}_{\star} \to \mathtt{Bool}_{\mathtt{low}}$ =
  λ b : $\mathtt{Bool}_{\star}$. true in
let i = private-input () in
let result = f i in
  publish result

$\sqsubseteq$

more precise

let f : $\mathtt{Bool}_{\mathtt{high}} \to \mathtt{Bool}_{\mathtt{low}}$ =
  λ b : $\mathtt{Bool}_{\mathtt{high}}$. 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 $\texttt{Ref}\ (\texttt{Bool}_{\texttt{low}})$. It points to a low memory location
- The type of the branch condition is $\texttt{Bool}_{\texttt{high}}$

✗ ~~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 $\mathrm{GSL}_{\mathsf{Ref}}$

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 $\mathrm{GSL}_{\mathsf{Ref}}$[2]

- The assignment fails because it is in a high-security branch and $\mathrm{GSL}_{\mathsf{Ref}}$ conservatively treats the reference's label ($\star$) 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.

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

Highlighted security labels default to `low` if omitted:

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

Timeline

I have completed the technical development of $\lambda^{\star}_{\mathtt{IFC}}$. I plan to submit the dissertation in early May. Planned chapters:

1. (DONE) Introduction
2. (DONE) Gradual IFC in $\lambda^{\star}_{\mathtt{IFC}}$
3. (DONE) The Cast Calculus $\lambda^{c}_{\mathtt{IFC}}$
4. (TODO) Compiling From $\lambda^{\star}_{\mathtt{IFC}}$ to $\lambda^{c}_{\mathtt{IFC}}$ (by early March)
5. (TODO) Type Safety of $\lambda^{\star}_{\mathtt{IFC}}$ (by mid March)
6. (TODO) Gradual Guarantee of $\lambda^{\star}_{\mathtt{IFC}}$ (by late March)
7. (TODO) Noninterference of $\lambda^{\star}_{\mathtt{IFC}}$ (by early April)
8. (TODO) Conclusion and Future Work (by early to mid April)

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! ☺