Mechanized Type Safety for Gradual Information Flow

Tianyu Chen    Jeremy G. Siek

Indiana University

Mosaic pattern, Samarkand. Wikimedia Image Archive
Road Map

- Why Gradual Information Flow Typing?
- Interpreting GLIO, in Agda
- Proving Type Safety
- Existing Designs and Future Directions
Consider an application:

- A user enters a string as input. Selected parts of the string are sensitive.
- Sensitive information in the input string must not be disclosed on the web page.
A Solution: *Information Flow Typing*!

▶ A solution is to implement the application in a *security-typed language*.
  ▶ The language regulates the flow of information and enforce *confidentiality*.
  ▶ It satisfies the *noninterference* property, so sensitive *input* does not interfere with publicly observable *output*: \[ \text{High} \not\Rightarrow \text{Low} \].

▶ Implementing the application in a security-typed language will guarantee the *confidentiality* of sensitive user information.
Defining the Grammar with Security Labels

Consider the following user input string:

{FirstName=Mad;LastName=Hatter;SSN=012-34-5678}
Defining the Grammar with Security Labels

Consider the following user input string:

\{FirstName=Mad;LastName=Hatter;SSN=012-34-5678\}

The input grammar:

- High-security terminals are in red; low-security ones in blue.
- The labels are propagated into further processing.
- Thanks to the language satisfying noninterference, high-security SSN digits will never be disclosed.

\[
\langle RECORD \rangle ::= \{ \text{FirstName} = \langle ID \rangle; \\
\quad \text{LastName} = \langle ID \rangle; \\
\quad \text{SSN} = \langle SSN \rangle \}
\]

\[
\langle ID \rangle ::= w, w \in \{A, \ldots Z, a, \ldots z\}^+
\]

\[
\langle SSN \rangle ::= \langle D \rangle \langle D \rangle \langle D \rangle - \langle D \rangle \langle D \rangle - \langle D \rangle \langle D \rangle \langle D \rangle \langle D \rangle \langle D \rangle \langle D \rangle \\
\langle D \rangle ::= d, d \in \{0, \ldots 9\}
\]
Why Gradual Typing?

Developer can choose between:

- ✔ Putting in effort to make the program type check at compile time.
- ✔ Leaving out the annotations to defer the enforcement until runtime.

Our work is based upon GLIO, a gradual security-typed language first introduced by de Amorim et al.

Road Map

- Why Gradual Information Flow Typing?
- Interpreting GLIO, in Agda
- Proving Type Safety
- Existing Designs and Future Directions
Enforcing IFC in GLIO, Fully Dynamically

Example $M^d$:

```ocaml
let f = \x : (Lab i Bool). publish x in
let g = \x : (Lab i Bool). (f x) in
let v = to-label High true in
  g v
```

Consider the example program above, $M^d$:

- The function `publish` publishes a low-security value to publicly visible output.
- Function $f$, $g$ both take boolean with statically unknown label $i$.
- Variable $v$ is bound to a value of high-security.
Enforcing IFC in GLIO, Fully Dynamically

Example $M^d$:

```ocaml
let f = \x : (Lab $i$ Bool). publish x in
let g = \x : (Lab $i$ Bool). (f x) in
let v = to-label High true in
g v
```

Consider the example program above, $M^d$:

- The function `publish` publishes a low-security value to publicly visible output.
- Function $f$, $g$ both take boolean with statically unknown label $i$.
- Variable $v$ is bound to a value of high-security.

✓ $M^d$ is well-typed.
Enforcing IFC in GLIO, Fully Dynamically

Example $M^d$:

```ocaml
let f = \x : (Lab \text{¿} Bool) . \text{publish} x in
let g = \x : (Lab \text{¿} Bool) . (f x) in
let v = to-label \text{High} true in
    g v
```

Consider the example program above, $M^d$:

- The function \text{publish} publishes a \text{low}-security value to publicly visible output.
- Function $f$, $g$ both take boolean with \textit{statically unknown label} \text{¿}.
- Variable $v$ is bound to a value of \text{high}-security.

$M^d$ is well-typed.

Now let's see what happens when we run $M^d$!
Security Checking at *Runtime*

let \( f = \lambda x : (\text{Lab } \diamond \text{ Bool}) . \text{publish } x \) in

let \( g = \lambda x : (\text{Lab } \diamond \text{ Bool}) . (f x) \) in

let \( v = \text{to-label } \text{High} \text{true} \) in

g v

The implicit casts serve as security checks and catch information flow violation at runtime.

- **1\text{st} cast:** \( \text{Lab High Bool } \Rightarrow \text{Lab } \diamond \text{ Bool} \) - permitted ✔
Security Checking at Runtime

let f = \( \lambda x : (\text{Lab } \llcorner \text{ Bool}) \). \text{publish } x \text{ in}

let g = \( \lambda x : (\text{Lab } \llcorner \text{ Bool}) \). (f x) in

let v = \text{to-label } \text{High} \text{ true in}
\hspace{7pt} g \hspace{2pt} v

The implicit casts serve as security checks and catch information flow violation at runtime.

- **1\(^{\text{st}}\) cast:** Lab High Bool \(\Rightarrow\) Lab \(\llcorner\) Bool - permitted ✔
- **2\(^{\text{nd}}\) cast:** Lab \(\llcorner\) Bool \(\Rightarrow\) Lab \(\llcorner\) Bool - permitted ✔
Security Checking at *Runtime*

```ocaml
defined_typed_let f = \x : (Lab ? B). publish x in
defined_typed_let g = \x : (Lab ? B). (f x) in
defined_typed_let v = to-label High true in
g v
```

The implicit casts serve as security checks and catch information flow violation at runtime.

- **1st cast:** Lab High Bool ⇒ Lab ? Bool - **permitted ✓**
- **2nd cast:** Lab ? Bool ⇒ Lab ? Bool - **permitted ✓**
- **3rd cast:** Lab ? Bool ⇒ Lab Low Bool - **rejected ✗**

Execution is terminated due to `castError` and thus information leakage is prevented.
Enforcing IFC in GLIO, Fully Statically

Example $M^s$:

```latex
let f = \lambda x : (Lab \text{ Low} \text{ Bool}) . \text{publish} x \text{ in}
let g = \lambda x : (Lab \text{ Low} \text{ Bool}) . (f x) \text{ in}
let v = \text{to-label} \text{ High} \text{ true} \text{ in}

\text{g v x}
```

- $M^s$ is the fully statically typed counterpart of $M^d$.
- We annotation $f$ and $g$ with static labels Low.
- The program is rejected by the type checker because $\text{High} \not\leq \text{Low}$. Security is enforced statically.
Example $M$:

```
let f = \ x : (Lab Low Bool) . publish x in
let g = \ x : (Lab \ i\ Bool) . (f x) in
let v = to-label High true in
  g v
```

The program $M$ is partially annotated, $f$ has a static annotation while $g$ does not.

- The program is statically well-typed, unlike $M_s$.
- But compared to $M^d$, the security violation can be detected earlier! We shall see on the next slide when $M$ runs.
Detecting Security Violation in $M$

```latex
let f = \lambda x : \text{(Lab Low Bool)}. \text{publish x in}

let g = \lambda x : \text{(Lab \text{? Bool)}. (f x) in}

let v = \text{to-label High true in}

let v = \text{to-label High true in}

\text{g v}
```

Similar to $M^d$, security is enforced at runtime:

- 1\textsuperscript{st} cast: Lab High Bool $\Rightarrow$ Lab \text{? Bool} - permitted $\checkmark$

Execution is terminated due to \textit{castError} earlier than $M^d$ because the program is more annotated.
Detecting Security Violation in $M$

\[
\text{let } f = \lambda x : \text{(Lab Low Bool)}. \text{ publish } x \text{ in }
\]

\[
\text{let } g = \lambda x : \text{(Lab ¿ Bool)}. (f x) \text{ in }
\]

\[
\text{let } v = \text{to-label High true in } g \ v
\]

Similar to $M^d$, security is enforced at runtime:

- 1\text{st cast}: Lab High Bool $\Rightarrow$ Lab ¿ Bool - permitted ✓
- 2\text{nd cast}: Lab ¿ Bool $\Rightarrow$ Lab Low Bool - rejected ×

Execution is terminated due to castError earlier than $M^d$ because the program is more annotated.
The Definitional Interpreter $\mathcal{V}$

$\mathcal{V} : \forall \Gamma T \hat{\ell}_1 \hat{\ell}_2 . (\gamma : \text{Env}) \to (M : \text{Term})$
$\to (\mu : \text{Store}) \to (\text{pc} : \mathcal{L}) \to (k : \mathbb{N})$
$\to \text{Result Conf}$

- $\gamma : \text{Env}$ Maps variables to values.
- $M : \text{Term}$ Runs on a well-typed term.
- $\mu : \text{Store}$ Maps addresses to type-value pairs $\langle T, v \rangle$.
- $\text{pc} : \mathcal{L}$ The program counter label to start with.
- $k : \mathbb{N}$ Gas; so that the interpreter is total.
- Result Conf The evaluation result configuration.
An Example Evaluation (of $M$)

As expected, evaluating $M$ yields a **cast error** at runtime:

\[
\text{Evaluating } M : \\
\text{run-M : } \forall [] \ M [] \ \text{Low 42} \equiv \text{error castError} \\
\text{run-M = refl}
\]
Road Map

- Why Gradual Information Flow Typing?
- Interpreting GLIO, in Agda
- Proving Type Safety
- Existing Designs and Future Directions
Desirable Language Properties

- de Amorim et al. prove that:
  - **Noninterference:** \( GLIO \) is secure.
  - **Gradual Guarantees:** Removing annotations, the term remains well-typed and has the same runtime behavior.

- We prove that:
  - **Type Safety:** Undefined behavior never occurs in \( GLIO \).

- Future work:
  - **Blame theorem:** A cast cannot be blamed if its source type and target type satisfy subtyping.
  - **Space Efficiency:** Casts are compressed so they do not grow in an unbounded fashion.
Why Care About *Type Safety*?

- Distinguish between different types of errors: 
  
  All errors \( \{ \text{Trapped}, \text{Untrapped} \} \)

- *Untrapped errors* are bad because they are undefined behavior; can be used to hack a program.

- **Type safety**: *untrapped errors never occur!*
Why Care About *Type Safety*?

- Distinguish between different types of errors:
  
  All errors \(\{\text{Trapped}, \text{Untrapped}\}\)

- *Untrapped errors* are bad because they are undefined behavior; can be used to hack a program.

- **Type safety:** *untrapped errors never occur!*

- Machine configuration and evaluation result:

  \[ L = \{\text{Low, High}\} \]

  \[ c \in \text{Store} \times \text{Value} \times L, e \in \text{Error} \]

  Result ::= timeout | error \(e\) | conf \(c\)
Theorem Statement of Type Safety

Theorem (Type safety)

If term $M$ is well-typed: $\emptyset \vdash \hat{l}_1, \hat{l}_2 \quad M : T$, the evaluation result of $M$ is also well-typed:

$$\vdash \forall \emptyset \ M \ _\emptyset \ pc \ k : T$$

Untrapped error is ruled out by well-typedness:

\begin{align*}
\vdash \text{timeout} : T \\
\vdash \text{error} \ e : T \\
\mu \vdash \mu \\
\mu \vdash \nu : T \\
\vdash \text{conf} \langle \mu, \nu, pc \rangle : T
\end{align*}
Road Map

► Why Gradual Information Flow Typing?
► Interpreting GLIO, in Agda
► Proving Type Safety
► Existing Designs and Future Directions 🤔
Gradual Security-Typed Language Properties

<table>
<thead>
<tr>
<th>System</th>
<th>Noninterference</th>
<th>Type Safety</th>
<th>Gradual guarantees</th>
<th>Blame theorem</th>
<th>Space efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_{gif}$</td>
<td>✓ Yes</td>
<td>✓ Yes</td>
<td>* Maybe</td>
<td>✓ Yes</td>
<td>✗ No</td>
</tr>
<tr>
<td>ML-GS</td>
<td>✓ Yes</td>
<td>✓ Yes</td>
<td>* Maybe</td>
<td>* Maybe</td>
<td>✗ No</td>
</tr>
<tr>
<td>GSL$_{Ref}$</td>
<td>✓ Yes</td>
<td>✓ Yes</td>
<td>✗ No</td>
<td>✗ No</td>
<td>* Maybe</td>
</tr>
<tr>
<td>GLIO</td>
<td>✓ Yes</td>
<td>✓ Yes</td>
<td>✓ Yes</td>
<td>✗ No</td>
<td>✗ No</td>
</tr>
</tbody>
</table>

- Two languages that satisfy the most properties are $\lambda_{gif}$ and GLIO. However, as is mentioned earlier, $\lambda_{gif}$ lacks mutable reference.

- The paper de Amorim et al. proves both noninterference and gradual guarantees for GLIO, resolving the tension proposed in the GSL$_{Ref}$ paper by having casts checking labels only, with classifying the data.

- Unfortunately, GLIO does not perform blame tracking. It would be difficult to add blame tracking to GLIO due to its heap model.

* We summarize our vision for a future design on the next slide.
Language Design Choices & Future Directions

To facilitate all five properties, we recommend the following design choices:

- **Value labeling:** Associating values with concrete labels (Low, High, ...); similar to GLIO.

- **Heap model:** Simple heap (no extra information stored) and reference proxies.

- **Surface language and cast insertion:** Having both surface language and cast insertion; similar to GSL_{Ref}.

- **Labeling granularity:** Fine-grained labeling; similar to λ_{gif}, ML-GS, and GSL_{Ref}.
Thank you!!

Any questions?