All Your IFCException Are Belong To Us

Cătălin Hrițcu, Michael Greenberg, Ben Karel,
Benjamin Pierce, Greg Morrisett

Robust exception handling mechanism for sound fine-grained dynamic information flow control

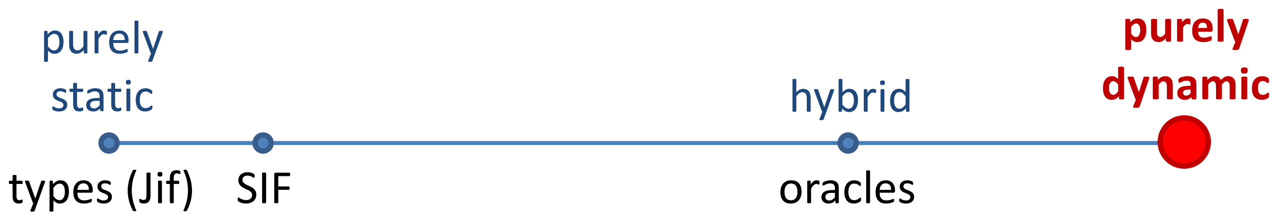
problem: exceptions can leak information

solution: public labels + delayed exceptions

Information flow control

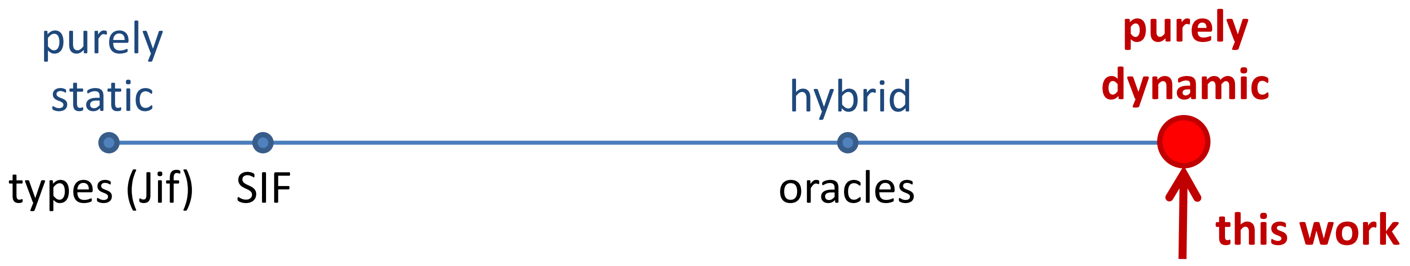
Protect secrecy and integrity
by assigning security levels (labels) to data
and preventing information leaks

Information flow control



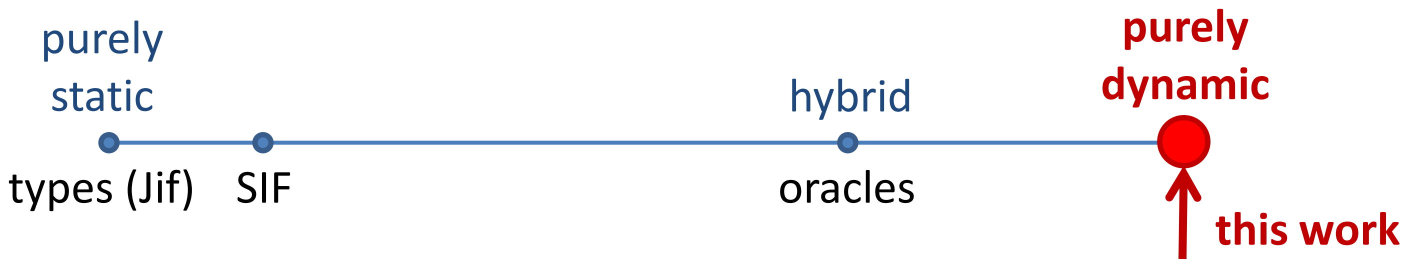
Protect secrecy and integrity
by assigning security levels (labels) to data
and preventing information leaks

Information flow control



Protect secrecy and integrity
by assigning security levels (labels) to data
and preventing information leaks

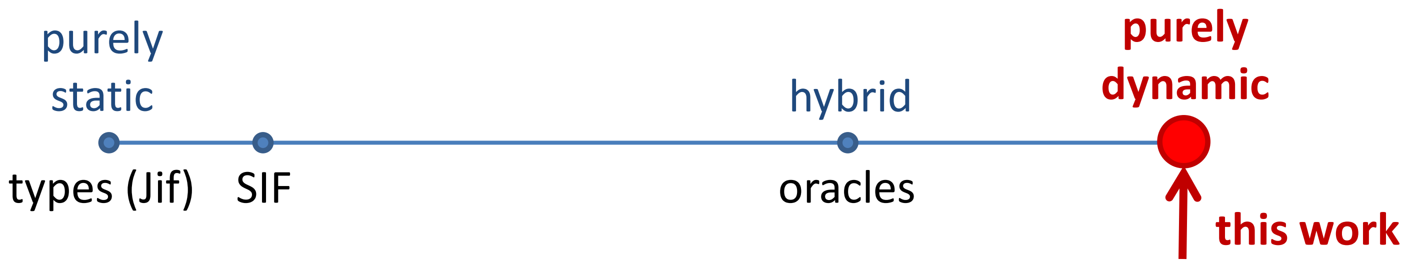
Information flow control



Why dynamic?

1. security policy is often dynamic
2. static analysis not always easily applicable, e.g.

Information flow control

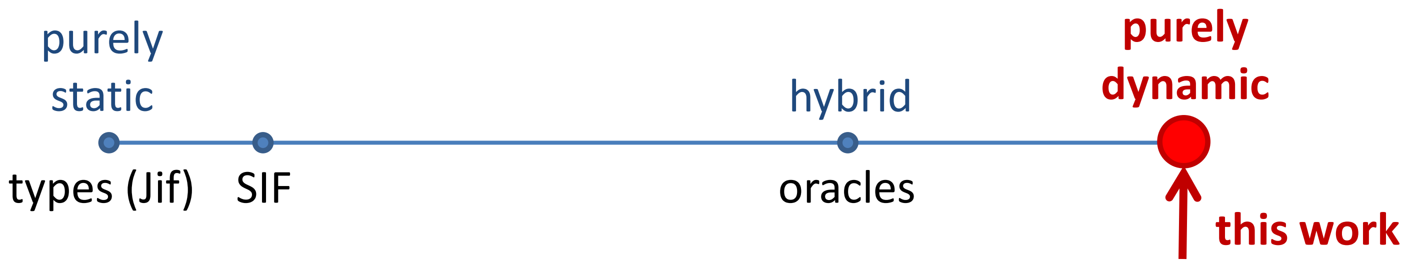


Why dynamic?

1. security policy is often dynamic
2. static analysis not always easily applicable, e.g.
   - high-level dynamic languages (JavaScript)

   - low-level machine code

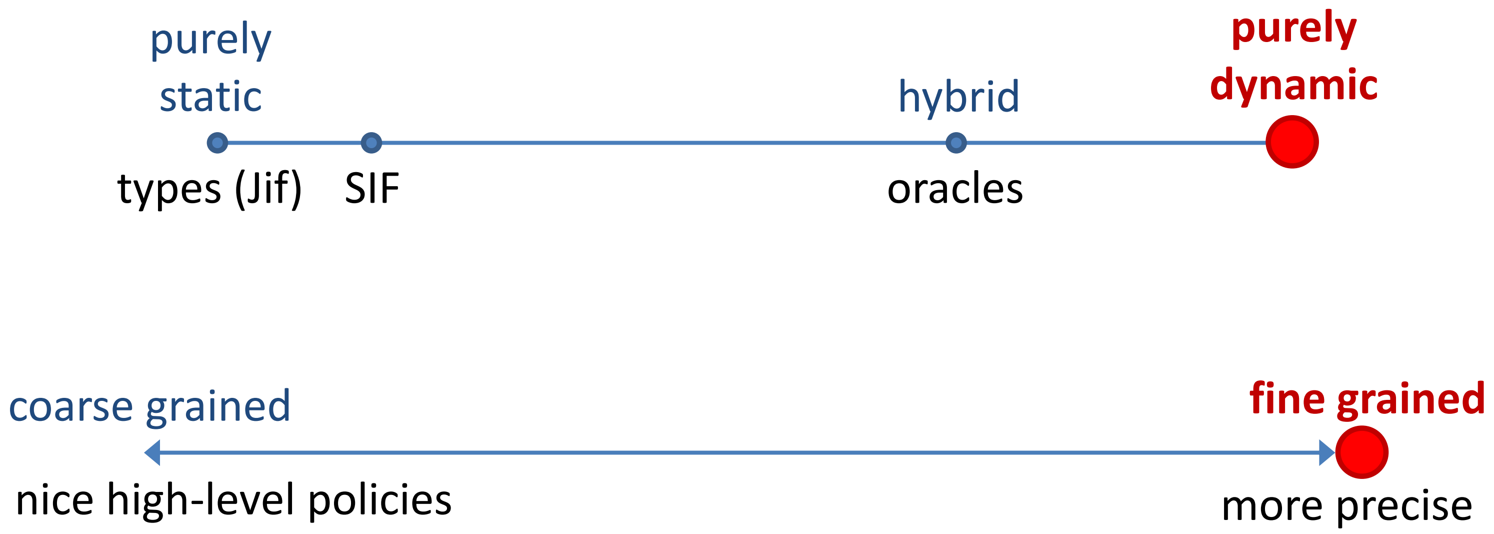
Information flow control



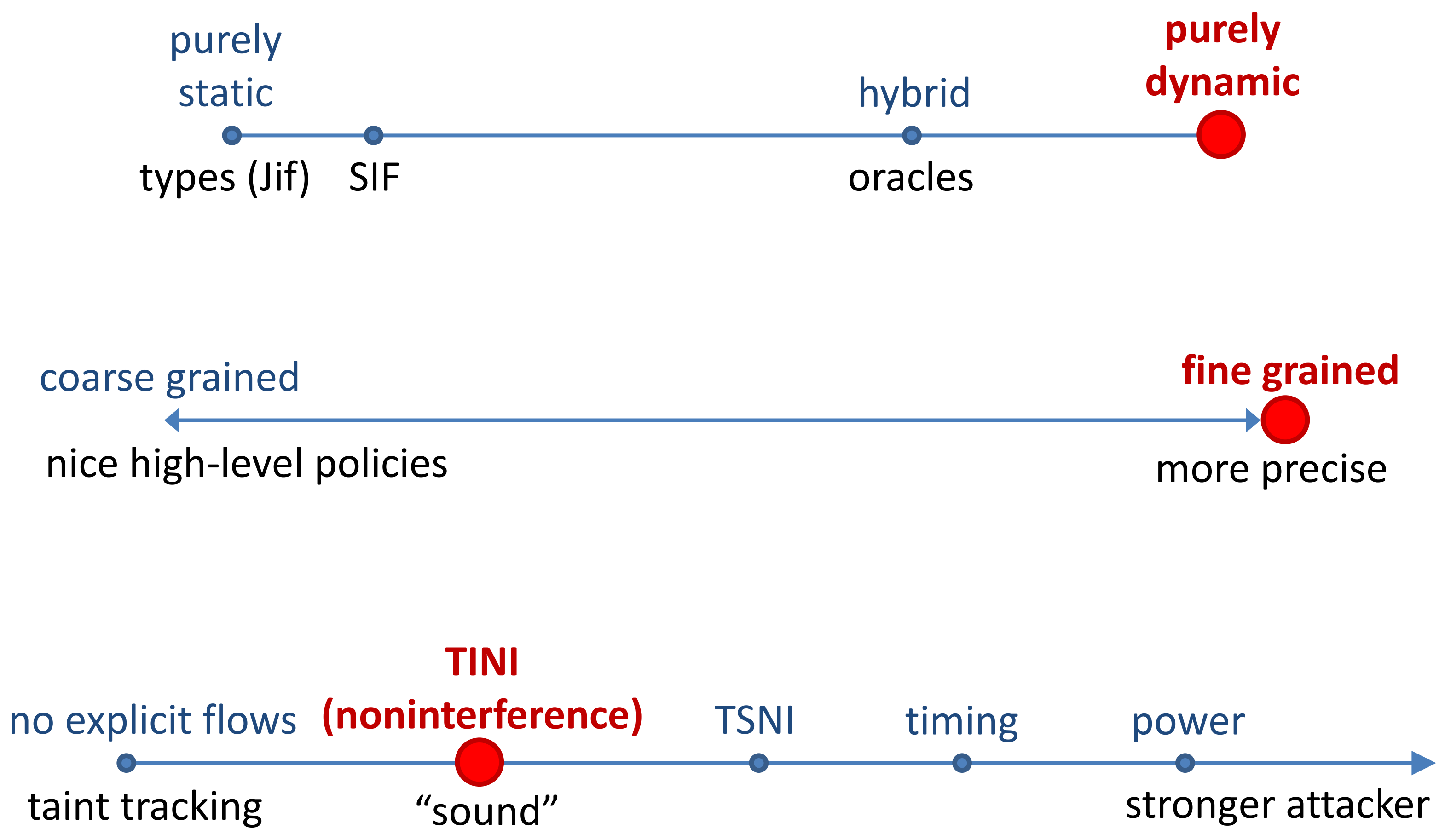
Why dynamic?

1. security policy is often dynamic
2. static analysis not always easily applicable, e.g.
   - high-level dynamic languages (JavaScript)
     - **this talk**: Breeze, new language (no legacy constraints)
   - low-level machine code
     - CRASH/SAFE project: OS+HW-supported IFC

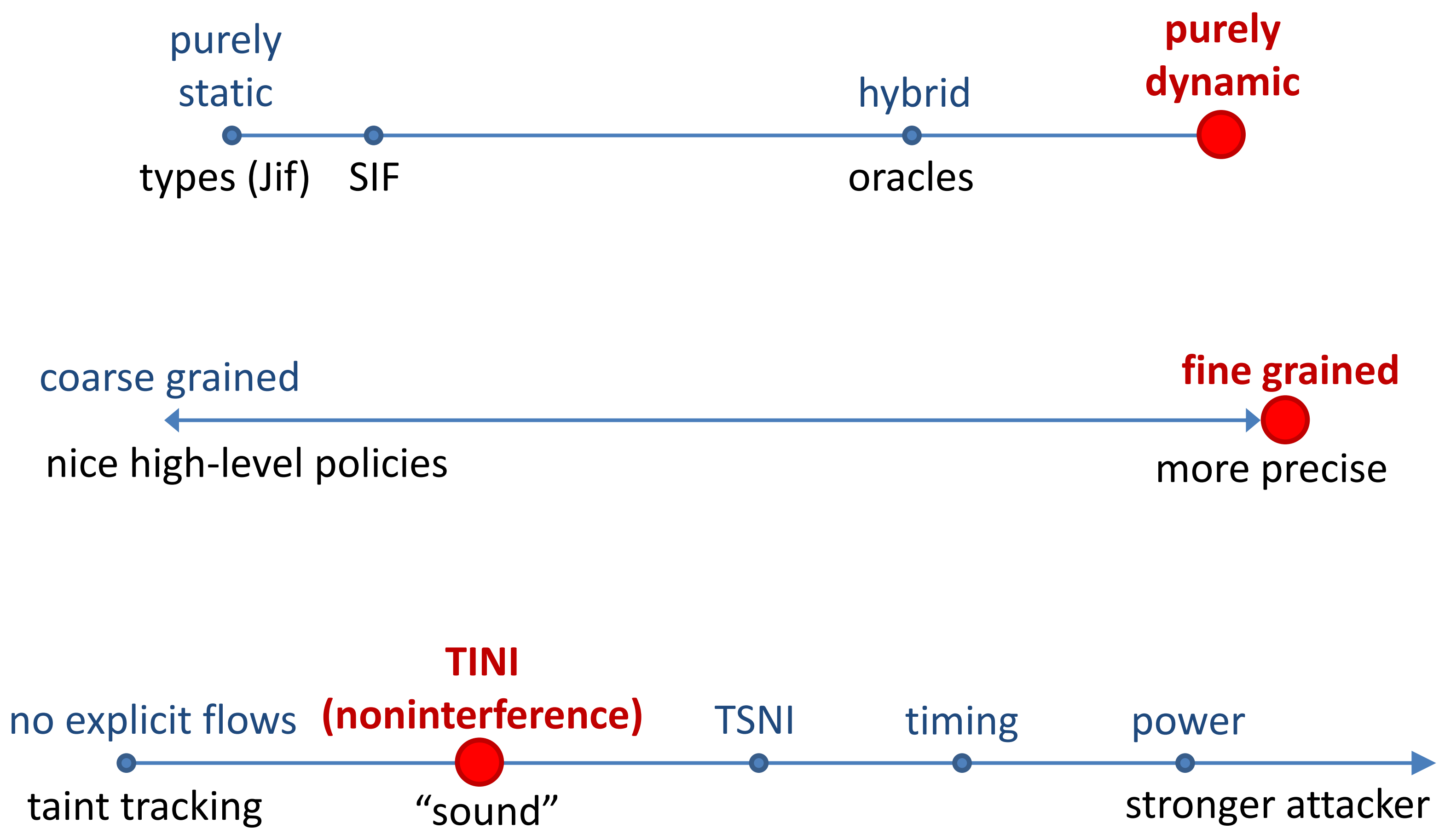
Information flow control



Information flow control



Is this even possible?



Yes, this is possible!

- TINI can be obtained purely dynamically!
  [Sabelfeld & Russo, 2009], [Austin & Flanagan, 2009]
- preventing implicit flows:
  - no low assignments in high contexts (branching on secrets)
  - `l:=false; if h {l:=true}; ...` is terminated

Yes, this is possible!

- TINI can be obtained purely dynamically!
  [Sabelfeld & Russo, 2009], [Austin & Flanagan, 2009]
- preventing implicit flows:
  - no low assignments in high contexts (branching on secrets)
  - `l:=false; if h {l:=true}; ...` is terminated
  - `l:=false; if h {l:=true}; l := false` has TINI
- TINI not a safety property [Fred Schneider, TISSEC ’00]
  - so we enforce a conservative approximation
  - incompleteness didn’t stop static enforcement either

Yes, this is possible!

- TINI can be obtained purely dynamically!

  [Sabelfeld & Russo, 2009], [Austin & Flanagan, 2009]

- preventing implicit flows:
  - no low assignments in high contexts (branching on secrets)
  - `l:=false; if h {l:=true}; ...` is terminated

- “stopping the world” not an option
  - can’t punt on availability / reliability to get secrecy / integrity

Contributions

- **showing that robust error handling is possible**
  - recovery from all errors, including IFC violations
  - without sacrificing soundness (TINI) or precision
- **identifying the 2 necessary ingredients**

  = solutions to 2 general problems:

  1. IFC exceptions can leak via labels → **public labels**
  2. all exceptions can leak via control → **delayed exceptions**
- **exploring the entire design space**
- **experimentally evaluating most radical design**

Contributions

rest of this talk focused on this part

- **showing that robust error handling is possible**
  - recovery from all errors, including IFC violations
  - without sacrificing soundness (TINI) or precision
- **identifying the 2 necessary ingredients**
  = solutions to 2 general problems:
  1. IFC exceptions can leak via labels → **public labels**
  2. all exceptions can leak via control → **delayed exceptions**
- **exploring the entire design space**
- **experimentally evaluating most radical design**

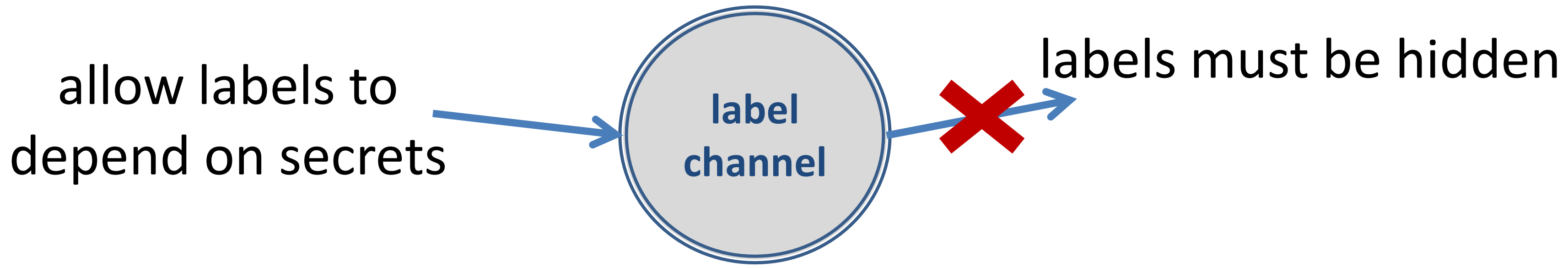
Problem #1:
IFC exceptions can leak via labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either ***into*** or ***out of*** label channel

label channel

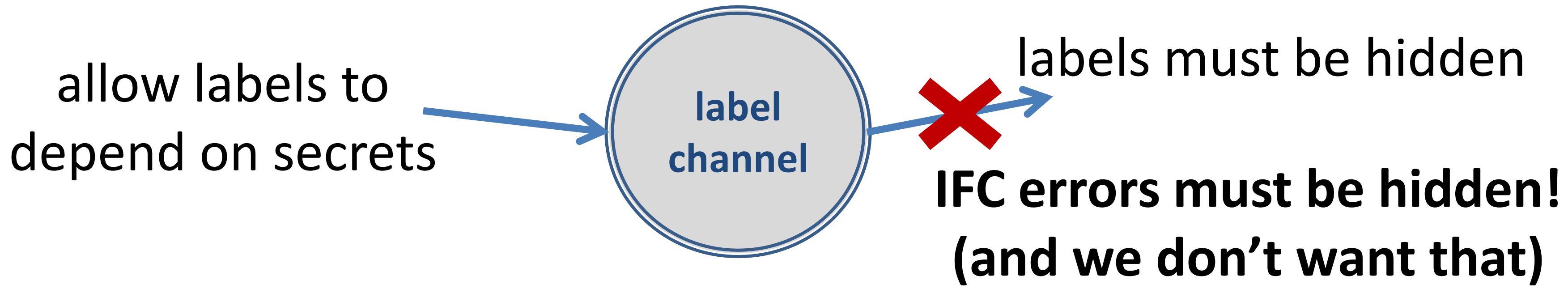
Problem #1: IFC exceptions can leak via labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either ***into*** or ***<u>out of</u>*** label channel



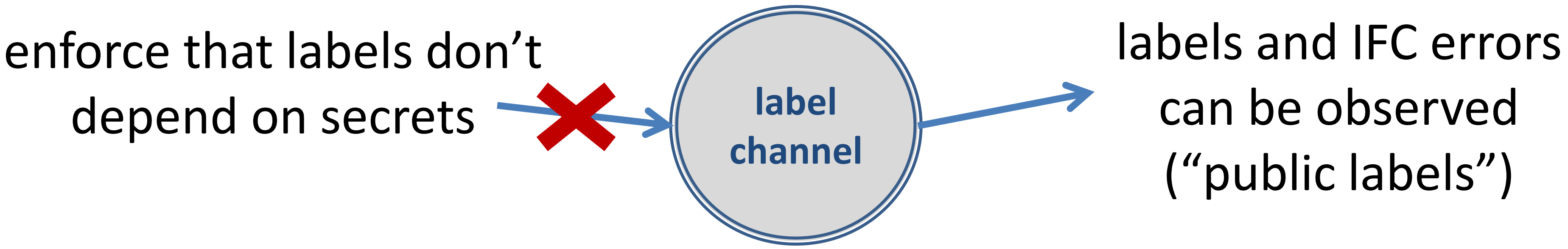
Problem #1: IFC exceptions can leak via labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking either ***into*** or ***out of*** label channel



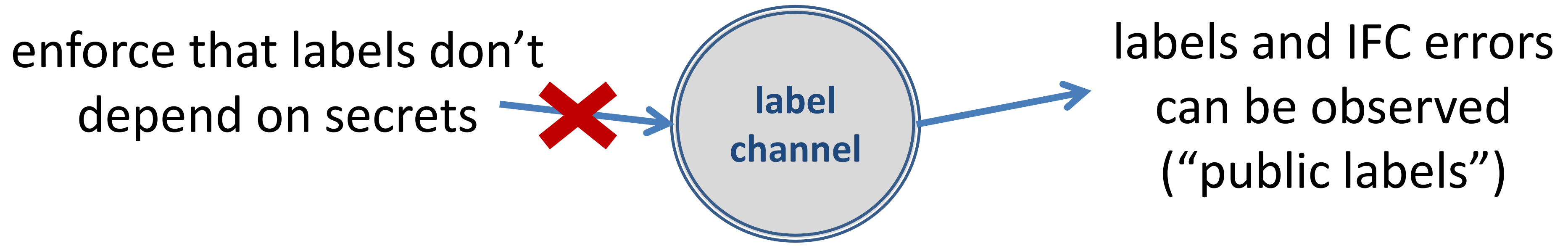
Problem #1:
IFC exceptions can leak via labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking ~~either~~ ***into*** ~~or~~ ~~***out of***~~ label channel


enforce that labels don't depend on secrets
label channel
labels and IFC errors can be observed ("public labels")

Problem #1:
IFC exceptions can leak via labels

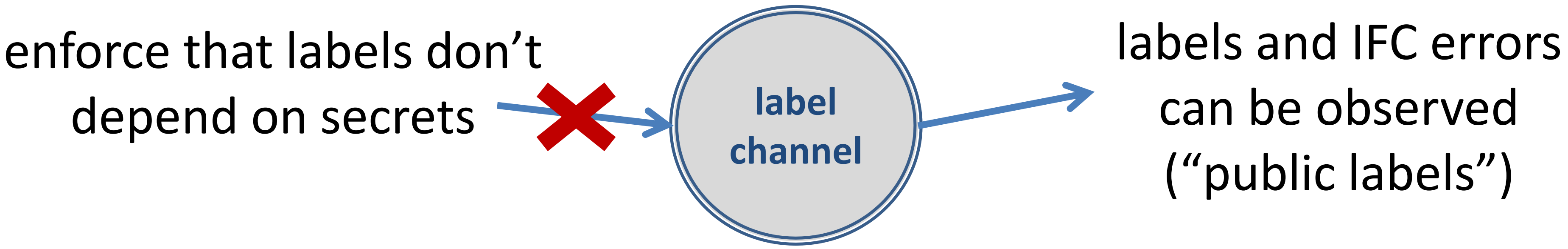
- labels are themselves information channels
- get soundness by preventing secrets from leaking ~~either~~ ***into*** ~~or **out of**~~ label channel


enforce that labels don't depend on secrets
label channel
labels and IFC errors can be observed ("public labels")

```
if s then ()@secret else ()@top-secret
```

Problem #1: IFC exceptions can leak via labels

- labels are themselves information channels
- get soundness by preventing secrets from leaking ~~either~~ ***into*** ~~or ***out of***~~ label channel


enforce that labels don't depend on secrets
label channel
labels and IFC errors can be observed ("public labels")

Solution #1: sound public labels via brackets

```
top-secret[if s then ()@secret else ()@top-secret]
```

[Deian Stefan et al., IFCP 2011]

Problem #2: Exceptions can leak via control

- ending brackets need to be control flow join points, otherwise...

  -
    ```
    try
      let _ = secret[if h then throw Ex] in
      false
    catch Ex => true
    ```

Problem #2:
Exceptions can leak via control

- ending brackets need to be control flow join points, otherwise...

  -
    ```
    try
      let _ = secret[if h then throw Ex] in
      false
    catch Ex => true
    ```
- brackets need to **delay** all exceptions!
  - `secret[if true@secret then throw Ex] => "(Error Ex)@secret"`
  - `secret [if false@secret then throw Ex] => "(Success ())@secret"`

Problem #2: Exceptions can leak via control

- ending brackets need to be control flow join points, otherwise...

  -
    ```
    try
      let _ = secret[if h then throw Ex] in
      false
    catch Ex => true
    ```
- brackets need to **delay** all exceptions!
  - `secret[if true@secret then throw Ex] => “(Error Ex)@secret”`
  - `secret [if false@secret then throw Ex] => “(Success ())@secret”`
- similarly for failed brackets
  - `secret[42@top-secret] => “(Error EBracket)@secret”`

Solution #2: Delayed exceptions

- **delayed exceptions unavoidable**
  - still have a choice how to propagate them
- we studied **two main alternatives**:
  1. **mix active and delayed exceptions** ($\lambda^{[\,]}{}_{\mathit{throw}}$)

Solution #2: Delayed exceptions

- **delayed exceptions unavoidable**
  - still have a choice how to propagate them
- we studied **two main alternatives**:
  1. **mix active and delayed exceptions** ($\lambda^{[\,]}_{throw}$)
  2. **only delayed exceptions** ($\lambda^{[\,]}_{NaV}$)
     - delayed exception = not-a-value (NaV)
     - NaVs are first-class replacement for values
     - NaVs propagated solely via data flow
     - NaVs are labeled and pervasive
     - simpler and more radical solution; implemented in Breeze

What’s in a NaV? Debugging aids!

- error message
  - `EDivisionByZero (“can’t divide %1 by 0”, 42)
- stack trace
  - pinpoints error **origin**
    - very different than for NullPointerException (the billion-dollar mistake)
- propagation trace
  - how did the error make it here?

Formal results

- proved TINI in Coq for $\lambda^{[\,]}$, $\lambda^{[\,]}_{NaV}$, and $\lambda^{[\,]}_{throw}$
  - for $\lambda^{[\,]}_{NaV}$ even with all debugging aids; **error-sensitive**
- some evidence that NaVs and catchable exceptions have **equivalent expressive power** (in theory)
  - translations validated by QuickChecking extracted code

$$\begin{tikzcd}
& \boxed{\lambda^{[\,]}} & \\
\boxed{\lambda^{[\,]}_{NaV}} \arrow[ur, bend left] \arrow[rr, leftrightarrow] & & \boxed{\lambda^{[\,]}_{throw}} \arrow[ul, bend right]
\end{tikzcd}$$

Formal results

- proved TINI in Coq for $\lambda^{[\,]}$, $\lambda^{[\,]}_{NaV}$, and $\lambda^{[\,]}_{throw}$
  - for $\lambda^{[\,]}_{NaV}$ even with all debugging aids; **error-sensitive**
- some evidence that NaVs and catchable exceptions have **equivalent expressive power** (in theory)
  - translations validated by QuickChecking extracted code

$$\begin{tikzcd}
& \lambda^{[\,]} & \\
\lambda^{[\,]}_{NaV} \arrow[ur, bend left] \arrow[rr, leftrightarrow] & & \lambda^{[\,]}_{throw} \arrow[ul, bend right]
\end{tikzcd}$$

New work:
Testing Noninterference, Quickly

Conclusion

- reliable error handling **possible** even for sound fine-grained dynamic IFC systems
- two mechanisms ($\lambda^{[\,]}_{NaV}$ and $\lambda^{[\,]}_{throw}$) and variants
  - **all errors recoverable**, even IFC violations
  - **necessary** ingredients: **sound public labels** (brackets)
    + **delayed exceptions**
  - quite **radical design** (not backwards compatible!)
  - **delayed exceptions applicable to static IFC**