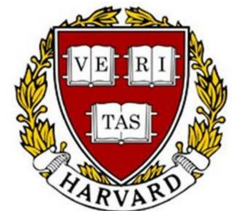
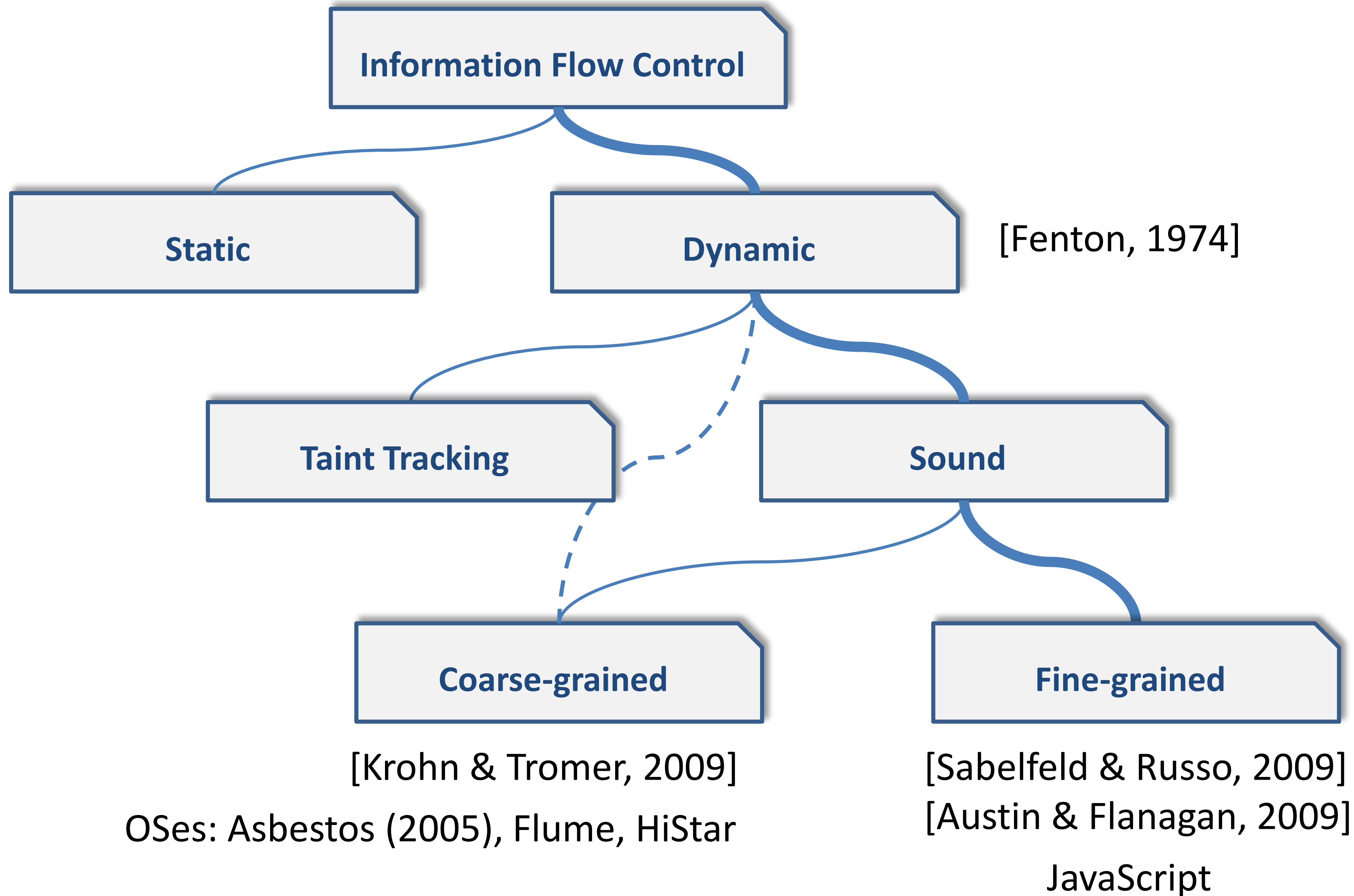
# All Your IFCException Are Belong To Us

Cătălin Hrițcu

(joint work with Michael Greenberg, Ben Karel,
Benjamin Pierce, Greg Morrisett, and more)

2012-11-05 -- WG 2.8 meeting in Annapolis





Information Flow Control
Static
Dynamic
[Fenton, 1974]
Taint Tracking
Sound
Coarse-grained
Fine-grained
[Krohn & Tromer, 2009]
OSes: Asbestos (2005), Flume, HiStar
[Sabelfeld & Russo, 2009]
[Austin & Flanagan, 2009]
JavaScript

# Breeze

- **sound fine-grained dynamic IFC**
- **label-based discretionary access control**
  - clearance helps prevent covert channels
- **functional core** (λ) + state(!) + concurrency (π)
  - from Pict/CML towards something more Erlang-ish
- **dynamically typed**
  - directly reflects capabilities of CRASH/SAFE HW
  - dynamically-checked first-class **contracts**

# Exception handling

- we wanted all Breeze errors to be **recoverable**
  - **including IFC violations! (IFCException)**
- however, existing work* assumes errors are **fatal**
  - makes some things easier ... at the expense of others

**+secrecy** **+integrity** **–availability**

*There are 2 very recent (partial) exceptions:
[Stefan et al., 2012] and [Hedin & Sabelfeld, 2012]

# Poison-pill attacks

channels only do top-level label checks

```
let cin = chan low;
let cout = chan low;

fun process_max x y =
  if x <= y then y else x

fun rec max_server_loop () =
  let (x,y) = recv cin in
  let res = process_max x y in
  send cout res;
  max_server_loop ()
```

3@low <= 2@high = false@high
pc=high → result is high

x=3@low y=2@high
res=3@high

**max_server gets killed because of IFC violation!?**

```
let client = send cin (3, 5)@low; recv cout = 5

let bclient = send cin (3, 5)@high

let attacker = send cin (3, 2@high)@low
```

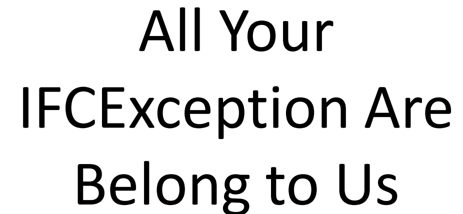
**bclient gets killed**

# Wishful thinking

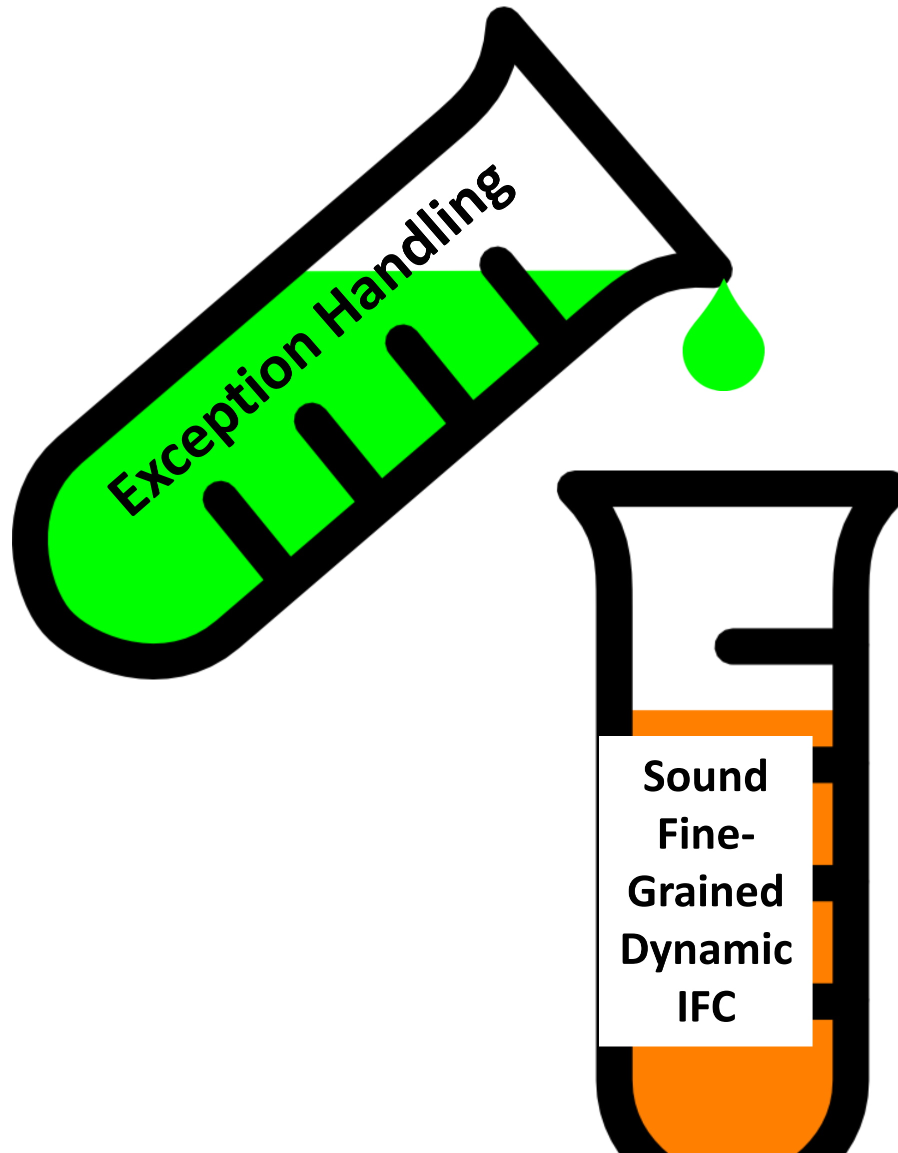
```
let cin = chan low;
let cout = chan low;

fun process_max (x,y) =
  if x <= y then y else x

fun rec max_server_loop' () =
  try
    send cout (process_max (recv cin))
  catch x => log x;
  max_server_loop' ()
```



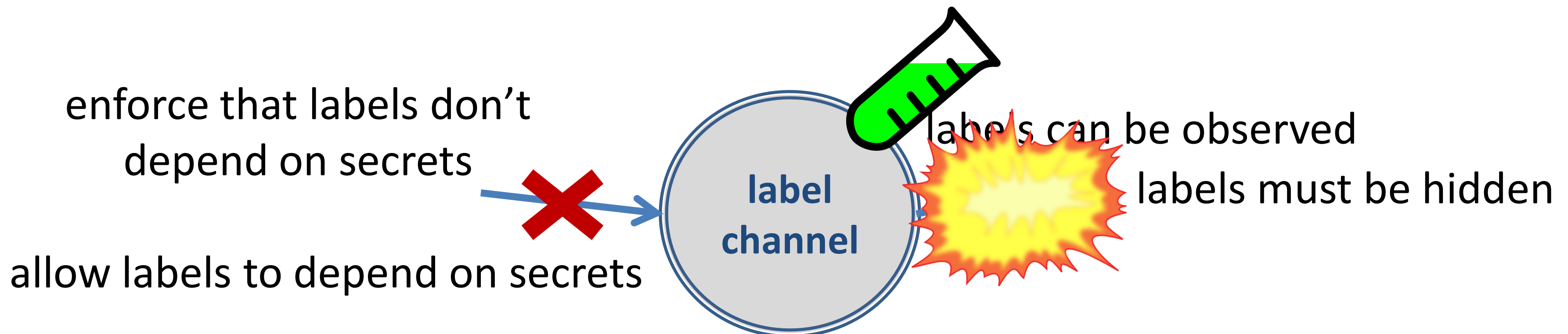
# But there is a problem **... in fact two!**



# But there is a problem ... **in fact two!**

# Labels are information channels

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



# Problem #1: IFC exceptions make labels public

- ... and that's unsound if labels can depend on secrets

- secret bit: h@high  `low <: high <: top`

```
let href = ref high () in
.......
try
  href := (if h then ()@high
           else      ()@top );
  true
catch IFCException => false
```

encode h into label

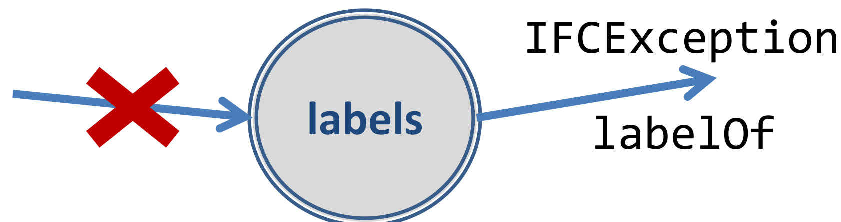
pc automatically restored to low once the if branches merged

vorks
on

so false/true is low

# Solution to problem #1: brackets

- no longer automatically restore pc
  - `pc=low` `if h then ()@high else ()@top` `pc=high`
- instead, restore pc manually using **brackets**
  - choose label before branching on secrets
  - `pc=low` `top[if h then ()@high else ()@top]` `pc=low`
  - brackets are not declassification!
  - sound even when annotation is incorrect (more later)
- **labels can now be soundly made public**
  - bracket annotations can be dynamically computed



# Problem #2: exceptions destroy control flow join points

- ending brackets have to be control flow join points
  - ```
    try
      let _ = high[if h then throw Ex] in
      false
    catch Ex => true
    ```
- brackets need to delay all exceptions!
  - `high[if true then throw Ex] => "(Inr Ex)@high"`
  - `high[if false then throw Ex] => "(Inr ())@high"`
    ```

# Solution #2: Delayed exceptions

- **delayed exceptions unavoidable**
  - still have a choice how to propagate them
- we studied **two alternatives** for error handling:
  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
     - more radical solution; implemented by Breeze

# NaV-lax vs. NaV-strict behavior

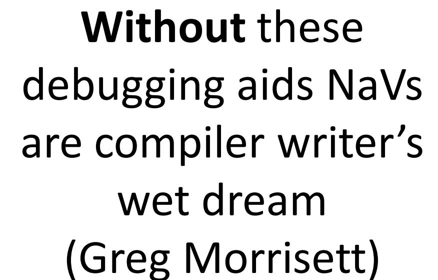
- all non-parametric operations are NaV-strict
  - `NaV@low + 42@high => NaV@high`
- for parametric operations we can chose:

```
                          NaV-lax          or    NaV-strict
– (fun x => 42) NaV  => 42               or    => NaV
– Cons NaV Nil       => Cons NaV Nil     or    => NaV
– (r := NaV,r=7)     => ((),r=NaV)       or    => (NaV,r=7)
```

- NaV-strict behavior reveals errors earlier
  - but it also introduces additional IFC constraints
- in Breeze the programmer can choose
  - in formal development NaV-lax everywhere

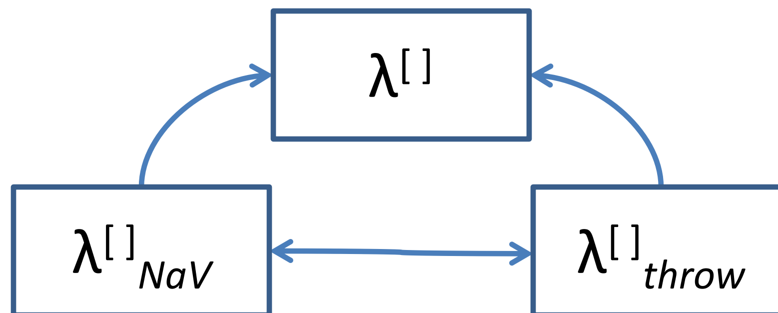
# What’s in a NaV?

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



# Formal results

- proved **error-sensitive non-interference** in Coq for $\lambda^{[\,]}$, $\lambda^{[\,]}_{NaV}$, and $\lambda^{[\,]}_{throw}$ (termination-insensitive)
  - for $\lambda^{[\,]}_{NaV}$ even with all debugging aids
- **conjecture**: in our setting NaVs and catchable exceptions have equivalent expressive power
  - translations validated by quick-checking code extracted from Coq (working on Coq proofs)



# Conclusion

- reliable error handling *possible* even for sound fine-grained dynamic IFC systems
- we study two mechanisms ($\lambda^{[\,]}_{NaV}$ and $\lambda^{[\,]}_{throw}$)
  - **all errors recoverable**, even IFC violations
  - necessary ingredients:<br>**sound public labels** (brackets) + **delayed exceptions**
  - quite radical design (not backwards compatible!)
- practical experience with NaVs
  - issues are surmountable
  - writing good error recovery code is still hard

# THE END