All Your IFCException Are Belong To Us

Cătălin Hrițcu

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

[Denning, 1977]

Static

Dynamic

[Fenton, 1974]

type systems & program analysis
Jif, FlowCaml, ...
Information Flow Control

Static

Dynamic

Taint Tracking

only explicit flows
Perl Taint mode, ...

[Fenton, 1974]
Information Flow Control

Static

- only explicit flows
- Perl Taint mode, ...

Dynamic

- Sound + Complete Impossible

[Fenton, 1974]

[Schneider, 2000]

non-interference not a trace property

can’t be precisely enforced by EM
Information Flow Control

Static

Taint Tracking

Dynamic

Sound

enforce stronger property (incomplete)
changing language semantics allowed
also prevents implicit flows
non-interference proofs

[Krohn & Tromer, 2009]  [Sabelfeld & Russo, 2009]

[Austin & Flanagan, 2009]
Information Flow Control

Static

Dynamic

Taint Tracking

Sound

Coarse-grained

Fine-grained

[Krohn & Tromer, 2009]

OSes: Asbestos (2005), Flume, HiStar

[Sabelfeld & Russo, 2009]

JavaScript

[Fenton, 1974]
Preventing implicit flows

• let lref = ref low false in

  if h then
    lref := true;
  lref := false

  pc=high
  bad flow -> halt program
  false alarm (program non-interferent)

• even purely functional code can leak via control flow:
  – if h then true else false
  – semantics of conditional:
    • if true@high then true else false => true@high
Breeze

• sound fine-grained dynamic IFC
• label-based discretionary access control
  – clearance helps prevent covert channels
• functional core \((\lambda) + \text{state}(!) + \text{concurrency}(\Pi)\)
  – 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]
But there is a problem
But there is a problem
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

enforce that labels don’t depend on secrets

allow labels to depend on secrets

labels can be observed

labels must be hidden
Problem #1: IFC exceptions reveal information about labels

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

```ml
let href = ref high () in
let href = ref high () in

try
    href := (if h then ()@high
    else ()@top );
    href := (if h then ()@high
    else ()@top );

true
true

catch IFCException => false

IFC errors must be hidden too (not low observable)
we don’t want this restriction!
```

allow labels to depend on secrets

labels must be hidden
Solution to problem #1: brackets

• prevent labels from depending on secrets so that labels are public
• no longer automatically restore pc
  – \( pc=\text{low} \ if \ h \ then \ ()@\text{high} \ else \ ()@\text{top} \ \ pc=\text{high} \)
• instead, restore pc manually using \textbf{brackets}
  – choose label on result before branching on secrets
  – \( pc=\text{low} \ \ \text{top}[if \ h \ then \ ()@\text{high} \ else \ ()@\text{top}] \Rightarrow ()@\text{top} \ \ pc=\text{low} \)
  – brackets are not declassification!
  – sound even when annotation is incorrect (next slide)
  – bracket annotations can be dynamically computed (labelOf)
Problem #2: exceptions destroy control flow join points

• ending brackets have to be control flow join points
  – try
    \[
    \text{let } _ = \text{high[if } h \text{ then throw Ex] in}
    \text{false}
    \text{catch Ex => true}
    \]

• brackets need to delay all exceptions!
  – \text{high[if true@high then throw Ex]} \Rightarrow "(Inr Ex)@high"
  – \text{high[if false@high then throw Ex]} \Rightarrow "(Inl ())@high"

• similarly for failed brackets
  – \text{high[(]@top]} \Rightarrow "(Inr EBracket)@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 \((\lambda[\text{throw}])\)
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 ($\lambda^\text{[]}_{\text{throw}}$)
  2. only delayed exceptions ($\lambda^\text{[]}_{\text{NaV}}$)
    • 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
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?

NaVs are compiler writer’s dream, especially if compiler is allowed to be imprecise about these debugging aids (Greg Morrisett)
Formal results

• proved termination-insensitive **non-interference** in Coq for \( \lambda[^1] \), \( \lambda[^1]_{NaV} \), and \( \lambda[^1]_{\text{throw}} \)
  – for \( \lambda[^1]_{NaV} \) even with all debugging aids; **error-sensitive**

• **conjecture**: in our setting NaVs and catchable exceptions have equivalent expressive power
  – translations validated by QuickChecking 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[^{\text{\text{\[}}}NaV$ and $\lambda[^{\text{\text{\[}}}\text{throw}$)
  – all errors recoverable, even IFC violations
  – key ingredients: sound public labels (brackets) + delayed exceptions
  – quite radical design (not backwards compatible!)
• our practical experience with NaVs:
  – issues are surmountable
  – writing good error recovery code is still hard
THE END