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

Taint Tracking

Coarse-grained

[krohn & tromer, 2009]

OSes: Asbestos (2005), Flume, HiStar

[kent, 1974]

Sound

Fine-grained

[sabelfeld & russo, 2009]

[ Austin & Flanagan, 2009]

JavaScript
• 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]
Poison-pill attacks

```ocaml
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 ()

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

channels only do top-level label checks

```
3@low <= 2@high = false@high
pc=high  \(\rightarrow\) result is high
```

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

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

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

```
let client = send cin (3, 5)@low; recv cout = 5
let bclient = send cin (3, 5)@high  \(\text{bclient gets killed}\)
let attacker = send cin (3, 2@high)@low
```
Wishful thinking

```ocaml
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' ()
```

All Your IFCException Are Belong to Us
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

![Diagram showing label channel with arrows indicating enforcing that labels don't depend on secrets and labels can be observed, labels must be hidden, allowing labels to depend on secrets are incorrect.]

enforce that labels don’t depend on secrets
labels can be observed
labels must be hidden
allow labels to depend on secrets
Problem #1: IFC exceptions make labels public

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

- secret bit: h@high
  
  \[
  \text{let } \text{href} = \text{ref high }() \text{ in }
  \]
  
  try
  
  encode h into label
  
  \[
  \text{href := (if h then } ()@\text{high}
  \]
  
  \[
  \text{else } ()@\text{top });
  \]
  
  true

  catch IFCException => false

  pc automatically restored to low once the if branches merged

  so false/true is low
Solution to problem #1: brackets

• no longer automatically restore pc
  – pc=low \textbf{if h then } ()@high \textbf{ else } ()@top \textbf{ pc=high}

• instead, restore pc manually using \textbf{brackets}
  – choose label before branching on secrets
  – pc=low \textbf{top[if h then } ()@high \textbf{ else } ()@top\textbf{]} \textbf{pc=low}
  – brackets are not declassification!
  – sound even when annotation is incorrect (more later)

• \textbf{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 \((\lambda[^1]_{\text{throw}})\)
  2. only delayed exceptions \((\lambda[^1]_{NaV})\)
    • 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:

<table>
<thead>
<tr>
<th>NaV-lax</th>
<th>or</th>
<th>NaV-strict</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fun x =&gt; 42) NaV =&gt; 42</td>
<td>or</td>
<td>=&gt; NaV</td>
</tr>
<tr>
<td>Cons NaV Nil =&gt; Cons NaV Nil</td>
<td>or</td>
<td>=&gt; NaV</td>
</tr>
<tr>
<td>(r := NaV,r=7) =&gt; ((),r=NaV)</td>
<td>or</td>
<td>=&gt; (NaV,r=7)</td>
</tr>
</tbody>
</table>

• 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?

Without these debugging aids NaVs are compiler writer’s wet dream
(Greg Morrisett)
Formal results

• proved **error-sensitive non-interference** in Coq for \( \lambda^[] \), \( \lambda^[]_{NaV} \), and \( \lambda^[]_{\text{throw}} \) (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 ($\lambda[^1]_{Nav}$ and $\lambda[^1]_{throw}$)
  – all errors recoverable, even IFC violations
  – necessary ingredients:
    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