All Your IFCException Are Belong To Us
or
Exception Handling in Breeze

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

Static

Dynamic

Taint Tracking

Coarse-grained

[Denning, 1977] type systems & program analysis, Jif, FlowCaml, ...

only explicit flows
Perl Taint mode, ...

Sound

Fine-grained

also implicit flows
non-interference proofs

[Fenton, 1974] [Sabelfeld & Russo, 2009] [Austin & Flanagan, 2009]

OSes: Asbestos, Flume, HiStar

[Krohn & Tromer, 2009]

[Krohn & Kohler & ..., 2005]
• sound fine-grained dynamic IFC
• label-based discretionary access control
  – clearance
• functional core \((\lambda) + \text{state}(!)) + \text{concurrency} (\Pi)\)
  – from Pict/CML towards something more Erlang-ish
• dynamically typed (for now)
  – directly reflects capabilities of SAFE HW
  – dynamically-checked first-class contracts
Exception handling

- we wanted all Breeze errors to be recoverable
  - including IFC violations!
- however, existing work* assumes errors are fatal
  - makes some things easier ... at the expense of others
  
  +secrecy  +integrity  —availability

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

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 client = send cin (3, 5)@low; recv cout = 5
let attacker = send cin (3, 2@high)@low

3@low <= 2@high = false@high
pc=high

max_server gets killed because of IFC violation!?
Wishful thinking

```plaintext
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
But there is a problem ... in fact two!
Labels are information channels

- well-known fact: labels that change are themselves information channels
- more than one label channel:
  - labels on reference contents (strong updates)
  - vs. labels on values and components of values
- get soundness by preventing secrets from leaking either into or out of label channel

no strong updates or "no-sensitive-upgrade" [Austin & Flanagan, ’09]

labels can be observed
Labels are information channels

• well-known fact:
  – labels that change are themselves information channels

• more than one label channel:
  – labels on reference contents (strong updates)
  – vs. labels on values and components of values

• get soundness by preventing secrets from leaking either into or out of label channel

allow labels to depend on secrets

labels must be hidden
e.g. “permissive upgrades”
[Austin & Flanagan, ’10]
Labels are information channels

• well-known fact:
  – labels that change are themselves information channels
• more than one label channel:
  – labels on reference contents (strong updates)
  – vs. labels on values and components of values
• get soundness by preventing secrets from leaking either into or out of label channel

allow labels to depend on secrets

labels must be hidden e.g. “permissive upgrades” [Austin & Flanagan, ’10]
Problem #1: IFC exceptions make all label channels public

• we disallow strong updates
• still need to close label channel on values
• secret bit: h@high
  \[
  \text{low} \ll \text{high} \ll \text{top}
  \]
• let \( h_{\text{ref}} = \text{ref high ()} \) in

\[
\begin{align*}
\text{try} & \quad \text{encode h into label} \\
\text{href} := \begin{cases} 
\text{if h then } ()@\text{high} \\
\text{else } ()@\text{top}
\end{cases}; \\
\text{true}
\end{align*}
\]
\[
\text{catch IFCException} \Rightarrow \text{false}
\]

Automatic pc restoring just doesn’t work!
Solution to problem #1: brackets

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

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

• \textbf{labels are now 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 else ()] in false
catch Ex => true
    ```

• failed brackets cannot raise exceptions
  – let lref = ref low false in
    try
      let _ = high[if h then ()@high else ()@top] in
      lref := true
catch EBrk => ()

• brackets need to delay all exceptions!
Solution #2: Delayed exceptions

• delayed exceptions unavoidable
  – still have a choice how to propagate them

• we study two alternatives for error handling:
  1. mix active and delayed exceptions \( (\lambda[^1] \text{throw}) \)
  2. only delayed exceptions \( (\lambda[^1] \text{NaV}) \)

• 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
**NaV-lax vs. NaV-strict behavior**

• all non-parametric operations are NaV-strict
  – $\text{NaV}@\text{low} + 42@\text{high} \Rightarrow \text{NaV}@\text{high}$

• 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>$(\text{fun } x =&gt; 42) \text{ NaV} \Rightarrow 42$</td>
<td>or</td>
<td>$\Rightarrow \text{ NaV}$</td>
</tr>
<tr>
<td>$\text{Cons NaV Nil} \Rightarrow \text{Cons NaV Nil}$</td>
<td>or</td>
<td>$\Rightarrow \text{ NaV}$</td>
</tr>
<tr>
<td>$(r := \text{NaV}, r=7) \Rightarrow ((), r=\text{NaV})$</td>
<td>or</td>
<td>$\Rightarrow (\text{NaV}, r=7)$</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 %2”, 42@high, 0@low)
  – high clearance code can obtain:
    “EDivisionByZero: can’t divide 42@high by 0@low”@high
  – all code can obtain:
    “EDivisionByZero: can’t divide <hidden>@high by 0@low”@low

• 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 $\lambda^{[]}_{\text{throw}}$ (termination-insensitive)
  – for $\lambda^{[]}_{NaV}$ even with all debugging aids

• **conjecture**: 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 ($\lambda^{[\cdot]}_{NaV}$ and $\lambda^{[\cdot]}_{throw}$)
  – all errors recoverable, even IFC violations
  – necessary ingredients:
    - public labels (via 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
INTEGRITY
Integrity

Sound taint tracking

Endorsement

default label = bottom (trusted)
(same as for secrecy)
you know “who” is untrusted and trust everyone else

IFC-endorsement

pc starts at integrity top
stronger guarantees
(by “being paranoid”)
coarser-grained

Signatures

not tracking implicit/explicit flows
simple and natural model
not IFC!
finer-grained
Q: Should A’s endorsement be preserved?
A1: No! (IFC-endorsement)
A2: Yes! (Signatures)
Signature labels

• Very much like digital signatures
  – P’s signing authority
  – P’s name
  – P’s signing key
  – P’s public verification key

• Lattice structure useful
  – conjunctive labels [[P],[Q]]
  – disjunctive labels [[P,Q]]
  – multi-signatures
  – group signatures

• Unforgeable
  – New atoms start out “unsigned” (integrity top)
  – Just passing around atoms preserves signatures
Data abstraction by signing

---

**Bool module**
- private principal B
- true
- false
- not

**secrecy part:**
only Bool can access this data (enforced by clearance, not IFC)

**integrity part:**
only Bool can create this signature

1@([[B]]; [[B]])
Data abstraction by signing

More flexible than dynamic sealing:
- no extra boxing;
- secrecy separate from integrity;
- multiple signers and “decrypters”

Bool module
- private principal B
- true
- false
- not

A (trusts B to access his data, but not to declassify it)
more restrictive

- discretionary access control (clearance)
- IFC
- secrecy

more restrictive

- public key encryption
- homomorphic encryption
- IFC
- digital signatures
- endorsement integrity
- signature labels

- more restrictive

- more restrictive
Signature labels are no silver bullet
Signature labels are no silver bullet

A \rightarrow B \rightarrow C

(42°, 10am, 2012-10-15)@A

(42°, 10am, 2012-10-15)@A
Signature labels are no silver bullet

Only sign “self-contained” (+immutable) messages
Signature labels are no silver bullet

Only sign “self-contained” (+immutable) messages
Signature labels are no silver bullet

Only sign “self-contained” (+immutable) messages
Signatures alone don’t guarantee freshness
- Linear/unique signatures? (could work in a closed system)
Rules $((\lambda[^{\cdot}])_{NaV})$

Boxes and atoms

\[
\begin{align*}
  b & ::= v \ | \ \delta \hspace{1em} \text{except} \\
  a & ::= b@L
\end{align*}
\]

\[
\frac{\rho(x) = a}{\rho \vdash x, pc \downarrow a, pc}
\]

\[
\frac{\rho \vdash (\lambda x.t), pc \downarrow (\rho, \lambda x.t)@\bot, pc}{\rho \vdash (\lambda x.t), pc \downarrow (\rho, \lambda x.t)@\bot, pc}
\]

\[
\frac{\rho(x_1) = (\rho', \lambda x. t)@L \hspace{1em} \rho(x_2) = a \hspace{1em} (\rho', x \mapsto a) \vdash t, (pc \vee L) \downarrow a', pc'}{\rho \vdash (x_1 \ x_2), pc \downarrow a', pc'}
\]

\[
\frac{\rho(x) = v@L}{\rho \vdash \text{labelOf } x, pc \downarrow L@\bot, pc}
\]
Rules $\left(\lambda^{[1]}_{Nav}\right)$

\[
\rho(x) = b@L' \quad \text{tagOf } b \neq \text{T Lab}
\]

\[
\rho \vdash x[t], pc \downarrow (\delta (prEx b))@\perp, (pc \lor L')
\]

\[
\rho(x) = L@L' \quad \rho \vdash t, (pc \lor L') \downarrow b@L'', pc'
\]

\[
L'' \lor pc' \subseteq L \lor (pc \lor L')
\]

\[
\rho \vdash x[t], pc \downarrow b@L, (pc \lor L)
\]

\[
\rho(x) = L@L' \quad \rho \vdash t, (pc \lor L') \downarrow b@L'', pc'
\]

\[
L'' \lor pc' \not\subseteq L \lor (pc \lor L')
\]

\[
\rho \vdash x[t], pc \downarrow (\delta \text{EBrk})@L, (pc \lor L')
\]

\[
prEx (\delta \text{excp}) = \delta \text{excp}
\]

\[
prEx _\_ = \text{ETYPE}
\]