The SLam Calculus

Nevin Heintze and Jon Riecke

Presented by: Sam Tobin-Hochstadt

CSG 399    April 27, 2006
(print-to low-security-port
  (if (read high-security)
      #t #f))
BAD!

(print-to low-security-port
 (if (read high-security)
    #t #f))
(print-to low-security-port
  (if (read high-security(H,H))
    #t(L,L) #f(L,L)))
Rejected by typechecker

\((\text{print-to low-security-port}
  \ (\text{if} \ (\text{read high-security}_{(H,H)})
  \ #t_{(L,L)} \ #f_{(L,L)}))\)
Fundamental Idea:
Fundamental Idea:
Add security properties to types
Security properties:
Who can read this data
Who can be influenced by this data
(r, \textit{ir})
r: readers
ir: indirect readers
readers can directly inspect an object
indirect readers can be given acess to some information about an object
Security Levels
We will just consider two
More complex: unix users and groups, roles, etc
The Lambda Calculus
Functions: \((\text{lambda } (x) \ e)\)
Variables: x
Function Application: \((e_1 \ e_2)\)
That's it!
But not quite ...
Pairs: \((\text{cons} \; e_1 \; e_2)\)
Projection: \((\text{proj}_i e)\)
Recursive Functions: $(\text{rec } f \ e)$
Case: \((\text{case } e \text{ of } (\text{inj}_1 e_1) \mid (\text{inj}_2 e_2))\)
Now we add types ...
Where do types go?
Functions: \((\text{lambda } (x : s) \ e)\)
Recursive Functions: \((\text{rec } f : s e)\)
What are types?
Unit: \( t = () \)
Sum Type: \( s + s \)
Product Type: \( (s * s) \)
Function Type: \( \text{\( s \rightarrow s \)} \)
But what was s?
Security Properties
Security properties: $k \equiv (r, ir)$
Security properties: $k = (r,ir)$

Types: $s = (t,k)$
Now we use these in the grammar
Basic values:

\[ \textbf{bv} = () \mid (\text{inj}_i \mathbf{v}) \mid (\text{cons} \mathbf{v} \mathbf{v}) \mid (\text{lambda} (x : s) \mathbf{e}) \]
Labeled Values: \( v = bv_{(r, ir)} \)
Labeled Constructors: \((\text{inj}_i \ e)_{(r, ir)}\)
Labeled Destructors: $\text{(e e)}_r$
One More Construct
(\text{protect}_{ir} \ e)
(\texttt{protect}_{ir} \ e)

Increases the security level on e
How Do We Check?
Constructors:
\((\text{cons } e_1 e_2)_k : (s_1 * s_2, k)\)
\((\text{cons } e_1 e_2)_k : (s_1 * s_2 , k)\)

provided \(e_1 : s_1\) and \(e_2 : s_2\)
\((\text{inj}_i \ e)_k : (s_1 + s_2 , k)\)
\[(\text{inj}_i \ e)_k : (s_1 + s_2 , k)\]

provided \(e : s_i\)
Destructors:
\((e_1 \ e_2)_{r^*} : s_2 \cdot ir\)
\[(e_1 e_2)_{r^*} : s_2 \cdot ir\]

provided \(e_1 : (s_1 \rightarrow s_2), (r, ir)\) and \(e_2 : s_1\)
\[(e_1 \ e_2)_{r^*} : s_2 \cdot ir\]

provided \(e_1 : (s_1 \rightarrow s_2), (r, ir)\) and \(e_2 : s_1\)

and \(r \leq r^*\)
Wait, what was that •
s · ir
\( s \cdot ir \)

means increase the security of \( s \) to \( ir \)
So checking a destructor involves 3 things:
So checking a destructor involves 3 things:

That the basic types match
So checking a destructor involves 3 things:

That the basic types match

That the reader has sufficient access to read the value
So checking a destructor involves 3 things:

That the basic types match

That the reader has sufficient access to read the value

That the result has the appropriate indirect security
And that's it!
We can now prove non-interference:
We can now prove non-interference:

That is, the result of a (low-security) program does not depend on its high-security portions
Also, we can prove an erasure property:
Also, we can prove an erasure property:

We don't need to do any security checking at runtime
Some Weaknesses
Nontermination - not considered
(let ([halt-if-true
      (lambda (x : bool_{H,H})
        (if x ()_{H,H}
          (halt-if-true x)))]))
(halt-if-true secret-bool)
#t_{L,L}
Timing - not considered
(let* ([t1 : int_{L,L} (get-time)]
    [tmp (if secret-bool
          (long-comp)
          (short-comp))]
    [t2 : int_{L,L} (get-time)])
  (> (- t2 t1) time-for-short-comp))
Extending the system
Mutation
and Concurrency
and Bears, Oh My!
Everything is harder with concurrency
Add reference cells, and spawn to the language
(box e)
(set-box! e₁ e₂)
(unbox e)
(spawnir e)
Add reference types
(ref s)
Add an effect system to the type system
Add latent effects to function types
Basically, what will happen when this function runs
Also take into account the context of actions
$$\text{(set-box! } e_1 e_2) : s$$

in context ir
(set-box! e₁ e₂) : s

in context ir

if e₁ : (ref s) and e₂ : s
(set-box! e₁ e₂) : s

in context ir

if e₁ : (ref s) and e₂ : s

and s • ir = s
(let ([halt-if-true
    (lambda (x : bool (H,H))
      (if x () (H,H)
        (halt-if-true x)))]
  (halt-if-true secret-bool)
  (set-box! y #t (L,L)))
(let ([halt-if-true
        (lambda (x : bool_{H,H})
            (if x ()_{H,H}
                (halt-if-true x)))])
    (halt-if-true secret-bool)
    (set-box! y #t_{L,L}))

Only high-security readers can read y
Non-termination still not fixed
No non-interference result
What is non-interference for parallel systems?
"Timing attacks" are part of the point
Integrity checking
Adds creators and indirect creators
So now security properties are a 4-tuple
I'll spare you the details
But this system obeys non-interference
Conclusions
We can apply standard PL techniques to design a secure language
And to prove theorems about it
Questions?