A Library Approach to Information Flow Security in Haskell

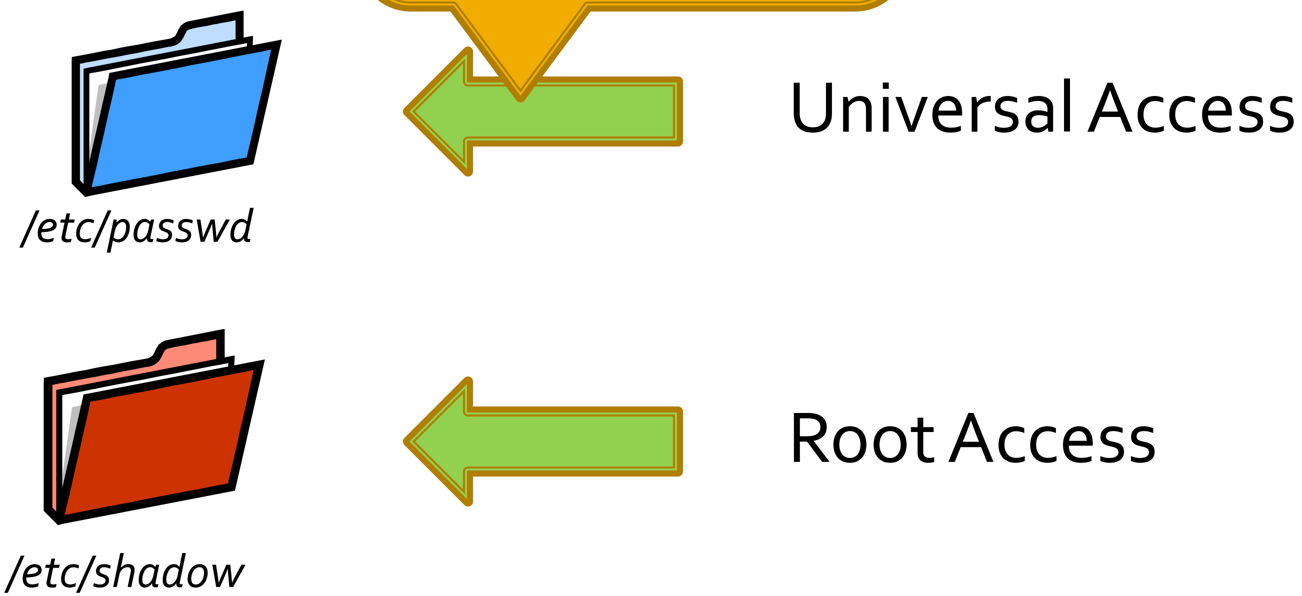
Koen Lindström Claessen,
Alejandro Russo, John Hughes
Chalmers University of Technology

WG2.8, Park City, Utah,
June 2008

Motivating Example

- Passwords ms

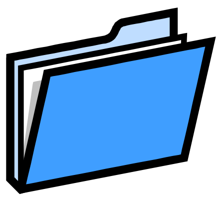
Dictionary attacks, offline attacks, …



Motivating Example

- Passwords ms

Dictionary
attac
atta



/etc/passwd

/etc/shadow

Linux Shadow Password HOWTO: Adding shadow support to a C program

"Adding shadow support to a program is actually fairly straightforward. The only problem is that the program must be run by root in order for the the program to be able to **access** the /etc/shadow file."

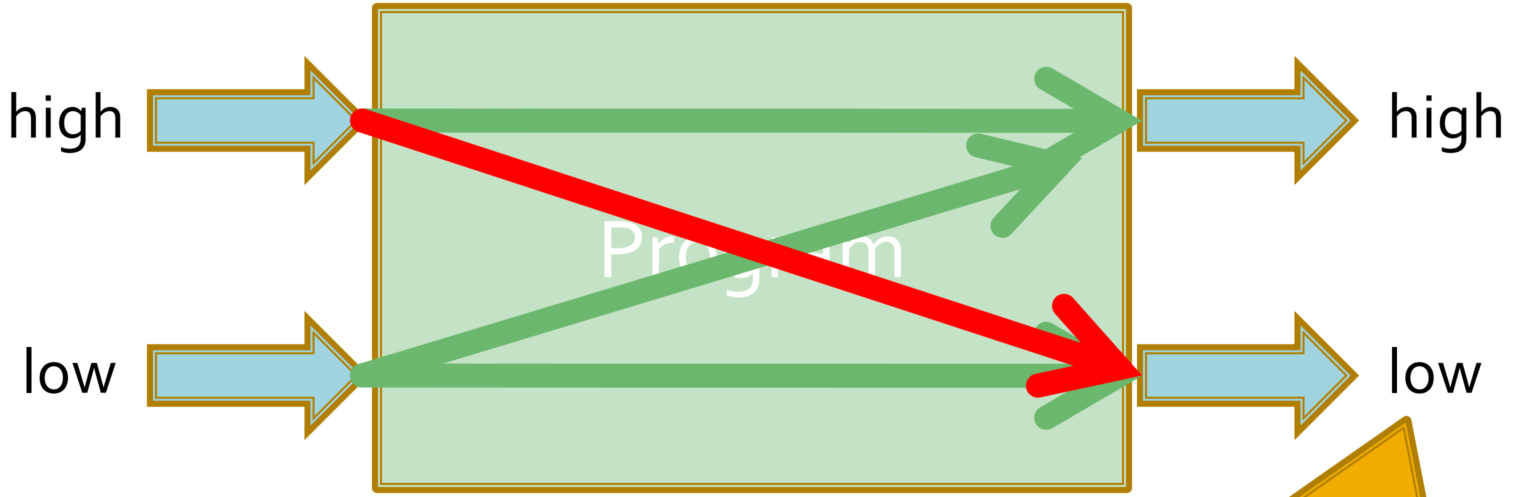
The Problem

- For the sake of
  - Intruders
  - People we let in (plug-ins)
  - Ourselves
- We want to restrict
  - Access to data
  - Where does data go?
  - Where is it used?

Confidentiality (aot integrity)

"Information-flow security"

The Model



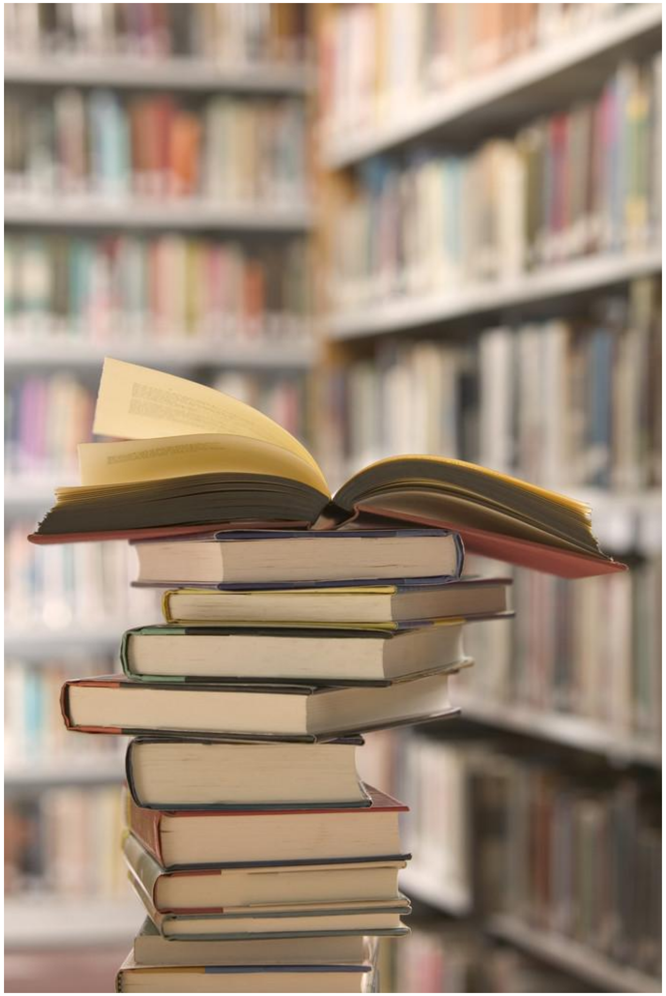
Non-interference: *Varying high inputs should not affect low inputs*

“Attacker”

- Attacker
  - Not trusted
  - Intruder
  - Programmer
  - Yourself
- Everyone (including the attacker) can observe low security outputs

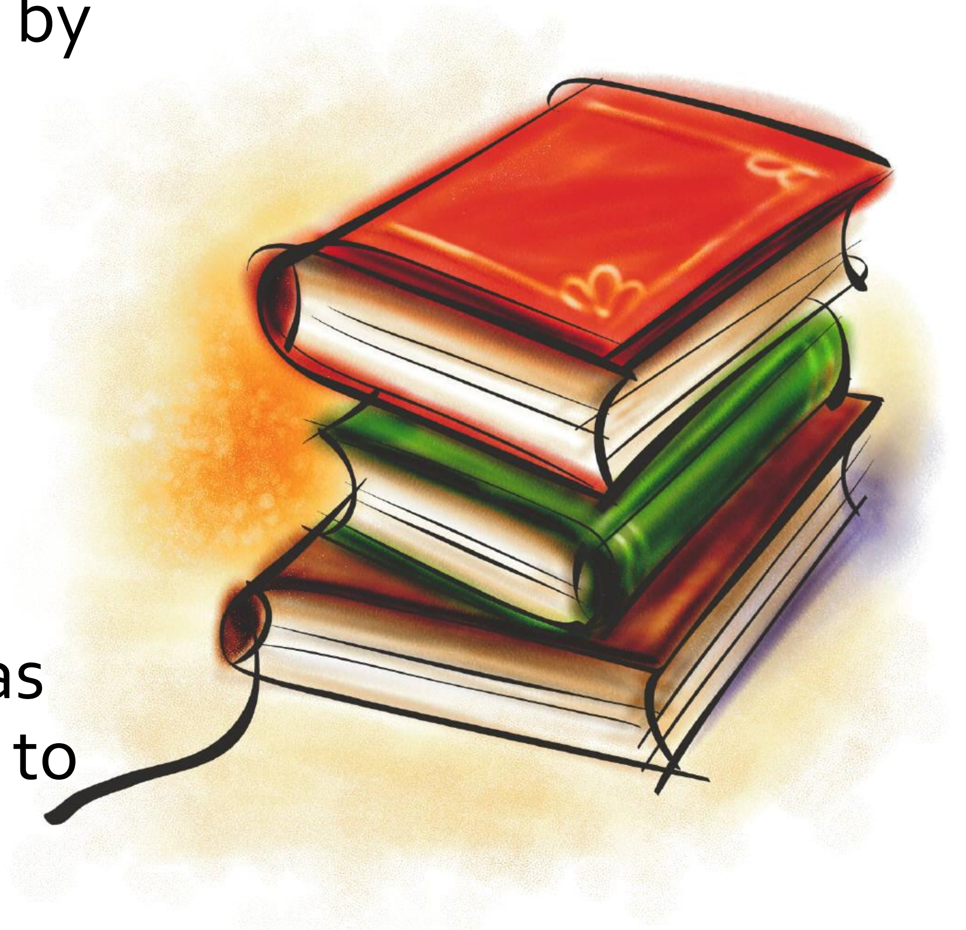
Information-Flow Security

- Study for ~30 years
- Active research field
- Compilers
  - JIF (Java) 2001
    - Cornell University
  - FlowCaml (ML) 2002
    - INRIA (not actively developed)
- Impact on practice
  - Limited!



Encoding IF in Haskell

- Possible to guarantee IF by a library
  - [Zdancewic & Li, 06]
  - Haskell
  - Arrows
- No need to write a compiler from scratch
- DSEL approach: Quick experimenting with ideas
- No restriction on the PL to use due to security



Encoding IF in Haskell

- Limitations
  - No side effects
- Extension to the library [Tsai, Russo, Hughes'07]
  - Major changes in the implementation of the library
  - New arrows combinators
  - Lack of arrow notation
- Why arrows?
  - Zdancewic and Li mention that monads are **not suitable** for the design of the library

Our Approach

- Light-weight
- Library-based
- Monad-based (not arrows)
- Restrict capabilities
  - Abstract types
  - Use of the module system
- Practical (?)

Why Haskell?

- Pure language
  - No side effects
  - (Controlled side effects)
- Strong type system
  - Cannot ”cheat”
- No implicit information flow!
  - Only explicit

```
if secret == 3 then
   print(1)
else
   print(2)
```

Example

```
f :: (Int {-secret-}, Char)
  -> (Int {-secret-}, Char)

f (n, c) = (n + 1, chr (ord c + 1))
```

YES

```
f (n, c) = (n + ord c, ’a’)
```

YES

```
f (n, c) = (n + ord c, chr n)
```

NO

```
f (n, c) | n > 0     = (42, c)
         | otherwise = (1, chr (ord c + 1))
```

NO

Simple Security API

```
type Sec a -- abstract

sec  :: a -> Sec a
open :: Sec a -> Key -> a


data Key = TheKey -- hidden


instance Functor Sec
instance Monad Sec
```

strict!

Non-Interference?

```
type A
type B
type C
type D

f :: (Sec A, B) -> (Sec C, D)
```

```
f (a1,b) = (c,d)  =>  f (a2,b) = (c',d)
```

Multiple Security Levels

```
type Sec s a -- abstract

sec  :: a -> Sec s a

open :: Sec s a -> s -> a
```

Security Lattice

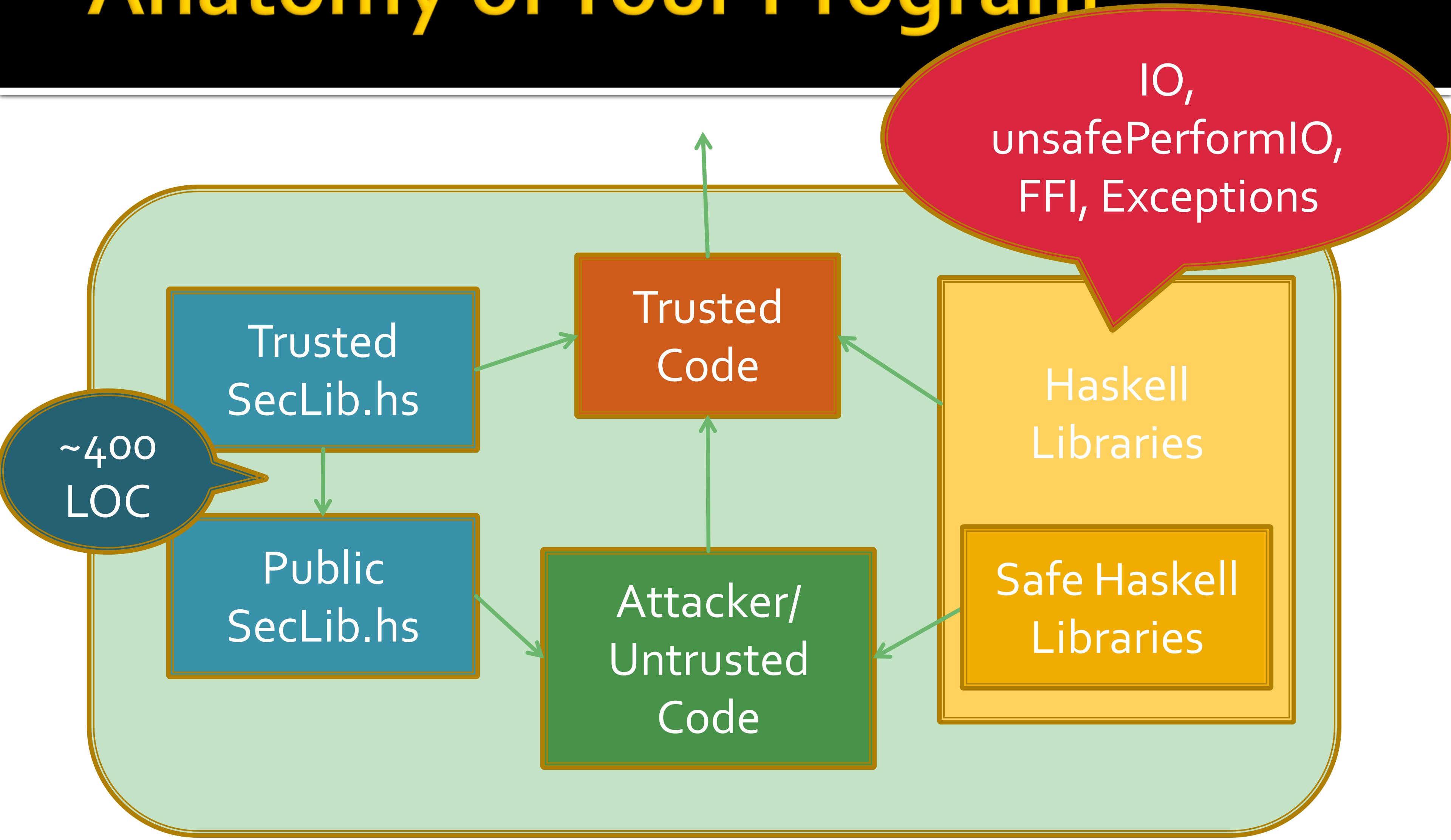
```
data H = H -- abstract
data L = L -- public

class Less low high where
  up :: Sec low a -> Sec high a

instance Less L H
instance Less L L
instance Less H H
```

$\boxed{\texttt{Sec L a} \quad \sim= \quad \texttt{a}}$

Anatomy of Your Program



How About IO?

- IO features
  - File IO
  - stdin/stdout
  - State references
  - Channels
  - ...
- This talk: Only File IO

First Try

```
type File s -- abstract

readFileSec  :: File s -> IO (Sec s String)

writeFileSec :: File s -> Sec s String -> IO ()
```

High Control / Implicit Flow

- "Depending on a high value, write to file1 or file2"
- Leads to result types
  - IO (Sec H a)
  - Sec H (IO (Sec H a))
  - IO (Sec H (IO (Sec H a)))
  - ...
- Need a new type for "secure IO"

SecIO

* Read from level s or lower
* Write to level s or higher
* Produce a value at level s

```
type SecIO s a -- abstract

peek         :: Sec s a -> SecIO s a
readFileSec  :: File s -> SecIO s String
writeFileSec :: File s -> String -> SecIO s ()

run          :: SecIO s a -> IO (Sec s a)
```

Side effects escape "Sec s"!

SecIO

```
example :: Sec H Int -> SecIO s ()
example secret =
  do x <- peek secret
     if x == 42
       then writeFileSec file1 ”foo”
       else writeFileSec file2 ”bar”
```

Anatomy of Your Program

```
shadow :: File H
passwd :: File L

main = … Untrusted.main shadow passwd …
```

```
main :: File H -> File L -> IO (Sec H Answer)
main shadow passwd = run (…)
```

File Capabilities

```
type File m s -- abstract

data R
data W

readFileSec  :: File R s -> SecIO s String
writeFileSec :: File W s -> String -> SecIO s ()
```

```
passwd   :: File R L
shadow   :: File R H
database :: File m H -- polymorphic
```

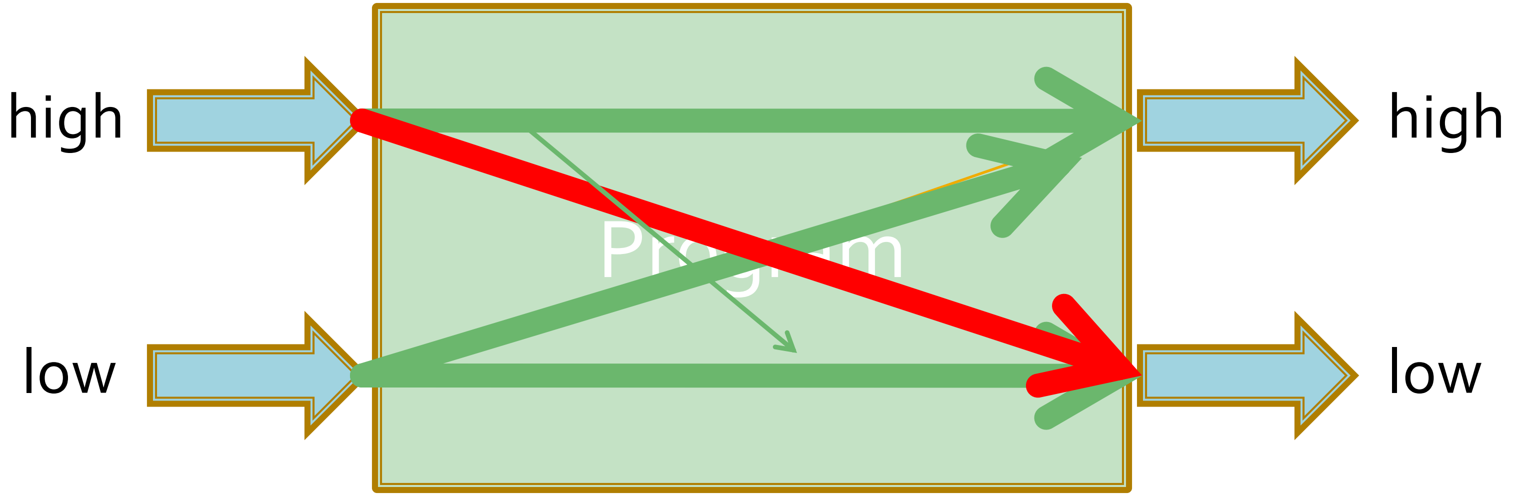
Information Flow in Practice

- Login program
  - Get password from user input
  - Check if it is correct (compare with shadow)
  - Act accordingly
- It is necessary to leak information that depends on secrets!
  - cypher inp == pwd
  - Not non-interferent

Declassification

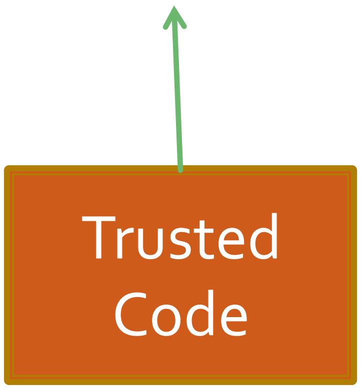
- Dimensions and principles of declassificaiton [Sabelfeld and Sands, 06]
  - **What** information can be leaked?
  - **When** can information be leaked?
  - **Where** in the program is it safe to leak information?
  - **Who** can leak information?
- How to be certain that our programs leak what they are supposed to leak?

The Adapted Model



Introducing Declassification in the Library

- Our library should be able to handle different kind of *declassificaiton policies*
- Policies are programs!
  - Trusted users of the library implement them
  - Controlled at run-time
- A module defines combinators for different declassification policies (**what**, **when**, **who**)



Escape Hatches

- Declassification is performed by functions
  - Terminology: **escape hatches** [Sabelfeld and Myers, 2004]
- In our library:

```
type Hatch sH sL a b = Sec sH a -> Sec sL b

hatch :: (a -> b) -> Hatch sH sL a b -- hidden
```

- Example: checking password

monomorphic

```
check :: Hatch H L (String,Passwd) Bool
check = hatch (\(inp,pwd) -> cypher inp == pwd)
```

Escape Hatches

- We want to restrict capabilities of escape hatches

```
type Hatch sH sL a b =
  Sec sH a -> IO (Maybe (Sec sL b))
```

internal state

may fail

Declassification Combinators

```
-- restricting "what" (how often)
nTimes :: Int -> Hatch sH sL a b ->
                             IO (Hatch sH sL a b)

-- example
check =
  nTimes 3
    (hatch (\(inp,pwd) -> cypher inp == pwd))
```

Implementation

```
-- restricting "what" (how often)
nTimes :: Int -> Hatch sH sL a b ->
                          IO (Hatch sH sL a b)
nTimes n hatch =
  do ref <- newIORef n
     return (\x ->
       do k <- readIORef ref
          if k >= 0
             then do writeIORef ref (k-1)
                     hatch x
             else do return Nothing)
```

Declassification Combinators

```
-- restricting "when" (flow locks)
data Open  = Open (IO ())   -- hidden
data Close = Close (IO ()) -- hidden

when :: Hatch sH sL a b ->
             IO (Hatch sH sL a b, Open, Close)
```

Declassification Combinators

```
-- restricting "who" (flow locks)
data Authority s = Auth Open Close -- hidden

who :: Hatch sH sL a b ->
             IO (Hatch sH sL a b, Authority sH)

-- for use by attacker
certify :: s -> Authority s -> IO a -> IO a
```

Declassification Combinators

- Powerful
- Expressive

- Theory of declassification is in its infancy
  - One dimension only
  - Weak results
- In practice, we want to combine things
  - Pragmatic

Correctness Proof

- "Sec" -- obvious and trivial
- All other things
  - SecIO
  - Files
  - References
  - ...
- On top of Sec: also obvious
- With slight modification: small proof

To do

To do

QuickChecking Correctness

- Modelled library + language as a Haskell datatype
  - Evaluate function
- Written a random generator
  - Respecting types
- Expressed non-interference as a QuickCheck property
  - Counter-examples for unsound versions of the library

Conclusions

- Light-weight library (~400 LOC)
- Practical
  - Simple (Monads)
  - Features: files, stdio/stdout, references
  - Declassification
  - Examples: login system, bidding,banking system prototype,…
- Limitations
  - Timing leaks
  - Static security lattice