Secure Programming via Libraries

Secure Multi-Execution in Haskell

Alejandro Russo (russo@chalmers.se)

Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

CHALMERS

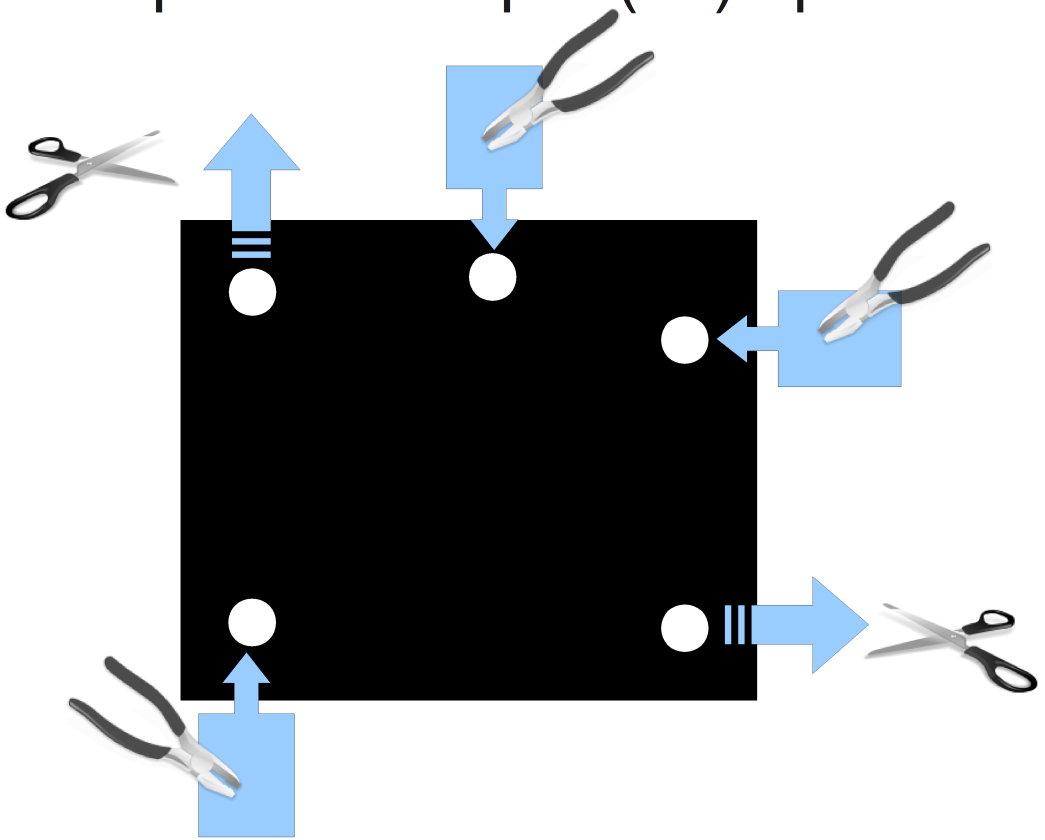
Enforcement for non-interference

- It is usually given as
  - Type-system
    [Volpano Smith Irnive 96]
  - Monitor
    [Volpano 99][Le Guernic et al. 06]
- Monitors are more permissive than traditional type-systems
  [Sabelfeld, Russo 09]
- Inspection of the code is necessary

Secure Multi-Execution

[Devriese, Piessens 10]

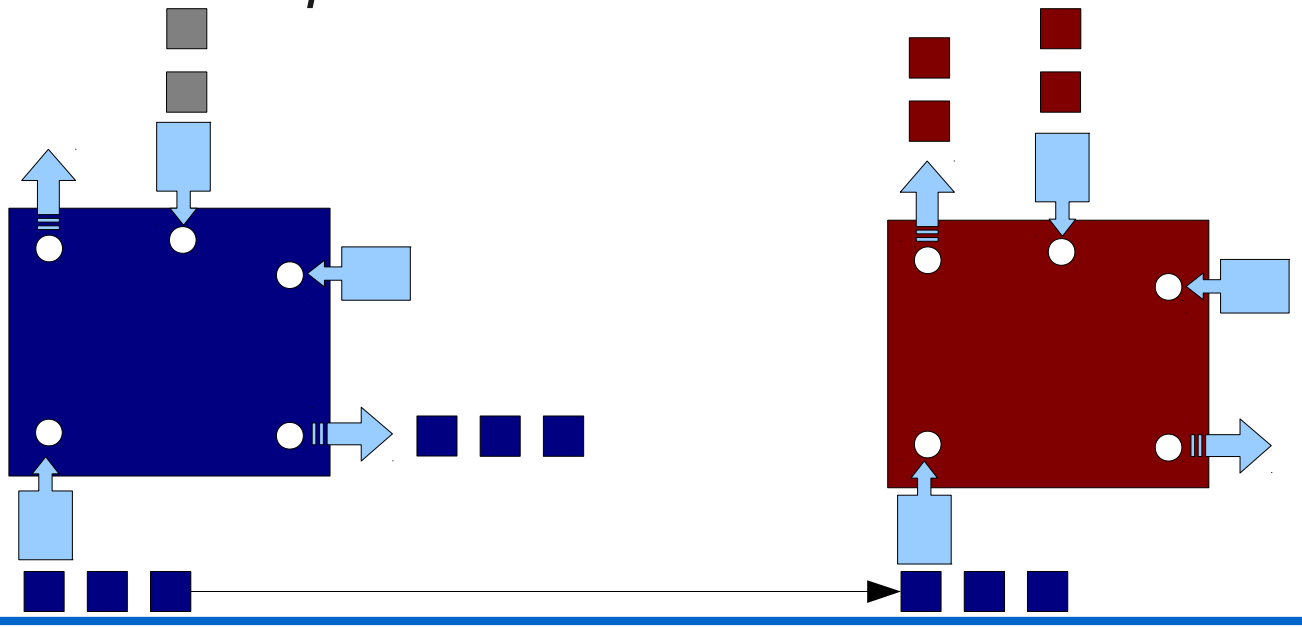
- Black box approach to enforce non-interference
  - No need to inspect the code
  - Manipulate input and output (IO) operations



Secure Multi-Execution

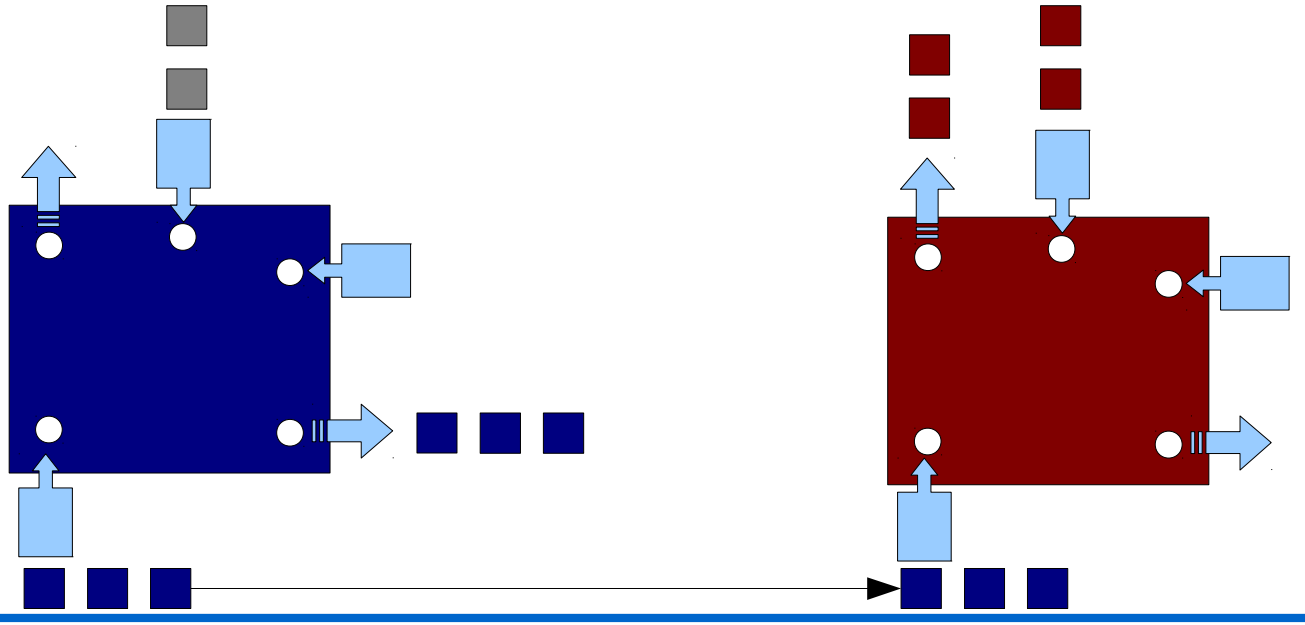
[Devriese, Piessens 10]

- Execute the program once for each security level.
- *Outputs are only produced in the execution linked to their security level*
- *Inputs are replaced by default inputs in executions linked to security levels lower than the security level of the input*
- *The high execution reuses inputs obtained in the low execution*



Guarantees?

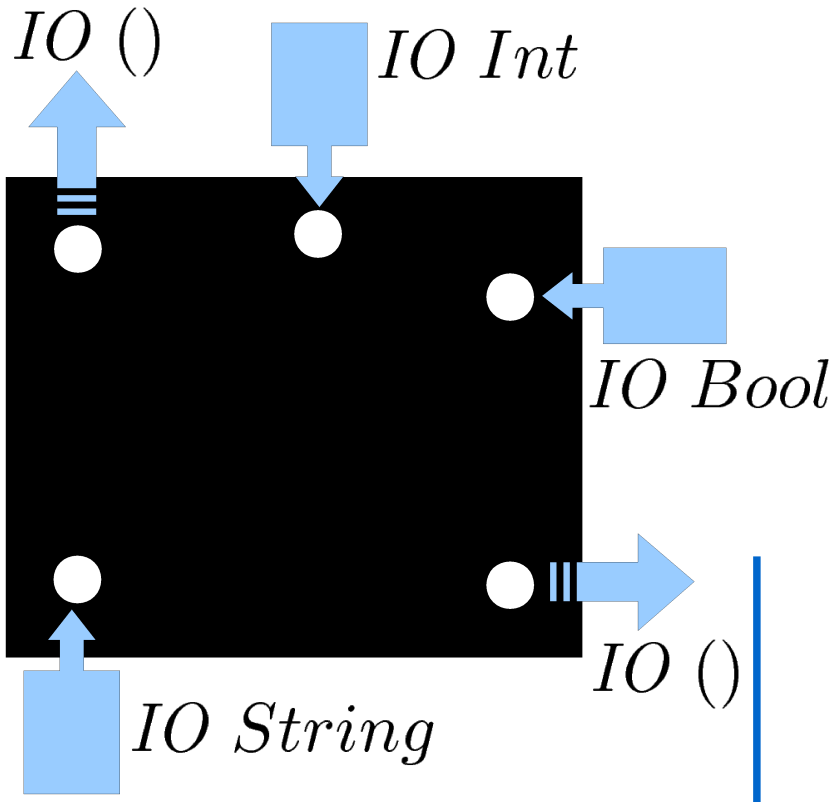
- Executed program satisfies non-interference
  - No explicit and implicit flows
- The secure multi-execution produces the same results
- Otherwise, the semantics changes to preserve security



Secure Multi-Execution in Haskell

[Jaskelioff, Russo 11]

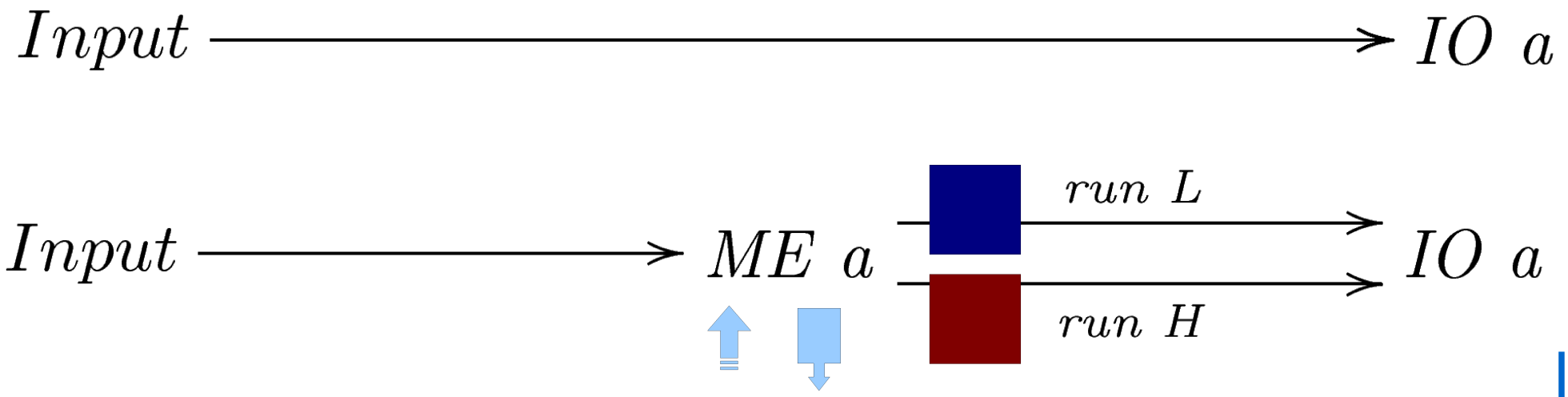
- Clear separation of pure computations with those with side-effects
- Every computation with side-effects is encapsulated into the monad *IO*
- Identify where IO is performed



Secure Multi-Execution in Haskell

[Jaskelioff, Russo 11]

- For simplicity, consider IO operations on files
- Reading produces a visible side-effect for the attacker
  - Actualization of access time



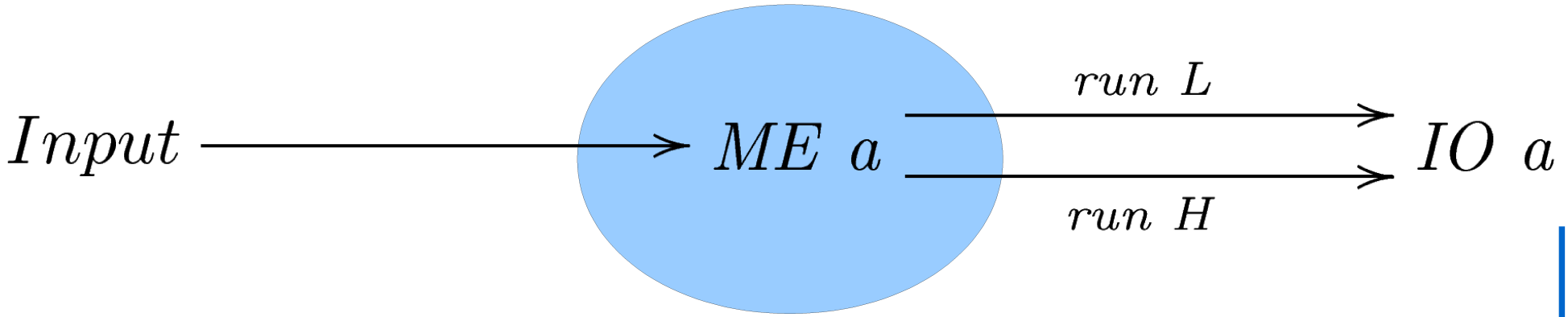
Monad ME

- It models the IO operations in a pure manner [Swierstra,Altenkirch 06]

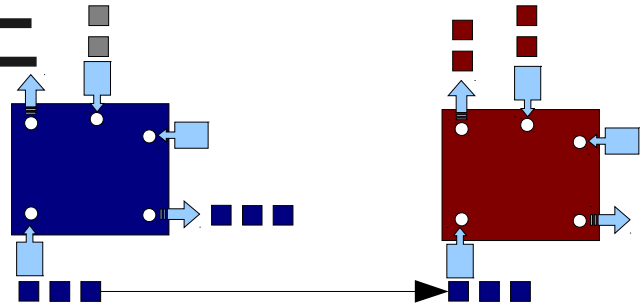
```
data ME a    =  Return a
             |  Write FilePath String (ME a)
             |  Read FilePath (String -> ME a)


writeFile          :: FilePath -> String -> ME ()
writeFile file s = Write file s (return ())

readFile           :: FilePath -> ME String
readFile file      = Read file return
```

Monad ME

```
data ME a    =  Return a
             |  Write FilePath String (ME a)
             |  Read FilePath (String -> ME a)

instance Monad ME where
    return x                    = Return x
    (Return x)          >>= f   = f x
    (Write file s p)    >>= f   = Write file s (p >>= f)
    (Read file g)       >>= f   = Read file (\i -> g i >>= f)
```



Interpreter for ME



```
run :: Level -> ChanMatrix -> ME a -> IO a
run l _ (Return a)                 = return a
run l c (Write file o t)
    | level file == l              = do  IO.writeFile file o
                                         run l c t
    | otherwise                    = run l c t
run l c (Read file f)
    | level file ==  l             = do  x <- IO.readFile file
                                         broadcast c l file x
                                         run l c (f x)
   | sless (level file) l          = do  x <- reuseInput c l file
                                         run l c (f x)
   | otherwise                     = run l c (f (defvalue file))

defvalue :: FilePath -> String
```

$$Input \longrightarrow \boxed{ME\ a \xrightarrow[run\ H]{run\ L} IO\ a}$$

Example Scenario

- Credit terms $discount/discount\ period\ \mathbf{net}/credit\ period$
- Invoice 1000 can have the term $2/10\ \mathbf{net}/30$
  - 2% discount if you cancel the credit before 10 days
  - The total credit should be paid in 30 days
- **Financial company** wants to compute
  - Total interest paid by the customer
    - $loan - loan \times (1 - discount/100)$
    - $\$1000 - \$1000 \times (1 - .2) = \$20$
  - Cost of credit
    - $\frac{discount}{100 - discount} \times \frac{360}{credit\ period - discount\ period}$
    - $\frac{2}{98} \times \frac{360}{20} = .3673$

Example Scenario

- The financial company wants to preserve the confidentiality of their clients
  - *Amount of every loan is secret*
- The cost of credit is public information
  - It can be used for statistics
- Implement a calculator that computes the interested obtained as well as the costs of credit
  - Be sure that confidentiality is preserved

Security Policy

```
level :: FilePath -> Level
level "Client"              = H
level "Client-Terms"        = L
level "Client-Interest"     = H
level "Client-Statistics" = L
level file                  = error $ "File " ++ file ++
                                      " has no security level"


defvalue :: FilePath -> String
defvalue "Client"           = "0 % 1"
defvalue "Client-Interest" = "0 % 1"
defvalue f                  = error "No default value for " ++ f
```

Example: Code

```
data CreditTerms = CT { discount :: Rational,
                        ddays    :: Rational,
                        net      :: Rational }
                  deriving Read

calculator :: ME ()
calculator =
    do  loanStr    <- readFile "Client"
        termsStr   <- readFile "Client-Terms"
        let  loan       = read loanStr
             terms      = read termsStr
             interest   = loan - loan * (1 - discount terms / 100)
             disct      = discount terms / (100 - discount terms)
             ccost      = disct * 360/(net terms - ddays terms)
        writeFile "Client-Interest" (show interest)
        writeFile "Client-Statistics" (show ccost)
```

- It looks like if it was implemented using IO
  - However, it uses the monad ME
- Does it work?

Example: Malicious Code

```
data CreditTerms = CT { discount :: Rational,
                        ddays    :: Rational,
                        net      :: Rational }
                  deriving Read

calculator :: ME ()
calculator =
    do  loanStr    <- readFile "Client"
        termsStr   <- readFile "Client-Terms"
        let  loan       = read loanStr
             terms      = read termsStr
             interest   = loan - loan * (1 - discount terms / 100)
             disct      = discount terms / (100 - discount terms)
             ccost      = disct * 360/(net terms - ddays terms)
        writeFile "Client-Interest" (show interest)
        writeFile "Client-Statistics" (show loan)
```

- Secure Multi-Execution avoids the leak!
- Does it work?

Future Work

- Take Secure Multi-Execution in Haskell to a library
  - Easy map different IO actions into monad ME
  - Not only IO actions related to file operations
    - References
    - Sockets
    - Etc
- Declassification
  - Challenging subject
  - Difficult to enforce without braking the black-box approach
  - Open question

Final Remarks

- The first approach to consider secure multi-execution in Functional Programming
- Core part of Secure Multi-Execution (interpreter) fits in one slide
- Implementation is available on request
  - Approximately 130 lines of code
- Challenges
  - Secure Multi-Execution as a library
  - Declassification