Secure Programming via Libraries

LIO: a monad for dynamically tracking information-flow

Alejandro Russo (russo@chalmers.se)

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

Motivation

- Mass used systems often present dynamic features
  - Facebook
    - Users come and go
    - People make (and get rid of) “friends”
    - New applications are created everyday
  - Android
    - New applications are installed in your phone
    - New features are added with updates

Motivation

- One of the main motivations is **permissiveness**
  - To secure as many programs as possible
- Therefore, we need technology that is able to
  - provide confidentiality and integrity guarantees
  - adapt security policies at run-time
  - express the interest of different parties involved in a computer system

LIO

[Stefan, Russo, Mitchell, Mazieres 11]

- It is a monad that provides:
  - Information-flow control dynamically
    - It is know that dynamic method are more **permissive** [Sabelfeld, Russo 09] but equally secure as traditional static ones
  - Some for of discretionary access control
    - It helps to deal with covert channels
    - Information-flow control is not perfect!
- It is implemented as a library in Haskell
- It has recently accepted for the Haskell Symposium 2011, Tokyo, Japan.

SecIO vs LIO

- They share the concepts about how to use monads in order to provide information-flow security
- SecIO provides information-flow security statically, while LIO does it dynamically
  - LIO is more **permissive** than SecIO
- SecIO is simpler than LIO
  - LIO provides information-flow control and a form of discretionary access control, while SecIO only provides the former
- SecIO provides an specific monad for pure values (Sec), while LIO does not
  - LIO can still manipulate pure values

Tracking information-flow dynamically

- `LIO` can perform side-effects or just compute with pure values
- `LIO` takes ideas from the operating systems into language-based security
- `LIO` protects every value in lexical scope by a single, and mutable, *current label*
  - Part of the state of the `LIO` monad
- It implements a notion of *floating label* for the current label
  - The current label “floats” above the label of the data observed so far

Floating Current Label

Program written using `LIO`

There is a current label at any point of the computation

```
program
    = do xs <- code1
         ys <- code2
         let z = [(e1,e2)| e1 <- xs, e2 <- ys ]
         return z
```

It is low

It is high

We assume that it is initially low

Floating Current Label

Program written using `LIO`

There is a current label at any point of the computation

```
program
   = do xs <- code1
        ys <- code2
        let z = [(e1,e2)| e1 <- xs, e2 <- ys ]
        return z
```

After this line, no public data can be affected (no *write-down*)

It is low

It is high

ys

xs

It continues low

lbl

```
program' =
   do result <- program
      ....
```

It cannot write to public data

Discretionary Access Control

- `LIO` also provides a form of discretionary access control
- `LIO` has a notion of *current clearance*
  - Part of the state of `LIO`
- It imposes an upper bound in the *current floating-label*
- Therefore, it restricts data access and manipulation
  - One manner to deal with covert channels (time, energy consumption, etc)
  - One manner to assure that some confidential data is not copied to be accessed in the future

Clearance

Program written using `LIO`

There is a current clearance at any point of the computation

```
program
    = do xs <- code1
         ys <- code2
         let z = [(e1,e2) | e1 <- xs, e2 <- ys ]
         return z
```

The program finishes its execution here!

It is low

It is high

It is low, i.e. the piece of code cannot access secret data

ys

xs

clr

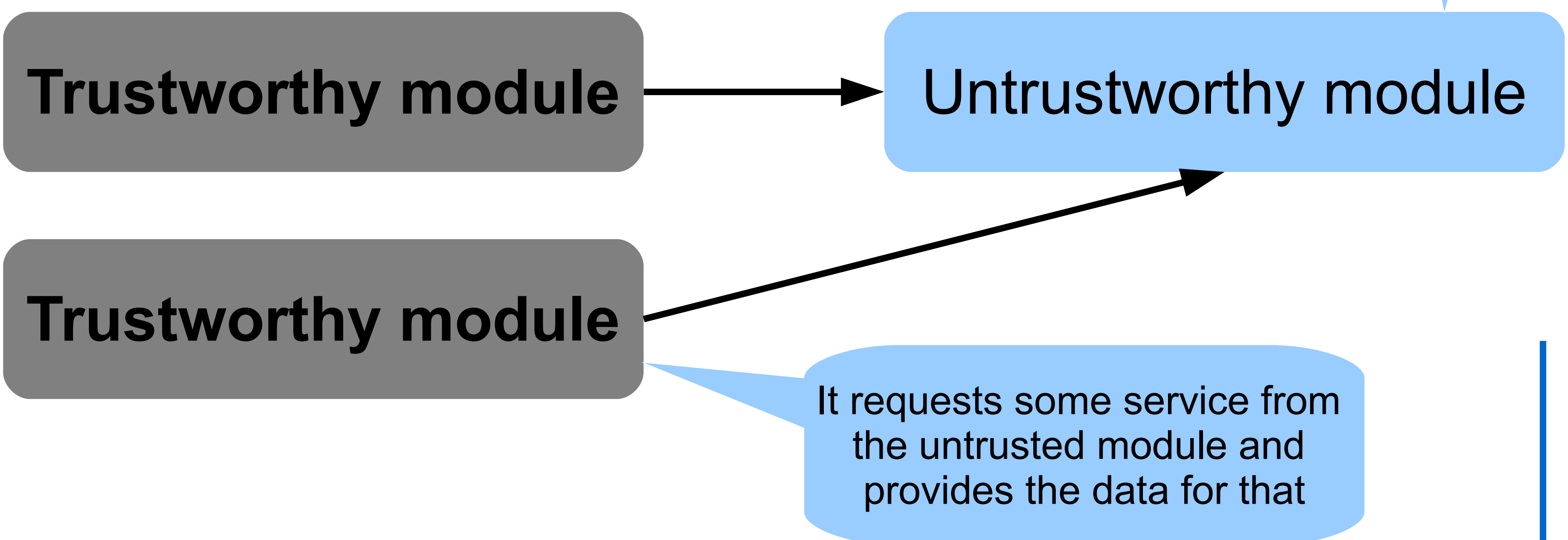
lbl

The label must float above the level `ys`, but `clr` does not allowed

Architecture

- Similar to the one for `SecIO`
- We have trustworthy and untrustworthy modules
- Depending on the type of the module, we import different modules from the library LIO

It export some services that required security policies



API: `label` and `unlabel`

It does not modify the current label and clearance!

We ignore this parameter

```
label :: (Label l) => l -> a -> LIO l s (Labeled l a)
```

- Given a label `l` (**between the current label and the clearance**) and a value of type `a`, it returns a value protected by `l`
- In other words, it assigns the security level described by `l` to the value of type `a`

`lbot` is `bottom` in DCLabels

`ltop` is `top` in DCLabels

Using DCLabels!

```
public :: LIO DCLabel () (Labeled DCLabel String)
public = label lbot "PublicData"

secret :: LIO DCLabel () (Labeled DCLabel String)
secret = label ltop "SecretData"

bob :: LIO DCLabel () (Labeled DCLabel String)
bob = label (newDC ("Alice" .\/. "Bob") "Bob") "BobData"
```

API: `label` and `unlabel`

We ignore this parameter

```
unlabel :: (Label l) => Labeled l a -> LIO l s a
```

- Given a labeled value of type `a` with security level `l`, it returns the value of type `a` and **raises the current label** (clearance permitting) to the join of the current label (`lbl`) and `l`
- Observe that after executing `unlabel`, the value of type `a` can be involved in computations and therefore the current label should float about it!

clr

lbl

:: Labeled DCLabel String
We cannot compute with the string!

We want to compute with the string

```
computation = do l_sec_str <- secret
                 sec_str   <- unlabel l_sec_str
                 return sec_str ++ sec_str
```

sec_str

Example (trustworthy code)

```
module ExampleUnLabelT where

import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB


import ExampleUnLabelU (computation)

public :: LIO DCLabel () (Labeled DCLabel String)
public = label lbot "PublicData"

secret :: LIO DCLabel () (Labeled DCLabel String)
secret = label ltop "SecretData"

execute = do (result, label) <- evalLIO (computation public secret) ()
             putStrLn $ "The result is: " ++ result
             putStrLn $ "With the label: " ++ prettyShow label
```

Only to be imported by trustworthy code!

It imports the service from the untrustworthy code

It provides some data to the service and executes it!

Example (untrustworthy code)

```haskell
module ExampleUnLabelU where


import LIO.DCLabel
import LIO.LIO

computation p s = do l_public_string <- p
                     l_secret_string <- s
                     public_string <- unlabel l_public_string
                     secret_string <- unlabel l_secret_string
                     return $ public_string ++ secret_string
```

To be imported by untrustworthy code!

After this point, any subsequent computation cannot write to public files

API: `toLabeled`

We ignore this parameter

```
toLabeled :: (Label l) => l -> LIO l s a -> LIO l s (Labeled l a)
```

- This primitive avoids creeping of the current label
  - Otherwise, after we read a secret, we cannot do any other computation that involves writing to public data
- It is similar to the primitive `plug` (from `SecIO`)
- Given a label `l` (**between the current label and the clearance**) , and a computation `m`, it executes `m` and returns its result in a value protected by `Labeled` **without raising the current label**
- Computation `m` cannot read data about level `l`

Example (trustworthy code)

```
module ExampleToLabeledT where

import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB


import ExampleToLabeledU (computation')

public :: LIO DCLabel () (Labeled DCLabel String)
public = label lbot "PublicData"

secret :: LIO DCLabel () (Labeled DCLabel String)
secret = label ltop "SecretData"

execute = do (result, label) <- evalLIO (computation' public secret) ()
             putStrLn $ "The result is: " ++ show result
             putStrLn $ "With the label: " ++ prettyShow label
```

The same as before but using a service provided by `computation'`

Remember that this executes `label`

Example (untrustworthy code)

```
module ExampleToLabeledU where


import LIO.DCLabel
import LIO.LIO

computation p s = do l_public_string <- p
                     l_secret_string <- s
                     public_string <- unlabel l_public_stri
                     secret_string <- unlabel l_secret_strin
                     return $ public_string ++ secret_string

computation' p s = do _ <- computation p s
                      l_public_string <- p
                      public_string   <- unlabel l_public_string
                      return public_string
```

At this point, computatoin `p` wants to create a `Labeled` value with label `lbot`.However, it cannot do it due to the current label

clr

lbl

CHALMERS

Example (untrustworthy code)

```
module ExampleToLabeledU where

import LIO.DCLabel
import LIO.LIO

computation p s = do l_public_string <- p
                     l_secret_string <- s
                     public_string <- unlabel l_public_string
                     secret_string <- unlabel l_secret_string
                     return $ public_string ++ secret_string

computation' p s = do _  <- toLabeled ltop (computation p s)
                      l_public_string <- p
                      public_string   <- unlabel l_public_string
                      return public_string
```

The current label is raised when computing `computation` as before

It is not raised when executing `toLabeled`

clr

lbl

API: `labelOf`

```
labelOf :: (Label l) => Labeled l a -> l
```

- It just returns the label of a Labeled value
- The labels are public information in the sense that they can be examined any time

Example (trustworthy code)

It will return 0 if the argument receive is secret and 1 otherwise

```
import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB


import ExampleLabelOfU (computation)

public :: LIO DCLabel () (Labeled DCLabel String)
public = label lbot "PublicData"

secret :: LIO DCLabel () (Labeled DCLabel String)
secret = label ltop "SecretData"

execute = do (result, label) <- evalLIO (computation secret) ()
             putStrLn $ "The result is: " ++ show result
             putStrLn $ "With the label: " ++ prettyShow label
```

Example (untrustworthy code)

```
module ExampleLabelOfU where

import LIO.DCLabel
import LIO.LIO

computation c  = do labeled <- c
                    l <- return $ labelOf labeled
                    if l == lbot then return 1
                                 else return 0
```

API: References

We ignore this parameter

```
newLIORef :: (Label l) => l -> a -> LIO l s (LIORef l a)
```

- Given a label `l` (**between the current label and the clearance**) , it creates a reference to a value of type `a` protected by `l`

```
readLIORef :: (Label l) => LIORef l a -> LIO l s a
```

- It reads the content of the reference and, similar to unlabeled, **raises the current label** (clearance permitting) to the join of the current label (`lbl`) and `l`

API: References

We ignore this parameter

```
writeLIORef :: (Label l) => LIORef l a -> a -> LIO l s ()
```

- It writes a value of type `a` into a given reference as long as, similar to label, the label of the reference is **between the current label and the clearance.**

Example (trustworthy code)

It is almost the same code as module `ExampleToLabeledT`

References

We use references instead of `Labeled` values

```
module ExampleRefsT where

import LIO.LIORef
import DCLabel.PrettyShow
import LIO.DCLabel
import LIO.TCB

import ExampleRefsU (computation)

public :: LIO DCLabel () (LIORef DCLabel String)
public = newLIORef lbot "PublicData"

secret :: LIO DCLabel () (LIORef DCLabel String)
secret = newLIORef ltop "SecretData"

execute = do (result, label) <- evalLIO (computation public secret) ()
             putStrLn $ "The result is: " ++ show result
             putStrLn $ "With the label: " ++ prettyShow label
```

Example (untrustworthy code)

```
module ExampleRefsU where

import LIO.LIORef
import LIO.DCLabel
import LIO.LIO

computation p s = do ref_l <- p
                     ref_s <- s
                     s <- readLIORef ref_s
                     writeLIORef ref_l s
                     return ()
```

It reads the content, then the current label is set to `ltop`

It fails to perform the writing!

Final Remarks

- We present a library for dynamically tracking information-flow
- More permissive than previous static approaches
- It also provides some form of discretionary access control
  - Covert channels
- Simple to use and parametric on the label system being used
  - You can use DCLabels!
- As `SecIO`, the correcness of the library relies on type safety and module abstraction
- SafeHaskell is coming for GHC 7.2