CHALMERS | GÖTEBORG UNIVERSITY

Secure Programming via Libraries

Alejandro Russo
(russo@chalmers.se)

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2011

CHALMERS | GÖTEBORG UNIVERSITY

Introduction to Haskell
Introduction to information-flow security
Introduction to Sec

Secure Programming via Libraries

Introduction

Alejandro Russo (russo@chalmers.se)

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

This Course: What is it?

- Programming language technology
  - Type-systems (`void main () { return ; }`)
  - Monitoring
- Theory and practice
  - Haskell
  - Python
- Focus on providing security via a library
- Based on recent research results

This Course: Learning Outcomes

- Security policies
  - Intended behavior of secure systems
- Identify programming languages concepts **useful** to provide security via libraries
- Practical experience with Haskell and Python
- Identify the **scope** of certain security libraries and programming language abstractions or concepts
- Some experience on **formalization** of security mechanisms
  - To prove that they do what they claim!

Organization

- Web page of the course
  - http://www.cse.chalmers.se/~russo/eci2011/
- Discussion email list
  - http://groups.google.com/group/eci-2011-security?hl=es
  - eci-2011-security@googlegroups.com
- 5 Lectures (3hs, 20-25 minutes break)
  - Exercises
- Exam in the end of the course
  - Describe how is going to be

Secure Programming via Libraries

Overview Haskell

Alejandro Russo (russo@chalmers.se)

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

Haskell in a Nutshell

- Purely functional language
  - Functions are **first-class** citizens!
  - Referential transparency

```
int plusone(int x) {return x+1;}

int plusone(int x) {calls++ ;
                    return x+1;}
```

  - Lazy evaluation
    - Expressions are evaluated at most once
  - Advance type system

Haskell Overview

- Definition of functions

```
plusone :: Int -> Int
plusone x = x + 1
```

- Hindley-Milner Polymorphism

```
first :: forall a b. (a,b) -> a
first (x,_) = x
```

- Built-in lists

```
lst1 = [1,2,3,4]
lst2 = 5 : []
```

```
lst3 = lst1 ++ lst2
```


Haskell Overview

- User-defined data types

```
data Nationality = Argentinian | Swedish

f :: Nationality -> String
f Argentinian = "Asado"
f Swedish    = "Surströmming"

data Tree a = Leaf | Node a (Tree a) (Tree a)

nodes :: Tree a -> [a]
nodes Leaf             = []
nodes (Node a t1 t2) = a : (nodes t1 ++ nodes t2)
```


Haskell Overview

- Type classes

```
bcmp x y = x == y
```

- What is the type for the function?

```
bcmp :: forall a. a -> a -> Bool

bcmp :: forall a. (Eq a) => a -> a -> Bool
```

- Type classes

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
instance Eq Int where ...
instance Eq Float where ...
instance Eq a => Eq [a] where ....
```


Haskell Overview

- Input and Output (IO)

```
hello :: IO ()
hello = do putStrLn "Hello! What is your name?"
           name <- getLine
           putStrLn $ "Hi, " ++ name ++ "!"
```

- If computations produce side-effects (IO) is **reflected** in the types!
  - Distinctive feature of Haskell.
  - Very useful for security!

Monads in Haskell

- What is a monad? (Explanation for the masses)
  - ADT denoting a computation that produces a value.
    - We call values of this special type *monadic values* or *monadic computations*
  - **Two operations** to build complex computations from simple ones
    - ***return*** creates monadic computations from simple values like integers, characters, float, etc.
    - ***bind*** takes to monadic computations and **sequentialize** them. The result of the first computation can be used in the second one.
- Examples: IO

Monads in Haskell

- Bind

```
getLine :: IO String        putStrLn :: String -> IO ()

c :: IO ()
c = do name <- getLine
       putStrLn $ "Hi, " ++ name ++ "!"

hello :: IO ()
hello = do putStrLn "Hello! What is your name?"
           name <- getLine
           putStrLn $ "Hi, " ++ name ++ "!"
```

Monads in Haskell

- return

```
return :: a -> IO a
return 42 :: IO Int

nextPrime :: Int -> Int
nextPrime = ....

prim :: IO (Int,Int)
prim = do number <- getLine
          let n = toInt number
          return (n, nextPrime n)
```


Exercise

- Write programs that do the following

```
*Overview> quiz1
What day were you born?
28
Not interesting.
*Overview>
```

```
*Overview> quiz1
What day were you born?
11
It is a prime number!
*Overview>
```

```
quiz1 :: IO ()
quiz1 = do putStrLn "What day were you born?"
           (n, np) <- prim
           if n == np
              then putStrLn $ "It is a prime number!"
              else putStrLn $ "Not interesting."
```


Why Monads?

- Monads represent computations.
- Different kind of monads represent different kind of computations
  - IO monad represents computation with inputs and outputs
- In this course, we will define a monad to represent *secure computations*
  - Computations where security is preserved

Secure Programming via Libraries

Information-Flow Security

Alejandro Russo (russo@chalmers.se)

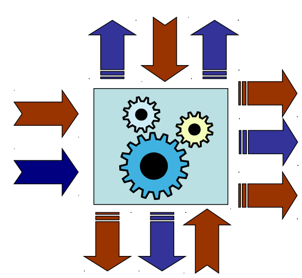
Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

Introduction

- Computer systems usually **send**, **receive**, and **store** *confidential information*
- *Computer networks provides benefits but exposes systems to attacks (malicious code)*
- *We want to preserve confidentiality*
  - *End-to-end security policy*

End-to-end Security Policies

- Security policies (intended behavior) that speaks about end-points of the system
- End-points?
  - Inputs and outputs!
- Confidentiality?



Language-based Security

[Kozen 99]

- How to to guarantee and end-to-end security requirements as confidentiality?
- Language-based security technology **inspects** the code of applications to guarantee security policies.
  - Fusion of programming languages technology and computer security
- Information-flow security

Language-based Information-Flow Security

[Sabelfeld, Myers 03]

- Programming languages techniques to **track** how data flows inside programs
  - Preserve confidentiality
  - Preserve some integrity of data
    - Corrupt data does not influence security critical operation
- It can be performed
  - Statically
    - Type-system [Volpano Smith Irnive 96]
  - Dynamically
    - Monitor [Volpano 99] [Le Guernic et al. 06]
  - Hybrid [Le Guernic et al. 06] [Russo, Sabelfeld 10]
- Comparison between static and dynamic techniques [Sabelfeld, Russo 09]

Security Lattice

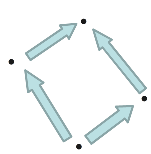
- Assign security levels to data representing their confidentiality
- Security levels are placed in a lattice (security lattice)
  - Information can flow from low to high positions in the lattice
- For simplicity, we only consider two security levels

$L \sqsubseteq H$ and $H \not\sqsubseteq L$

$L \sqcup H = H$ $\quad L \sqcap H = L$

$L \sqcup L = L$ $\quad L \sqcap L = L$

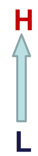
$H \sqcup H = H$ $\quad H \sqcap H = H$

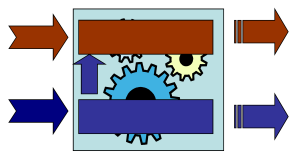


Non-interference

[Goguen Meseguer 82]

- Security policy to preserve confidentiality
- Given the two-point security lattice, then *non-interference* establishes that public outputs should not depend on secret data
- Programs have secret and public inputs and outputs, respectively





- More formally,

$$\forall O_L \exists I_L \forall I_H \cdot low(P(I_L, I_H)) = O_L$$

Types of Illegal Flows

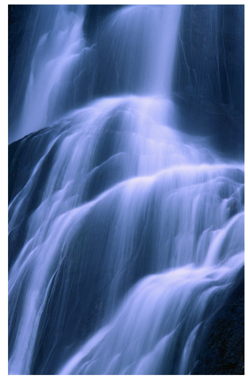
[Denning, Denning 77]

- Explicit flows

```
l := h
```

- Implicit flows

```
if h>0
   then l:=1
   else l:=2
```



Covert Channels

- Besides explicit and implicit flows, programs can leak information by other means
  - Not originally designed for that purpose
- It depends on the attacker observational power
- Energy consumption (e.g. Smartcards [Messerges et al])
- External timing
  - Arbitrarily precise stopwatch [Agat 00]
  - Cache attacks [Jackson et al 06]
  - Termination [Askarov et al 08]
- Internal timing
  - No precise stopwatch, but rather affecting the behavior of threads depending on the secret [Russo 08]

Termination Insensitive Non-interference

[Askarov et al 08]

- Non-interference security policy that ignores leaks due to termination

$$\forall O_L \exists I_L \forall I_H \cdot terminates(P) \Rightarrow low(P(I_L, I_H)) = O_L$$

- Main information-flow compilers ignore leaks due termination [Jif] [FlowCaml]
- What is the bandwidth of this covert channel?
  - A secret cannot be leaked in polynomial time
  - For uniform distributed secrets, the advantage to gain when guessing the secrets (after a polynomial amount of observation) is negligible

```
l:=0 ;
if h>0
  then while true do skip ;
```

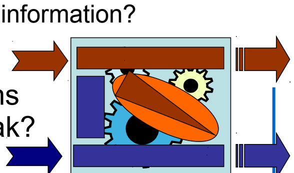
h<=0 → l = 0 (Ok)
h>0 → (Loop)

- From now on, we ignore termination.
  - Non-interference means termination insensitive non-interference

Declassification

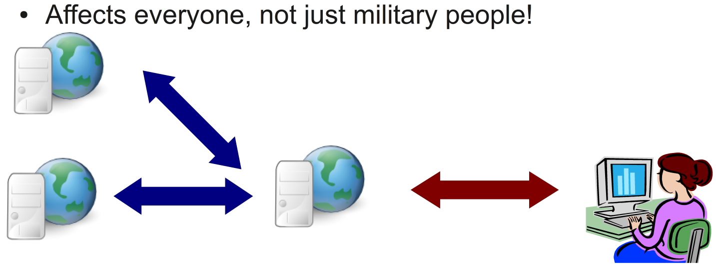
[Sabelfeld, Sands 07]

- Useful system **intentionally** release information as part of its behavior
  - Password checker (pwd == input)
- Dimensions and principles of declassification
  - **What** information can be leak?
  - **When** can information be leaked?
  - **Where** in the program is safe to leak information?
  - **Who** can leak information?
- How to be certain that our programs leak what they are supposed to leak?



Where Information-flow security is useful?

- It originally emerges from military settings
  - Mandatory access control [Bell and LaPadula 73]
- Nowadays, the web is an exciting scenario to apply information-flow control [FlowSafe Mozilla]
  - Affects everyone, not just military people!



Where Information-flow security is useful?

- It originally emerg
  - Mandatory acce
- Nowadays, the wel information-flow co
  - Affects everyon

Web Security and Information-flow

[OWASP 10]

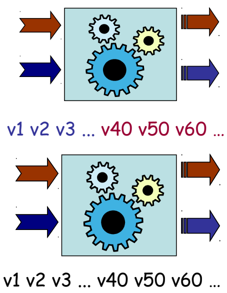
- Ten most frequent attacks
  - A1 – Injection (SQL, OS, etc)
    - **Information-flow**
  - A2 – Cross Site Scripting (XSS)
    - **Information-flow**
  - A3 – Broken Authentication and Session Management
    - **Information-flow** helps here as well
  - A4 – Insecure Direct Object References
    - **Information-flow**
  - ....
- Very hot area at the moment for doing research

Static vs. Dynamic Enforcement for Information-flow

- Security policy: secrets must no be leaked!
  - Termination insensitive non-interference
- Some purely dynamic mechanisms are as secure as traditional type-systems [Sabelfeld, Russo 09]
- Should we go dynamic or static?
  - Several arguments are possible to argue against [Le Guernic et al, 06] [Shroff et al, 07] [Vogt et al, 07]
  - In favor of dynamic monitors
    - Permissiveness
    - Dynamic code evaluation (eval in JavaScript)
- Web applications *permissiveness* is very important !

Flow-sensitive and Flow-insensitive Enforcement for Non-interference [Hunt, Sands 06]

- Traditional enforcements
  - Avoid illegal explicit and implicit flows
  - Fix sources of secret and public inputs and outputs
- Flow-insensitive (FI)
  - Each variable has a fix security level during the execution of the program
- Flow-sensitive (FS)
  - Variables can change their security level during execution according to the data stored at a given time
  - More convenient for programmers!
- A program accepted by traditional FS type-system is also accepted by traditional FI type-system (rewriting)



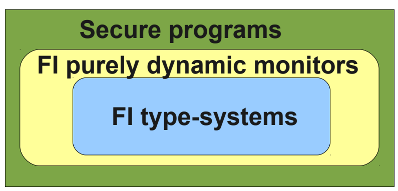

```
v1 := h ;
v2 := v1+l ;
v1 := l ;
h  := v1 + v2 ;
```


Information-flow Security

- Active research area
  - No more only motivated by military applications
- Web security and information-flow is a hot topic!
  - Companies are showing interests on this technology
- During the 70's dynamic techniques were pioneers
  - Operating system security
- During the 90's static techniques were dominant
  - Language-based security
- During 00's, dynamic techniques are back!
  - We can see combination of both
- Exiting times to do research on the area!

Flow-sensitive and Flow-insensitive Enforcement for Non-interference [Sabelfeld, Russo 09] [Russo, Sabelfeld 10]

- Hunt and Sands compare two static enforcements
  - FI and FS type-systems
- Flow-insensitive
  - FI monitor is as secure as traditional FI type-sytems
  - Monitor accepts more programs



- Flow-sensitive
  - No possible to obtain a sound and more permissive purely dynamic monitor (than a FS type-system)
  - To recover the picture above for FS, static analysis is needed!
  - Is it desired to recover the picture above? [Austin, Flanagan 09]
    - Open question

Secure Programming via Libraries

Introduction

Alejandro Russo (russo@chalmers.se)

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

This Course: What is it?

- Programming language technology
  - Type-systems (`void main () { return ; }`)
  - Monitoring
- Theory and practice
  - Haskell
  - Python
- Focus on providing security via a library
- Based on recent research results

This Course: Learning Outcomes

- Security policies
  - Intended behavior of secure systems
- Identify programming languages concepts **useful** to provide security via libraries
- Practical experience with Haskell and Python
- Identify the **scope** of certain security libraries and programming language abstractions or concepts
- Some experience on **formalization** of security mechanisms
  - To prove that they do what they claim!

Organization

- Web page of the course
  - http://www.cse.chalmers.se/~russo/eci2011/
- Discussion email list
  - http://groups.google.com/group/eci-2011-security?hl=es
  - eci-2011-security@googlegroups.com
- 5 Lectures (3hs, 20-25 minutes break)
  - Exercises
- Exam in the end of the course
  - Describe how is going to be

Secure Programming via Libraries

Overview Haskell

Alejandro Russo (russo@chalmers.se)

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

Haskell in a Nutshell

- Purely functional language
  - Functions are **first-class** citizens!
  - Referential transparency

```
int plusone(int x) {return x+1;}

int plusone(int x) {calls++ ;
                    return x+1;}
```

  - Lazy evaluation
    - Expressions are evaluated at most once
  - Advance type system

Haskell Overview

- Definition of functions

```
plusone :: Int -> Int
plusone x = x + 1
```

- Hindley-Milner Polymorphism

```
first :: forall a b. (a,b) -> a
first (x,_) = x
```

- Built-in lists

```
lst1 = [1,2,3,4]
lst2 = 5 : []
```

```
lst3 = lst1 ++ lst2
```


Haskell Overview

- User-defined data types

```
data Nationality = Argentinian | Swedish

f :: Nationality -> String
f Argentinian = "Asado"
f Swedish    = "Surströmming"

data Tree a = Leaf | Node a (Tree a) (Tree a)

nodes :: Tree a -> [a]
nodes Leaf             = []
nodes (Node a t1 t2) = a : (nodes t1 ++ nodes t2)
```


Haskell Overview

- Type classes

```
bcmp x y = x == y
```

- What is the type for the function?

```
bcmp :: forall a. a -> a -> Bool

bcmp :: forall a. (Eq a) => a -> a -> Bool
```

- Type classes

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
instance Eq Int where ...
instance Eq Float where ...
instance Eq a => Eq [a] where ....
```


Haskell Overview

- Input and Output (IO)

```
hello :: IO ()
hello = do putStrLn "Hello! What is your name?"
           name <- getLine
           putStrLn $ "Hi, " ++ name ++ "!"
```

- If computations produce side-effects (IO) is **reflected** in the types!
  - Distinctive feature of Haskell.
  - Very useful for security!

Monads in Haskell

- What is a monad? (Explanation for the masses)
  - ADT denoting a computation that produces a value.
    - We call values of this special type *monadic values* or *monadic computations*
  - **Two operations** to build complex computations from simple ones
    - ***return*** creates monadic computations from simple values like integers, characters, float, etc.
    - ***bind*** takes to monadic computations and **sequentialize** them. The result of the first computation can be used in the second one.
- Examples: IO

Monads in Haskell

- Bind

```
getLine :: IO String     putStrLn :: String -> IO ()

c :: IO ()
c = do name <- getLine
       putStrLn $ "Hi, " ++ name ++ "!"

hello :: IO ()
hello = do putStrLn "Hello! What is your name?"
           name <- getLine
           putStrLn $ "Hi, " ++ name ++ "!"
```

Monads in Haskell

- return

```
return :: a -> IO a
return 42 :: IO Int

nextPrime :: Int -> Int
nextPrime = ....

prim :: IO (Int,Int)
prim = do number <- getLine
          let n = toInt number
          return (n, nextPrime n)
```


Exercise

- Write programs that do the following

```
*Overview> quiz1
What day were you born?
28
Not interesting.
*Overview>
```

```
*Overview> quiz1
What day were you born?
11
It is a prime number!
*Overview>
```

```
quiz1 :: IO ()
quiz1 = do putStrLn "What day were you born?"
           (n, np) <- prim
           if n == np
              then putStrLn $ "It is a prime number!"
              else putStrLn $ "Not interesting."
```


Why Monads?

- Monads represent computations.
- Different kind of monads represent different kind of computations
  - IO monad represents computation with inputs and outputs
- In this course, we will define a monad to represent *secure computations*
  - Computations where security is preserved

Secure Programming via Libraries

Information-Flow Security

Alejandro Russo (russo@chalmers.se)

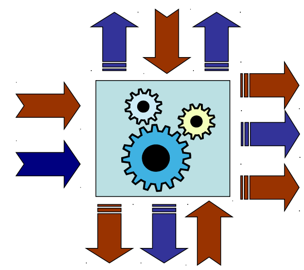
Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

Introduction

- Computer systems usually **send**, **receive**, and **store** *confidential information*
- *Computer networks provides benefits but exposes systems to attacks (malicious code)*
- *We want to preserve confidentiality*
  - *End-to-end security policy*

End-to-end Security Policies

- Security policies (intended behavior) that speaks about end-points of the system
- End-points?
  - Inputs and outputs!
- Confidentiality?



Language-based Security
[Kozen 99]

- How to to guarantee and end-to-end security requirements as confidentiality?
- Language-based security technology **inspects** the code of applications to guarantee security policies.
  - Fusion of programming languages technology and computer security
- Information-flow security

Language-based Information-Flow Security
[Sabelfeld, Myers 03]

- Programming languages techniques to **track** how data flows inside programs
  - Preserve confidentiality
  - Preserve some integrity of data
    - Corrupt data does not influence security critical operation
- It can be performed
  - Statically
    - Type-system [Volpano Smith Irnive 96]
  - Dynamically
    - Monitor [Volpano 99] [Le Guernic et al. 06]
  - Hybrid [Le Guernic et al. 06] [Russo, Sabelfeld 10]
- Comparison between static and dynamic techniques [Sabelfeld, Russo 09]

Security Lattice

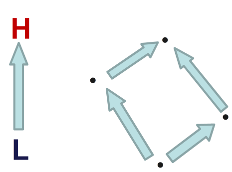
- Assign security levels to data representing their confidentiality
- Security levels are placed in a lattice (security lattice)
  - Information can flow from low to high positions in the lattice
- For simplicity, we only consider two security levels

$L \sqsubseteq H$ and $H \not\sqsubseteq L$

$L \sqcup H = H$ $\quad L \sqcap H = L$

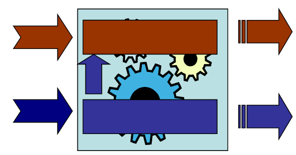
$L \sqcup L = L$ $\quad L \sqcap L = L$

$H \sqcup H = H$ $\quad H \sqcap H = H$



Non-interference
[Goguen Meseguer 82]

- Security policy to preserve confidentiality
- Given the two-point security lattice, then *non-interference* establishes that public outputs should not depend on secret data
- Programs have secret and public inputs and outputs, respectively
- More formally,



$$\forall O_L \exists I_L \forall I_H \cdot low(P(I_L, I_H)) = O_L$$

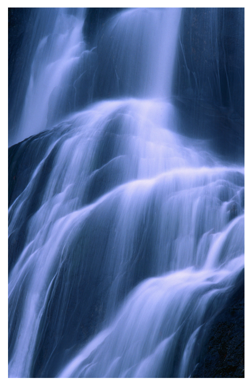
Types of Illegal Flows
[Denning, Denning 77]

- Explicit flows

```
l := h
```

- Implicit flows

```
if h>0
   then l:=1
   else l:=2
```



Covert Channels

- Besides explicit and implicit flows, programs can leak information by other means
  - Not originally designed for that purpose
- It depends on the attacker observational power
- Energy consumption (e.g. Smartcards [Messerges et al])
- External timing
  - Arbitrarily precise stopwatch [Agat 00]
  - Cache attacks [Jackson et al 06]
  - Termination [Askarov et al 08]
- Internal timing
  - No precise stopwatch, but rather affecting the behavior of threads depending on the secret [Russo 08]

Termination Insensitive Non-interference

[Askarov et al 08]

- Non-interference security policy that ignores leaks due to termination

$$\forall O_L \exists I_L \forall I_H \cdot terminates(P) \Rightarrow low(P(I_L, I_H)) = O_L$$

- Main information-flow compilers ignore leaks due termination [Jif] [FlowCaml]
- What is the bandwidth of this covert channel?
  - A secret cannot be leaked in polynomial time
  - For uniform distributed secrets, the advantage to gain when guessing the secrets (after a polynomial amount of observation) is negligible

```
l:=0 ;
if h>0
  then while true do skip ;
```

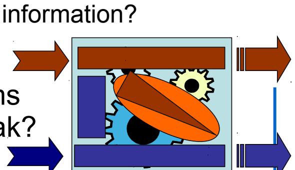
h<=0 → l = 0 (Ok)
h>0 → (Loop)

- From now on, we ignore termination.
  - Non-interference means termination insensitive non-interference

Declassification

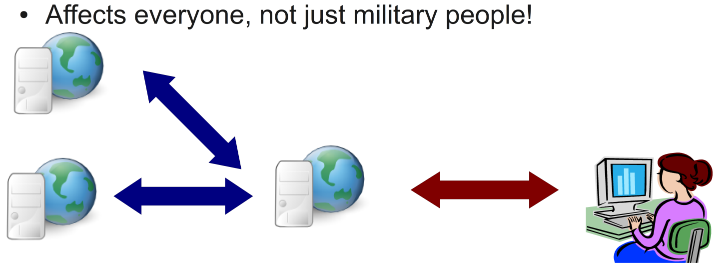
[Sabelfeld, Sands 07]

- Useful system **intentionally** release information as part of its behavior
  - Password checker (pwd == input)
- Dimensions and principles of declassification
  - **What** information can be leak?
  - **When** can information be leaked?
  - **Where** in the program is safe to leak information?
  - **Who** can leak information?
- How to be certain that our programs leak what they are supposed to leak?



Where Information-flow security is useful?

- It originally emerges from military settings
  - Mandatory access control [Bell and LaPadula 73]
- Nowadays, the web is an exciting scenario to apply information-flow control [FlowSafe Mozilla]
  - Affects everyone, not just military people!



Where Information-flow security is useful?

- It originally emerg
  - Mandatory acce
- Nowadays, the wel information-flow co
  - Affects everyon

Web Security and Information-flow

[OWASP 10]

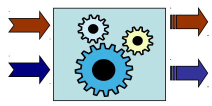
- Ten most frequent attacks
  - A1 – Injection (SQL, OS, etc)
    - **Information-flow**
  - A2 – Cross Site Scripting (XSS)
    - **Information-flow**
  - A3 – Broken Authentication and Session Management
    - **Information-flow** helps here as well
  - A4 – Insecure Direct Object References
    - **Information-flow**
  - ....
- Very hot area at the moment for doing research

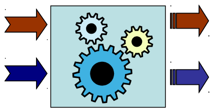
Static vs. Dynamic Enforcement for Information-flow

- Security policy: secrets must no be leaked!
  - Termination insensitive non-interference
- Some purely dynamic mechanisms are as secure as traditional type-systems [Sabelfeld, Russo 09]
- Should we go dynamic or static?
  - Several arguments are possible to argue against [Le Guernic et al, 06] [Shroff et al, 07] [Vogt et al, 07]
  - In favor of dynamic monitors
    - Permissiveness
    - Dynamic code evaluation (eval in JavaScript)
- Web applications *permissiveness* is very important !

Flow-sensitive and Flow-insensitive Enforcement for Non-interference [Hunt, Sands 06]

- Traditional enforcements
  - Avoid illegal explicit and implicit flows
  - Fix sources of secret and public inputs and outputs
- Flow-insensitive (FI)
  - Each variable has a fix security level during the execution of the program
- Flow-sensitive (FS)
  - Variables can change their security level during execution according to the data stored at a given time
  - More convenient for programmers!
- A program accepted by traditional FS type-system is also accepted by traditional FI type-system (rewriting)

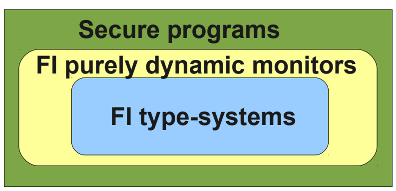





```
v1 := h ;
v2 := v1+l ;
v1 := l ;
h  := v1 + v2 ;
```


Flow-sensitive and Flow-insensitive Enforcement for Non-interference [Sabelfeld, Russo 09] [Russo, Sabelfeld 10]

- Hunt and Sands compare two static enforcements
  - FI and FS type-systems
- Flow-insensitive
  - FI monitor is as secure as traditional FI type-sytems
  - Monitor accepts more programs



- Flow-sensitive
  - No possible to obtain a sound and more permissive purely dynamic monitor (than a FS type-system)
  - To recover the picture above for FS, static analysis is needed!
  - Is it desired to recover the picture above? [Austin, Flanagan 09]
    - Open question

Information-flow Security

- Active research area
  - No more only motivated by military applications
- Web security and information-flow is a hot topic!
  - Companies are showing interests on this technology
- During the 70's dynamic techniques were pioneers
  - Operating system security
- During the 90's static techniques were dominant
  - Language-based security
- During 00's, dynamic techniques are back!
  - We can see combination of both
- Exiting times to do research on the area!

Secure Programming via Libraries

A library for information-flow in Haskell

Alejandro Russo (russo@chalmers.se)

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

Encoding information-flow in Haskell
[Li, Zdancewic 06]

- Show that it is possible to guarantee IFC by a library
- Implementation in Haskell using Arrows [Hughes 98]
- Arrows? A generalization of Monads [Wadler 01]
- Pure values only
  - No side-effects
- One security label for data
  - All secret or all public!

Encoding information-flow in Haskell
[Tsai, Russo, Hughes 07]

- Extend the library by Li and Zdancewic
  - More than one security label for data
  - Concurrency
- Major changes in the library
  - New arrows
  - Lack of arrow notation
- Why arrows?
  - Li and Zdancewic argue that *monads are not suitable for the design of such a library*

A lightweight library for Information-flow in Haskell
[Russo, Claessen, Hughes 08]

- Lightweight
  - Approximately 325 lines of code
  - Static type-system of Haskell to enforce non-interference
  - Dynamic checks when declassification occurs
- Use Monads (not Arrows!)
  - Programmers are more familiar with Monads than Arrows

A lightweight library for Information-flow in Haskell
[Russo, Claessen, Hughes 08]

- The library **relies on** Haskell
  - Capabilities to maintain abstraction of data types
    - Haskell module system
  - Haskell is **strongly typed**
    - We cannot cheat!
- There are extensions of Haskell that break these two requirements!
- For a full list, please visit the proposal of SafeHaskell
  - An extension of Haskell to disallow those dangerous features than can jeopardize security
  - Join work with Prof. Mazieres et al. at Stanford university.

```
unsafePerformIO :: IO a -> a
unsafeCoerce :: a -> b
```


Why Haskell?

- Clear separation of pure computations with those with side-effects
- Every computation with side-effects is encapsulated into the IO monad
- Side-effects can encode information about secret data
- It is necessary to control them
  - It is known where they occur! Just look at the type!

Side-effects and IO

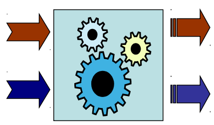
- Just look at the type!

```
f1 :: Eq a => a -> [a] -> ([a], Bool)

f2 :: (Show a, Eq a) => Int -> a -> ([a], IO Bool)
```

- All bets are off if an IO computation comes from untrustworthy code
  - It is not known the side-effects that it will produce

```
f1 x xs = (take 10 (cycle xs), elem x xs)

f2 n x = (take n (iterate id x),
          do putStrLn "Hi!"
             putStrLn "The arguments of the function are"
             putStrLn $ "x = " ++ show x
             putStrLn $ "n = " ++ show n
             return True )
```


Secure Pure Computations



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

f (c, i) = ( chr(ord c + i), i)
```
YES

```
f (c, i) = (chr(ord c + i), ord c)
```
NO

```
f (c, i) = (chr(ord c + 1), i+1)
```
YES

```
f (c, i) | c > 65     =  (c, 42)
         | otherwise  =  (c, i)
```
NO

A Security Monad for Pure Computations

- Security monad
  - It assigns a security level to data
  - Once inside the monad, it is not possible to escape!

```
data Sec s a -- abstract
instance Monad (Sec s)
```

  - We represent security levels by singleton types

```
secret :: Sec H Int
secret = ...
```
H

```
known :: Sec L Int
known = ...
```
L

Using Sec

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

f' :: (Sec H Char, Int)
      -> (Sec H Char, Int)

f (c, i) = ( chr(ord c + i), i)
```
YES

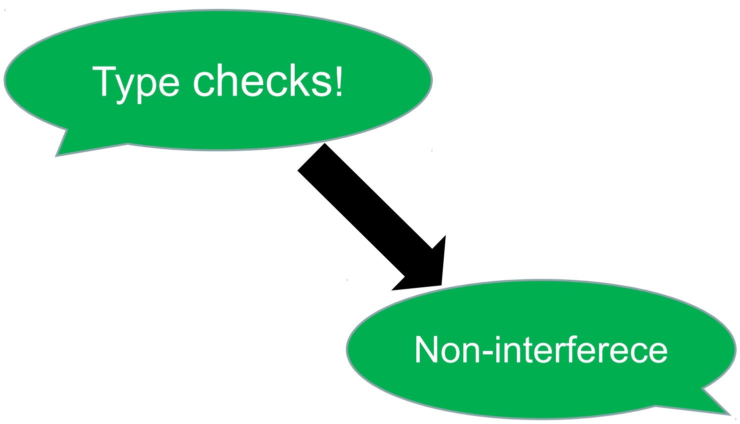
```
f' (sec_c, i) = (do c <- sec c
                    return (chr(ord c + i))
                ,i)
```
YES

Using Sec

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

f' :: (Sec H Char, Int)
      -> (Sec H Char, Int)

f (c, i) = (chr(ord c + i), ord c)
```
NO

```
f' (sec_c, i) = ( do c <- sec c
                     return (chr(ord c + i))
                ,do c <- sec c
                    return (ord c) )
```
NO

Security Guarantee

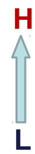


A Security Monad for Pure Computations

- Security monad
  - It assigns a security level to data
  - Once inside the monad, it is not possible to escape!

```
data Sec s a -- abstract
instance Monad (Sec s)
```

  - We represent security levels by singleton types
  - What about the security lattice?



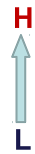
Security Lattice

- We model it using type classes in Haskell
  - Constrains to polymorphic types

```
class Less s s' where
      less :: s -> s' -> ()
```

  - Encoding two-point lattice is just provide instances for that type class

```
instance Less L H where
   less _ _ = ()

instance Less L L where
   less _ _ = ()

instance Less H H where
   less _ _ = ()
```



Security Monad and the Security Lattice

- Push up information in the security lattice

```
up :: Less s s' => Sec s a -> Sec s' a
```

- It allows to convert public values into secrets

```
fup :: Sec L Int -> Sec H Char
fup sec_i = do i <- up (sec_i)
               return (chr i)
```

- What if it is possible to make the following instance?

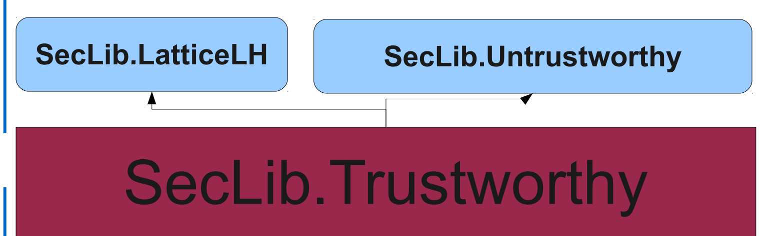
```
instance Less H L where
   less _ _ = ()
```


Security Monad and the Security Lattice

- The library works as long as
  - Attackers cannot define method `less` for arbitrary instances of the type class `Less`
- How to ensure that?
  - Mainly by the abstraction power of Haskell's module system

Arquitecture

```
module X where

import SecLib.Untrustworthy
import SecLib.LatticeLH

...
```



Importing SecLib.Trustworthy

- `SecLib.Trustworthy` must not be imported by untrustworthy code
  - Otherwise, no security guarantees are possible

```
instance Less H L where
   less _ _ = ()
```

Other Assumptions

- The monad `Sec s` must remain abstract
  - Guarantee by the installation of the library
  - `Sec.hs` is not an exposed module
- Use of unsafe Haskell extensions
  - `StandaloneDeriving`
  - `System.IO.Unsafe`
    - `unsafePerformIO, unsafeIterleaveIO, etc.`
  - `OverlappingInstances`
- Check SafeHaskell (work-in-progress)
  - A Haskell extension to safely execute untrusted Haskell code

Security API for Pure Computations

```
data Sec s a -- abstract
instance Monad (Sec s)


up :: Less s s' => Sec s a -> Sec s' a


module X where

import SecLib.Untrustworthy
import SecLib.LatticeLH
```

CHALMERS | GÖTEBORG UNIVERSITY

Introduction to SecIO

Secure Programming via Libraries

A library for information-flow in Haskell (side-effects)

Alejandro Russo (russo@chalmers.se)

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

Side-effects?

[Russo, Claessen, Hughes 08]

- What about trying to do side-effects inside of the security monad?

NO `Sec H (IO ())` YES

- Would you run the IO computation?

Malicious Code

- The following code shows malicious side-effects

```
func :: Sec H Char -> Sec H (IO ())
func sec_c = do c <- sec_c
                return $ do putStrLn "The secret is gone!"
                            writeFile "PublicFile" [c]
```

- Important Haskell feature for security: ***by looking the type of a piece of code, it is possible to determine if it performs side-effects!***

Side-effects and Sec

- Trustworthy code

```
module SideEffectsSecT where

import Data.Char
import SecLib.LatticeLH
import SecLib.Trustworthy

import SideEffectsSecU -- Import the untrustworthy function unsafe

secret :: Sec H Char    -- This is the secret to be manipulated by the
                        -- untrustworthy code
secret = return 'X'

execute :: IO ()
execute = reveal $ unsafe func
```


Side-effects and Sec

- Untrustworthy code

```
module SideEffectsSecU where

import Data.Char
import SecLib.LatticeLH
import SecLib.Untrustworthy


-- Do not execute IO operations inside Sec!
func :: Sec H Char -> Sec H (IO ())
func sec_c = do c <- sec_c
                return $ do putStrLn "The secret is gone!"
                            writeFile "PublicFile" [c]
```


Little Quiz

- What about programs of the following type?

NO `Sec H (IO (Sec L Int))`

NO `Sec H (Sec L (IO Char))`

NO `Sec L (Sec H (IO ()))`

YES `Sec L (IO (Sec H Char))`

Side-effects?

[Russo, Claessen, Hughes 08]

- What about trying to do side-effects inside of the security monad?

NO

```
Sec H (IO ())
```

YES

- We do not know if the side-effects are safe to perform
- What should we do?
- IO is a monad that encapsulates side-effects
- **Let us make another monad that encapsulates safe side-effects!**

Monad SecIO

- It is a monad that performs secure side-effects
  - Side-effects that preserve confidentiality!

It is a computation that
a) writes to security locations above s and
b) which **result, of type a, has confidentiality level at least a**

```
data SecIO s a -- abstract
instance Monad (SecIO s)
```


Monad SecIO

- We show how it works for files
  - It also works for references and sockets (check the library)

```
data SecIO s a
```

It is a computation that
a) writes to security locations above s and
b) which **result, of type a, has confidentiality level at least a**

```
c1 :: SecIO H Int
```

It writes to secret files and returns a secret integer

```
c2 :: SecIO L (Sec H Int)
```

It writes to public and secret files and returns a secret integer

```
c3 :: SecIO L Int
```

It writes to public and secret files and returns public integer

API for SecIO

```
data SecIO s a
instance Monad (SecIO s)

type File s
```

It is a file which content has confidentiality level s

The secure version of the operations to read and write files in Haskell

```
readFileSecIO  :: File s -> SecIO s' (Sec s String)

writeFileSecIO :: File s -> String -> SecIO s ()
```

```
readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO ()
```

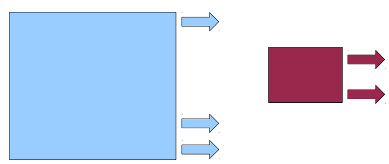

API for SecIO

```
value :: Sec s a -> SecIO s a
```

It pushes any pure secure value into a side-effectful computation

```
plug :: Less sl sh =>
        SecIO sh a -> SecIO sl (Sec sh a)
```

It plugs computations that perform side-effects at a higher level into computations that perform side-effect into lower levels



```
-- Only trustworthy code (breaks the abstraction)
revealSecIO :: SecIO s a -> IO (Sec s a)
```


Small Example

- We want to write a function that copy contents of files
- We do not want *the function to leak information*
- The function should allow copying:
  - a public file into another public file,
  - a secret file into another secret file,
  - a public one into a secret one
- It must avoid *copying a secret file into a public one*
- We will use the library to get the security part of the code right!

Small Example: Trustworthy code

```haskell
module CopyT where

import SecLib.LatticeLH
import SecLib.Trustworthy

import CopyU (copy)

secret_file :: File H
secret_file = mkFile "SecretFile"

public_file :: File L
public_file = mkFile "PublicFile"

trusted_copy :: Less s s' => (File s -> File s' -> SecIO s' ())
                          -> File s -> File s' -> IO ()

trusted_copy copy_func fs fs' = do sec <- revealIO $ copy_func fs fs'
                                   return $ reveal sec
execute :: IO ()
execute = trusted_copy copy public_file secret_file
```

It imports the untrustworthy copying function

It establishes the confidentiality level of the files

Type for the untrustworthy copying function

It executes the untrustworthy function. Does it preserve confidentiality?

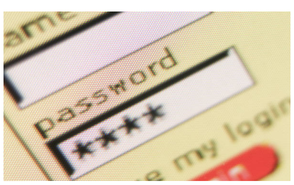
Small Example: Untrustworthy code

```haskell
module CopyU where

import SecLib.LatticeLH
import SecLib.Untrustworthy

copy :: Less s s' => File s -> File s' -> SecIO s' ()
copy file1 file2  = do sec_str <- readFileSecIO file1
                       str     <- value (up sec_str)
                       writeFileSecIO file2 str
```

It provides a function with the type requested by module CopyT

- Can you write the function above in such a way that copies the content of a secret file into a public one?
  - Try it out!
- The type-checker will not allow it

Constructing a Secure Password Administrator



- Linux Password Administrator
  - /etc/passwd

```
bjorn:x:1003:100::/home/andrei:/bin/bash
hana:x:500:100::/home/tsa:
josef:x:1006:100::/home/john:/bin/bash
```

  - /etc/shadow

```
bjorn:$1$0ID5oZxB$0tdKR1VQWWQlkJR1Uj7na0:13397:0:99999:7:::
hana:$1$.28fO/M9$aaNMN4SWEKZiGPYoEq9996:13460:0:::::0
josef:$1$UP1uD.28$hi3vYEa20.zgWYNVN/Lq81:13539:0:99999:7:::
```

- 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 (or SUID root) in order for the the program to be able to **access** the /etc/shadow file.

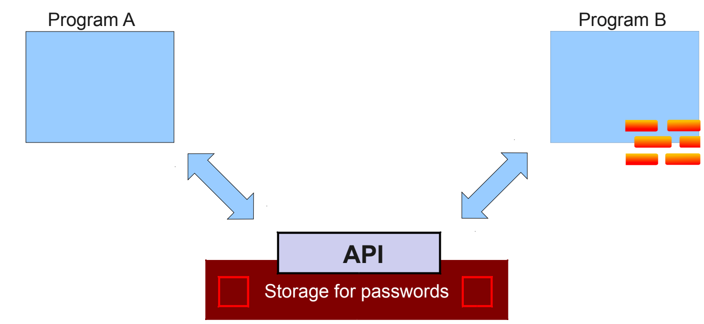
Password Administrator

- What are the security concerns?
  - Give root permission to a program that only needs to authenticate a user
  - Password might be leaked (un)intentionally (dictionary attacks)
- Linux provides an API to access `/etc/shadow`

```c
#ifdef HAS_SHADOW
#include <shadow.h>
#include <shadow/pwauth.h>
#endif
```

- File `/etc/shadow` can be accessed by other means (not only by the API)
- We assume the opposite (e.g. in kernel space, remote server, etc)

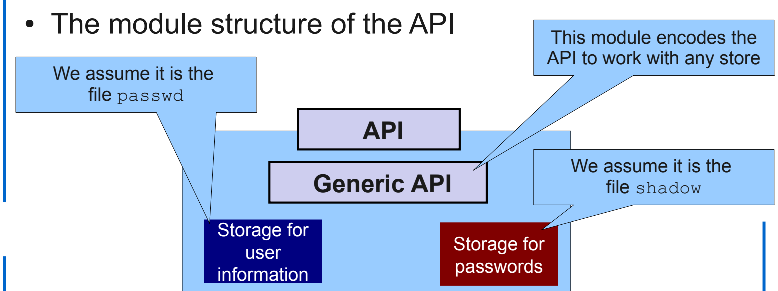
More graphically



	Required root access	Confidentiality
C program + shadow.h	**YES**	**NO**
Haskell program + SecLib	**NO**	**YES**

Password Administrator

- Let us implement the API in Haskell
  - Recall that shadow password are only accessible via the API
- The module structure of the API



GenericAPI

```
module GenericAPI
 ( getSpwdName, putSpwd, getNames )
where

import Spwd
```

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Cypher }
```

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
```

Store for user information

Store for password

```
putSpwd :: FilePath -> Spwd -> IO ()
```

Store for password

```
getNames :: FilePath -> IO [Name]
```

Store for user information

API

```
module API
 (
     getSpwdName
   , putSpwd
   , getNames
 )
where

import Spwd
import qualified GenericAPI as GenericAPI (getSpwdName, putSpwd, getNames)

passwd :: FilePath
passwd = "./passwd"

shadow :: FilePath
shadow = "./shadow"

getSpwdName :: Name -> IO (Maybe Spwd)
getSpwdName = GenericAPI.getSpwdName passwd shadow

putSpwd :: Spwd -> IO ()
putSpwd = GenericAPI.putSpwd shadow

getNames :: IO [Name]
getNames = GenericAPI.getNames passwd
```

Store of user information

Store for passwords

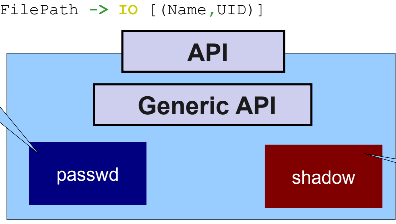
The module applies the generic API interface to specific stores

Implementing getSpwdName

- Some internals of the implementation
  - It is not the most advance password administrator
  - You can do it better!
  - It is only for pedagogical purposes

```
parse_passwd :: FilePath -> IO [(Name,UID)]
```

[(Name, UID)]



[(UID, Cypher)]

```
parse_shadow :: FilePath -> IO [(UID,Cypher)]
```


Implementing getSpwdName

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
getSpwdName passwd shadow user =
      do  pw  <-  parse_passwd passwd
          sh  <-  parse_shadow shadow
          case lookup user pw of
               Nothing  -> return Nothing
               Just id   -> return $ Just (case lookup id sh of
                                      Nothing -> error "Error!"
                                      Just c  -> Spwd { uid = id ,
                                                        cypher = c})
```

pw :: [(Name, UID)]

sh :: [(UID, Cypher)]

```
parse_passwd :: FilePath -> IO [(Name,UID)]
parse_shadow :: FilePath -> IO [(UID,Cypher)]
```


Using the API

- Programs using that API can build up more sophisticated functions

```
module Auxiliaries where

import Data.Maybe
import Spwd
import API

-- Function to suggest a user name
suggest_name :: Name -> IO Name
suggest_name name =
    do ns <- getNames
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b == False)
                                     [ (name', name' `elem` ns)
                                       | n <- [0..], let name'= name ++ show n]
```

- How does it work?
  - User “david” is in the system, then it suggests “david0”. If “david0” is in the system, then it suggests “david1”, etc.
- Could someone implement some unintended behaviour in this function?

Using the API

```
suggest_name :: Name -> IO Name
suggest_name name =
    do ns <- getNames
       f ns
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b == False)
                                     [ (name', name' `elem` ns)
                                      | n <- [0..], let name' = name ++ show n ]


f :: [Name] -> IO ()
f ns = do lst <- f' ns
          writeFile "foo" (show lst)
          return ()

       where f' []     = return []
             f' (n:ns) = do spwd <- getSpwdName n
                            lst  <- f' ns
                            return $ (n, (cypher $ fromJust spwd)) : lst
```

What is this?

It is copying the passwords to a file

Modifying the API?

- We see two versions of `suggest_name`
  - Built on the password adminstrator API
- To identify the one violating confidentiality, we looked at the code and think for a bit
  - *Code revision*
- Let us use the `SecLib` to automatically enforce confidentiality
  - In that manner, we do not need to do code review!
  - Of course, we still need to do testing for correctness

Marking the Secret Data

- How do we start?
  - Indicating which are the secrets (passwords) in our program

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Cypher }
```

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }
```


GenericAPI: Secure Version

```
module GenericAPI
  ( getSpwdName, putSpwd, getNames )
where
import SecLib.LatticeLH
import SecLib.Untrustworthy
import Spwd

-- getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
-- putSpwd :: FilePath -> Spwd -> IO ()
-- getNames :: FilePath -> IO [Name]

getSpwdName :: File L -> File H -> Name -> SecIO s (Maybe Spwd)

putSpwd :: File H -> Swpd -> SecIO H ()

getNames :: File L -> SecIO s [Name]
```

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }
```

Store for user information

Store for password

This function does not write to any file

This function writes to a secret file

Store for password

This function does not write to any file

API: Secure Version

```
module API
  (
      getSpwdName
    , putSpwd
    , getNames
  )
where

import Spwd
import qualified GenericAPI as GenericAPI (getSpwdName, putSpwd, getNames)

import SecLib.Trustworthy
import SecLib.LatticeLH

passwd :: File L
passwd = mkFile "./passwd"

shadow :: File H
shadow = mkFile "./shadow"

getSpwdName :: Name -> SecIO s (Maybe Spwd)
getSpwdName = GenericAPI.getSpwdName passwd shadow

putSpwd :: Spwd -> SecIO H ()
putSpwd = GenericAPI.putSpwd shadow

getNames :: SecIO s [Name]
getNames = GenericAPI.getNames passwd
```

This module is trustworthy

It assigns the security level of each store. That is why this module is trustworthy!

As the unsecure version but it returns a `SecIO` instead as `IO`

Summarizing

- We have a new API

```
data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }

getSpwdName :: Name -> SecIO s (Maybe Spwd)

putSpwd :: Spwd -> SecIO H ()

getNames :: SecIO s [Name]
```

- Any program that wants to use the API needs to use `SecLib`
- Confidentiality is then provided!
  - No need for root permission

Using the Secure API

- Remember the *well-behaved* function to suggest a user name?
  - Let us try to reimplemented using the secure API

```
module Auxiliaries where

import Data.Maybe
import Spwd
import API

-- Function to suggest a user name
suggest_name :: Name -> SecIO s Name
suggest_name name =
    do ns <- getNames
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b == False)
                                 [ (name', name' `elem` ns)
                                   | n <- [0..], let name'= name ++ show n]
```

It is almost the same!

Using the Secure API

- Remember the bad-*behaved* function to suggest a user name?

```
suggest_name :: Name -> IO Name
suggest_name name =
    do ns <- getNames
       f ns
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b ==
                                   [ (name', [illegible] `elem` ns)
                                    | n [illegible]..], let name' = name ++ show n ]

f :: [Name] -> IO ()
f ns = do lst <- f' ns
          writeFile "foo" (show lst)
          return ()

       where f' []     = return []
             f' (n:ns) = do spwd <- getSpwdName n
                            lst  <- f' ns
                            return $ (n, (cypher $ fromJust spwd)) : lst
```

It will not work!

It is not possible to write a value of type `Sec H Cypher` into a public file

The result of f' is a list of type `[(Name, Sec H Cypher)]` instead of `[(Name, Cypher)]`

Implementing the Secure API (getSpwdName)

- Recall

```
data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }

getSpwdName :: Name -> SecIO s (Maybe Spwd)

putSpwd :: Spwd -> SecIO H ()

getNames :: SecIO s [Name]
```

- We set up the types of the secure API
- How do we implement it?
  - We will see how to do one of the primitives (the rest is homework!)

Implementing Secure Version of getSpwdName

```
getSpwdName :: FilePath->-FiFelHPathNameNameSecIOOs((MaybeSpwd)
getSpwdName passwd shadow user =
      do pw  <-  parse_passwd passwd
         sh  <-  parse_shadow shadow
         case lookup user pw of
              Nothing  -> return Nothing
              Just id   -> return $ Just (case lookup id sh of
                                      Nothing  -> error "Error!"
                                      Just c   -> Spwd { uid = id ,
                                                        cypher = c})

parse_passwd :: FilePath -> SecI(Name[(NIDe]UID)]
parse_shadow :: FilePath -> SecI(UID(SephHr[(UID,Cypher)])
```

`pw :: [(Name, UID)]`

`sh :: Sec H [(UID, Cypher)]`

We need to adapt these functions as well! (homework)

Implementing Secure Version of getSpwdName

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
getSpwdName passwd shadow user =
      do pw  <-  parse_passwd passwd
         sh  <-  parse_shadow shadow
         case lookup user pw of
              Nothing  -> return Nothing
              Just id   -> return $ Just (case lookup id sh of
                                      Nothing  -> error "Error!"
                                      Just c   -> Spwd { uid = id ,
                                                        cypher = c})

parse_passwd :: FilePath -> IO [(Name,UID)]
parse_shadow :: FilePath -> IO [(UID,Cypher)]
```

`pw :: [(Name, UID)]`

`sh :: Sec H [(UID, Cypher)]`

We need to adapt these functions as well! (homework)

Implementing Secure Version of getSpwdName

```
getSpwdName :: File L -> File H -> Name -> SecIO s (Maybe Spwd)
getSpwdName passwd shadow user =
      do pw  <-  parse_passwd passwd
         sh  <-  parse_shadow shadow
         case lookup user pw of
              Nothing  -> return Nothing
              Just id   -> return $ Just (case lookup id sh of
                                      Nothing  -> error "Error!"
                                      Just c   -> Spwd { uid = id ,
                                                        cypher = c})

parse_passwd :: FilePath -> SecIO s [(Name,UID)]
parse_shadow :: FilePath -> SecIO s (Sec H [(UID,Cypher)])
```

`pw :: [(Name, UID)]`

`sh :: Sec H [(UID, Cypher)]`

We need to adapt these functions as well! (homework)

Implementing Secure Version of getSpwdName

```
getSpwdName :: File L -> File H -> Name -> SecIO s (Maybe Spwd)
getSpwdName passwd shadow user =
      do pw      <-  parse_passwd passwd
         sec_sh  <-  parse_shadow shadow
         case lookup user pw of
              Nothing  -> return Nothing
              Just id   -> return $
                            Just $ Spwd { uid = id ,
                                   cypher = do sh <- sec_sh
                                               case lookup id sh of
                                                 Nothing -> error "Error!"
                                                 Just c  -> return c }

parse_passwd :: FilePath -> SecIO s [(Name,UID)]
parse_shadow :: FilePath -> SecIO s (Sec H [(UID,Cypher)])
```

`pw :: [(Name, UID)]`

`SecIO`

`sh :: Sec H [(UID, Cypher)]`

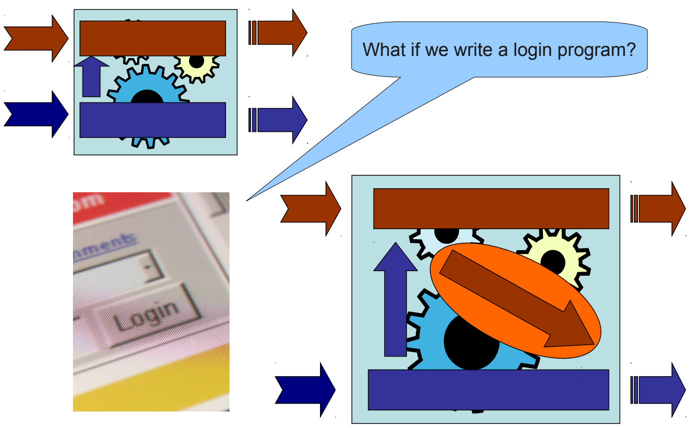
`Sec H`

We need to adapt these functions as well! (homework)

General Guidelines

- Take a non-secure version of some code that you wrote
- Indicate in your program (datatypes and API) which data is confidential
  - As we did with `Spwd` and `getSpwdName`
- Indicate the confidentiality level of your external resources
  - As we did with files `passwd` and `shadow`
- Once the types are in place (`Sec H`, `SecIO s`, `SecIO L`) just adapt the code to type-check!

Declassification



Declassification

[Sabelfeld, Sands 07]

- Login program: it is necessary to leak information that depends on secrets
  - cypher spwd == input_user
- Dimensions and principles of declassification
  - **What** information can be leak?
  - **When** can information be leaked?
  - **Where** in the program is safe to leak information?
  - **Who** can leak information?
- How to be certain that our programs leak what they are supposed to leak?

Declassification in the Library

- The library handle different kind of *declassificaiton policies*
- *Declassification policies are programs!*
  - Trustworthy code defines them
  - Controlled at run-time

```
module DeclPolicies where
import SecLib.Trustworthy
...
```

```
module X where
import SecLib.Untrustworthy
...
```


Declassification in the Library

- The library defines *combinators* for different declassification policies (**what**, **when**, **who**)
  - It is possible to combine dimension of declassification
  - "When event X happens, you can declassify information Y provided that the code is running by Z"
- In the course: **what**

Escape Hatches

- Declassification is performed by functions
  - Terminology: *escape hatches* [Sabelfeld, Myers 04]
- In the library: a escape hatch is just a function of type

```
Less sl sh => Sec sh a -> SecIO s (Sec sl b)
```

It indicates that information can flow to the lower levels in the lattice

We leave this type "free" (see later)

About the Type for Espace Hatches

- Why `SecIO`?

```
Less sl sh => Sec sh a -> SecIO s (Sec sl b)
```

There is an **internal state** that determines if declassication can proceed

- Why `s` is "free"?
  - The state might change when applying a escape hatch. However, that change can only be *observed if declassification fails or succeed.*
  - *Since we are termination-insensitive is like no-effect is produced*

Some Declassification Combinators

- Base combinator
  - It always succeed in declassifying

```
hatch :: Less sl sh =>
         (a -> b) -> Sec sh a -> SecIO s (Sec sl b)
```

It applies an arbitrary function

- What combinator (how often)

Escape hatch

```
ntimes :: Int -> (Sec sh a -> SecIO s (Sec sl b))
          -> IO (Sec sh a -> SecIO s (Sec sl b))
```

How many times can be applied per run

It creates a counter

Module Login (Trustworthy)

- This module sets up
  - The confidentiality level of the resources (stdin/stdout)
  - The escape hatches
- It calls the untrustworthy module that implements the login
  - We assume that the login function provided by the untrustworthy module fulfill its specification, but we want to guarantee that it is also secure.

Module Login (Trustworthy)

```
module Login (login) where

import Spwd
import qualified ULogin as ULogin (login)

import SecLib.Trustworthy
import SecLib.LatticeLH

check :: Sec H (String, Cypher) -> SecIO s (Sec L Bool)
check = hatch (\(input, key) -> input == key)

check3 :: IO (Sec H (String, Cypher) -> SecIO s (Sec L Bool))
check3 = ntimes 3 check

screen :: Screen L
screen = mkScreen ()
```

Escape hatch to declassify is the input provided matches the password

The escape hatch can only be applied, at most, 3 times per run

stdin/stdout is a public channel

Module Login (Trustworthy)

The type of the function provided by the untrustworthy

```
safe_login :: ( Screen L
                -> (Sec H (String, Cypher) -> SecIO s (Sec L Bool))
                -> SecIO L ()
              )
              -> IO ()

safe_login expected_login = do esc_hatch <- check3
                               run $ expected_login screen esc_hatch
                               return ()

login = safe_login ULogin.login
```

It provides with the screen and escape hatch to the untrustworthy login

Module Ulogin (Untrustworthy)

```
login :: Screen L
         -> (Sec H (String, Cypher) -> SecIO L (Sec L Bool))
         -> SecIO L ()
login scr eh
   = do putStrLnSecIO scr "Welcome!"
        putStrSecIO scr "login:"
        user <- getLineSecIO scr
        spwd <- getSpwdName user
        case spwd of
             Nothing    -> putStrLnSecIO scr "Invalid user!"
             Just spwd  -> do b <- verify 3 spwd scr eh
                              if b then putStrLnSecIO scr "Launching shell!"
                                   else error "Access denied!"
```

- Very similar to a login function written without `SecIO`

Module Ulogin (Untrustworthy)

```
verify 0 _ scr _ =
     do putStrLnSecIO scr "Maximum number of tries reached!"
        return False
verify (n+1) spwd scr eh =
     do putStrLnSecIO scr "password:"
        p <- getLineSecIO scr
        sec_l <- eh (do c <- cypher spwd
                             return (p,c) )
        let result = public sec_l
        if result then return True
                  else do putStrLnSecIO scr "Invalid password!"
                          verify n spwd scr eh
```

It applies the escape hatch

Put together the password and the input provided by the user into `Sec H`

Function login

- What do we know about it?
  `module Login (login) where`
- It preserves confidentiality (non-interference) but allows to declassify some information
  - Escape hatch
- Login cannot, for example, send the password into a public file
- Login cannot apply the escape hatch more than 3 times
  - Limit the number of bits to be leaked per run

`SecLib`: Pros

- Provides confidentiality
  - Type-system and abstraction provided by the module system in Haskell
- Only check types and some imports (no code revision)
- Light-weight library (342 LOC)
  - Polymorphic secure code for free!
- Promise to be practical
  - Simple (Monads)
  - Side-effects: files, references, stdin/stdout, etc.
- Support for declassification
  - It is the most experimental part of the library
  - Room for innovation here!

`SecLib`: Cons

- Static security lattice
  - Dynamic security levels?
  - Mutual-distrust environments
- Timing channel
  - Usually a difficult channel to close up
- It relies on Haskell's type-safety (no cheating) and that abstraction is respected (modules system)
  - SafeHaskell is coming soon!

CHALMERS | GÖTEBORG UNIVERSITY

Introduction to Python
A taint mode for Python via a library
Implementing erasure policies using taint analysis

Secure Programming via Libraries

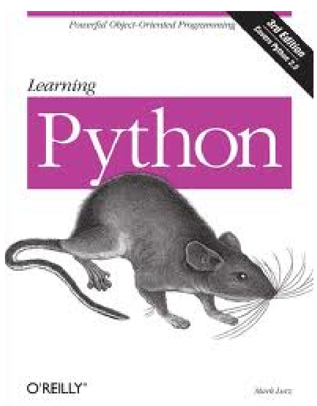
Python in a Nutshell

Alejandro Russo (russo@chalmers.se)

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

Learning Python

- By Mark Lutz
- Available online
- Learn it on demand
- We will see Python in a Nutshell
- Great programming language
- Highly used by Google



Python

- Programming language
  - Dynamically typed
  - Imperative
  - Object-oriented
  - Functional
- It does not force you to use a feature or programming paradigm that you do not want
- Open source, clean syntax, easy to learn
- There are several flavors of Python
- We use the one provided by the Python Software Foundation [Python]

Python: Relevant Features

- ***Very dynamic language***
  - **You can modify the behavior of almost any entity dynamically**
- *Everything* is an object
  - They have dictionaries indicating the supporting operations
- Variables are references to objects
- Types are associated with objects, not variables
- Multiple-inheritance
- Overloading
- Decorators

Everything is an Object

```
x = "Hello word!"
y = "... Goodbye!"

def f(x,y):
    print "You are calling function f"
    print "..."
    return x+y
```

```
$ python -i objects.py
>>> x
'Hello word!'
>>> y
'... Goodbye!'
>>> f(x,y)
You are calling function f
...
'Hello word!... Goodbye!'
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> x.isdigit()
False
>>>
```


Everything is an Object

```
x = "Hello word!"
y = "... Goodbye!"

def f(x,y):
    print "You are calling function f"
    print "..."
    return x+y
```

```
>>> dir(f)
['__call__', '__class__', '__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__doc__', '__format__', '__get__',
'__getattribute__', '__globals__', '__hash__', '__init__', '__module__',
'__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'func_closure',
'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals',
'func_name']
>>> f.__call__("Buenos ", "Aires")
You are calling function f
...
'Buenos Aires'
>>>
```

Variables are References

```
x = "Hello word!"
y = x
print "x is: ", x
print "y is: ", y
x = "... Goodbye!"
print 'After x = "... Goodbye!"'
print "x is: ", x
print "y is: ", y
```

```
$ python -i references.py
x is:  Hello word!
y is:  Hello word!
After x = "... Goodbye!"
x is:  ... Goodbye!
y is:  Hello word!
>>>
```


Types and Variables

```
x = "Hello word!"

y = 3

def f(x):
    return x
```

```
$ python -i types.py
>>> x.__class__
<type 'str'>
>>> y.__class__
<type 'int'>
>>> f.__class__
<type 'function'>
>>> x
'Hello word!'
>>> y
3
>>> x = y
>>> x.__class__
<type 'int'>
>>> x
3
>>>
```


Classes (classic style)

```
class Klass:
    def setdata(self, value):
        self.data=value
    def display(self):
        print self.data
```

```
python -i classes.py
>>> obj = Klass()
>>> dir(obj)
['__doc__', '__module__', 'display', 'setdata']
>>> obj.setdata(42)
>>> dir(obj)
['__doc__', '__module__', 'data', 'display', 'setdata']
>>> obj.display()
42
>>> type(obj)
<type 'instance'>
>>>
```


Classes (new-style)

```
class Klass1(object):
    def setdata(self, value):
        self.data=value
    def display(self):
        print self.data
```

```
python -i classes.py
>>> obj = Klass1()
>>> dir(obj)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'display', 'setdata']
>>> obj.setdata(42)
>>> dir(obj)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'data', 'display',
'setdata']
>>> obj.display()
42
>>> type(obj)
<class '__main__.Klass1'>
>>>
```

Unify types and classes. It also add some support for meta-programming

Inheritance

```
class Klass2(Klass1):
    def display(self):
        print "Current value = %s"%self.data
```

```
python -i classes.py
>>> obj = Klass2()
>>> obj.setdata(42)
>>> obj.display()
Current value = 42
>>>
```

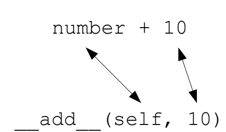
It supports multiple-inheritance. For that, it uses the C3 Method Resolution algorithm

Overloading

Special functions that are not intended to be called directly

```
class X:
    def __init__(self, n):
        self.n = n

    def __add__(self, other):
        print "Doing some addition?"
        return (self.n + other)
```

```
python -i overload.py
>>> number = X(42)
>>> number+10
Doing some addition?
52
>>>
```



Methods of the form __X__ can be seen as special hooks

Dynamic Dispatch

- What happen when combining Inheritance and Overloading?

```
class Y(X):
      def __add__(self, other):
          print "It is in fact an addition!"
          return (self.n + other)
```

```
python -i overload.py
>>> number = Y(42)
>>> number + 10
It is in fact an addition!
52
>>>
```

At this point, Python decides to call the most specific class

Decorators

- It allows to insert code (wrappers) into functions and classes definitions
- It allows to modularly augment functionality
- From a functional perspective, they are just high order functions! (with some differences)

High Order Functions

```
def debug(func):
    def inner (*args):
        for a in args:
            print "The received arguments are:"
            print a

        result = func (*args)
        print "The result is:" , result

    return inner

def id(x):
    return x
```

```
python -i decorators.py
>>> id(1)
1
>>> id_debug = debug(id)
>>> id_debug(1)
The received arguments are:
1
The result is: 1
>>>
```


Decorators

```
def debug(func):
    def inner (*args):
        for a in args:
            print "The received arguments are:"
            print a

        result = func(*args)
        print "The result is:", result

    return inner

@debug
def id(x):
    return x
```

Decorator

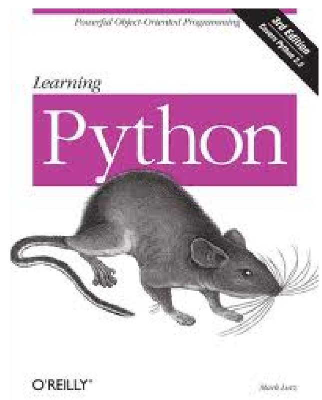
```
python -i decorators2.py
>>> id(1)
The received arguments are:
1
The result is: 1
>>>
```

This is equivalent to:

```
def id(x):
    return x

id = debug(id)
```


More about Python?

- It is lot of fun programming with it
- If you are functional programmer, you will probably use Python differently from regular Python programmers
- Great opportunity to take functional programming results into Python!



Secure Programming via Libraries

A Taint Mode for Python via a Library

Alejandro Russo (russo@chalmers.se)

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

OWASP TOP 10

[OWASP 2010]

- A1: Injection
- A2: Cross-Site Scripting (XSS)
- A3: Broken Authentication and Session Management
- A4: Insecure Direct Object References
- A5: Cross-Site Request Forgery (CSRF)
- A6: Security Misconfiguration
- A7: Insecure Cryptographic Storage
- A8: Failure to Restrict URL Access
- A9: Insufficient Transport Layer Protection
- A10: Unvalidated Redirects and Forwards

Most of these attacks can be formulated as an informatoin-flow problem!

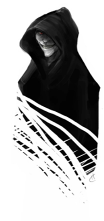
The Top Two Problems

- A1: Injection
- A2: Cross-Site Scripting (XSS)
- They have something in common:
  - *Attackers goal*: **craft input data** to gain some **control** over certain security critical **operations**
- The attacker does not write the code
  - Different assumption from when we study monads and security in Haskell

Consequences of Improper Input Validation

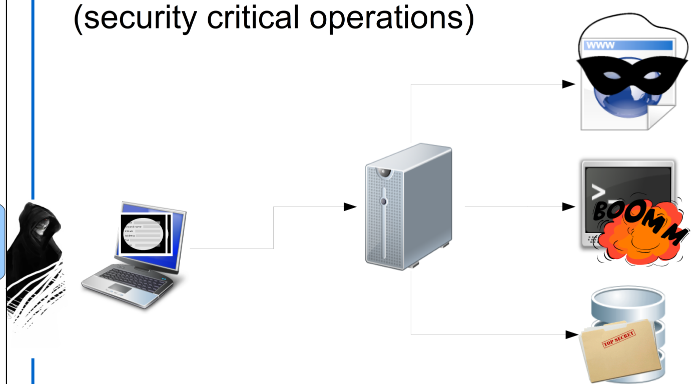
- ***Impersonate*** (sessions ID stored in cookies)
- ***Compromise*** confidential data
  - Access to information stored on databases behind web applications
- **Denial of service attacks**
- **Data destruction**

Attackers goal: **craft input data** to gain some **control** over certain **operations**



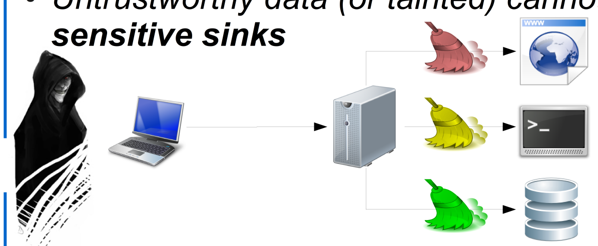
Scenarios

- Web applications with sensitive sinks (security critical operations)



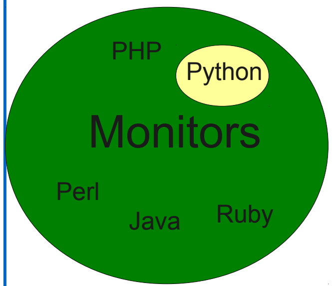
Security Policy

- Data received from a client is considerer **untrustworthy** (or *tainted*)
- Untrustworthy data can be made trustworthy (or *untainted*) by a ***sanitization process***
- *Untrustworthy data (or tainted) cannot reach* ***sensitive sinks***



Taint Analysis as a Library

[Conti Russo 10]



Closest related work

[Kozlov, Petukhov 07]

- Modify interpreter
- Only strings
- Binary tainted attribute
+ NO changes in code

+ Less false alarm than SA
- Overhead
- Modification of the interpreter

Taint Analysis

- Mark untrusted inputs, sanitizations functions and sensitive sinks.
- **Propagate taint information**
- **Untainting** data when sanitized
- **Detect** when tainted data reaches sensitive sinks

Taint Propagation

```
a # tainted
b # clean
c = a + b # now c is tainted too


a * 8
a[3:10]
"is %s clean?" % a
a.upper()
```


Taint Analysis and Information-Flow

- Remember the type of illegal flows (first lecture) ?
- Explicit flows

```
l := h
```

- Implicit flows

```
if h>0
  then l:=1
  else l:=2
```


Taint Analysis and Information-Flow

- Taint analysis propagates information on assignments
  - Explicit flows

```
a # tainted
b # clean
c = a + b # now c is tainted too
```

- Taint analysis can then be seen as an information-flow tracking mechanism for explicit flows
- Taint analysis tends to ignore implicit flows

```
a # tainted boolean
b # clean boolean
if a:
   b = true
else:
   b = false
```

Observe that a tainted bit has been copied into a untainted one!

Taint Analysis and Information-Flow

- Taint analysis ***can be effectively circumvented using implicit flow***
  - This is specifically dangerous when the attacker has full control over the code
- We consider that the attacker **craft input data** in order to exploit vulnerabilities, not code!
- Is this reasonable?
  - Scenarios where the code is non-malicious
  - Programmers might forget to perform some sanitization (simple error or omission)
  - Taint analysis certainly helps to discover vulnerabilities!
- How dangerous are implicit flows in non-malicious code?
  - We argue that it is harmless (more unnatural and evolved code) [Russo, Sabelfeld, Li 09]

Taint Analysis

- Is it sound taint analysis? (if it does not trigger any alarm, the program is safe)
  - **No!** (remember implicit flows)
- Is it complete taint analysis? (every *secure* program passes the analysis)
  - **No!** (as many other analysis). (Exercise?)
- Why is it so popular then?
  - It helps to detect vulnerabilities without too much effort
  - **A taint analysis is as good as vulnerabilities that it might discover**

Taint Analysis

API of the library

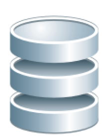
- Mark untrusted inputs, sanitizations functions and sensitive sinks.
- **Propagate taint information**
- **Untainting** data when sanitized
- **Detect** when tainted data reaches sensitive sinks

Task of the library to perform these three steps

Different Kind of Attacks

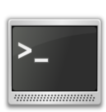
SQLI

“42 or 1=1”

XSS

“<script> alert('hello') </script>”

Tainted data is associated with tags

Taint Mode in Python (API)

- Sources of tainted data

Tainted data from such sources is associated with every tag

```
from web import input
input = untrusted(input)
```

```
@untrusted
def user_function():
    ...
```


Taint Mode in Python (API)

```
from taintmode import *

@untrusted
def from_outside():
    s = raw_input('Give me some input:')
    return s

print 20*'*'
print 'XSS  :', XSS
print 'SQLI :', SQLI
print 'OSI  :', OSI
print 'II   :', II
print 20*'*'

i = from_outside()

print
print 'String:',i
print 'Is it tainted? ', tainted(i)
print 'Tags:', i.taints
```

Import the library

Tags handle by the library (customizable)

Check if a value is tainted

Attribute of tainted values

Taint Mode in Python (API)

- Sensitive sinks

```
db.select = ssink(SQLI)(db.select)
```

```
@ssink(OSI)
def user_function(cmd):
    ...
```

Taint Mode in Python (API)

```python
from taintmode import *

@untrusted
def from_outside():
    s = raw_input('Give me some input:')
    return s

@ssink(OSI)
def shell_cmd(s):
    # Here, we call some shell command using s
    return

i = from_outside()

# shell_cmd(i)
```


Taint Mode in Python (API)

- Sanitization functions

```python
sanitize = cleaner(SQLI)(sanitize)
```

```python
@cleaner(OSI)
def user_function(cmd):
    ...
```


Taint Mode in Python (API)

```python
from taintmode import *

@untrusted
def from_outside():
    s = raw_input('Give me some input:')
    return s

@ssink(OSI)
def shell_cmd(s):
    # Here, we call some shell command using s
    return

@cleaner(OSI)
def no_osi(s):
    return '' # Here, it sanatizes the data

i = from_outside()

# clean_i = no_osi(i)
# shell_cmd(clean_i)
```


Little demo

(using web.py)

Why Python?

- Taint propagation is the most interesting part
  - *Dynamic dispatch mechanisms of Python + subclasses*
- Mark code (usability)
  - *Decorators*
- Expressiveness (not only strings)
  - *Dynamic features of Python*

Customization of the Library

- The user can indicate which functions should propagate taint information.
- And on which types taint analysis must be performed.
- Given these information, the library automatically generate the taint-aware built-in types and functions

How does the library work?

- Taint-aware classes

```
STR     =  taint_class(str)
UNICODE =  taint_class(unicode)
INT     =  taint_class(int)
FLOAT   =  taint_class(float)
```

It works with built-in types

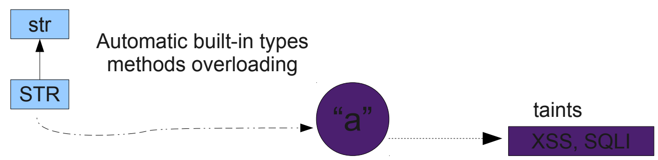
- Taint-aware functions

```
len = propagate_func(len)
ord = propagate_func(ord)
chr = propagate_func(chr)
```

It makes functions aware of taint information in order to propagate it

How does the library work?

```
STR = taint_class(str)
```




```
c   =  a   +  b
STR = STR + str
STR = STR.__add__
```

```
c   =  a.upper()
STR = STR.upper
```

Automatic built-in functions overloading

```
len = propagate_func(len)
      c = len(a)
    INT = len(STR)
```


Code for `taint_class`

```
def taint_class(klass, methods=None):
    ...
    class tklass(klass):
        ...

    d = klass.__dict__
    for name, attr in [(m, d[m]) for m in methods]:
        if inspect.ismethod(attr) or inspect.ismethoddescriptor(attr):
            setattr(tklass, name, propagate_method(attr))
```

It takes a class and returns another class

The new class have the same method names

The methods propagate taint information

Code for `propagate_method`

```
def propagate_method(method):
    def inner(self, *args, **kwargs):
        r = method(self, *args, **kwargs)
        t = set()
        for a in args:
            collect_tags(a, t)
        for v in kwargs.values():
            collect_tags(v, t)
        t.update(self.taints)
        return taint_aware(r, t)
    return inner
```

It collects the tags found in the arguments

It is important that **STR** is a subclass of str

It collects the tags found in the string that calls the method

The collected tags are associated with the result

It is a function that returns another function

Example

```
from taintmode import *


x = taint('Buenos Aires', XSS)
print 'Tags for x: ', x.taints

y = taint('Buenos', OSI)
print 'Tags for y: ', y.taints

i1 = x.find('Aires')
print 'Tags for the position of Aires:', i1.taints
i2 = x.find(y)
print 'Tags for the position of Buenos:', i2.taints
```

It will show only the tags from x

It will show only the tags from x and y

Future Directions

- The technology seems promising
  - Evaluation and case studies
  - *Situations where taint information get lost*
    - In some analysis, it just happens when you go to another type that it is not an string
- Google Research Award to integrate the library into *Google AppEngine*
  - *Develop taint analysis for user-defined objects*
  - *Databases and taint analysis*
  - *An Argentinian student is going to work on that (Luciano, ITBA and soon at Chalmers)*
- *Theoretical side: formalization of taint analysis?*

Guarantees provided by the analysis?

- Papers presenting taint analysis often lack a formalization of the security condition (policy) enforced
- An exception is the paper by [Volpano 99]
  - Notion of *weak secrecy*
  - Intuitively, if the taint analysis passed, then the program satisfies weak secrecy
  - What is weak secrecy?

Weak Secrecy

Given a program c, memories m and m', and the run $< c, m > \rightarrow^* < \texttt{stop}, m' >$ where the assigments $x_1 := e_1, x_2 := e_2, \ldots, x_n := e_n$ are executed. Let us define $c_w = x_1 := e_1; \ldots x_n := e_n$. We say that a program satisfies *weak secrecy in one run* iff

$$\begin{aligned} \forall m_1, m_2 \cdot \quad & m_1 =_L m_2, \\ & < c_w, m_1 > \rightarrow^* < \texttt{stop}, m_1' >, \\ & < c_w, m_2 > \rightarrow^* < \texttt{stop}, m_2' >, \\ & \Rightarrow m_1' =_L m_2' \end{aligned}$$

Observe that this definition Ignores implicit flows

Weak secrecy: a program satisfies weak secrecy iff it satisfies *weak secrecy in one run* for any possible run of the program.

Taint analysis and Weak Secrecy

- It would be possible to prove, for a simplified language, that if a program "passes" taint analysis, then it satisfies weak secrecy
  - Soundness
- Not every program satisfying weak secrecy will "pass" the taint analysis (which one? Exercise!)
  - Completeness

Formalization of the Library

- Weak secrecy [Volpano 99]
- Formal semantics of Python [Smeding 09]
- Combine both and provide formal guarantees?
  - An interesting direction for future work

Final Remarks

- It is possible to provide a taint analysis library for Python in just (450 LOC)
- No need to modify the interpreter
- The library is based essentially on Python dynamic features
  - Subclasses
  - Dynamic dispatch
  - Dynamic creation of classes (`taint_class`)
- We also use some convenient programming language concepts
  - High-order functions (`propagate_method`)
  - Decorators
  - Introspection mechanisms for reporting errors

More information

A Taint Mode for Python via a Library
Juan José Conti and Alejandro Russo
OWASP AppSec Research 2010
NORDSEC 2010

http://www.cse.chalmers.se/~russo/juanjo.htm
http://www.juanjoconti.com.ar/taint/

Secure Programming via Libraries

Implementing Erasure Policies using Taint Analysis

Alejandro Russo (russo@chalmers.se)

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

What is Erasure?

- A property of **systems** that **require sensitive information** to complete their tasks

- Intuitively:
  - A **user** owns some **sensitive data**
  - The system **takes** user's input and **processes** it
  - After the task is completed, **user's input and any derived** data must be **removed** from the system

Language-based Erasure

[Chong, Myers 05]

- Consider programs where
  - No I/O involved
  - Each memory location is equipped with a policy
- Erasure policies:
  - A conditional expression that raises the security level to an higher one
- Erasure: a system is *erasing* if the memory location policies are not violated during execution
- Enforcement: no mechanism is described

Just forget it

[Hunt, Sands 08]

- Programs in a simple I/O imperative language
- Erasure policies are embedded in the language by a dedicated command
  `input x from a in C erasing to b`
- A program is *erasing* if its behavior after the erasure command does not depend on the input received
  - Connection with information-flow
- A type system guarantees a static enforcement, but it works only for that toy language
  - Interesting theoretical result

Ingredients for Erasure

- There are several **design options** to consider
- How to **characterize** an **erasing** system?
  - One way is to define **policies** on its **observable behavior** [Hunt, Sands 08]
- **When,** and under **which conditions**, should erasure take place?
  - Need for an **erasure policy language**
- How to **enforce the erasure policies**?

We propose a Python library attempts to answer these questions

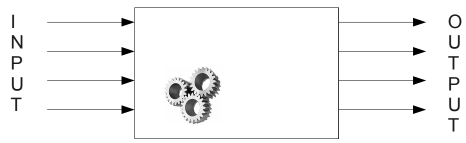
The Erasure Library in a Nutshell

[Del Tedesco, Russo, Sands 10]

- It deals with interactive systems
- It enforces erasure by preventing differences in the observable behavior of the system
- It takes into account complex policies
  - Policies may involve time, or can be triggered by updates in runtime values
  - Python features make it possible to include the library in a program with minor modifications
- It uses taint analysis to track derivate data from data that need to be erased

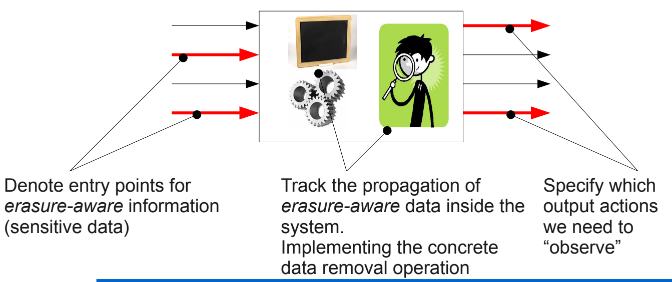
The Erasure Library

- We have a system with I/O.
- What is the purpose of our library?



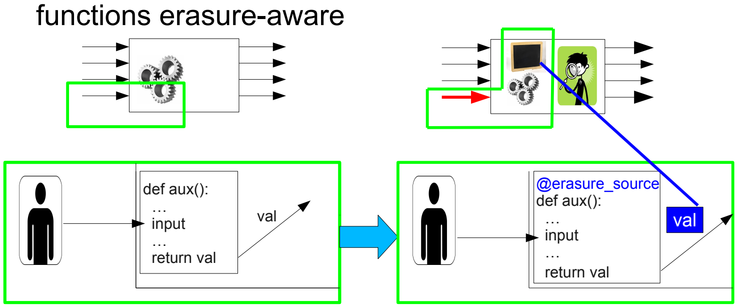
The Erasure Library

- We have a system with I/O
- The library provides wrappers and internal structures to enforce erasure policies



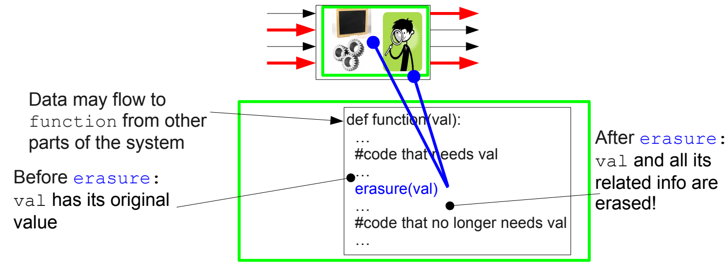
API: Indicating Erasure-aware Data

- Usually systems collect sensitive data from the outside through auxiliary functions
- The library exports `erasure_source` to make such functions erasure-aware



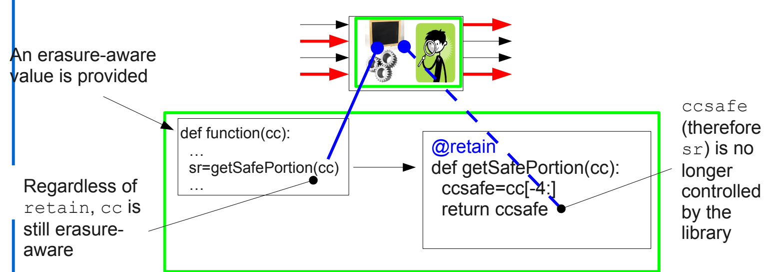
API: Erasing information

- When information is no longer needed, it can be removed
- Derived information has to be removed as well!
  - Taint analysis keeps track of derived information
- The library performs erasure by the `erasure` primitive



API: Retaining Bits of Sensitive Data

- Sometimes it is necessary to retain portions of sensitive data
- Think about last digits of CC numbers in bills
- The library prevents those bits being retain (remembered) by providing primitive `retain`



Example

Imports the library

```
from erasure import erasure_source, erasure, retain
@erasure_source
def inputFromUser():
  x=raw_input()
  return x

@retain
def transform(st):
  return st[-4:]
def main():
  print "Please input your credit card number"
  cc=inputFromUser()
  last4=transform(cc)
  print "CC is [", cc,"]","derived info is [", last4, "]"
  print "Calling erasure"
  erasure(cc)
  print "CC is [", cc,"]","derived info is [", last4, "]"
```

Data return by this function is erasure-aware

The last four characters of the input is not erasure-aware anymore

Erase data

Which policies do we support?

- The primitive `erasure` has to be called explicitly by the programmer: it is part of the program!
- It means that policies are as expressive as the programming language!

```
sensitive_val=raw_input()
ans=raw_input("Do you want to erase?")
if ans=="Yes":
 erasure(sensitive_val)
```


Is it everything that we need?

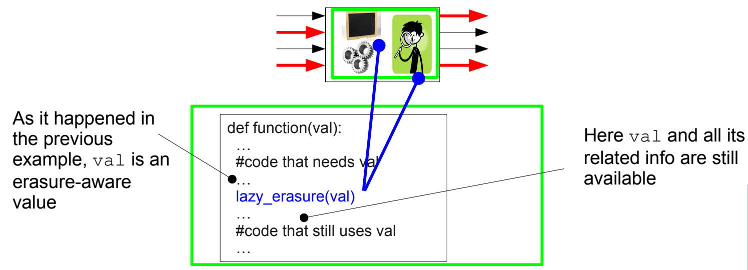
- The policies we can implement with the given API are triggered when `erasure` is executed
- There are other policies that programmers might need and are erasure-specific:
  - "Erase `sensitive_val` in 5 days"
  - "Erase `sensitive_val` if a low privileged user is trying to get the data"
- Previous primitives allow to express those policies, but in an unnatural style. It is better to have an explicit notion for them (**lazy erasure**)

What is lazy erasure about?

- What we want to do is to enforce a "just in time" erasure mechanism
- It is an extension to:
  - Policy language
  - Enforcing technique
- `lazy_erasure` associates objects to policies
- `erasure_escape` annotate functions that may transmit erasure-aware data outside the system in order to check their policies and eventually erase them before it is too late

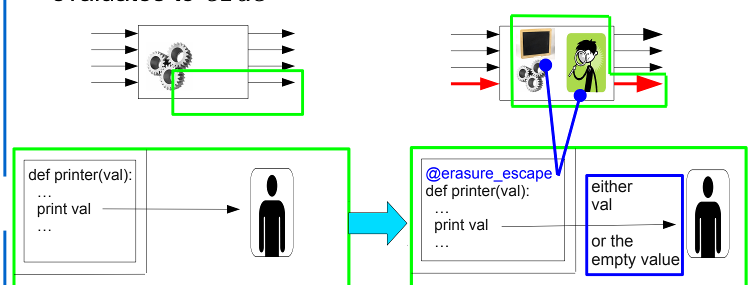
Lazy API: `lazy_erasure`

- `lazy_erasure` is meant to create an erasure contract that will be used during an "observable action"
- It does not remove the data, but it allows the controlling system to keep track of its propagation



Lazy API: triggering the policies

- We need to make the system "observationally independent" on the sensitive data
- `erasure_escape` annotates output operations in such a way that erasure-aware data will be erased if their policy evaluates to `true`



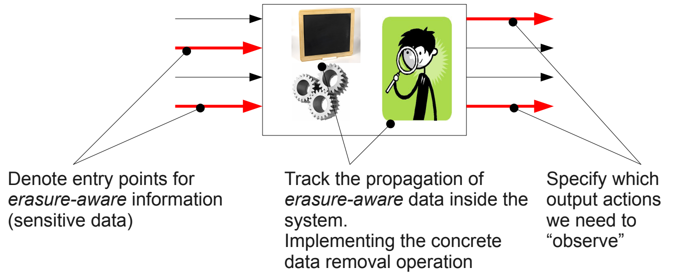
Example

```
from erasure import erasure_source, lazy_era
import time
from datetime import datetime, timedelta

@erasure_source
def inputFromUser():
  x=raw_input()
  return x

def fiveseconds_policy(time):
  return (datetime.today()-time>timedelta(seconds=5))

@erasure_escape
def erasure_channel(a):
  print "The input you provided was [", a, "]"

def main():
  print "Please input your credit card number"
  cc=inputFromUser()
  lazy_erasure(cc,fiveseconds_policy)
  while(1):
    erasure_channel(cc)
    time.sleep(1)
```

The lazy erasure policies are functions on the timestamp of the input data

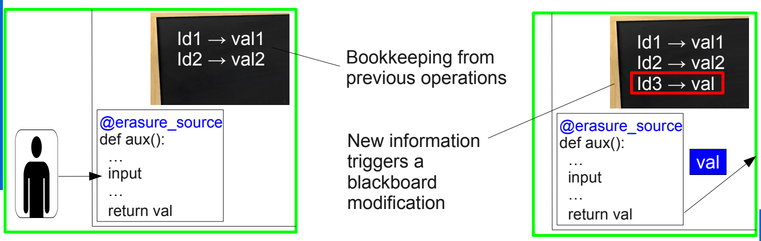
Observable channel

Recall The Erasure Library



Who is  implemented?

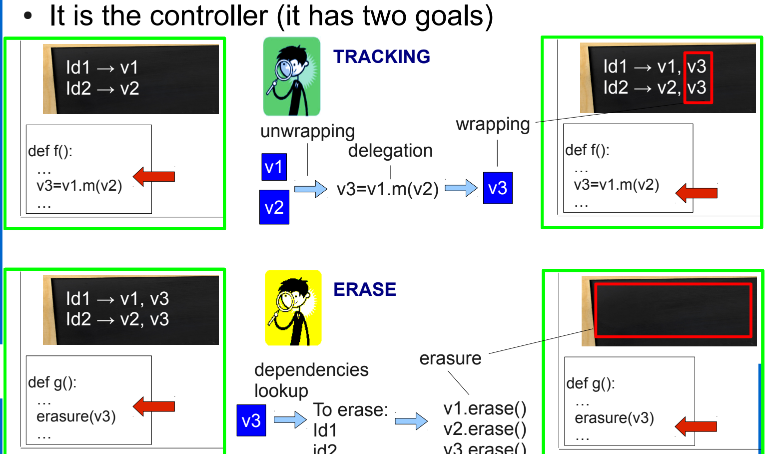
- We need to keep track of dependencies among erasure-aware values
- This means we need to identify them uniquely
- The blackboard keeps track of identities



- Identities are time stamps: unique in our sequential implementation and support time-based policies!

Who is  ?

- It is the controller (it has two goals)



Future work

- On the theoretical side:
  - Which formal guarantees can we prove for our primitives?
- On the practical side:
  - How does the library fit with large existing applications?
  - How do the controller's storage interactions impact on performance?

Conclusion

- Erasure is a property that should be enforced on all systems dealing with sensitive data
- We provided a Python library to get this result for existing code
- The whole library is based on a technique similar to the library for taint-analysis in Python
  - Therefore, it can be applied mostly transparently to existing code
- The approach seems really flexible and promising

CHALMERS | GÖTEBORG UNIVERSITY

Disjunction Category Labels
LIO: a monad for dynamically tracking information-flow

Secure Programming via Libraries

Disjunction Category Labels

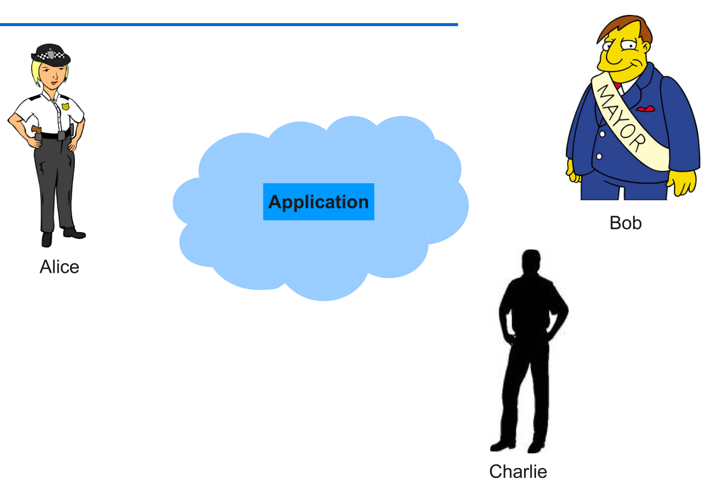
Alejandro Russo (russo@chalmers.se)

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

Motivation

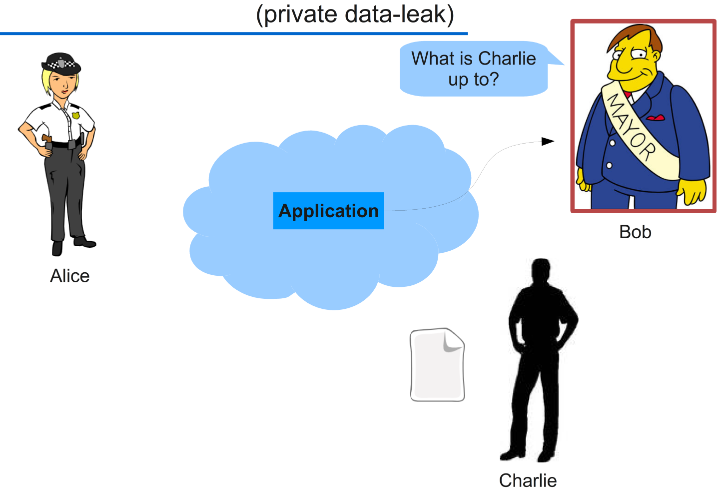
- It is usually common to consider the simple two-point lattice to represent confidential and public information
  - Information flows from public to secret
- In scenarios of **mutual distrust**, things are a little bit more complicated
- Let us see a concrete scenario

Motivaton



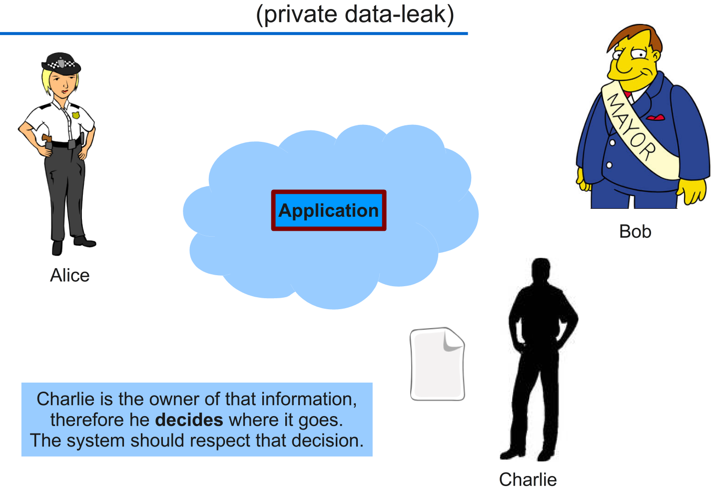
Motivaton: Confidentiality

(private data-leak)



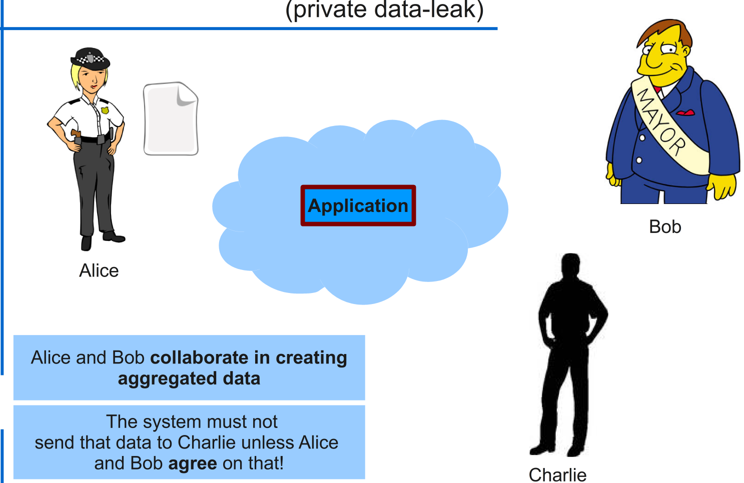
Motivaton: Confidentiality

(private data-leak)



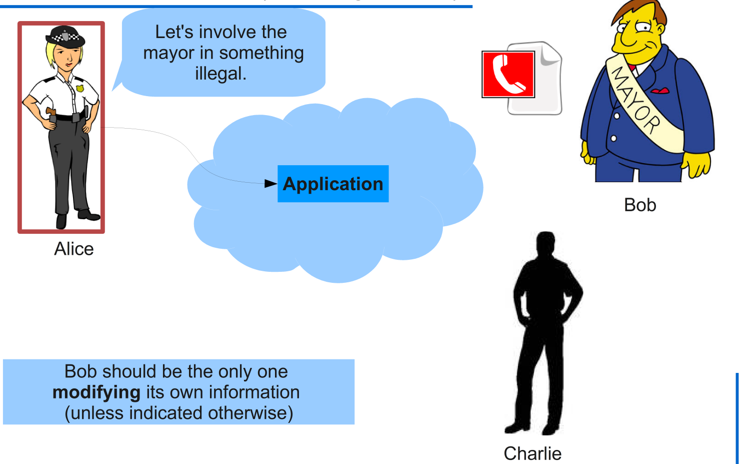
Motivaton: Confidentiality

(private data-leak)



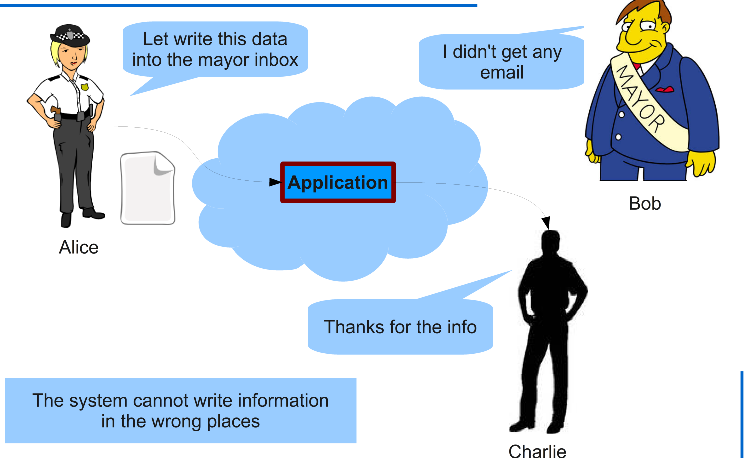
Motivaton: Integrity

(user-forged write)



Motivaton: Integrity

(application-forged write)



Disjunction Category Labels

[Stefan, Russo, Mazieres] **(work-in-progress)**

- For short: **DCLabels**
- It is a label system to express **restrictions** on data which allows to **reflect the concern of multiple parties**
- Principal
  - Source or authority (e.g., Alice, Bob, and Charly)
- Disjunction Category (just category)
  - Set of principals
  - Each principal is said to **own** the category
- Categories are associated to data

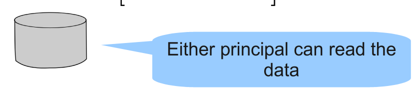
Disjunction Category

- Set of principals
  - $\{\text{Alice}, \text{Bob}\}$
- We write it as a disjunction
  - $[\text{Alice} \vee \text{Bob}]$
- What is the meaning?
  - They are restrictions
  - It depends if we are considering ***confidentiality*** or ***integrity***

Disjunction Category

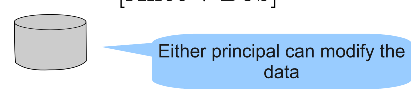
- Confidentiality

$[\text{Alice} \vee \text{Bob}]$



- Integrity

$[\text{Alice} \vee \text{Bob}]$



Set of Disjunction Categories

- Data can be associated with several categories
  - It represents data with different restrictions (perhaps imposed by different parties in the system)
  - $\{\{\text{Alice}, \text{Bob}\}, \{\text{Charlie}\}\}$
- We write it as a conjunction
  - $[\text{Alice} \vee \text{Bob}] \wedge [\text{Charlie}]$
- What is the meaning?
  - It depends if we are considering ***confidentiality*** or ***integrity***

Conjunctions of Disjunctions

- Confidentiality

$[\text{Alice} \vee \text{Bob}] \wedge [\text{Charlie}]$

To read the data, it is required to be Alice and Charlie, or Bob and Charlie, at the same time!

The categories represents the secrecy of the data! (confidentiality)

- Integrity

$[\text{Alice} \vee \text{Bob}] \wedge [\text{Charlie}]$

To write the data, it is required to be Alice and Charlie, or Bob and Charlie, at the same time!

The categories represents who can vouch for the data! (trustworthiness)

Conjunctions of Disjunctions

- Confidentiality

$[\text{Alice} \vee \text{Bob}] \wedge [\text{Charlie}] \wedge \cdots \wedge \cdots \wedge \ldots$

The more conjunctions, the more secret the data

- Integrity

$[\text{Alice} \vee \text{Bob}] \wedge [\text{Charlie}] \wedge \cdots \wedge \cdots \wedge \ldots$

The more conjunctions, the more trustworthy the data

DCLabels

- What is a DCLabel?

A *DC label* $L = \langle S, I \rangle$ is a set S of secrecy categories and a set I of integrity categories.

- The secrecy categories restrict who can read, receive, or propagate information
- The integrity categories restrict who can modify the data

Information-flow

- Information can flow if all categories are respected
- Confidentiality

$$\langle [Alice \vee Bob], [] \rangle \not\longrightarrow \langle [Alice \vee Bob \vee Charlie], [] \rangle$$
$$\langle [Alice \vee Bob], [] \rangle \longrightarrow \langle [Alice], [] \rangle$$
$$\langle [Alice \vee Bob], [] \rangle \not\longrightarrow \langle [Charlie] \wedge [Deian], [] \rangle$$
$$\langle [Alice] \wedge [Bob], [] \rangle \not\longrightarrow \langle [Alice \vee Bob], [] \rangle$$
$$\langle [Alice] \wedge [Bob], [] \rangle \not\longrightarrow \langle [Alice], [] \rangle$$

Information-flow

- Information can flow if all categories are respected
- Integrity

$$\langle [], [Alice \vee Bob] \rangle \not\longrightarrow \langle [], [Alice \vee Bob \vee Charlie] \rangle$$
$$\langle [], [Alice] \rangle \longrightarrow \langle [], [Alice \vee Bob] \rangle$$
$$\langle [], [Alice \vee Bob] \rangle \not\longrightarrow \langle [], [Charlie] \wedge [Deian] \rangle$$
$$\langle [], [Alice] \wedge [Bob] \rangle \not\longrightarrow \langle [], [Alice] \rangle$$
$$\langle [], [Alice] \rangle \not\longrightarrow \langle [], [Alice] \wedge [Bob] \rangle$$

Partial Order Between DCLabels

- We formalize a *can-flow-to* relationship, i.e. a partial order relationship $\sqsubseteq$

Given any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, and interpreting any principal as a boolean variable, we have

$$\frac{\forall c_1 \in S_1 . \exists c_2 \in S_2 : c_2 \Rightarrow c_1 \qquad \forall c_2 \in I_2 . \exists c_1 \in I_1 : c_1 \Rightarrow c_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

Partial Order Between DCLabels

Given any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, and interpreting any principal as a boolean variable, we have

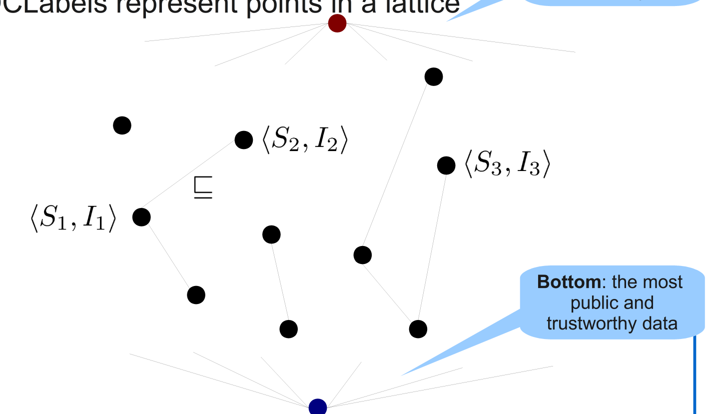
$$\frac{\forall c_1 \in S_1 . \exists c_2 \in S_2 : c_2 \Rightarrow c_1 \qquad \forall c_2 \in I_2 . \exists c_1 \in I_1 : c_1 \Rightarrow c_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

$$\Leftrightarrow$$

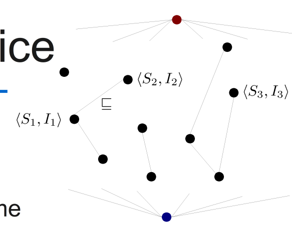
$$\frac{S_2 \Rightarrow S_1 \wedge I_1 \Rightarrow I_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

Lattice

- DCLabels represent points in a lattice



Dynamic Lattice



- Principals and DCLabels can be generated at run-time
  - This lattice might be modified on run-time
- In a system with several principals (e.g., users), it is difficult to fit the lattice in one picture
  - Gmail? Hotmail? Facebook?
- Top element: $\langle [P_1] \wedge [P_2] \wedge \cdots \wedge [P_n], [] \rangle$
- Bottom element: $\langle [], [P_1] \wedge [P_2] \wedge \cdots \wedge [P_n] \rangle$
- Problem?
  - We do not always know all the principals in the system
    - Principals can come and go

False: $\langle All, [] \rangle$

True: $\langle [], All \rangle$

Join and Meet Operations

- It is possible to define the join and meet operations and proof their correctness
  - The authors of DLM [Myers, Liskov 98] have not proved this formally
    - "The formula for meet is sound, but unlike the formula for join, it does not always produce the most restrictive label for all possible extensions P'"
    - "The result is that label inference must be conservative in some cases, which does not seem to be a significant problem"

Join and Meet for DCLabels

- They are simply defined as

  The join and meet for any two labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$ are respectively defined as:

$$L_1 \sqcup L_2 = \langle S_1 \wedge S_2, I_1 \vee I_2 \rangle$$
$$L_1 \sqcap L_2 = \langle S_1 \vee S_2, I_1 \wedge I_2 \rangle$$

- We proved that this is actually the join and meet (exercise)
- These operations might introduce categories which are redundant

Declassification/Endorsement

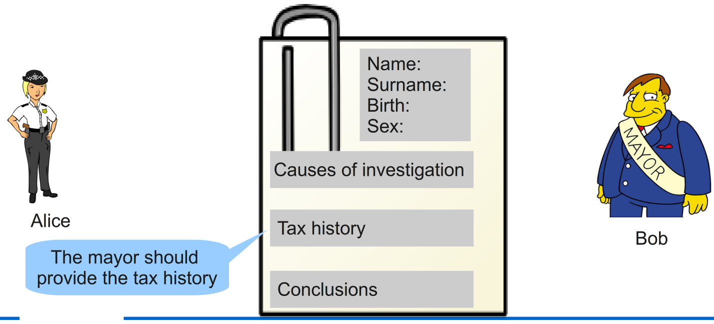
- Any system have some sort of intended release of information
- In a mutual distrust environment, it is necessary to declassify data after some collaborative effort

$$\langle [Alice] \wedge [Bob], [] \rangle \; \bullet\!\!\not\longrightarrow\!\!\bullet \; \langle [Alice], [] \rangle$$

- We describe a motivating example based on confidentiality but it also holds for integrity

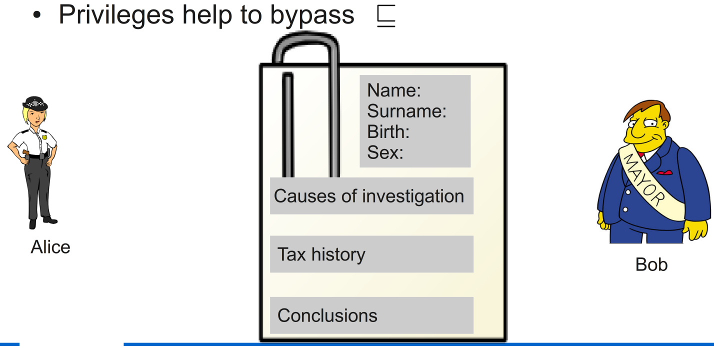
Declassification

- Alice is carrying out an investigation and she needs the tax history of the suspect



Declassification

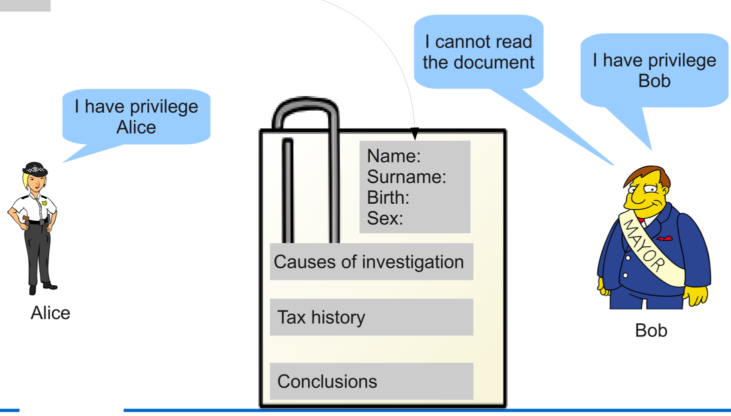
- The code that Alice is running has the privilege "Alice"
  - It allows to ignore the principal "Alice" in the DCLabels
  - Privileges help to bypass $\sqsubseteq$



Declassification

$\langle[Alice], []\rangle$

$\langle[Alice], []\rangle \bullet\!\!\not\longrightarrow\!\!\bullet \langle[Bob], []\rangle$



Privileges

Given any two DC labels $L_1 = \langle S_1, I_1\rangle$ and $L_2 = \langle S_2, I_2\rangle$, and a privilege set P, we can alternatively define the "can-flow-to given P" relation as follows:

$$\frac{\langle S_1, I_1 \wedge P\rangle \sqsubseteq \langle S_2 \wedge P, I_2\rangle}{\langle S_1, I_1\rangle \sqsubseteq_P \langle S_2, I_2\rangle}$$

$\langle[Alice] \wedge [Bob], []\rangle \bullet \not\sqsubseteq \bullet \langle[Alice], []\rangle$

$\langle[Alice] \wedge [Bob], []\rangle \bullet \sqsubseteq_{Bob} \bullet \langle[Alice], []\rangle$

Bob declassifies his data

$\langle[], [Alice]\rangle \bullet \not\sqsubseteq \bullet \langle[], [Alice] \wedge [Bob]\rangle$

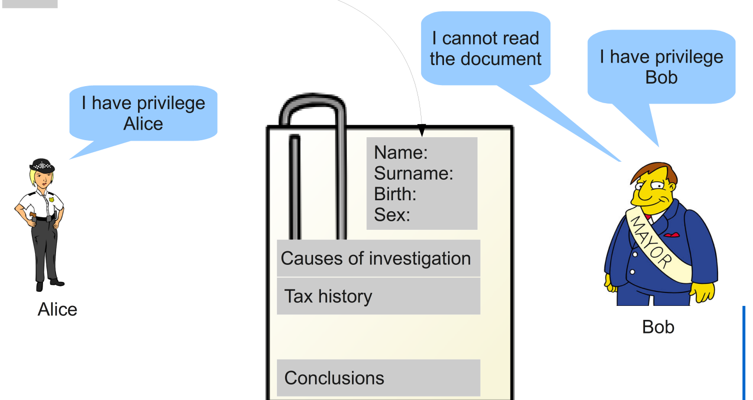
$\langle[], [Alice]\rangle \bullet \sqsubseteq_{Bob} \bullet \langle[], [Alice] \wedge [Bob]\rangle$

Bob endorses the data

Declassification

$\langle[Alice], []\rangle$

$\langle[Alice], []\rangle \bullet\!\!\not\longrightarrow\!\!\bullet \langle[Bob], []\rangle$

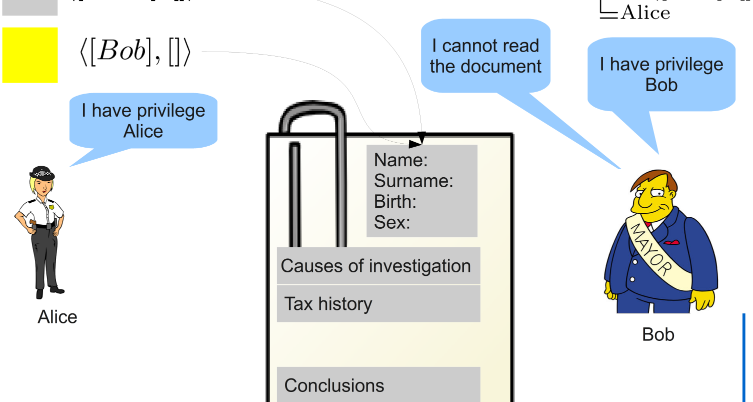


Declassification

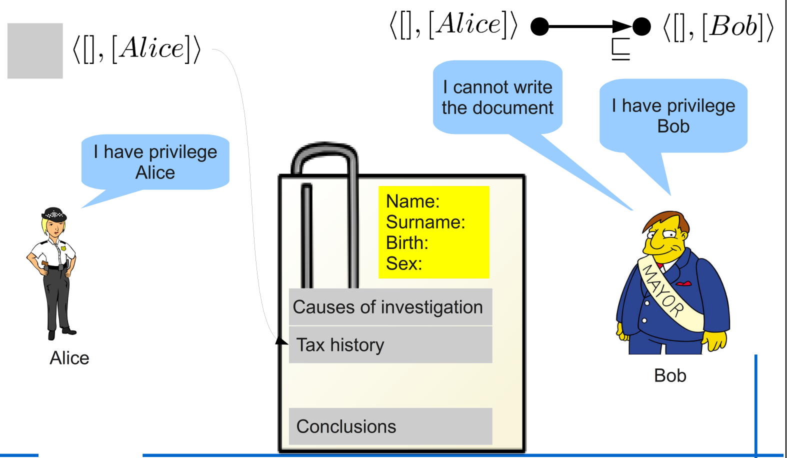
$\langle[Alice], []\rangle$

$\langle[Alice], []\rangle \bullet \sqsubseteq_{Alice} \bullet \langle[Bob], []\rangle$

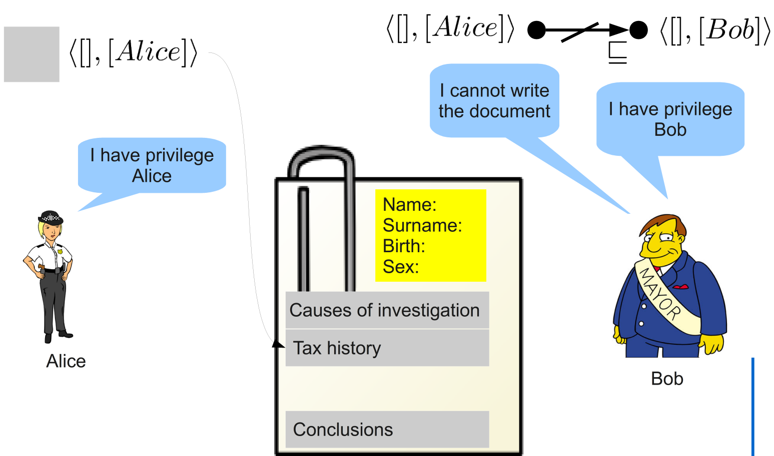
$\langle[Bob], []\rangle$



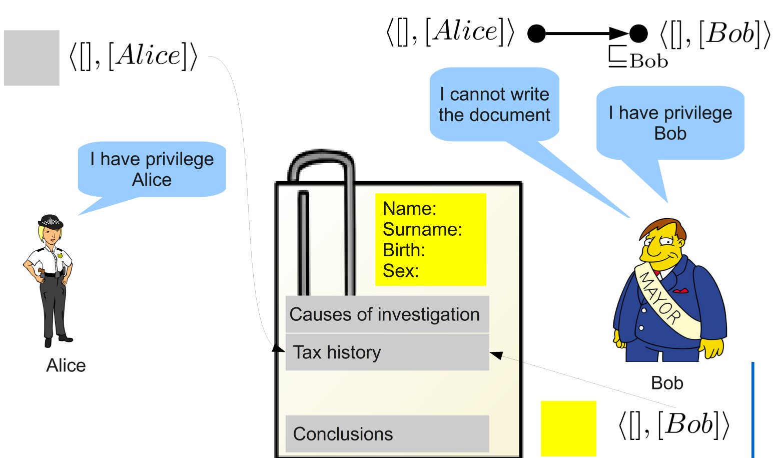
Endorsement



Endorsement



Endorsement



A Library for DCLabels in Haskell

- It is in a experimental phase
  - Remember that it is work-in-progress!
- I adapted the library for this course
- In the future, you might refer to the official release
- Check the webpage of the course to get the installation instructions

Creating DCLabels

```haskell
module Labels where

import DCLabel.Safe
import DCLabel.PrettyShow

c1 = "Alice" .\/. "Bob"

l1 =  "Alice" .\/. "Bob" ./\. "Carla"

l2 = "Alice" ./\. "Carla"

dc1 = newDC l1 l2

dc2 = newDC "Deain" "Alice"
```

It can use DCLabels without the capability to create privileges

Categories (disjunctions)

Labels (conjunctions of disjunctions)

DCLabels

Join, Meet, and ⊑

```
*ExamplesDCLabels> pShow dc1
<{["Alice" \/ "Bob"] /\ ["Carla"]} , {["Alice"] /\ ["Carla"]}>
*ExamplesDCLabels> pShow dc2
<{["Deain"]} , {["Alice"]}>
*ExamplesDCLabels> pShow $ join dc1 dc2
<{["Alice" \/ "Bob"] /\ ["Carla"] /\ ["Deain"]} , {["Alice"]}>
*ExamplesDCLabels> pShow $ meet dc1 dc2
<{["Alice" \/ "Bob" \/ "Deain"] /\ ["Carla" \/ "Deain"]} ,
 {["Alice"] /\ ["Carla"]}>
*ExamplesDCLabels> pShow dc1
<{["Alice" \/ "Bob"] /\ ["Carla"]} , {["Alice"] /\ ["Carla"]}>
*ExamplesDCLabels> pShow $ join dc1 top
<{ALL} , {}>
*ExamplesDCLabels> pShow $ join dc1 bottom
<{["Alice" \/ "Bob"] /\ ["Carla"]} , {["Alice"] /\ ["Carla"]}>

*ExamplesDCLabels> canflowto dc1 top
True
*ExamplesDCLabels> canflowto bottom dc1
True
```

Privileges

```haskell
import DCLabel.Core
import DCLabel.PrettyShow
import DCLabel.NanoEDSL

l1 =  "Alice" .\/. "Bob" ./\. "Carla"

l2 = "Alice" ./\. "Carla"

dc1 = newDC l1 l2

dc2 = newDC "Deain" "Alice"

pr = createPrivTCB (newDC ("Alice" ./\. "Carla") )
```

Only trusted code can create privileges

Creation

Privileges

```
*ExamplesDCLabels> pShow dc1
<{["Alice" \/ "Bob"] /\ ["Carla"]} , {["Alice"] /\ ["Carla"]}>
*ExamplesDCLabels> pShow dc2
<{["Deain"]} , {["Alice"]}>
*ExamplesDCLabels> canflowto dc1 dc2
False


*ExamplesDCLabels> pShow $ priv pr
{["Alice"] /\ ["Carla"]}
*ExamplesDCLabels> canflowto_p pr dc1 dc2
True
```

Secrecy category of `dc1` cannot be fullfiled by `dc2`

Now it is possible given privileges

Final Remarks

- Label system for mutual distrust scenarios (DCLabels)
  - Conjunction of categories
  - Categories are disjunction of principals
- It allows to express the interest of different parties
- Precisely compute join and meet
- Work-in-progress
  - Comparison with DLM (we have a precise meet)
- More systems need to be built using DCLabels

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

`lbl`

Floating Current Label

Program written using `LIO`

There is a current label at any point of the computation

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

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

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

The program finishes its execution here!

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

It is low

It is high

`ys`

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

`clr`

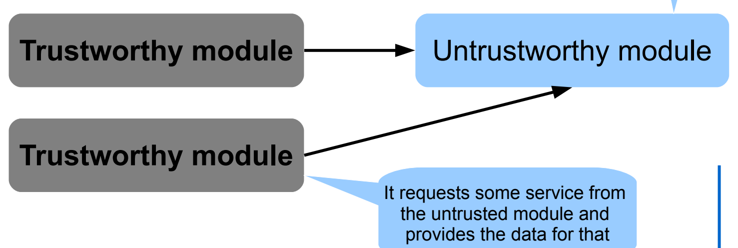
`xs`

`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



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

sec_str

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

```
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_str…
                     secret_string <- unlabel l_secret_stri…
                     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

Example (untrustworthy code)

```haskell
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
```

It is not raised when executing `toLabeled`

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

`clr`

`lbl`

API: `labelOf`

```haskell
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)

```haskell
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
```

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

Example (untrustworthy code)

```haskell
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

```haskell
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`

```haskell
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

```haskell
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)

```
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
```

It is almost the same code as module `ExampleToLabeledT`

References

We use references instead of `Labeled` values

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

CHALMERS | GÖTEBORG UNIVERSITY

Soundness of LIO
Secure Multi-Execution in Haskell

Secure Programming via Libraries

Soundness of LIO

Alejandro Russo (russo@chalmers.se)

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

Soudness for LIO

[Stefan, Russo, Mitchell, Mazieres 11]

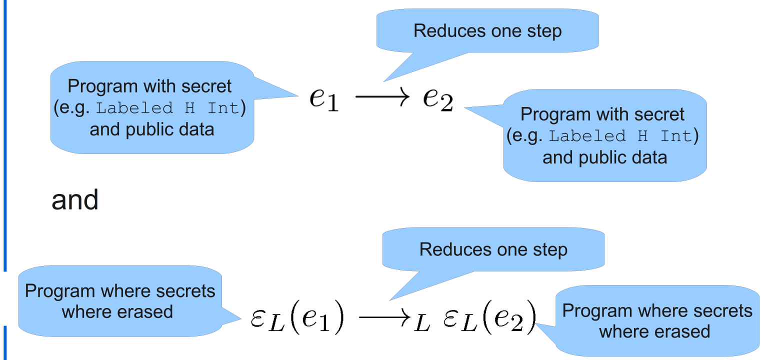
- Formalizes the non-interference guarantee provided by LIO
- For the proof, we consider a core and simple and functional language
  - Why not full Haskell?
  - λ-calculus extended with boolean values, pairs, recursion, monadic operations, references
- *We formally prove that the concept of monads works to guarantee non-interference*

Proof Technique

- Similar technique as the one used by Li and Zdancewic [Li, Zdancewic 10]
- Programs are expressions
- Main idea is simple:
  - If a program, that **involves secret and public information**, computes a public result, then the same public result can be obtained by a program that consists on the original one where **the secret data has been erased**!

Proof Technique

- More technically, we build a simulation between



The Language

The language

- The language and types

$$
\begin{array}{rl}
\text{Label:} & l \\
\text{Address:} & a \\
\text{Term:} & v ::= \texttt{true} \mid \texttt{false} \mid () \mid l \mid a \mid x \mid \lambda x.e \mid (e,e) \\
& \quad\mid \texttt{fix}\ e \mid \texttt{Lb}\ v\ e \mid (e)^{\text{LIO}} \mid \bullet \\
\text{Expression:} & e ::= v \mid e\ e \mid \pi_i\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \quad\mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{return}\ e \mid e \texttt{ >>= } e \mid \ldots \\
\text{Type:} & \tau ::= \texttt{Bool} \mid () \mid \tau \to \tau \mid (\tau,\tau) \\
& \quad\mid \ell \mid \texttt{Labeled}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau \mid \texttt{Ref}\ \ell\ \tau \\
\text{Store:} & \phi : \text{Address} \to \texttt{Labeled}\ \ell\ \tau
\end{array}
$$

The language

- The language and types

Special syntax node: *internal representation LIO computations*

Special syntax node: *it represents term erasure*

Special syntax node: *internal representation of Labeled values*

$$
\begin{array}{lrl}
\text{Address:} & & a \\
\text{Term:} & v ::= & \texttt{true} \mid \texttt{false} \mid () \mid l \mid a \mid x \mid \lambda x.e \mid (e,e) \\
& & \mid \texttt{fix}\ e \mid \texttt{Lb}\ v\ e \mid (e)^{\text{LIO}} \mid \bullet \\
\text{Expression:} & e ::= & v \mid e\ e \mid \pi_i\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ e \mid \ldots \\
\text{Type:} & \tau ::= & \texttt{Bool} \mid () \mid \tau \to \tau \mid (\tau,\tau) \\
& & \mid \ell \mid \texttt{Labeled}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau \mid \texttt{Ref}\ \ell\ \tau \\
\text{Store:} & & \phi : \text{Address} \to \texttt{Labeled}\ \ell\ \tau
\end{array}
$$

return and >>=

- Every monad has two operations: return and bind

$\texttt{return} :: a \to M a$

$\texttt{>>=} :: M a \to (a \to M b) \to M b$

- So far, we wrote programs using **do**

$e_1\ \texttt{>>=}\ \lambda x.e_2$ ⇒

```
do x ← e1
   e2
```


The Semantics

Operational Semantics

- It describes how a valid program is interpreted as a sequence of computational steps [Winskel]
- We describe the steps via evaluation contexts
- **Evaluation contexts**
  - An evaluation contexts E is just a term with a "hole"
  - $E[e]$ is the substitution of e into the hole
  - Intuitively, if a term M is being evaluated where $M = E[e]$
    - E is the context
    - e is the part of the term being evaluated

Evaluation Example

$e ::= \texttt{true} \mid \texttt{false} \mid (e,e) \mid \pi_i\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{el}$…

$E ::= [\cdot] \mid \pi_i\ E \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \ldots$

Expression to evaluate

$M = \texttt{if}\ \pi_1\ (\texttt{true},\texttt{false})\ \texttt{then}\ \texttt{true}\ \texttt{else}\ (\texttt{false},\texttt{true})$

$E_1 = \texttt{if}\ [\cdot]\ \texttt{then}\ \texttt{true}\ \texttt{else}\ (\texttt{false},\texttt{true})$

$M = E_1[\pi_1\ (\texttt{true},\texttt{false})]$ — Expressed in terms of evaluation contexts

$E_1[\pi_1\ (\texttt{true},\texttt{false})] \longrightarrow E_1[\texttt{true}]$ — Reduction step

$E_1[\texttt{true}] = \texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ (\texttt{false},\texttt{true})$

$E_1[\texttt{true}] = [\cdot][\texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ (\texttt{false},\texttt{true})]$

$[\cdot][\texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ (\texttt{false},\texttt{true})] \longrightarrow [\cdot][\texttt{true}] = \texttt{true}$

$M \longrightarrow^{*} \texttt{true}$

Reduction rules

$$E[\pi_i\ (e_1,e_2)] \longrightarrow E[e_i]$$
$$E[\texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow E[e_1]$$
$$E[\texttt{if}\ \texttt{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow E[e_2]$$

Operational Semantics for LIO

$v ::= \ \ldots \mid \texttt{Lb}\ v\ e \mid \ldots$

$e ::= \ \ldots \mid \texttt{label}\ e\ e \mid \texttt{unlabel}\ e \mid \texttt{toLabeled}\ e\ e$
$\quad\quad \mid \texttt{newRef}\ e\ e \mid \ldots$

$E ::= [\cdot] \mid \ldots$

- LIO computations have state
  - Current label, clearance, and an store for references

State of the LIO computation

$$\langle \Sigma, E[e] \rangle \longrightarrow \langle \Sigma', E[e'] \rangle$$

Reduction step

Current label: $\Sigma.\texttt{lbl}$

Current clearance: $\Sigma.\texttt{clr}$

Store: $\Sigma.\phi$

Operational Semantics for LIO

- The security checks are done in the semantics
  - Dynamic approach

It respects the current label and clearance

(LAB)

$$\frac{\Sigma.\mathtt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\mathtt{clr}}{\langle \Sigma, E[\mathtt{label}\ l\ e]\rangle \longrightarrow \langle \Sigma, E[\mathtt{return}\ (\mathtt{Lb}\ l\ e)]\rangle}$$

If the security checks are not fulfilled, the execution gets "stuck". In practice, it could be an uncaught exception, etc.

It evaluates to the internal representation

Operational Semantics for LIO

It is the join of the current label and the label that protects e

Clearance is respected

It sets a new current label

(UNLAB)

$$\frac{l' = \Sigma.\mathtt{lbl} \sqcup l \qquad l' \sqsubseteq \Sigma.\mathtt{clr} \qquad \Sigma' = \Sigma[\mathtt{lbl} \mapsto l']}{\langle \Sigma, E[\mathtt{unlabel}\ (\mathtt{Lb}\ l\ e)]\rangle \longrightarrow \langle \Sigma', E[\mathtt{return}\ e]\rangle}$$

A `Labeled` value which contents is e

It extracts the value e and returns it

Operational Semantics for LIO

The current label after executing e should be below l

The label of the result is among the current label and clearance

It executes the LIO computation e

(TOLAB)

$$\frac{\Sigma.\mathtt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\mathtt{clr} \qquad \langle \Sigma, e\rangle \longrightarrow^* \langle \Sigma', (v)^{\mathrm{LIO}}\rangle \qquad \Sigma'.\mathtt{lbl} \sqsubseteq l \qquad \Sigma'' = \Sigma'[\mathtt{lbl} \mapsto \Sigma.\mathtt{lbl}, \mathtt{clr} \mapsto \Sigma.\mathtt{clr}]}{\langle \Sigma, E[\mathtt{toLabeled}\ l\ e]\rangle \longrightarrow \langle \Sigma'', E[\mathtt{label}\ l\ v]\rangle}$$

The label of the result of computing e

Observe that this state has (only) the same current label and clearance values as when starting executing e

Operational Semantics for LIO

The allocated memory location is "new"

The store in the state gets modified

(NREF)

$$\frac{a\ \mathit{fresh} \qquad \Sigma.\mathtt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\mathtt{clr} \qquad \Sigma' = \Sigma.\phi[a \mapsto \mathtt{Lb}\ l\ e]}{\langle \Sigma, E[\mathtt{newRef}\ l\ e]\rangle \longrightarrow \langle \Sigma', E[\mathtt{return}\ a]\rangle}$$

It returns a memory location

Operational Semantics for LIO

- You have seen a few rules
- Check the paper for the rest of them
  [Stefan, Russo, Mitchell, Mazieres 11]
- You should be able to understand them after the lecture

The Types

Typing

- It is not very interesting for our library
  - It is a dynamic approach, not static one
- Typing judgment

$$\Gamma \vdash e : \tau$$

Assumptions about the type of free variables in e

Expression to type

Type

$\Gamma \vdash x + x : ?$

$\Gamma = [x : \texttt{Bool}]$ → $\Gamma \vdash x + x : ?$

$\Gamma = [x : \mathit{Int}]$ → $\Gamma \vdash x + x : \mathit{Int}$

Typing rules

- They indicate how to perform type-checking
  - Rules are usually syntax-directed rules
- An expression type-checks if we can construct a *type derivation* (application of the typing rules)

Type system (very simple)

$$\Gamma \vdash 1 : \texttt{Int}$$

$$\Gamma \vdash \texttt{true} : \texttt{Bool}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e, e') : (\tau, \tau')}$$

Here you have the type derivation!

$$\frac{\dfrac{\Gamma \vdash \texttt{true} : \texttt{Bool} \quad \Gamma \vdash 1 : \texttt{Int}}{\Gamma \vdash (\texttt{true}, 1) : (\texttt{Bool}, \texttt{Int})} \qquad \Gamma \vdash \texttt{true} : \texttt{Bool}}{\Gamma \vdash ((\texttt{true}, 1), \texttt{true}) : ((\texttt{Bool}, \texttt{Int}), \texttt{Bool})}$$

What is the type?

Interesting typing rules

Special syntax node: *internal representation of Labeled values*

Special syntax node: *internal representation LIO computations*

Special syntax node: *it represents term erasure*

Term: $v ::= \ \ldots \mid \texttt{Lb}\ v\ e \mid (e)^{\texttt{LIO}} \mid \bullet$

Store: $\phi : \text{Address} \to \texttt{Labeled}\ \ell\ \tau$

$$\frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{Lb}\ e_1\ e_2 : \texttt{Labeled}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e)^{\texttt{LIO}} : \texttt{LIO}\ \ell\ \tau}$$

$$\Gamma \vdash \bullet : \tau$$

- The rest of the typing rules are just like the ones implemented in Haskell

So far

- We have seen
  - The language
  - Semantics
  - Types
- What is coming now?
  - Combine all of them (and some other techniques) in order to prove non-interference in programs written using LIO

Soundness

Proof Technique

- More technically, we build a simulation between

Reduces one step

Program with secret (e.g. `Labeled H Int`) and public data

$$e_1 \longrightarrow e_2$$

Program with secret (e.g. `Labeled H Int`) and public data

and

Reduces one step

Program where secrets where erased

$$\varepsilon_L(e_1) \longrightarrow_L \varepsilon_L(e_2)$$

Program where secrets where erased

The Erasure Function

- Function ε_L
  - It is responsible for performing *term erasure*
  - It is often applied homomorphically

$$\varepsilon_L(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) =$$
$$\texttt{if}\ \varepsilon_L(e)\ \texttt{then}\ \varepsilon_L(e_1)\ \texttt{else}\ \varepsilon_L(e_2)$$

- Intuitively, the function removes values and expressions that are not below L
- L is the attacker level

The Erasure Function

$$\varepsilon_L(\bullet) = \bullet \qquad \varepsilon_L((e)^{\texttt{LIO}}) = (\varepsilon_L(e))^{\texttt{LIO}}$$

Idempotent

$$\varepsilon_L(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \bullet & l \not\sqsubseteq L \\ \texttt{Lb}\ l\ \varepsilon_L(e) & \text{otherwise} \end{cases}$$

It removes labeled values where the label Is not below L

$$\frac{\varepsilon_L(\Sigma.\phi) = \{(x, \varepsilon_L(\Sigma.\phi(x))) : x \in \text{dom}(\Sigma.\phi)\}}{\varepsilon_L(\Sigma) = \Sigma[\phi \mapsto \varepsilon_L(\Sigma.\phi)]}$$

It propagates the application of the erasure function to the labeled values stored by references

$$\varepsilon_L(\langle \Sigma, e \rangle) = \begin{cases} \langle \varepsilon_L(\Sigma), \bullet \rangle & \Sigma.\texttt{lbl} \not\sqsubseteq L \\ \langle \varepsilon_L(\Sigma), \varepsilon_L(e) \rangle & \text{otherwise} \end{cases}$$

Erasure in configurations (technical reasons)

Proof Technique

- More technically, we build a simulation between

$$e_1 \longrightarrow e_2$$

Program with secret (e.g. `Labeled H Int`) and public data

Reduces one step

Program with secret (e.g. `Labeled H Int`) and public data

and

$$\varepsilon_L(e_1) \longrightarrow_L \varepsilon_L(e_2)$$

Program where secrets where erased

Reduces one step

Program where secrets where erased

A new evaluation relationship

$$\frac{\langle \Sigma, e \rangle \longrightarrow \langle \Sigma', e' \rangle}{\langle \Sigma, e \rangle \longrightarrow_L \varepsilon_L(\langle \Sigma', e' \rangle)}$$

- Expressions under this evaluation relationship are evaluated as before
- It guarantees that confidential data (above L) is erased as soon as it is created

Simulation

- This is the main idea behind the proof

$$\begin{array}{ccc} \langle \Sigma, e \rangle & \longrightarrow^* & \langle \Sigma', e' \rangle \\ \downarrow \varepsilon_L & & \downarrow \varepsilon_L \\ \varepsilon_L(\langle \Sigma, e \rangle) & \longrightarrow_L^* & \varepsilon_L(\langle \Sigma', e' \rangle) \end{array}$$

Preliminaries

- In order to prove the simulation, it is necessary to show several auxiliary results
  - You can read it from the paper
- The proof consists on establishing the simulation in two phases
  - For expressions that do not execute any `toLabeled`
  - For expressions that execute n-`toLabeled`
- Why is that?
  - The semantics for `toLabeled` uses big-step semantics

Establishing the simulation

Lemma 1 (Single-step simulation without `toLabeled`). *If*

- $\Gamma \vdash e : \tau$, *and*
- $\langle \Sigma, e \rangle \longrightarrow \langle \Sigma', e' \rangle$

where `toLabeled` *is not executed, then*

i) $\Gamma \vdash e' : \tau$, *and*

ii) $\varepsilon_L(\langle \Sigma, e \rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', e' \rangle)$.

Subject reductoin

Subject reductoin

Establishing the simulation

- The proof going on case analysis on the expression being evaluated
  - Recall that evaluation is performed using evaluation contexts

Establishing the simulation

Case:
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}}{\langle \Sigma, E[\texttt{label}\ l\ e] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle}:$$

- $l \sqsubseteq L$:

$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, E[\texttt{label}\ l\ e] \rangle) \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\texttt{label}\ l\ \varepsilon_L(e)] \rangle \\
&\longrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\texttt{return}\ (\texttt{Lb}\ l\ \varepsilon_L(e))] \rangle) \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\texttt{return}\ (\texttt{Lb}\ l\ \varepsilon_L(e))] \rangle \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\varepsilon_L(\texttt{return}\ (\texttt{Lb}\ l\ e))] \rangle \\
&= \varepsilon_L(\langle \Sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle)
\end{aligned}$$

It applies the definition in a left-to-right manner

It just applies the definition

Idempotent erasure function

It applies the definition in a right-to-left manner

Establishing the simulation

Case:
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}}{\langle \Sigma, E[\texttt{label}\ l\ e] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle}:$$

- $l \not\sqsubseteq L$:

$$\begin{aligned}
&\varepsilon_L(\langle \Sigma, E[\texttt{label}\ l\ e] \rangle) \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\texttt{label}\ l\ \varepsilon_L(e)] \rangle \\
&\longrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\texttt{return}\ (\texttt{Lb}\ l\ \varepsilon_L(e))] \rangle) \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\texttt{return}\ (\texttt{Lb}\ l\ \bullet)] \rangle \\
&= \langle \varepsilon_L(\Sigma), \varepsilon_L(E)[\varepsilon_L(\texttt{return}\ (\texttt{Lb}\ l\ e))] \rangle \\
&= \varepsilon_L(\langle \Sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle)
\end{aligned}$$

It applies the definition in a left-to-right manner

It just applies the definition

Idempotent erasure function

It applies the definition in a right-to-left manner

Establishing the simulation

Lemma 2 (Simulation for expressions not executing `toLabeled`). *If*

- $\Gamma \vdash e : \tau$, *and*
- $\langle \Sigma, e \rangle \longrightarrow^* \langle \Sigma', e' \rangle$

where `toLabeled` *is not executed, then*

i) $\Gamma \vdash e' : \tau$, *and*

ii) $\varepsilon_L(\langle \Sigma, e \rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma', e' \rangle)$.

- The proof is on induction on $\longrightarrow^*$
- The base case is Lemma 1

Establishing the simulation

Lemma 3 (Simulation). *If* $\Gamma \vdash e : \tau$ *and* $\langle \Sigma, e \rangle \longrightarrow^* \langle \Sigma', e' \rangle$ *then* $\Gamma \vdash e' : \tau$ *and* $\varepsilon_L(\langle \Sigma, e \rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma', e' \rangle)$.

- The proof is on induction on the number of `toLabeled` being executed
- Base case is Lemma 2
- For the inductive case, we rewrite the big-step semantics into

$$\underbrace{\langle \Sigma, e \rangle \longrightarrow^* \langle \Sigma_0, E[}_{\text{first big-step}}\underbrace{\texttt{toLabeled}\ l\ e_0] \rangle \longrightarrow^* \langle \Sigma', e' \rangle}_{\text{second big-step}}$$

no `toLabeled`

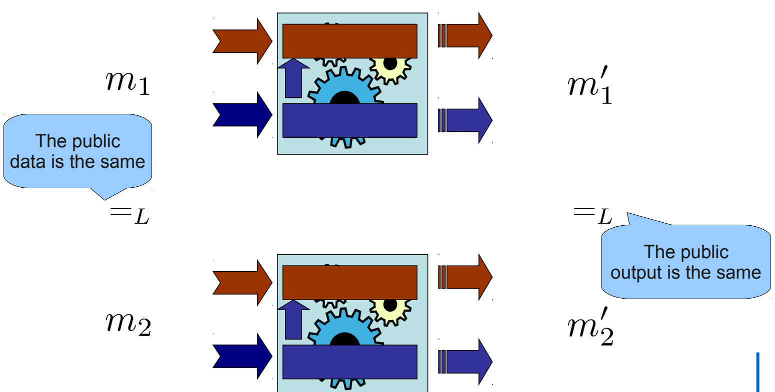
≤ k `toLabeled`

≤ k `toLabeled`

Non-interference

- Having the simulation established
- We proceed with a formulation of the theorem that proves non-interference
- The formulation is “standard”
- It requires a notion of low-equivalence
- It captures the observational power of the attacker
- If we run the program twice but with the same public input, the same public output must be observed

Low-equivalence



Low-equivalence

- We considered labeled values as the input and output of programs
- Intuitively, two expressions are low-equivalent if the are equal, modulo labeled values whose labels are above L

$$\frac{e \approx_L e' \qquad l \sqsubseteq L}{\texttt{Lb}\ l\ e \approx_L \texttt{Lb}\ l\ e'} \qquad \frac{l \not\sqsubseteq L}{\texttt{Lb}\ l\ e \approx_L \texttt{Lb}\ l\ e'}$$

If the label is not below L, then the content of labeled values it is not important

$\texttt{if true then (Lb}\ H\ \texttt{false) else false} \approx_L$
$\texttt{if true then (Lb}\ H\ \texttt{true) else false}$

Low-equivalence

- We define low-equivalence between stores as well
- Intuitively, two stores are low-equivalent if the stored labeled values below L are the same

Both stores contains the same *public* labeled values

The *public* labeled values are low-equivalent

$$\frac{\mathrm{dom}_L(\Sigma.\phi) = \mathrm{dom}_L(\Sigma.\phi') \qquad \forall a \in \mathrm{dom}_L(\Sigma.\phi) \cdot \Sigma.\phi(a) \approx_L \Sigma'.\phi(a)}{\Sigma.\phi \approx_L \Sigma'.\phi}$$

Low-equivalence

- We now define low-equivalence for configurations
  - It essentially means to have low-equivalence in the store and the expression to be evaluated when the current label is below L

$$\frac{\Sigma.\phi \approx_L \Sigma'.\phi \qquad \overset{e \approx_L e'}{\Sigma.\texttt{lbl} = \Sigma'.\texttt{lbl}} \qquad \Sigma.\texttt{clr} = \Sigma'.\texttt{clr} \qquad \Sigma.\texttt{lbl} \sqsubseteq L}{\langle \Sigma, e \rangle \approx_L \langle \Sigma', e' \rangle}$$

$$\frac{\Sigma.\phi \approx_L \Sigma'.\phi \qquad \Sigma.\texttt{lbl} \not\sqsubseteq L \qquad \Sigma'.\texttt{lbl} \not\sqsubseteq L}{\langle \Sigma, e \rangle \approx_L \langle \Sigma', e' \rangle}$$

Non-interference

Theorem 1 (Non-interference). *Given a computation e (with no $\bullet$, $(\ )^{\text{LIO}}$, or $\texttt{Lb}$) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \rightarrow \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau')$, initial environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \emptyset$, security label l, an attacker at level L such that $l \sqsubseteq L$, then*

$$\begin{aligned}
&\forall e_1 e_2 . (\Gamma \vdash e_i : \texttt{Labeled}\ \ell\ \tau)_{i=1,2} \\
&\quad \wedge (e_i = \texttt{Lb}\ l\ e_i')_{i=1,2} \wedge \langle \Sigma_1, e\ e_1 \rangle \approx_L \langle \Sigma_2, e\ e_2 \rangle \\
&\quad \wedge \langle \Sigma_1, e\ e_1 \rangle \longrightarrow^* \langle \Sigma_1', (\texttt{Lb}\ l_1\ e_1'')^{\text{LIO}} \rangle \\
&\quad \wedge \langle \Sigma_2, e\ e_2 \rangle \longrightarrow^* \langle \Sigma_2', (\texttt{Lb}\ l_2\ e_2'')^{\text{LIO}} \rangle \\
&\quad \Rightarrow \langle \Sigma_1', \texttt{Lb}\ l_1\ e_1'' \rangle \approx_L \langle \Sigma_2', \texttt{Lb}\ l_2\ e_2'' \rangle
\end{aligned}$$

Non-interference (specialized)

Theorem 1 (Non-interference)**.** *Given a computation e (with no $\bullet$, $(\,)^{\text{LIO}}$, or Lb) where $\Gamma \vdash e : \texttt{Labeled}\,\ell\,\tau \rightarrow \texttt{LIO}\,\ell\,(\texttt{Labeled}\,\ell\,\tau')$, initial environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \emptyset$, an attacker at level L, then*

$$\begin{aligned}
&\forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\,\ell\,\tau)_{i=1,2}\\
&\quad \wedge (e_i = \texttt{Lb}\;H\;e_i')_{i=1,2} \wedge \langle \Sigma_1, e\;e_1\rangle \approx_L \langle \Sigma_2, e\;e_2\rangle\\
&\quad \wedge \langle \Sigma_1, e\;e_1\rangle \longrightarrow^* \langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle\\
&\quad \wedge \langle \Sigma_2, e\;e_2\rangle \longrightarrow^* \langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle\\
&\quad \Rightarrow \langle \Sigma_1', \texttt{Lb}\;l_1\;e_1''\rangle \approx_L \langle \Sigma_2', \texttt{Lb}\;l_2\;e_2''\rangle
\end{aligned}$$

It should have use $(e_i = \texttt{Lb}\;L\;(\texttt{Lb}\;H\;e_i'))_{i=1,2}$ but for simplicity I did not

Proof Sketch

- We will use our simulation
- We asumme (you can prove it) that

$$\varepsilon_L(e) = \varepsilon_L(e') \Rightarrow e \approx_L e'$$

Proof Sketch II

$$\begin{aligned}
&(e_i = \texttt{Lb}\;H\;e_i')_{i=1,2} \wedge \langle \Sigma_1, e\;e_1\rangle \approx_L \langle \Sigma_2, e\;e_2\rangle\\
&\wedge \langle \Sigma_1, e\;(\texttt{Lb}\;H\;e_1')\rangle \longrightarrow^* \langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle\\
&\wedge \langle \Sigma_2, e\;(\texttt{Lb}\;H\;e_2')\rangle \longrightarrow^* \langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle
\end{aligned}$$

- By our simulation, we know that

By the simulation

$$\varepsilon_L(\langle \Sigma_1, e\;(\texttt{Lb}\;H\;e_1')\rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle)$$
$$\varepsilon_L(\langle \Sigma_2, e\;(\texttt{Lb}\;H\;e_2')\rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

Proof Sketch III

$$\varepsilon_L(\langle \Sigma_1, e\;(\texttt{Lb}\;H\;e_1')\rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle)$$
$$\varepsilon_L(\langle \Sigma_2, e\;(\texttt{Lb}\;H\;e_2')\rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

Erase function goes inside the configuration

- We expand it

$$\langle \varepsilon_L(\Sigma_1), \varepsilon_L(e\;(\texttt{Lb}\;H\;e_1'))\rangle \longrightarrow^* \varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle)$$
$$\langle \varepsilon_L(\Sigma_2), \varepsilon_L(e\;(\texttt{Lb}\;H\;e_2'))\rangle \longrightarrow^* \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

- A little bit more

$$\langle \varepsilon_L(\Sigma_1), \varepsilon_L(e)\;(\texttt{Lb}\;H\;\bullet)\rangle \longrightarrow^* \varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle)$$
$$\langle \varepsilon_L(\Sigma_2), \varepsilon_L(e)\;(\texttt{Lb}\;H\;\bullet)\rangle \longrightarrow^* \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

Proof Sketch IV

These are the same configurations

$$\langle \varepsilon_L(\Sigma_1), \varepsilon_L(e)\;(\texttt{Lb}\;H\;\bullet)\rangle \longrightarrow_L^* \varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle)$$
$$\langle \varepsilon_L(\Sigma_2), \varepsilon_L(e)\;(\texttt{Lb}\;H\;\bullet)\rangle \longrightarrow_L^* \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

- We know that $\longrightarrow_L^*$ is deterministic
- Then,

By equality and definition of erasure function

$$\varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle) = \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

- Which means,

By definition of erasure function

Remember what we assume in the begining

$$\varepsilon_L((\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}) = \varepsilon_L((\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}})$$

$$\varepsilon_L(\texttt{Lb}\;l_1\;e_1'') = \varepsilon_L(\texttt{Lb}\;l_2\;e_2'') \qquad \Rightarrow \texttt{Lb}\;l_1\;e_1'' \approx_L \texttt{Lb}\;l_2\;e_2''$$

Proof Sketch V

- Then,

$$\varepsilon_L(\langle \Sigma_1', (\texttt{Lb}\;l_1\;e_1'')^{\text{LIO}}\rangle) = \varepsilon_L(\langle \Sigma_2', (\texttt{Lb}\;l_2\;e_2'')^{\text{LIO}}\rangle)$$

- Which means,

By equality and definition of erasure function

$$\varepsilon_L(\Sigma_1'.\phi) = \varepsilon_L(\Sigma_2'.\phi) \quad \Rightarrow \mathrm{dom}_L\,(\Sigma_1'.\phi) = \mathrm{dom}_L\,(\Sigma_2'.\phi)$$

- For any "public" labeled value in the store, we have

$$\varepsilon_L(\Sigma_1'.\phi(x)) = \varepsilon_L(\Sigma_2'.\phi(x)),\ \text{for any } x \in \mathrm{dom}_L\,(\Sigma_1'.\phi)$$

By definition of erasure function and equality

$$\Rightarrow \Sigma_1'.\phi(x) \approx_L \Sigma_2'.\phi(x),\ \text{for any } x \in \mathrm{dom}_L\,(\Sigma_1'.\phi)$$

What we assume in the beginning

$$\Rightarrow \Sigma_1'.\phi \approx_L \Sigma_2'.\phi$$

By definition of low-equivalence for stores

Proof Sketch VI

- Now, we have that

$$\Sigma'_1.\phi \approx_L \Sigma'_2.\phi \qquad \texttt{Lb}\ l_1\ e''_1 \approx_L \texttt{Lb}\ l_2\ e''_2$$

- We still need to prove

$$\langle \Sigma'_1, \texttt{Lb}\ l_1\ e''_1 \rangle \approx_L \langle \Sigma'_2, \texttt{Lb}\ l_2\ e''_2 \rangle$$

- From the simulation, we had

$$\varepsilon_L(\langle \Sigma'_1, (\texttt{Lb}\ l_1\ e''_1)^{\text{LIO}} \rangle) = \varepsilon_L(\langle \Sigma'_2, (\texttt{Lb}\ l_2\ e''_2)^{\text{LIO}} \rangle)$$

- Which implies that

$$\Sigma'_1.\texttt{lbl} = \Sigma'_2.\texttt{lbl} \land \Sigma'_1.\texttt{clr} = \Sigma'_2.\texttt{clr}$$

Proof Sketch VII

- So, having

$$\Sigma'_1.\phi \approx_L \Sigma'_2.\phi \qquad \texttt{Lb}\ l_1\ e''_1 \approx_L \texttt{Lb}\ l_2\ e''_2$$

$$\Sigma'_1.\texttt{lbl} = \Sigma'_2.\texttt{lbl} \qquad \Sigma'_1.\texttt{clr} = \Sigma'_2.\texttt{clr}$$

- We can prove

$$\langle \Sigma'_1, \texttt{Lb}\ l_1\ e''_1 \rangle \approx_L \langle \Sigma'_2, \texttt{Lb}\ l_2\ e''_2 \rangle$$

- by just case analysis if $\Sigma'_1.\texttt{lbl} \sqsubseteq L$ and applying the definition of low-equivalence for configurations

Final Remarks

- We formalize the ideas behind LIO
  - Language: simple call-by-name lambda-calculus
- Semantics
  - Security checks
- Types (not very interesting)
- Simulation
- Low-equivalence
- Non-interference theorem

Secure Programming via Libraries

Secure Multi-Execution in Haskell

Alejandro Russo (russo@chalmers.se)

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

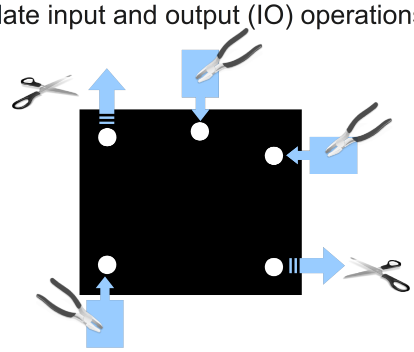
Enforcement for non-interference

- 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]

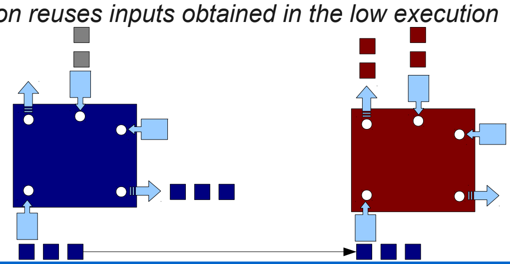
- Black box approach to enforce non-interference
  - No need to inspect the code
  - Manipulate input and output (IO) operations



Secure Multi-Execution

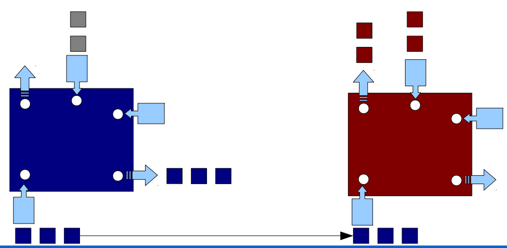
[Devriese, Piessens 10]

- 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?

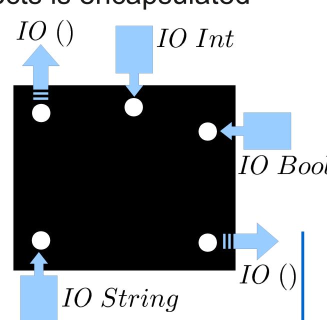
- Executed program satisfies non-interference
  - No explicit and implicit flows
- The secure multi-execution produces the same results
- Otherwise, the semantics changes to preserve security



Secure Multi-Execution in Haskell

[Jaskelioff, Russo 11]

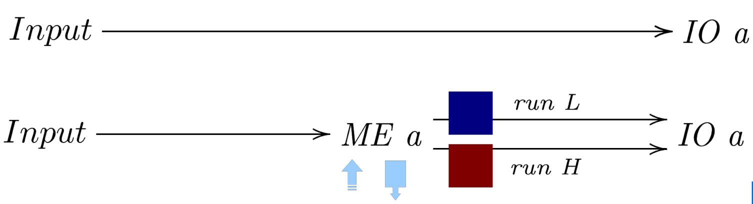
- Clear separation of pure computations with those with side-effects
- Every computation with side-effects is encapsulated into the monad *IO*
- Identify where IO is performed



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



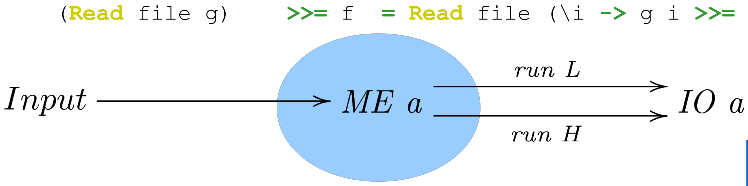
Monad ME

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

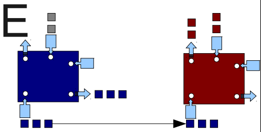
```
data ME a   =  Return a
            |  Write FilePath String (ME a)
            |  Read FilePath (String -> ME a)


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

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


Monad ME

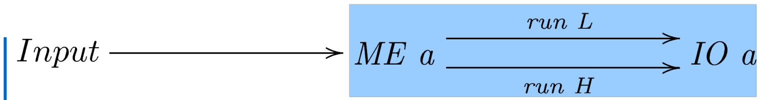
```
data ME a   =  Return a
            |  Write FilePath String (ME a)
            |  Read FilePath (String -> ME a)

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
```



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

```haskell
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

```haskell
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

```haskell
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