Secure Programming via Libraries

Introduction

Alejandro Russo (russo@chalmers.se)

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

CHALMERS

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 **useful** programming languages concepts 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

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)

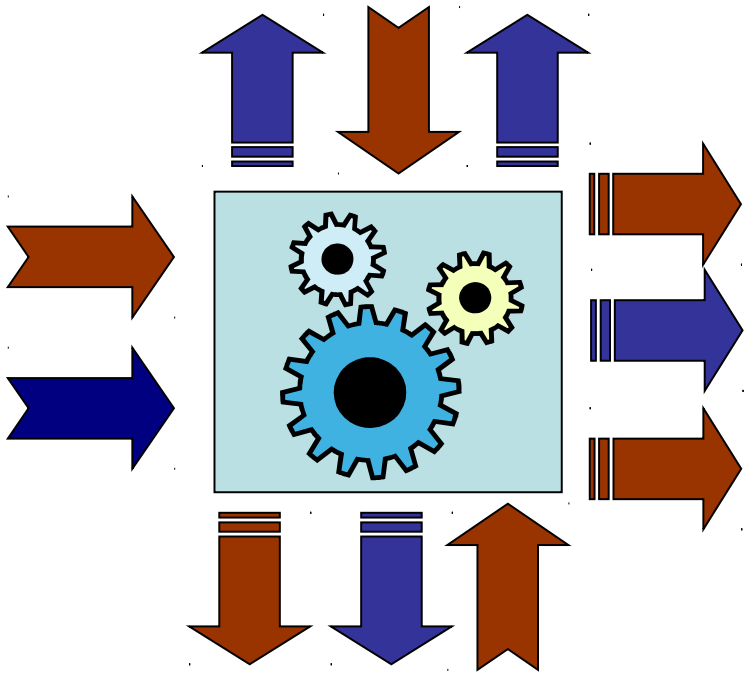
Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

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

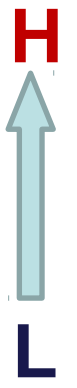
- Assign security levels to data representing their confidentiality
- Security levels are placed in a lattice (security lattice)
  - Information can flow from low to high positions in the lattice
- For simplicity, we only consider two security levels

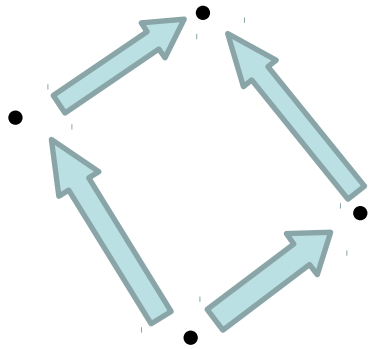
$$L \sqsubseteq H \text{ and } H \not\sqsubseteq L$$

$$L \sqcup H = H \qquad L \sqcap H = L$$

$$L \sqcup L = L \qquad L \sqcap L = L$$

$$H \sqcup H = H \qquad H \sqcap H = H$$

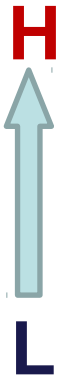




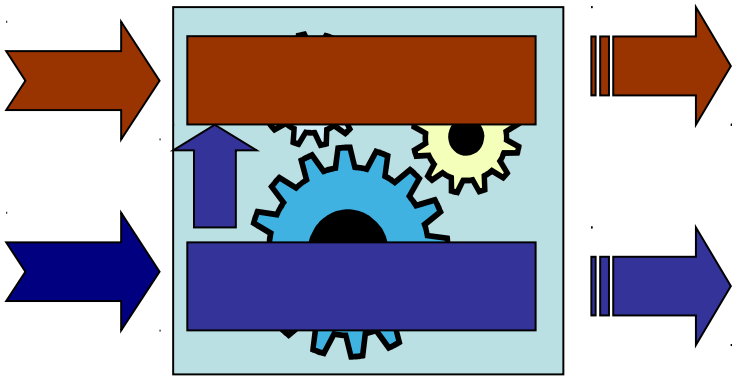
Non-interference

[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

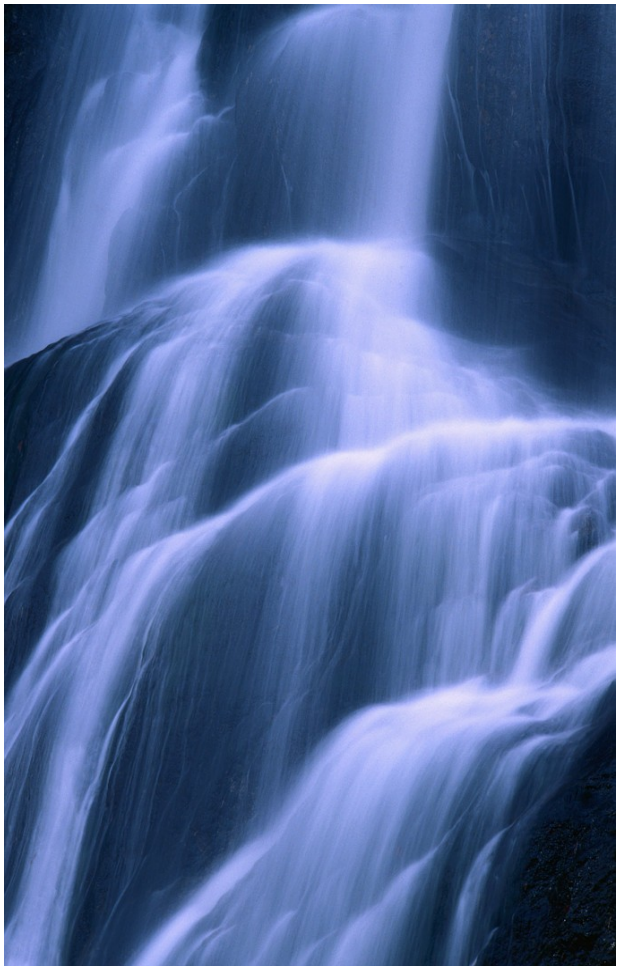
[Denning, Denning 77]

- 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 to 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 ;
```

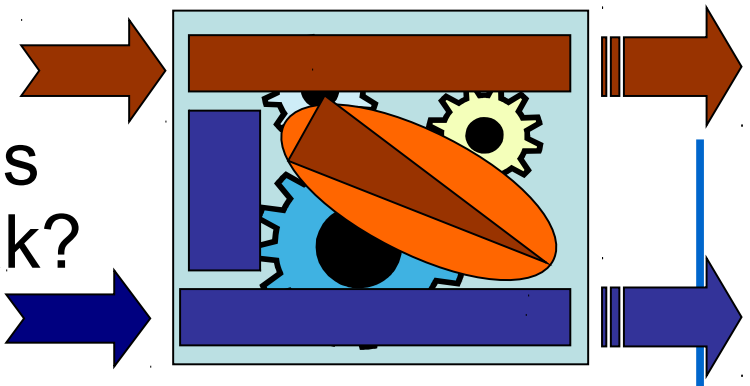
h<=0 → l = 0 (Ok)
h>0 → (Loop)

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

Declassification

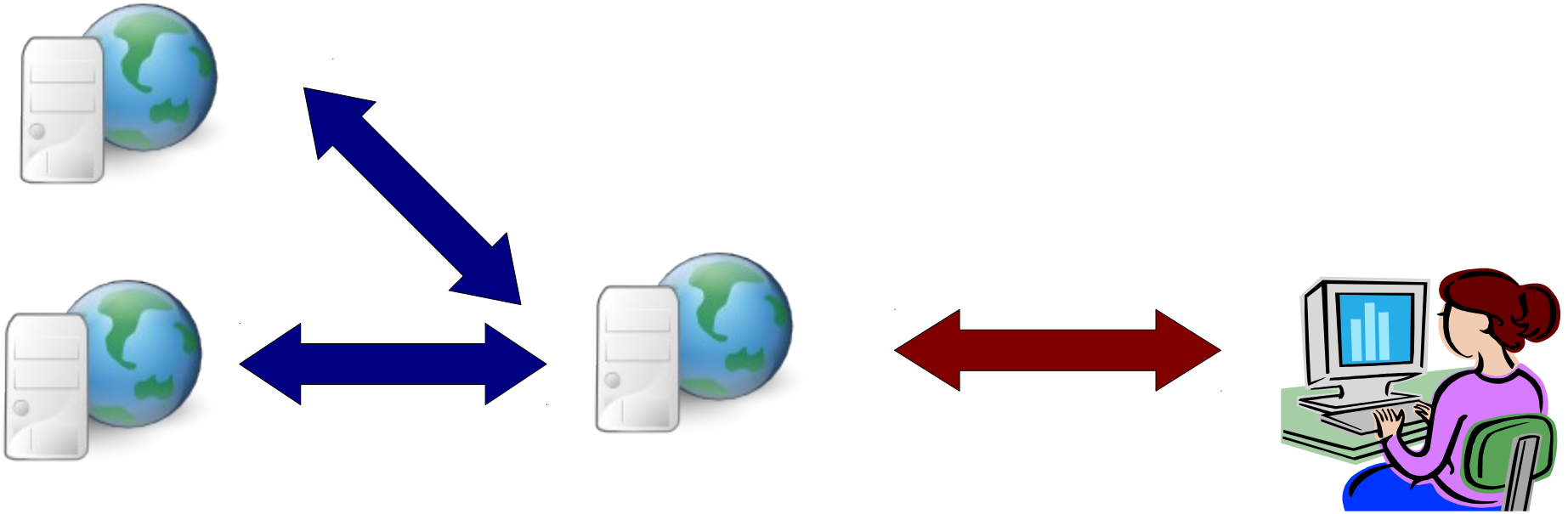
[Sabelfeld, Sands 07]

- 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]

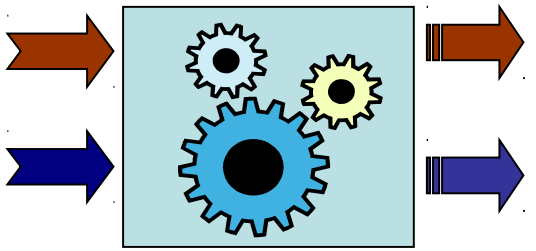
- Ten most frequent attacks
  - A1 – Injection (SQL, OS, etc)
    - **Information-flow**
  - A2 – Cross Site Scripting (XSS)
    - **Information-flow**
  - A3 – Broken Authentication and Session Management
    - **Information-flow** helps here as well
  - A4 – Insecure Direct Object References
    -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

Static vs. Dynamic Enforcement for Information-flow

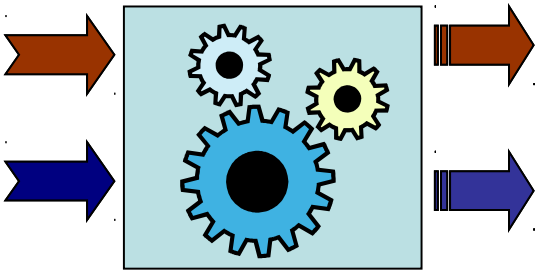
- Security policy: secrets must no be leaked!
  - Termination insensitive non-interference
- Some purely dynamic mechanisms are as secure as traditional type-systems [Sabelfeld, Russo 09]
- Should we go dynamic or static?
  - Several arguments are possible to argue against [Le Guernic et al, 06] [Shroff 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 v2 v3 ... v40 v50 v60 ...

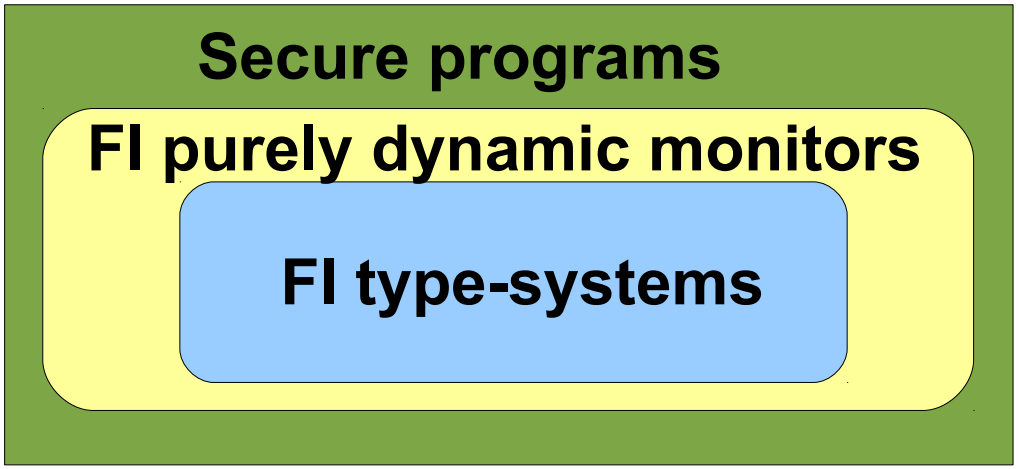


v1 v2 v3 ... v40 v50 v60 ...

```
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 (than a FS type-system) purely dynamic monitor
  - 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!