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)

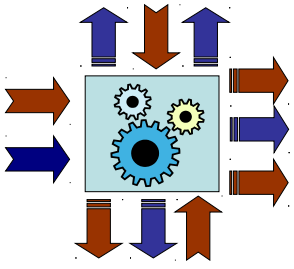
Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

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

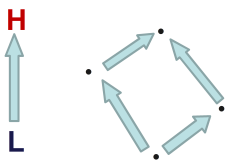
- Assign security levels to data representing their confidentiality
- Security levels are placed in a lattice (security lattice)
  - Information can flow from low to high positions in the lattice
- For simplicity, we only consider two security levels

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

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

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

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

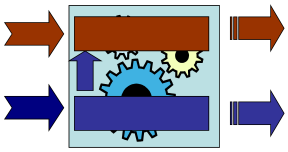


Non-interference

[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

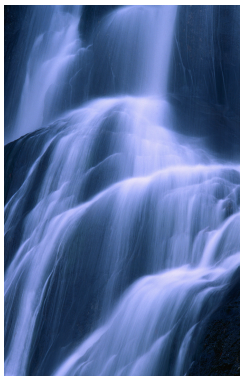
[Denning, Denning 77]

- 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

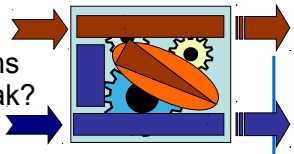
```
l:=0 ;
if h>0
   then while true do skip ;
```

h<=0 → l = 0 (Ok)
h>0 → (Loop)

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

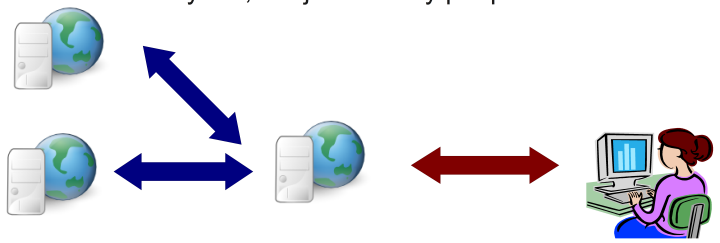
Declassification
[Sabelfeld, Sands 07]

- 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]

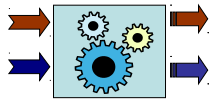
- Ten most frequent attacks
  - A1 – Injection (SQL, OS, etc)
    - **Information-flow**
  - A2 – Cross Site Scripting (XSS)
    - **Information-flow**
  - A3 – Broken Authentication and Session Management
    - **Information-flow** helps here as well
  - A4 – Insecure Direct Object References
    -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

Static vs. Dynamic Enforcement for Information-flow

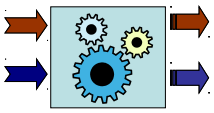
- Security policy: secrets must no be leaked!
  - Termination insensitive non-interference
- Some purely dynamic mechanisms are as secure as traditional type-systems [Sabelfeld, Russo 09]
- Should we go dynamic or static?
  - Several arguments are possible to argue against [Le Guernic et al, 06] [Shroff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



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

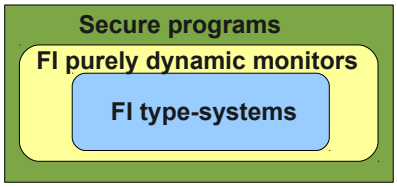
```
v1 := h ;
v2 := v1+l ;
v1 := l ;
h  := v1 + v2 ;
```


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)

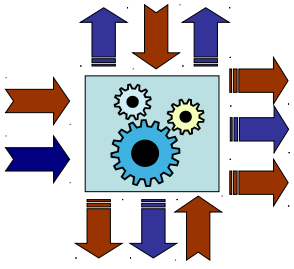
Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

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

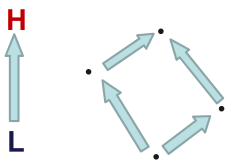
- Assign security levels to data representing their confidentiality
- Security levels are placed in a lattice (security lattice)
  - Information can flow from low to high positions in the lattice
- For simplicity, we only consider two security levels

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

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

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

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

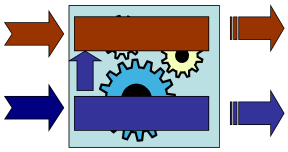


Non-interference

[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

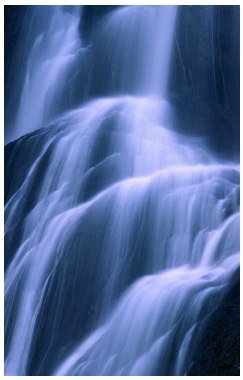
[Denning, Denning 77]

- 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

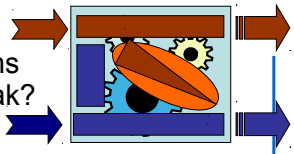
```
l:=0 ;
if h>0
   then while true do skip ;
```

h<=0 → l = 0 (Ok)
h>0 → (Loop)

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

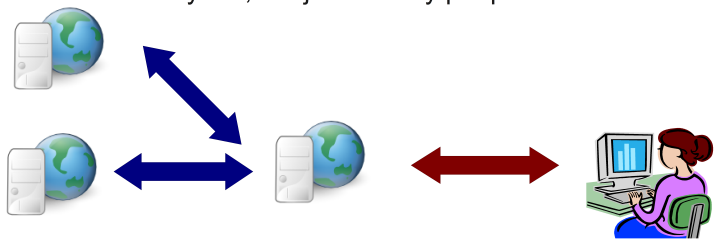
Declassification
[Sabelfeld, Sands 07]

- 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]

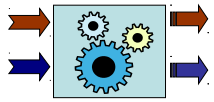
- Ten most frequent attacks
  - A1 – Injection (SQL, OS, etc)
    - **Information-flow**
  - A2 – Cross Site Scripting (XSS)
    - **Information-flow**
  - A3 – Broken Authentication and Session Management
    - **Information-flow** helps here as well
  - A4 – Insecure Direct Object References
    -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

Static vs. Dynamic Enforcement for Information-flow

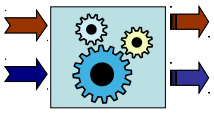
- Security policy: secrets must no be leaked!
  - Termination insensitive non-interference
- Some purely dynamic mechanisms are as secure as traditional type-systems [Sabelfeld, Russo 09]
- Should we go dynamic or static?
  - Several arguments are possible to argue against [Le Guernic et al, 06] [Shroff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



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

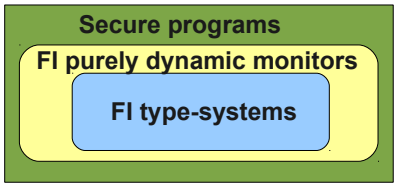
```
v1 := h ;
v2 := v1+l ;
v1 := l ;
h  := v1 + v2 ;
```


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

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!

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

- 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.

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

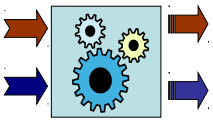
- Just look at the type!

```
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)

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

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

f (c, i) | c > 65     =  (c, 42)
         | otherwise  =  (c, i)
```

YES

NO

YES

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 = ...

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

H

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)

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

YES

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)

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

NO

NO

Security Guarantee

Type checks!

Non-interferece

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?

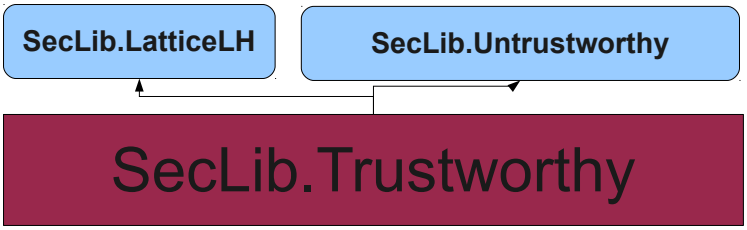
```
instance Less H L where
  less _ _ = ()
```


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
```

A Library for Light-Weight Information-Flow Security in Haskell

Alejandro Russo Koen Claessen John Hughes
Chalmers University of Technology, Gothenburg, Sweden
{russo,koen,rjmh}@chalmers.se

Abstract

Protecting confidentiality of data has become increasingly important for computing systems. *Information-flow* techniques have been developed over the years to achieve that purpose, leading to special-purpose languages that guarantee information-flow security in programs. However, rather than producing a new language from scratch, information-flow security can also be provided as a library. This has been done previously in Haskell using the arrow framework. In this paper, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve the same goals. We present a *monadic library* to provide information-flow security for Haskell programs. The library introduces mechanisms to protect confidentiality of data for pure computations, that we then easily, and modularly, extend to include dealing with side-effects. We also present combinators to dynamically enforce different declassification policies when release of information is required in a controlled manner. It is possible to enforce policies related to *what*, by *whom*, and *when* information is released or a combination of them. The well-known concept of monads together with the light-weight characteristic of our approach makes the library suitable to build applications where confidentiality of data is an issue.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Security, Languages

Keywords Information-flow, Declassification, Library, Monad

1. Introduction

Protecting confidentiality of data has become increasingly important for computing systems. Often, software is so complex that it is hard to see if a program can be abused by a malicious person to gain access to private data. This is important when developing software oneself, and becomes increasingly more important if one is forced to trust other people's code.

Information-flow techniques have been developed over the years to achieve this kind of protection. For example, as a result, two main stream compilers, Jif (based on Java) and Flowcaml (based on Ocaml) have been developed to guarantee information-flow security in programs.

However, it is a very heavy-weight solution to introduce a new programming language for dealing with information-flow. In this work, we explore the possibility of expressing restrictions on information-flow as a library rather than a new language.

We end up with a light-weight monadic approach to the problem of expressing and ensuring information-flow in Haskell. Code that exhibits information flows that are disallowed will be ill-typed and rejected by the type checker. Our approach is general enough to deal with practical concepts such as secure reading and writing to files (which can be generalized to capture any information exchange with the outside world) and declassification (a pragmatic way of allowing controlled information leakage (Sabelfeld and Sands 2005)).

Our library might be used in scenarios where we want to incorporate in our programs some code written by outsiders (untrusted programmers) to access our private information. Such code can be also allowed to interact with the outside world (for example by accessing the web). We would like to have a guarantee that the program will not send our private data to an attacker. A slightly different, but related, scenario is where we ourselves write the possibly unsafe code, but we want to have the help of the type checker to find possible security mistakes.

Li and Zdancewic (Li and Zdancewic 2006) have previously shown how to provide information-flow security also as a library, but their implementation is based on *arrows* (Hughes 2000), which naturally requires programmers to be familiar with arrows when writing security-related code. In this work, we show that arrows are not necessary to design such libraries and that a less general notion, namely monads, is sufficient to achieve very similar goals.

1.1 Motivating example

Consider a machine running Linux with the default installation of the Shadow Suite (Jackson 1996) responsible to store and manage users' passwords. In this machine, file `/etc/passwd` contains information regarding users such as user and group ID's, which are used by many system programs. This file must remain world readable. Otherwise, simple commands as `ls -l` stop working. Passwords are set in the file `/etc/shadow`, which can only be read and written by root. From now on, we refer to the passwords stored in this file as *shadow passwords*. Programs that verify passwords need to be run as root. From the security point of view, this requirement implies that very careful programming practices must be followed when creating such programs. For instance, if a program running as root has a shell escape, it is not desirable that such shell escape runs with root privileges. The process to verify a password usually consists of taking the input provided by the user, applying some cryptographic algorithms to it, and comparing the result of that with the user's information stored in `/etc/shadow`. Observe that an attacker can encrypt a dictionary of common passwords offline and then, given some file `/etc/shadow`, try to guess users' passwords by checking matches. This attack is known as an *offline dictionary attack* and is one of the most common methods for gain-

Haskell'08, September 25, 2008, Victoria, BC, Canada.

ing or expanding unauthorized access to systems (Narayanan and Shmatikov 2005). In order to obtain the content of `/etc/shadow`, the attacker needs to obtain root privileges, which is not impossible to achieve (Local Root Exploit 2008). Given these facts, we can conclude that there are mainly two security problems with shadow passwords: programs require having root privileges to verify passwords and offline dictionary attacks. We start dealing with these problems by firstly limiting the access to the password file. With this in mind, we assume that information stored in `/etc/shadow` is only accessible through an API. The following Haskell code shows an example of such API.

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

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

Data type `Spwd` stores users' identification number (`uid::UID`) and users' password (cypher :: Cypher). For a simple presentation, we assume that passwords are stored as plain text and not cyphers. Function `getSpwdName` receives a user name and returns his (her) password if such user exists. Function `putSpwd` takes a register of type `Spwd` and adds it to the shadow password file. This API is now the only way to have access to shadow passwords. We can still be more restrictive and require that such API is only called under root privileges, which is usually the case for Unix-like systems. Unfortunately, this restriction does not help much since attackers could obtain unauthorized root access and then steal the passwords. However, by applying information-flow techniques to the API and programs that use it, it is possible to guarantee that passwords are not revealed while making possible to verify them. In other words, offline dictionary attacks are avoided as well as some requirements as having root privileges to verify passwords. In Section 3.3, we show a secure version of this API.

1.2 Contributions

We present a *light-weight library* for *information-flow security* in Haskell. The library is *monadic*, which we argue is easier to use than arrows, which were used in previous attempts. The library has a pure part, but also deals with *side-effects*, such as the secure reading and writing of files. The library also provides novel and powerful means to specify *declassification policies*.

1.3 Assumptions

In the rest of the paper, we assume that the programming language we work with is a controlled version of Haskell, where code is divided up into *trusted* code, written by someone we trust, and *untrusted* code, written by the attacker. There are no restrictions on the trusted code. However, the untrusted code has certain restrictions; certain modules are not available to the untrusted programmer. For example, all modules providing IO functions, including exceptions (and of course `unsafePerformIO`) are not allowed. Our library will reintroduce part of that functionality to the untrusted programmer in a controlled, and therefore, secure way.

2. Non-interference for pure computations

Non-interference is a well-known *security policy* that preserves confidentiality of data (Cohen 1978; Goguen and Meseguer 1982). It states that public outcomes of programs do not depend on their confidential inputs.

In imperative languages, information leaks arise from the presence of *explicit* and *implicit* flows inside of programs (Denning and Denning 1977). Explicit flows are produced when secret data is placed explicitly into public locations by an assignment. Implicit flows, on the other hand, use control constructs in the language in order to reveal information. In a pure functional language, however, this distinction becomes less meaningful, since there are no assignments nor control constructs. For example, a conditional (if-then-else) is just a function as any other function in the language. In a pure language, all information-flow is explicit; information only flows from function arguments to function results.

```
newtype Sec s a

instance Functor (Sec s)
instance Monad (Sec s)

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

Figure 1. The Sec monad

To illustrate information leaks in pure languages, we proceed assuming that a programmer, potentially malicious, needs to write a function `f :: (Char,Int) -> (Char,Int)` where characters and integers are considered respectively secret and public data. We assume that attackers can only control public inputs and observe public results when running programs, and can thus only observe the second component of the pair returned by function `f`. For simplicity, we also assume that type `Char` represents ASCII characters.

If a programmer writes the code

```
f (c, i) = (chr (ord c + i), i+3)
```

then the function is non-interferent and preserves the confidentiality of `c`; the public output of `f` is independent of the value of `c` [1]. If a programmer instead writes

```
f (c, i) = (c, ord c)
```

then information about `c` is revealed, and the program is not non-interferent! Attackers might try to write less noticeable information leaks however. For instance, the code

```
f (c, i) = (c, if ord c > 31 then 0 else 1)
```

leaks information about the printability of the character `c` and therefore should be disallowed as well.

In this section, we show how monads can be used to avoid leaks and enforce the non-interference property for pure computations.

2.1 The Sec monad

In order to make security information-flow specific, we are going to make a distinction at the type level between *protected* data and *public* data. Protected data only lives inside a special *monad* (Wadler 1992). This security monad makes sure that only the parts of the code that have the right to do so are able to look at protected data.

In larger programs, it becomes necessary to talk about several *security levels* or *areas*. In this case, values are not merely protected or public, but they can be protected by a certain security level `s`.

Take a look at Fig. 1, which shows the API of an abstract type, `Sec`, which is a functor and a monad. There are two functions provided on the type `Sec`; `sec` is used to protect a value, and `open` is used to look at a protected value. However, to look at a protected value of type `Sec s a`, one needs to have a value of type `s`. Restricting access to values of different such types `s` by means of the module system allows fine control over which parts of the program can look at what data. (For this to work, `open` needs to be strict in its second argument.)

For example, if we define a security area `H` in the following way:

[1] Function `chr` returns an exception when the received argument does not represent an ASCII code. By observing occurrences of exceptions or computations that diverge, an attacker can deduce some information about secrets. However, we only consider programs that terminate successfully.

```
module Lattice where

data L = L
data H = H

class Less sl sh where
  less :: sl -> sh -> ()

instance Less L L where
  less _ _ = ()

instance Less L H where
  less _ _ = ()

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

Figure 2. Implementation of a two-point lattice

```
data H = H
```

then we can model the type of the function `f` given in the beginning of this section as follows:

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

The first, secure, example of `f` can be programmed as follows:

```
f (sc,i) = ((\c -> chr (ord c + i)) `fmap` sc,i+3)
```

However, the other two definitions can not be programmed without making use of `H` or breaking the type checker.

So, for a part of the program that has no means to create non-bottom values of a type `s`, direct access to protected values of type `Sec s a` is impossible. However, computations involving protected data are possible as long as the data stays protected. This can be formalized by stating that type `Sec` guarantees a *non-interference* property. For any type `A`, and values `a1, a2 :: A`, a function

```
f :: Sec H A -> Bool
```

will produce the same result for arguments `a1` and `a2`. See (Russo et al. 2008a) for more details.

We will later show the implementation of the type `Sec` and its associated functions.

2.2 Security lattice

Valid information flows inside of programs are determined by a *lattice on security levels* (Denning 1976). Security levels are associated to data in order to establish its degree of confidentiality. The ordering relation in the lattice, written $\sqsubseteq$, represents allowed flows. For instance, $l_1 \sqsubseteq l_2$ indicates that information at security level l_1 can flow into entities of security level l_2.

For simplicity, in this paper, we will only use a two-point lattice with security levels `H` and `L` where `L` $\sqsubseteq$ `H` and `H` $\not\sqsubseteq$ `L`. Security levels `H` and `L` denote secret (*high*) and public (*low*) information, respectively. The implementation of the lattice is shown in Figure 2. Type class `Less` encodes the relation $\sqsubseteq$ and security levels are represented as singleton types (Pierce 2004). The role of `less` is explained in Section 4. Public information is characterized by the security level `L`. Constructor `L` is then publicly available so that data at security level `L` can be observed by anyone, which also includes attackers.

As explained earlier, attackers must have no access to the constructor `H`. In Section 4, we describe how to achieve such restriction.

Finally, to capture the fact that valid information flows occur from *lower* (`L`) to higher (`H`) security levels, we introduce the function

```
up :: Less sl sh => Sec sl a -> Sec sh a
```

The function `up` can be used to turn any protected value into a protected value at a higher security level. The implementation of `up` will be shown later.

3. Non-interference and side-effects

The techniques described in Section 2 do not perform computations with side-effects. The reason for that is that side-effects involving confidential data cannot be executed when they are created inside of the monad `Sec s`.

Even if we allowed a restricted and secure form of file reading and writing in the IO-monad, that would still not be enough. For example, if we, read information from file A, and depending on the value of a secret, want to write either to a file B or file C, we would obtain a computation of type `IO (Sec H (IO ()))`. It is easy to see that these types quickly become unmanagable, and, more importantly, unusable.

In this section, we show how we can augment our security API to be able to deal with *controlled* side-effects while still maintaining non-interference properties.

In this paper, we concentrate how to provide an API that allows reading and writing protected data from and to files. For this to work properly, files need to contain a security level, so that only data from the right security level can be written to a file. We assume that the attacker has no way of observing what side-effects were performed, other than through our API. (The attacker, so to say, sits within the Haskell program and has no way of getting out[2].)

The ideas for reading and writing files can be extended to deal with many other controlled IO operations, such as creating, reading and writing secure references, communicating over secure channels, etc. We will however not deal with the details of such operations in this paper.

3.1 Secure files

We model all interactions with the outside world by operations for reading and writing files (Tanenbaum 2001). For that reason, we decide to include secure file operations in our library. We start by assigning security levels to files in order to indicate the confidentiality of their contents. More precisely, we introduce the abstract data type `File s`. Values of type `File s` represent names of files whose contents have security level `s`. These files are provided by the trusted programmer. We assume that attackers have no access to the internal representation of `File s`. In Section 4, we show how to guarantee such assumption.

A first try for providing secure file operations is to provide the following two functions:

```
readSecIO  :: File s -> IO (Sec s String)
writeSecIO :: File s -> Sec s String -> IO ()
```

These functions do not destroy non-interference, because they do not open up for extra information-flow between security levels. The data read from a file with security level `s` is itself protected with security level `s`, and any data of security level `s` can be written to a file of security level `s`.

However, the above functions are not enough to preserve confidentiality of data. Take a look at the following program:

```
writeToAFile :: Sec H String -> Sec H (IO ())
writeToAFile secs =
  (\s -> if length s < 10
           then writeSecIO file1 s
           else writeSecIO file2 s) `fmap` secs
```

[2] A situation where the attacker is in league with a hacker who has gotten access to our system, and can for example read log files, is beyond our control and the guarantees of our library.

```
newtype SecIO s a

instance Functor (SecIO s)
instance Monad (SecIO s)

value :: Sec s a -> SecIO s a

readSecIO  :: File s' -> SecIO s (Sec s' String)
writeSecIO :: File s -> String -> SecIO s ()

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

Figure 3. The SecIO monad

Here, `file1, file2 :: File H` is assumed to be defined elsewhere.

The behavior of the above function is indeed dependent on the protected data in its argument, as indicated by the result type. However, only the *side-effects* of the computation are dependent on the data, not the *result value*. Why is this important? Because we assume that the attacker has no way of observing from within the program what these side-effects are! (Unless the attacker can observe the results of the side-effects, namely the change of file contents in either `file1` or `file2`, but that information can only be obtained by someone with the appropriate security clearance anyway.) This assumption is valid for the scenarios described in Section 1.

In other words, since side-effects cannot be observed from within a program, we are going to allow the leakage of side-effects. Our assumption is only true if we restrict the IO actions that the attacker can perform.

3.2 The SecIO monad

To this end, we introduce a new monad, called `SecIO`. This monad is a variant of the regular IO monad that keeps track of the security level of all data that was used inside it.

Take a look at Fig. 3, which shows the API for an abstract type `SecIO`, which is a functor and a monad. Values of type `SecIO s a` represent computations that can securely read from any file, securely write to files of security level `s` (or higher), and look at data protected at level `s` (or lower).

The function `value` can be used to look at a protected value at the current security level. The function `readSecIO` reads protected data from files at any security level, protecting the result as such. The function `writeSecIO` writes data to files of the current security level.

The function `plug` is used to import computations with side-effects at a high level into computations with side-effects at a low level of security. Observe that only the side-effects are "leaked", not the result, which is still appropriately protected by the high security level. This function is particularly suitable to write programs that contain loops that depend on public information and perform, based on secret and public data, side-effects on secret files in each iteration.

These functions together with the return and bind operations for `SecIO s` constitute the basic interface for programmers.

Based on that, more convenient and handy functions can then be defined. For instance,

```
s_read :: Less s' s => File s' -> SecIO s String
s_read file =  do ss <- readSecIO file
                  value (up ss)

s_write :: Less s' s =>
           File s -> String -> SecIO s' (Sec s ())
s_write file str = plug (writeSecIO file str)
```

Observe that `s_read` and `s_write` have simpler types while practically providing the same functionality as `readSecIO` and `writeSecIO`, respectively.

In the next section, we show how to implement the core part of our library: the monads `Sec s` and `SecIO s`. We continue this section with an example that shows how these APIs can be used.

3.3 Developing a secure shadow passwords API

As an example of how to apply information-flow mechanisms, we describe how to adapt the API described in the introduction to guarantee that neither API's callers or the API itself reveal shadow passwords. Specifically, passwords cannot be copied into public files at all. Hence, offline dictionary attacks are avoided as well as the requirement of having root privileges to verify passwords. As mentioned in the introduction, we assume that the contents of `/etc/shadow` is only accessible through the API. For simplicity, we assume that this file is stored in the local file system, which naturally breaks the assumption we have just mentioned (user root has access to all the files in the system). However, it is not difficult to imagine an API that establishes, for example, a connection to some sort of password server in order to get information regarding shadow passwords.

We firstly start adapting our library to include the two-point lattice mentioned in Section 2. We decide to associate security level `H`, which represents secret information, to data regarding shadow passwords. Then, we indicate that file `/etc/shadow` stores secret data by writing the following lines

```
shadowPwds :: File H
shadowPwds  = MkFile "/etc/shadow"
```

We proceed to modify the API to indicate what is the secret data handled by it. More precisely, we redefine the API as follows:

```
getSpwdName :: Name -> IO (Maybe (Sec H Spwd))
putSpwd     :: Sec H Spwd -> IO ()
```

where values of type `Spwd` are now "marked" as secrets [3]. The API's functions are then adapted, without too much effort, to meet their new types. In order to manipulate data inside of the monad `Sec H`, API's callers need to import the library in their code. Since `/etc/shadow` is the only file with type `File H` in our implementation, this is the only place where secrets can be stored after executing calls to the API. By marking values of type `Spwd` as secrets, we restrict how information flows inside of the API and API's callers while making possible to operate with them. In Section 5, we show how to implement a login program using the adapted API.

4. Implementation of monads Sec and SecIO

In this section, we provide a possible implementation of the APIs presented in the previous two sections.

In Fig. 4 we show a possible implementation of `Sec`. `Sec` is implemented as an identity monad, allowing access to its implementation through various functions in the obvious way. The presence of `less` in the definition of function `up` includes `Less` in its typing constrains. Function `unSecType` is used for typing purposes and has no computational meaning. Note the addition of the function `reveal`, which can reveal any protected value. This function is not going to be available to the untrusted code, but the trusted code might sometimes need it. In particular, the implementation of `SecIO` needs it in order to allow the leakage of side-effects.

In Fig. 5 we show a possible implementation of `SecIO`. It is implemented as an IO computation that produces a safe result. As

[3] Values of type `Maybe` are not included inside of `Sec H` since the existence of passwords is linked to the existence of users in the system, which is considered public information.

```
module Sec where

-- Sec
newtype Sec s a = MkSec a

instance Monad (Sec s) where
  return x = sec x

  MkSec a >>= MkSec k =
    MkSec (let MkSec b = k a in b)

sec :: a -> Sec s a
sec x = MkSec x

open :: Sec s a -> s -> a
open (MkSec a) s = s `seq` a

up :: Less s s' => Sec s a -> Sec s' a
up sec_s@(MkSec a) = less s s' `seq` sec_s'
                     where (sec_s') = MkSec a
                           s        = unSecType sec_s
                           s'       = unSecType sec_s'

-- For type-checking purposes (not exported).
unSecType :: Sec s a -> s
unSecType _ = undefined

-- only for trusted code!
reveal :: Sec s a -> a
reveal (MkSec a) = a
```

Figure 4. Implementation of Sec monad

an invariant, the IO part of a value of type `SecIO s a` should only contain unobservable (by the attacker) side-effects, such as the reading from and writing to files.

There are a few things to note about the implementation. Firstly, the function `reveal` is used in the implementation of monadic bind, in order to leak the *side-effects* from the protected IO computation. Remember that we assume that the performance of side-effects (reading and writing files) cannot be observed by the attacker. Some leakage of side-effects is unavoidable in any implementation of the functionality of `SecIO`. Secondly, the definition of the type `File` does not make use of its argument `s`. This is also unavoidable, because it is only by a promise from the trusted programmer that certain files belong to certain security levels. Thirdly, function `plug`, similarly to function `up`, includes `less` and an auxiliary function (`unSecIOType`) to properly generate type constraints.

The modules `Sec`, `SecIO`, and `Lattice` can only be used by trusted programmers. The untrusted programmers only get access to modules `SecLibTypes` and `SecLib`, shown in Fig. 6. They import the three previous modules, but only export the trusted functions. Observe that the type `L` and its constructor `L` are exported, but for `H`, only the type is exported and not its constructor. Method `less` is also not exported. Therefore, functions `up` and `plug` are only called with the instances of `Less` defined in `Lattice.hs`.

In order to check that a module is safe with respect to information-flow, the only thing we have to check is that it does not import trusted modules, in particular:

- `Sec` and `SecIO`
- any module providing exception handling, for example `Control.Monad.Exception`,
- any module providing unsafe extensions, for example `System.IO.Unsafe`

```
module SecIO where
import Lattice
import Sec

-- SecIO
newtype SecIO s a = MkSecIO (IO (Sec s a))

instance Monad (SecIO s) where
  return x = MkSecIO (return (return x))

  MkSecIO m >>= k =
    MkSecIO (do sa <- m
                let MkSecIO m' = k (reveal sa)
                m')

-- SecIO functions
value :: Sec s a -> SecIO s a
value sa = MkSecIO (return sa)

run :: SecIO s a -> IO (Sec s a)
run (MkSecIO m) = m

plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)
plug ss_sh@(MkSecIO m)
     = less sl sh `seq` ss_sl
       where
         (ss_sl) = MkSecIO (do sha <- m
                               return (sec sha))
         sl      = unSecIOType ss_sl
         sh      = unSecIOType ss_sh

-- For type-checking purposes (not exported).
unSecIOType :: SecIO s a -> s
unSecIOType _ = undefined

-- File IO
data File s = MkFile FilePath

readSecIO :: File s' -> SecIO s (Sec s' String)
readSecIO (MkFile file) =
  MkSecIO ((sec . sec) `fmap` readFile file)

writeSecIO :: File s' -> String -> SecIO s ()
writeSecIO (MkFile file) s =
  MkSecIO (sec `fmap` writeFile file s)
```

Figure 5. Implementation of SecIO monad

5. Declassification

Non-interference is a security policy that specifies the absence of information flows from secret to public data. However, real-word applications release some information as part of their intended behavior. Non-interference does not provide means to distinguish between intended releases of information and those ones produced by malicious code, programming errors, or vulnerability attacks. Consequently, it is needed to relax the notion of non-interference to consider *declassification* policies or intended ways to leak information. In this section, we introduce run-time mechanisms to enforce some declassification policies found in the literature.

Declassification policies have been recently classified in different dimensions(Sabelfeld and Sands 2005). Each dimension represents aspects of declassification. Aspects correspond to *what*, *when*, *where*, and by *whom* data is released. In general, type-systems to enforce different declassification policies include different features, e.g rewriting rules, type and effects, and external analysis (Myers and Liskov 2000; Sabelfeld and Myers 2004; Chong and Myers 2004). Encoding these features directly into the Haskell type system would considerably increase the complexity of our library. For

```
module SecLibTypes ( L (..), H,  Less () ) where
import Lattice


module SecLib
 (   Sec, open, sec, up
   , SecIO, value, plug, run,
   , File, readSecIO, writeSecIO, s_read, s_write
 )
where

import Sec
import SecIO
```

Figure 6. Modules to be imported by untrusted code

the sake of simplicity and modularity, we preserve the part of the library that guarantees non-interference while orthogonally introducing run-time mechanisms for declassification. More precisely, declassification policies are encoded as programs which perform run-time checks at the moment of downgrading information. In this way, declassification policies can be as flexible and general as programs! Additionally, we provide functions that automatically generate declassification policies based on some criteria. We call such programs *declassification combinators*. We provide combinators for the dimensions *what*, *when*, and *who* (*where* can be thought as a particular case of *when*). As a result, programmers can combine dimensions by combining applications of these combinators.

5.1 Escape Hatches

In our library, declassification is performed through some special functions. By borrowing terminology introduced in (Sabelfeld and Myers 2004), we call these functions "escape hatches" and we represent them as follows.

```
type Hatch s s' a b = Sec s a -> IO (Maybe (Sec s' b))
```

Escape hatches are functions that take some data at security level `s`, perform some computations with it, and then probably return a result depending if downgrading of information to security level `s'` is allowed or not. Arbitrary escape hatches can be included in the library depending on the declassification policies needed for the built applications. In fact, escape hatches are just functions. Types `IO` and `Maybe` are present in the definition of `Hatch s s' a b` in order to represent run-time checks and the fact that declassification may not be possible on some circumstances. By placing `Maybe` outside of monad `Sec s'`, the fact that declassification is possible or not is public information and programs can thus take different actions in each case. Consequently, it is important to remark that declassification policies should not depend on secret values in order to avoid unintended leaks (we give examples of such policies later). Otherwise, it would be possible to reveal information about secrets by inspecting the returned constructor (`Just` or `Nothing`) when applying escape hatches.

As mentioned in the beginning of the section, we include some *declassification combinators* that are responsible for generating escape hatches. The simplest combinator creates escape hatches that always succeed when downgrading information. Specifically, we define the following combinator.

```
hatch :: Less s' s => (a -> b) -> Hatch s s' a b
hatch f = \sa -> return(Just(return(f (reveal sa))))
```

Basically, `hatch` takes a function and returns an escape hatch that applies such function to a value of security level `s` and returns the result of that at security level `s'` where `s'` $\sqsubseteq$ `s`. Observe how the function `reveal` is used for declassification.

The idea is that the function `hatch` is used by trusted code in order to introduce a controlled amount of leaking to the attacker. Note that it is possibly dangerous for the trusted code to export a *polymorphic* escape hatch to the attacker! A polymorphic function can often be used to leak an unlimited amount of information, by for example applying it to lists of data. In general, escape hatches that are exported should be monomorphic.

5.2 The What dimension

In general, type systems that enforce declassification policies related to "what" information is released are somehow conservatives (Sabelfeld and Myers 2004; Askarov and Sabelfeld 2007; Mantel and Reinhard 2007). The main reason for that is the difficulty to statically predict how the data to be declassified is manipulated or changed by programs. Inspired by quantitative information-theoretical works (Clark et al. 2002), we focus on "how much" information can be leak instead of determining exactly "what" is leaked. In this light, we introduce the following declassification combinator.

```
ntimes :: Int -> Hatch s s' a b -> IO (Hatch s s' a b)
ntimes n f
 = do ref <- newIORef n
      return (\sa -> do k <- readIORef ref
                        if k <= 0
                           then do return Nothing
                           else do writeIORef ref (k-1)
                                   f sa )
```

Essentially, `ntimes` takes a number `n` and an escape hatch `h`, and returns a new escape hatch that produces the same result as `h` but that can only be applied at most `n` times. To achieve that, the combinator creates the reference `ref` to the number of times (`n`) that the escape hatch (`h`) can be applied. Every application of the escape hatch then checks if the maximum number of allowed applications has been reached by observing the condition `k <= 0`. Additionally, every application of the escape hatch also reduce the number of possible future applications by executing `writeIORef ref (k-1)`. The generated escape hatch returns `Nothing` if the policy is violated as a manner to avoid leaking more information than intended. Inspecting if the result of applying an escape hatch is `Nothing` or not can be considered as a covert channels by itself when happening inside of computations related to confidential data. Fortunately, escape hatches applied inside of computations depending on secrets are never executed. For instance, if we try to apply an escape hatch inside of some secret computation, it will have the type `Sec H (IO (Maybe (Sec L b)))` for some type `b`. Declassification is performed inside of the `IO` monad and it is not possible to extract `IO` computations from the monad `Sec H` unless than another escape hatches is declared to release `IO` computations. Therefore, escape hatches must be introduced to release pure values rather than side-effecting computations, which seems to be the case for most applications.

Note that the function `ntimes` is safe to be exported to the attacker, since it only restricts the use of existing hatches.

As an example of how `ntimes` can be used, we write a login program that uses the secure shadow password API described in Section 3.3. It is not possible to write such program without having means for declassification. The login program must release some information about users' passwords: if access is granted, then the attacker knows that his input matches the password, otherwise he knows that it does not. We present the program in Figure 7. Module `Policies` introduces declassification policies for our login program and states that a shadow password can be compared by equality at most three times. This module is trusted and must not be imported by untrusted code. Otherwise, attackers can create an unrestricted number of escape hatches in order to leak secrets!

```
module Policies ( declassification ) where
import SecLibTypes ; import Declassification
import SpwdData

declassification
 = ntimes 3 (hatch (\(spwd,c) -> cypher spwd == c))
   :: IO (Hatch H L (Spwd, String) Bool)


module Main ( main ) where
import Policies
import Login

main = do match <- declassification
          login match


module Login ( login ) where
import SecLibTypes ; import SecLib
import SpwdData ; import Spwd
import Maybe

check :: (?match :: Hatch H L (Spwd, Cypher) Bool)
         => Sec H Spwd -> String -> Int -> String
            -> IO ()
check spwd pwd n u =
  do acc <- ?match ((\s -> (s, pwd)) `fmap` spwd)
     if (public (fromJust acc))
               then putStrLn "Launching shell..."
               else do putStrLn "Invalid login!"
                       auth (n-1) u spwd

auth 0 _ spwd = return ()
auth n u spwd = do putStr "Password:"
                   pwd <- getLine
                   check spwd pwd n u

login match
  = do let ?match = match
       putStrLn "Welcome!"
       putStr "login:"
       u <- getLine
       src <- getSpwdName u
       case src of
            Nothing    -> putStrLn "Invalid user!"
            Just spwd  -> auth 3 u spwd
```

Figure 7. Secure login program

Module `SecLibTypes`, described in Section 4, is extended to include type definitions related to declassification as, for instance, `Hatch s s' a b`. Module `Declassification` introduces declassification combinators (e.g. `ntimes`). These modules are part of our trusted base. Module `Declassification` must not be imported by untrusted code for the same reasons given for module `Policies`. Modules `SpwdData` and `Spwd` respectively include the data type declaration of `Spwd` and the API described in Section 3.3. Module `Main` extracts declassification policies defined in `Policies` and pass them to the `login` function. In general, this module determines what functions are called from untrusted code in order to run the program. In this case, it determines that `login` must be called to perform the login procedure. Since the module imports module `Policies`, it also belongs to the trusted base. The most interesting module is `Login`. This module does not belong to our trusted base and therefore it may contain code written by possibly malicious programmers. Because declassification policies can be applied at any part of the untrusted code, we place them into implicit parameters (Lewis et al. 2000). Implicit parameters can be thought as some kind of global variables and they are declared by

```
module Bid ( bid ) where

obtainBid :: FilePath -> IO Int
obtainBid file = do s <- readFile file
                    return (read s :: Int)

bid  = do putStrLn "Bid system!"
          putStrLn "-----------"
          putStrLn ""
          putStrLn "Obtaining the bids..."
          a <- obtainBid "bidA"
          -- writeFile "bidB" (show (a+1))
          b <- obtainBid "bidB"
          putStrLn (if a > b then "A wins!"
                             else "B wins!")
```

Figure 8. Insecure bidding system

writing variable names starting with the symbol ?. Module `Login` contains three functions: `check`, `auth`, and `login`. Function `check` takes the password `spwd :: Sec H Spwd` stored in the system for the user `u :: String` and checks, by applying the escape hatch placed in `?match`, if the user's input `pwd :: String` matches the password stored in the field `cypher` of `spwd`. Assuming that is possible to perform the declassification described by `?match`, variable `acc` stores if the access is granted or not. We assume that untrusted code has access to the function `public :: Sec L a -> a` to extract the public values from monad `Sec L`. In the example, `public` is applied to values returned by `?match`. If the access is denied, `check` might give another chance to the user by calling the function `auth`. Function `auth` is responsible to ask the user's password and validates it at most `n` times. Function `login` asks for the user name and checks that the user is registered in the system by calling the function `getSpwdName` from the secure shadow password API.

Since program in Figure 7 type-checks, it respects the declassification policies defined in module `Policies`, i.e. the password can be compared for equality only three times. To illustrate that, we place our selves in the role of the attacker and modify function `check` to call `auth n u spwd` instead. As a result, it would be now possible to try as many passwords as the user wants and thus increasing the amount of information leak by unit of time. Observe that this situation is particularly dangerous when passwords have short length as PIN numbers in ATMs. Nevertheless, if we try to run the modified code, we get the message `*** Exception: Maybe.fromJust: Nothing` after the user tries more than three times to check if the access can be granted or not.

5.3 The When dimension

As a motivating example for handling this dimension, we can consider the scenario described in (Chong and Myers 2004) of a sealed auction where each bidder submits a single secret bid in a sealed envelope. Once all bids are submitted, the envelopes are opened and the bids are compared. The highest bidder wins. One security property that is important for this program is that no bidder knows any of the other bids until all the bids have been submitted. Program in Figure 8 simulates this process for two bidders: `A` and `B`. We represent envelopes as files. Function `obtainBid` opens an envelope and extracts the bid. The rest of the program is self-explanatory. It is possible to incorrectly implement the auction protocol by mistake or intentionally. For instance, if we uncommented the line in Figure 8, the program uses the bid from user `A` to make user `B` the winner. However, no information about `A`'s bid must be available until `B` submits his (her) own bid.

The library introduces the when dimension by associating events in the system that indicates at which time release of in-

formation may occur. For instance, "releasing a software key may occur after the payment has been confirmed". Inspired by (Broberg and Sands 2006), we implement boolean flags called *flow locks* [4] that, when open, allow downgrading of information.

Flow locks are introduced by the following combinator.

```
when :: Hatch s s' a b ->
        IO (Hatch s s' a b, Open, Close)
when f = do ref <- newIORef False
            return (\sa -> do b <- readIORef ref
                              if b then f sa
                                   else return Nothing
                   , writeIORef ref True
                   , writeIORef ref False)
```

Basically, `when` takes an escape hatch `h` and returns a new escape hatch that produces the same result as `h` but that has associated a flow lock to it. The combinator creates the reference `ref` to an initially close flow lock represented as `False`. The returned escape hatch can only be applied when the associated flow lock is open (i.e. the corresponding boolean flag is set to `True`). Observe that, by inspecting the value of `b`, every application of the escape hatch checks that the flow lock is open before declassifying information. The combinator also returns computations to open and close the lock, which respectively have type `Open` and `Close`. These computations must be only used by trusted code. Otherwise, the attacker can execute them at any time in the untrusted code and thus ignoring the events that indicate when declassification may occur. `Open` and `Close` are just synonymous type declarations for `IO ()`.

We can then implement a secure bidding system. We firstly define our security lattice composed by the security levels `A`, `B`, and `L`, where $\texttt{L} \sqsubseteq \texttt{A}$ and $\texttt{L} \sqsubseteq \texttt{B}$. Security levels `A` and `B` are respectively associated to information coming from users `A` and `B`, while `L` denotes public information. We implement these security levels as singleton types with constructors `A :: A`, `B :: B`, and `L::L`. The described security lattice is very simple and therefore we omit details about its implementation. The secure bidding system is shown in Figure 9. At first glance, it might seem that this implementation is much more complex than the insecure one. However, the module `Bid`, the core of the bidding system, has approximately the same size as before. The rest of the modules are related to properly setting up the security level of different resources in the program as well as the corresponding declassification policies. Module `Files` declares the security level `A` and `B` for the files that store the bids of users `A` and `B`, respectively. Module `Policies` defines the escape hatches `ha` and `hb` to release information that belongs to users `A` and `B`, respectively. Computations `openA` and `closeA` (`openB` and `closeB`) open and close the flow lock associated to `hA`(`hB`), respectively. As mentioned before, the opening and closing of locks are produced by trusted code. In this case, the opening of locks happens when bids are read from files. We then place function `obtainBid` in the trusted module `Main`. We also adapt such function to read files at security level `s` and return their contents, but opening the flow lock received as argument. Function `main` obtains the escape hatches from `declassification` and defines trusted function responsible for opening flow locks. Function `obtainBidA` (`obtainBidB`) reads the bid of user `A` (`B`) and opens the lock for releasing the bid of user `B` (`A`). Differently from the insecure version in Figure 8, function `bid` receives as arguments escape hatches and functions to obtain bids. Module `Bid` is written by the attacker or possibly

[4] The notion presented here about flow locks is not exactly the same that is introduced in Broberg and Sands's paper. For instance, their work can statically check if a program respects the declassification policies determined by the flow locks. Moreover, the state of the locks is not related with the state of programs at all. We differ from these two points due to the dynamic nature of our approach. However, the intuitive idea of allowing downgrading of information when locks are open is preserved in our implementation.

```
module Files ( bidAF, bidBF ) where
import Sec (secret, File (File) ) ; import Lattice

bidAF :: File A
bidAF = MkFile "bidA"

bidBF :: File B
bidBF = MkFile "bidB"


module Policies ( declassification ) where
import SecLibTypes ; import Declassification

declassification
 = do (pA :: Hatch A L Int Int, openA, closeA)
         <- when (hatch id)
      (pB :: Hatch B L Int Int, openB, closeB)
         <- when (hatch id)
      return (pA, openA, closeA, pB, openB, closeB)


module Main ( main ) where
import Policies ; import Files ; import SecLib
import Bid

obtainBid :: File s -> Open -> IO (Sec s Int)
obtainBid file open
  = do sec <- run (do r <- s_read file
                      return (read r :: Int))
       open
       return sec

main = do (hA, openA, closeA,
          hB, openB, closeB) <- declassification
          let obtainBidA = obtainBid bidAF openB
              obtainBidB = obtainBid bidBF openA
          bid hA obtainBidA hB obtainBidB


module Bid ( bid ) where
import SecLibTypes ; import SecLib

bid hA obtainBidA hB obtainBidB
 = do putStrLn "Bid system!"
      putStrLn "-----------"
      putStrLn ""
      putStrLn "Obtaining the bids..."
      bidA   <- obtainBidA
      -- Just cheat <- hA bidA
      bidB   <- obtainBidB
      Just seca <- hA bidA
      Just secb <- hB bidB
      putStrLn(if (public seca) > (public secb)
                  then "A wins!"
                  else "B wins!")
```

Figure 9. Secure bidding system

malicious programmer. In this module, function `bid` obtains the bids to later compare them. In order to compare bids, they need to be extracted from values of type `Sec A Int` and `Sec B Int` through the escape hatches `ha` and `hb`, respectively. It is then not possible to determine which bid is the highest before obtaining all for them. For instance, if we uncommented the line in function `bid`, we obtain a program that tries to release the bid from user `A` before getting the bid for user `B`, which is clearly a non-desirable behavior for the auction system. However, if we run the program, we get the message `*** Exception: Maybe.fromJust: Nothing` since the flow lock associated to release `A`'s bid is not open. In order to open it, we firstly need to get `B`'s bid!

To illustrate why flow locks may need to be closed, we take the example on step further by thinking of a bidding system that allows the users to bid more than once. In this case, function `bid` is called several times and flow locks related to `hA` and `hB` must be closed between each call. Otherwise, all the flow locks are open at the second call of `bid`, which allows bids to be released at any time. It is not difficult to imagine this implementation by considering that function `main` calls computations `closeA` and `closeB` before each call of `bid`.

For simplicity, we considered an auction system with only two users. However, it is possible to use flow locks when more users are present in the auction. Indeed, we can create escape hatches that are associated to as many flow locks as users. In order to do that, we can compose `when` with itself as many times as users we have in the system. In this way, the escape hatch obtained in the end is associated to as many flow locks as users. Then, when a user submits its bid, his corresponding flow lock is open.

Attackers can still write programs that wrongly implement the auction system. For instance, we can write a program that makes user `A` the winner all the time by just replacing the `if-then-else` in Figure 9 by `putStrLn "The user A wins!"`. However, user `A` is going to be the winner because the program is not implemented correctly, but not because the program "cheated" by inspecting `B`'s bid. Correctness of programs are stronger properties than those ones captured by declassification policies.

5.4 The Who dimension

In the *Decentralized Label Model* (DLM) (Myers and Liskov 1997, 1998, 2000) data is marked with a set of principals who owns the information. While executing a program, the code is authorized to act on behalf of some set of principals known as *authority*. Then, declassification makes a copy of the released data and marked it with the same principals as before the downgrading but excluding those ones appearing in the authority of the code. We do not consider situations where some principals can act on behalf of others.

Similarly to (Li and Zdancewic 2006), we adapt the idea of DLM to work on a security lattice. Authorities are assigned with a security level l in the lattice and they are able to declassify data at that security level. To achieve that, we introduce a declassification combinator that checks the authority of the code before applying an escape hatch. As indicated in (Broberg and Sands 2006), DLM can be expressed using flow locks. Fortunately, our implementation is also suitable for that. More precisely, we have the following declassification combinator.

```
data Authority s = Authority Open Close

who :: Hatch s s' a b -> IO (Hatch s s' a b, Authority s)
who f = do (whof, open, close) <- when f
           return (whof, Authority open close)

certify :: s -> Authority s -> IO a -> IO a
certify s (Authority open close) io =
  s `seq` (do open ; a <- io ; close ; return a)
```

Combinator `who` takes an escape hatch an returns another escape hatch that is associated with a flow lock. The main idea here is that the flow lock is open when the code runs under the same authority as the security level appearing as the argument of the escape hatch. The mechanisms to open and close the flow lock are placed inside of the data type `Authority s`. The constructor of this data type is not accessible for attackers. Otherwise, they can avoid the certification process to determine that some piece of code runs under some authority. Such certification process is carried out by the function `certify`. This function takes an element of security type `s`, an `Authority s`, and a computation `IO a`. In Section 4, we explain that constructors that belongs to security levels above the security level of the attacker are not exported. For instance, in the two-point lattice considered so far, attackers can only observe data at security level `L`, and thus constructor `H :: H` is not exported to untrusted modules. This assumption needs to be relaxed in order to consider this dimension for declassification. To certify that some code has authority `s`, we require that such code, possibly malicious, has only access to the constructors for security level `s` and the security level denoting public information. In this way, it is reflected that code running under authority `s` can freely declassify data from security level `s` as expected in DLM. Function `certify` checks that it receives a valid constructor for the security type `s` by applying `seq` to it, and then respectively opens and closes a flow lock before and after running the `IO` computation received as argument. Observe that this function can be freely used by attackers since it requires to provide the right constructor for some security level `s` and only authorities at that level must have it. Therefore, assignments of authorities to pieces of code must be clearly part of the trusted code.

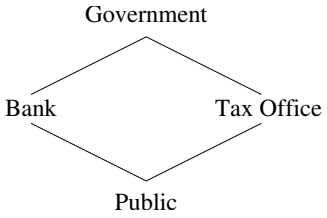


Figure 10. Security lattice

As a motivating example for this dimension, we start consider the security lattice in Figure 10. We have the security levels: Government, Bank, Tax Office, and Public to represent information related to citizens that is used for such entities. Unless that information is made public, banks cannot have access to information stored in the tax office and vice versa. Government, on the other hand, can have full access to the information stored at banks and the tax office, which can be debatable for any real government. However, we made such assumption to simplify the example and rather illustrate how functions `who` and `certify` can be used. We implement the security levels Government, Bank, Tax Office, and Public with the singleton types `G`, `B`, `T`, and `L`, respectively. The described security lattice is very simple and therefore we omit details about its implementation. We assume that the declassification polices are the followings: banks can declassify the status of their accounts (i.e. if an account is open or close), the tax office can release the address of the citizens, and the government can provide information about new immigrants to the tax office as well as revealing results of financial studies related to the economy of the country to the banks. Observe that, for instance, it is possible for the government to declassify some information to a bank, and then the bank divulges that information to the public by opening or closing some accounts. In order to avoid that, a more complex security lattice needs to be encoded. However, for simplicity, we tighten to the lattice in Figure 10. In Figure 11, we give the skeleton of an application that uses these security levels and the mentioned declassification policies. Module `Policies` declares declassification policies constructed by combinator `who`. Accounts, status of accounts, citizens, addresses, immigrants, financial studies, and outcomes of financial studies are represented by data types `Account`, `Status`, `Citizen`, `Address`, `Immigrant`, `Study`, and `Result`, respectively. Functions `status`, `address`, `immigrant`, and `study` have types `Account -> Status`, `Citizen -> Address`, `Immigrant -> Citizen`, and `Study -> Result`, respectively. These functions together with declarations of data types related to the application are placed in the module `Data`. Function `declassification`

```
module Policies ( declassification ) where
import SecLibTypes ; import Declassification
import Data

declassification
 = do (hB :: (Hatch B L Account Status),
       authBank) <- who (hatch status)
      (hT :: (Hatch T L Citizen Address),
       authTax)  <- who (hatch address)
      (hG :: (Hatch G T Immigrant Citizen),
       authG) <- who (hatch inmigrants)
      (hG' :: (Hatch G B Study Result),
       authG') <- who (hatch studies)
      return ((hB, authBank), (hT, authTax),
              (hG, authG), (hG', authG'))

module Bank ( bank ) where
import SecLibTypes ; import SecLib
import Data

bank :: B -> (Hatch B L Account Status,
              Authority B) -> IO ()
bank = ...

module TaxOffice ( taxoffice ) where

import SecLibTypes ; import SecLib
import Data

taxoffice
 :: T -> (Hatch T L Citizen Address, Authority T)
    -> IO ()
taxoffice = ...

module Government ( government ) where
import SecLibTypes ; import SecLib
import Data

government
 :: G -> (Hatch G T Immigrant  Citizen,
    Authority G) -> (Hatch G B Study Result,
    Authority G) -> IO ()
government = ...


module Main ( main ) where
import Policies ; import Lattice
import Bank ; import TaxOffice ; import Government

main
  = do (whohB,  whohT, whohG, whohG')
         <- declassification
       bank B whohB
       taxoffice T whohT
       government G whohG whohG'
       return ()
```

Figure 11. Skeleton for an application

implements the declassification policies described before. Modules `Bank`, `TaxOffice`, and `Government` are untrusted and they might include malicious code. Functions `bank`, `taxoffice`, and `government` receive the escape hatches together with values of type `Authority s` for some corresponding instances of `s`. Observe that `bank`, `taxoffice`, and `government` expects to receive the constructor for security types `B`, `T`, and `G`, respectively. In other words, the authority for `bank`, `taxoffice`, and `government` is set to `B`, `T`, and `G`, respectively. Consequently, it is then possible for those functions to apply `cerfity` with escape hatches that release information at their authority level. Module `Main` sets the authority for each of the given functions while providing the corresponding escape hatches. Observe how constructors `B :: B`, `T :: T`, and `G :: G` are given to functions `bank`, `taxoffice`, and `government`, respectively. Malicious code placed in one function only compromises confidential information related to its authority's security level. For instance, if function `bank` contains malicious code, then confidential information related to the bank may be at risk. However, if `government` is compromised, all the information in the system may be affected. Function `government` should be carefully designed, or perhaps other restrictions regarding the application of the escape hatch must be imposed in this function (see next subsection). This phenomenon also occurs in `DLM` when a process running with the authority of all the principals in the system contains malicious code.

5.5 Combining dimensions

For some application, declassification policies are not so simple as those ones captured by the dimensions of what, when, and who. For those scenarios, the user of the library has basically two options. One one hand, the user can program his own policy, which provides enough flexibility. However, such flexibility could be dangerous when declassification policies are not implemented carefully. For instance, an escape hatch must not decide if declassification is possible by inspecting confidential data. Otherwise, attackers learn information about secrets when applying escape hatches by inspecting if the returned values are `Nothing` or not. On the other hand, users can specify more interesting declassification policies by combining applications of `ntimes`, `when`, and `who` together. For instance, we extend the *what*-policy from the example given in Section 5.2 to consider more dimensions as follows.

```
comb = do h <- ntimes 3
              (hatch (\(spwd,c) -> cypher spwd == c))
          (h', open, close) <- when h
          (h'', auth) <- who h'
          return (h''':: Hatch H L (Spwd, String) Bool,
                  open, close, auth)
```

Observe how `comb` defines an escape hatch that releases information if it is applied in a piece of code with authority `H` when some events that execute `open` happened and information has not been previously released more than three times. Other combinations are also possible. To the best of our knowledge, this is the first implementation of mechanisms to enforce more than one dimension for declassification.

6. Related work

Much previous related work addresses non-interference and functional languages consider reduced programming languages (Heintze and Riecke 1998; Volpano et al. 1996; Volpano and Smith 1997) or require designing compilers from scratch (Pottier and Simonet 2002; Simonet 2003). Rather than implementing compilers, Li and Zdancewic (Li and Zdancewic 2006) show how to provide information-flow security as a library for a real programming language. They provide an implementation for Haskell based on arrows combinators(Hughes 2000), which naturally requires programmers to be familiar with arrows when writing security-related code. Their library still imposes restrictions on what kind of programs can be written. In particular, their approach does not generalize naturally in the presence of side-effects or information composed of data with different security levels. To incorporate these features, the library requires major changes as well as the introduction of new combinators (Tsai et al. 2007).

In this paper, we show that a less general notion, namely monads, is enough to provide information-flow security as a library. We propose a light-weight library ($\sim$ 400 LOC) able to handle side-effecting computations and that requires programmers to be familiar with monads rather than arrows. Moreover, by just placing data

into corresponding `Sec s` monads, our library is also able to handle data composed of elements with different security levels. However, there exists one restriction in our approach w.r.t. to the arrow approach. Since our security levels are represented by types, all of them have to be known statically at compile-time[5], whereas in the arrow approach, they can be constructed at run-time.

Abadi et. al. developed the *dependency core calculus* (DCC) (Abadi et al. 1999) based on a hierarchy of monads to guarantee non-interference. Similarly, `Sec` constructs a hierarchy of monads when applied to security levels `s`. However, DCC uses non-standard typing rules for its *bind* operations while our library just provides instances of the type class `Monad`. Tse and Zdancewic translate DCC to System F and show that non-interference can be stated using the *parametricity* theorem for F (Tse and Zdancewic 2004). They also provide an implementation in Haskell for a two-point lattice. Their implementation encodes each security level as an abstract data type constructed from functions and binding operations to compose computations with permitted flows. The same kind of ideas relies behind `Sec s`, `open`, and `close` (see Section 4). Their implementation requires, at most, $O(n^2)$ definitions for binders for n-points lattices. Since they consider the same non-standard features for binders as in DCC, they provide as many definitions for binders as different type of values produced after composing secure computations. Moreover, their implementation needs to be compiled with the flag `-fallow-undecidable-instances` in GHC. On one hand, our library requires, at most, $O(n^2)$ instantiations on the type class `Less` for n-points lattices, but it does not provide more than one definition for binders nor requires allowing undecidable instances in GHC [6]. DCC and Tse and Zdancewic's approach do not consider computations with side-effects. Moreover, Tse and Zdancewic leaves as an open question how to encode more expressive policies, such as declassification, directly in the type system of Haskell.

Harrison and Hook show how to implement an abstract operating system called *separation kernel* (Harrison and Hook 2005). Programs running under this multi-threading operating system are non-interferent. To achieve that, the authors rely on properties related to monad transformers as well as state and resumption monads. Basically, each thread is represented as an state monad that have access to the locations related to the thread's security level while state monad transformers act as parallel composition. Interleaving and communication between threads is carried out by plugging a resumption monads on top of the parallel composition of all the threads in the system. Non-interference is then enforced by the scheduler implementation, which only allow signaling threads at the same, or higher, security level as the thread that issued the signal. Different from that, our library enforces non-interference by typing. The authors also use monads differently than we do since their goals are constructing secure kernels rather than providing information-flow security as a library. For instance, we do not use state monads, state transformers, or resumption monads since we do not model threads. As a result, our library is simpler and more suitable to write sequential programs in Haskell. It is stated as a future work how to extend our library to include concurrency.

Crary et. al. design a monadic calculus for non-interference for programs with mutable state(Crary et al. 2003). Their language distinguishes between *term* and *expressions*, where terms are pure and expressions are (possibly) effectful computations. The calculus mainly tracks flow of information by inspecting the security levels of effects produced by expressions. Expressions can be included at the term level as an element of the monadic type $\bigcirc_{(r,w)} A$, which denotes a suspended computation where the security level r is an upper bound on the security levels of the store locations that the suspended computation reads, while w is a lower bound on the security level of the store locations to which it writes. Authors introduce the notion of *informativeness* in order to relax some typing rules so that reading and writing into secret store locations can be included in large computations related to public data. A type A is informative at security level r or above if its values can be used or observed by computations that may read data from security level r or above. In our library, the type `SecIO s a` makes the value of type `a` only informative at security level `s`. In principle, the value of type `a` cannot be used anywhere but inside the monad `SecIO s`. Considering a two-point lattice, we introduce the function `plug :: Less L H => SecIO H a -> SecIO L (Sec H a)` to allow reading and writing secret files into computations related to public data. Observe that the function preserves the informativeness of `a` by placing it inside of the monad `Sec H`.

Recently, several approaches have been proposed to dynamically enforce non-interference (Guernic et al. 2006; Shroff et al. 2007; Nair et al. 2007). In order to be sound, these approaches still need to perform some static analysis prior to or at run-time. Authors argue, in one way or another, that their methods are more precise than just applying an static analysis to the whole program. For instance, if there is an insecure piece of dead code in a program, most of the static analysis techniques will reject that program while some of their approaches will not. The reason for that relies in the fact that dead code is generally not executed and therefore not analyzed by dynamic enforcement mechanisms. Our library also combines static and dynamic techniques but in a different way. Non-interference is statically enforced through type-checking while run-time mechanisms are introduced for declassfication. By dynamically enforcing declassification policies, we are able to modularly extend the part of the library that enforce non-interference to add downgrading of information and being able to enforce several dimensions for declassification in a flexible and simple manner. To the best of our knowledge, this is the first implementation of declassification policies that are enforced at run-time and the first implementation that allows combining dimensions for declassifications.

7. Conclusions

We have presented a light-weight library for information-flow security in Haskell. Based on specially designed monads, the library guarantees that well-typed programs are non-interferent; i.e. secret data is not leaked into public channels. When intended release of information is required, the library also provides novel means to specify declassification policies, which comes from the fact that policies are dynamically enforced and it is possible to construct complex policies from simple ones in a compositional manner.

Taking ideas from the literature, we show examples of declassification policies related to what, when, and by whom information is released. The implementation of the library and the examples described in this paper are publicly available in (Russo et al. 2008a). The well-known concept of monads together with the light-weight and flexible characteristic of our approach makes the library suitable to build Haskell applications where confidentiality of data is an issue.

Acknowledgments

We wish to thank to Aslan Askarov, Ulf Norell, Andrei Sabelfeld, David Sands, Josef Svenningsson, and the anonymous reviewers for useful comments and discussions about this work. This work was funded in part by the Information Society Technologies program of the European Commission,

[5] We are investigating the use of polymorphic recursion to alleviate this – this remains future work however.

[6] All the code shown in the paper works with the Glasgow Haskell Compiler (GHC) with the flag `-fglasgow-exts`

Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM.

N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In Peter Sestoft, editor, *Proc. European Symp. on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *QAPL'01, Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state, 2003.

D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Annual Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 75–89. Springer-Verlag, December 2006.

W. L. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 16–30, Washington, DC, USA, 2005. IEEE Computer Society.

N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

M. H. Jackson. Linux shadow password howto. Available at `http://tldp.org/HOWTO/Shadow-Password-HOWTO.html`, 1996.

J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, New York, NY, USA, 2000. ACM.

P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

P. Li and S. Zdancewic. Arrows for secure information flow. Available at `http://www.seas.upenn.edu/~lipeng/homepage/lz06tcs.pdf`, 2007.

Local Root Exploit. Linux kernel 2.6 local root exploit. Available at `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=465246`, February 2008.

H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In Rocco De Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 141–156. Springer, 2007. ISBN 978-3-540-71314-2.

A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

S. K. Nair, P. N.D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, September 2007.

A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 364–372, New York, NY, USA, 2005. ACM.

B. C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004. ISBN 0262162288.

F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. Software release and documentation. Available at `http://www.cs.chalmers.se/~russo/seclib.htm`, 2008a.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. Technical Report. Chalmers University of Technology. To appear., October 2008b.

A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE Computer Society, 2005.

P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 203–217, 2007.

V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.

A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130313580.

T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007.

S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 115–125, New York, NY, USA, 2004. ACM.

D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997.

D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

P. Wadler. Monads for functional programming. In *Marktoberdorf Summer School on Program Design Calculi*, August 1992.

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 ()
```

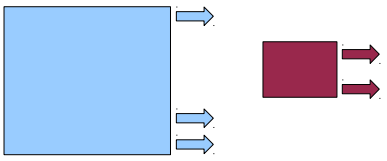
```
readFile :: FilePath -> IO String

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

```
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

```
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
```

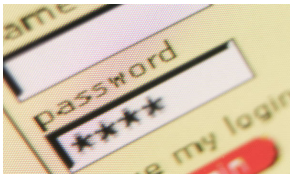
It provides a function with the type requested by module CopyT

- 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.

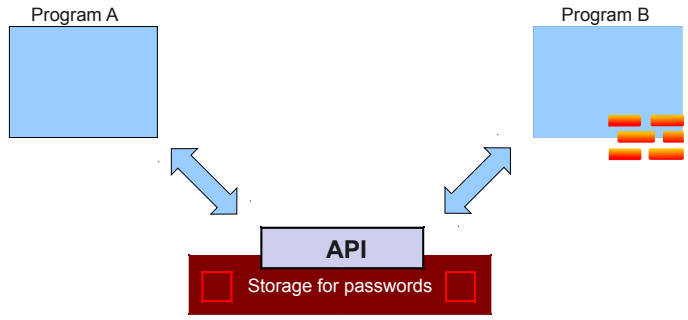
Password Administrator

- 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`

```
#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)

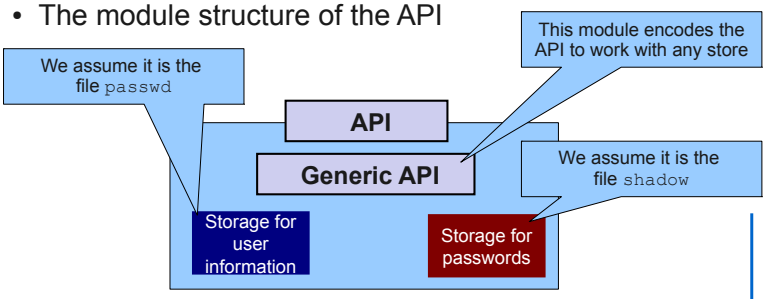
More graphically



	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

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

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

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

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

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

Store for user information

Store for password

Store for password

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

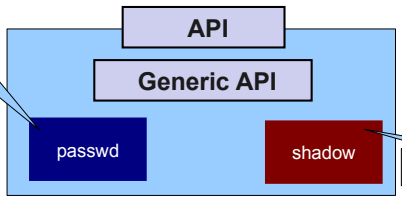
The module applies the generic API interface to specific stores

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 ]
```

What is this?

```
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 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

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

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

-- 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]
```

Store for user information

Store for password

This function does not write to any file

Store for password

This function writes to a secret file

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' ,  elem` ns)
                                       | n' <- [1..], 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 -> 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 :: Sec H [(UID, Cypher)]`

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

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

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

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

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

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

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

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

`SecIO`

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

`Sec H`

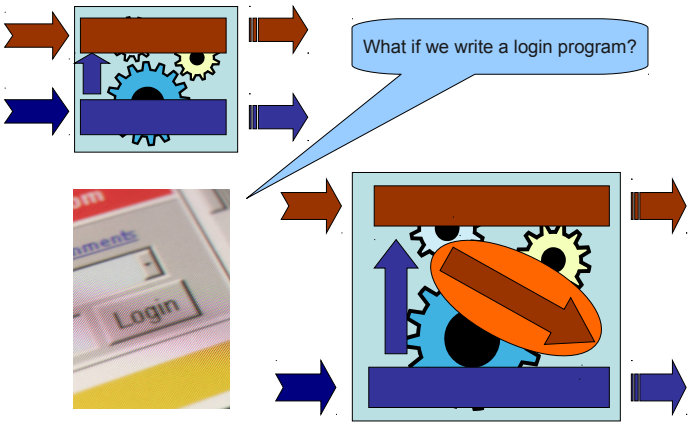
```
parse_passwd :: FilePath -> SecIO s [(Name,UID)]
parse_shadow :: FilePath -> SecIO s (Sec H [(UID,Cypher)])
```

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)

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

Escape hatch

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 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 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)

```
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
```

The type of the function provided by the untrustworthy

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

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

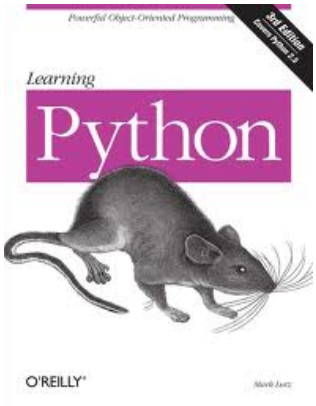
Python in a Nutshell

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

```
$ 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
>>>
```

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

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


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

```
$ 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
>>>
```

```
x = "Hello word!"

y = 3

def f(x):
    return x
```


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
>>>
```

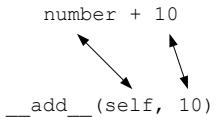
It supports multiple-inheritance. For that, it uses the C3 Method Resolution algorithm

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

```
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:

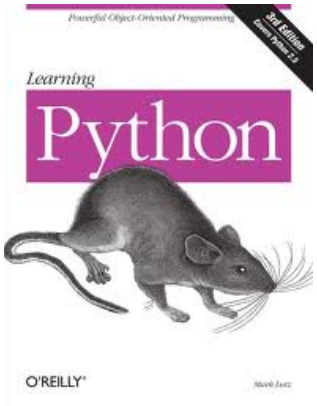
```
def id(x):
    return x

id = debug(id)
```


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)

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!



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
>>>
```

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

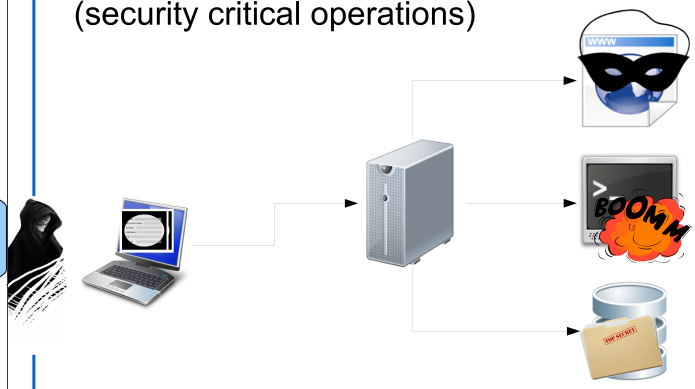
Consequences of Improper Input Validation

- ***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**

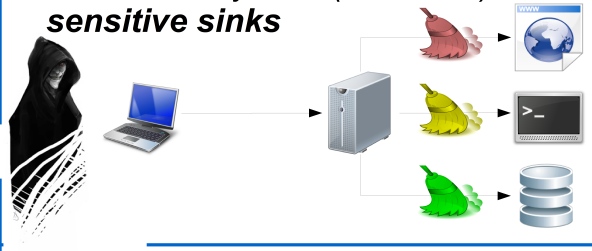
Scenarios

- Web applications with sensitive sinks (security critical operations)



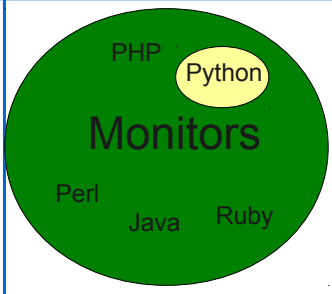
Security Policy

- 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)

Tainted data from such sources is associated with every tag

- Sources of tainted data

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

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


Taint Mode in Python (API)

Import the library

```
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
```

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)

```
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

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

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


Taint Mode in Python (API)

```
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

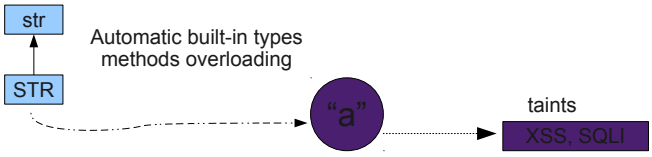
- Taint-aware functions

```
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 (Lucian , 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

$$\forall m_1, m_2 \cdot \; m_1 =_L m_2, \; < c_w, m_1 > \rightarrow^* < \texttt{stop}, m_1' >, \; < c_w, m_2 > \rightarrow^* < \texttt{stop}, m_2' >, \; \Rightarrow m_1' =_L m_2'$$

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

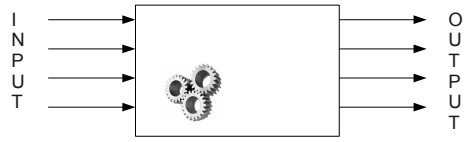
The Erasure Library in a Nutshell

[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

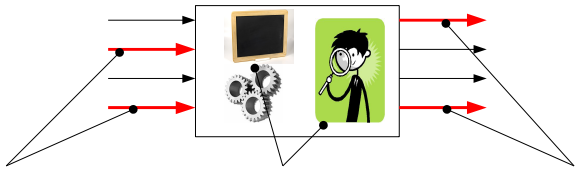
The Erasure Library

- 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



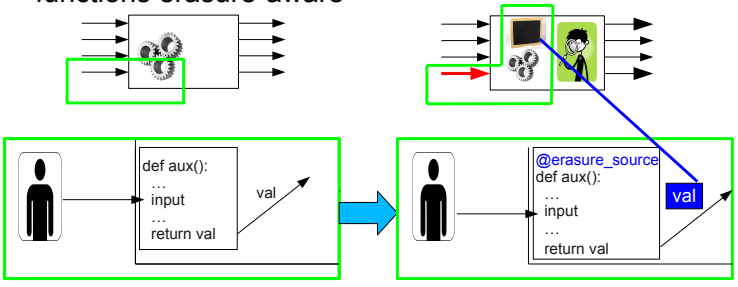
Denote entry points for *erasure-aware* information (sensitive data)

Track the propagation of *erasure-aware* data inside the system. Implementing the concrete data removal operation

Specify which output actions we need to "observe"

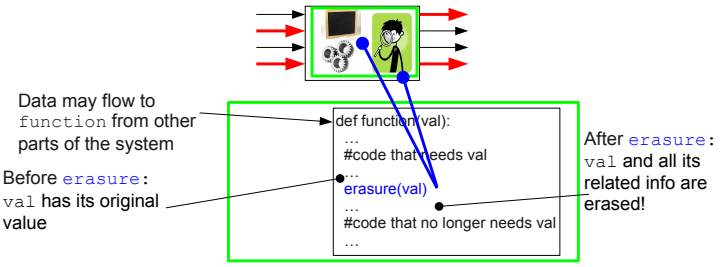
API: Indicating Erasure-aware Data

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



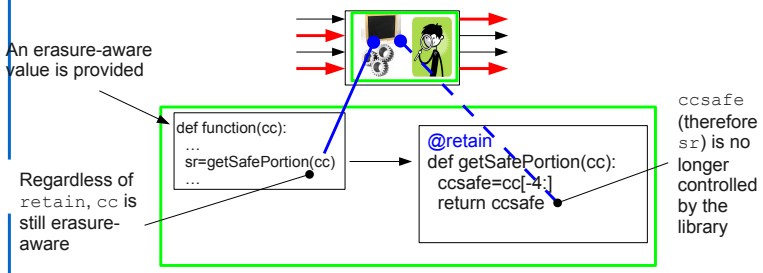
API: Erasing information

- 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?

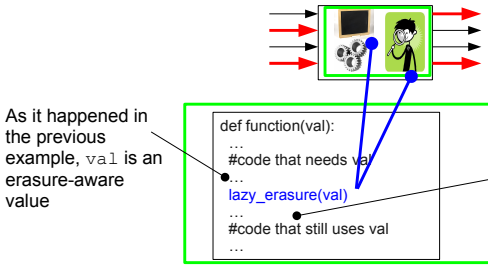
- The policies we can implement with the given API are triggered when `erasure` is executed
- There are other policies that programmers might need and are erasure-specific:
  - "Erase `sensitive_val` in 5 days"
  - "Erase `sensitive_val` if a low privileged user is trying to get the data"
- Previous 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

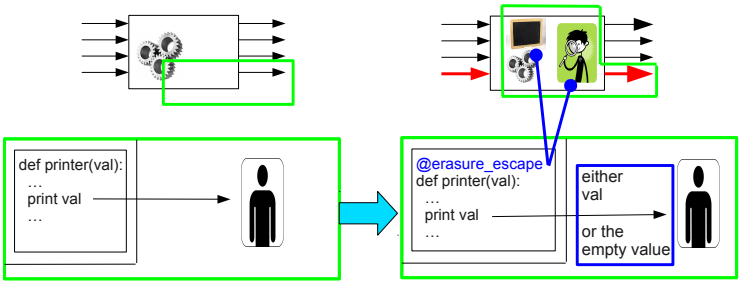
Lazy API: `lazy_erasure`

- `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`



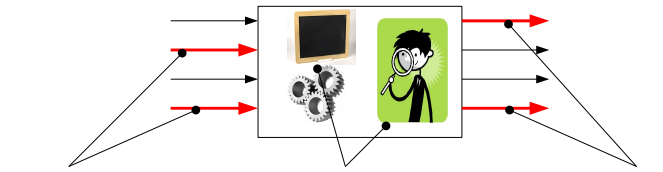
Example

```
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



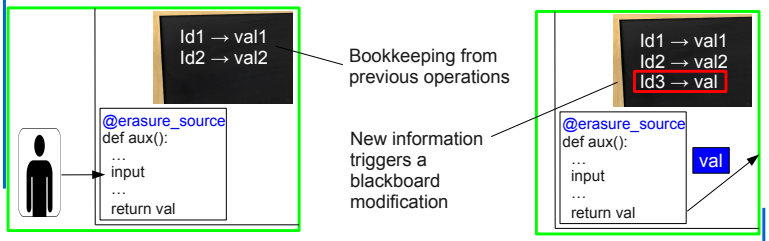
Denote entry points for *erasure-aware* information (sensitive data)

Track the propagation of *erasure-aware* data inside the system. Implementing the concrete data removal operation

Specify which output actions we need to "observe"

Who is  implemented?

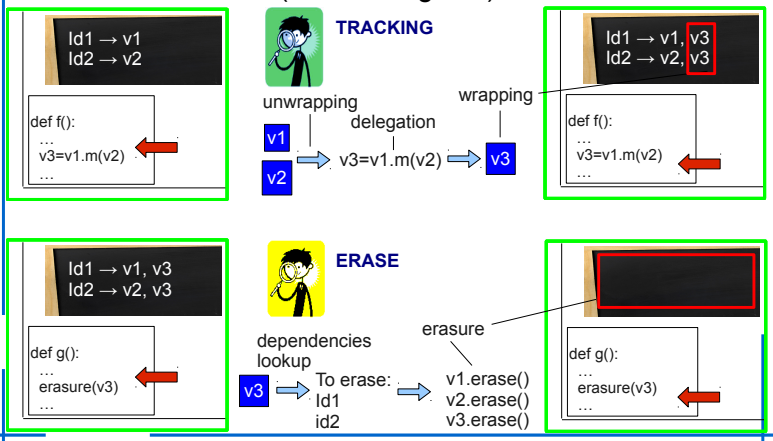
- We need to keep track of dependencies among erasure-aware values
- This means we need to identify them uniquely
- The blackboard keeps track of identities



- 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

A Taint Mode for Python via a Library

Juan José Conti[1] and Alejandro Russo[2]

[1] Universidad Tecnológica Nacional, Facultad Regional Santa Fe, Argentina
[2] Chalmers University of Technology, Sweden

Abstract. Vulnerabilities in web applications present threats to on-line systems. SQL injection and cross-site scripting attacks are among the most common threats found nowadays. These attacks are often result of improper or none input validation. To help discover such vulnerabilities, popular web scripting languages like Perl, Ruby, PHP, and Python perform taint analysis. Such analysis is often implemented as an execution monitor, where the interpreter needs to be adapted to provide a taint mode. However, modifying interpreters might be a major task in its own right. In fact, it is very probably that new releases of interpreters require to be adapted to provide a taint mode. Differently from previous approaches, we show how to provide taint analysis for Python via a library written entirely in Python, and thus avoiding modifications in the interpreter. The concepts of classes, decorators and dynamic dispatch makes our solution lightweight, easy to use, and particularly neat. With minimal or none effort, the library can be adapted to work with different Python interpreters.

1 Introduction

Over the past years, there has been a significant increase on the number of activities performed on-line. Users can do almost everything using a web browser (e.g. watching videos, listening to music, banking, booking flights, planing trips, etc). Considering the size of Internet and its number of users, web applications are probably among the most used pieces of software nowadays. Despite its wide use, web applications suffer from vulnerabilities that permit attackers to steal confidential data, break integrity of systems, and affect availability of services. When development of web applications is done with little or no security in mind, the presence of security holes increases dramatically. Web-based vulnerabilities have already outplaced those of all other platforms [4] and there are no reasons to think that this tendency has changed [12].

According to OWASP [32], cross-site scripting (XSS) and SQL injection (SQLI) attacks are among the most common vulnerabilities on web applications. Although these attacks are classified differently, they are produced by the same reason: *user supplied data is sent to sensitive sinks without a proper sanitation*. For example, when a SQL query is constructed using an unsanitize string provided by a user, SQL injection attacks are likely to occur. To harden applications against these attacks, the implementations of some popular web scripting languages perform taint analysis in a form of execution monitors [23, 2]. In that manner, not only run interpreters code, but they also perform security checks. Taint analysis can also be provided through static analysis [15, 16]. Nevertheless, execution monitors usually produce less false alarms than traditional static techniques [28]. In particular, static techniques cannot deal with dynamic

code evaluation without being too conservative. Most of the modern web scripting languages are capable to dynamically execute code. In this paper, we focus on dynamic techniques.

Taint analysis is an automatic approach to find vulnerabilities. Intuitively, taint analysis restricts how tainted or untrustworthy data flow inside programs. Specifically, it constrains data to be untainted (trustworthy) or previously sanitized when reaching sensitive sinks. Perl was the first scripting language to provide taint analysis as an special mode of the interpreter called *taint mode* [6]. Similar to Perl, some interpreters for Ruby [30], PHP [22], and recently Python [17] have been carefully modified to provide taint modes. Adapting interpreters to incorporate taint analysis present two major drawbacks that directly impact on the adoption of this technology. Firstly, incorporating taint analysis into an interpreter might be a major task in its own right. Secondly, it is very probably that it is necessary to repeatedly adapt an interpreter at every new version or release of it.

Rather than modifying interpreters, we present how to provide a taint mode for Python via a library written entirely in Python. Python is spreading fast inside web development [1]. Besides its successful use, Python presents some programming languages abstractions that makes possible to provide a taint mode via a library. For example, Python decorators [20] are a non-invasive and simple manner to declare sources of tainted data, sensitive sinks, and sanitation functions. Python's object-oriented and dynamic typing mechanisms allows the execution of the taint analysis with almost no modifications in the source code.

```
1 import sys
2 import os
3
4 usermail = sys.argv[1]
5 file = sys.argv[2]
6
7 cmd = 'mail -s "Requested file" '
8       + usermail + ' < ' + file
9 os.system(cmd)
```

Fig. 1. Code for `email.py`

The library provides a general method to enhance Python's built-in classes with tainted values. In general, taint analysis tends to only consider strings or characters [23, 22, 14, 17, 13, 29]. In contrast, our library can be easily adapted to consider different built-in classes and thus providing a taint analysis for a wider set of data types. By only considering tainted strings, the library provides a similar analysis than in [17], but without modifying the Python interpreter. To the best of our knowledge, a library for taint analysis has not been considered before.

1.1 A motivating example

We present an example to motivate the use of taint analysis in order to discover and repair vulnerabilities. The example considers an scenario of a web application where users can send their remotely stored files by email. Figure 1 shows the simple module `email.py` that is responsible to perform such task. For simplicity, the code takes the user input from the command line (lines 4 and 5) rather than from the web server. Figure 2 shows some invocations to the module from the shell prompt. Line 1 shows a request from Alice to send her own file `reportJanuary.xls` to her email address

```
1 python email.py alice@domain.se ./reportJanuary.xls
2 python email.py devil@evil.com '/etc/passwd'
3 python email.py devil@evil.com '/etc/passwd ; rm -rf / '
```

Fig. 2. Different invocations for `email.py`

`alice@domain.se`. In this case, Alice's input produces a behavior which matches the intention of the module. In contrast, lines 2 and 3 show how attackers can provide particular inputs to exploit unintended or unforeseen behaviors of `email.py`. Line 2 exploits the fact that `email.py` was written assuming that users only request their own files. Observe how `devil@evil.com` gets information regarding users accounts by receiving the file `/etc/passwd`. Line 3 goes an step further and injects the command `rm -rf /` after sending the email. These attacks demonstrate how, what was intended to be a simple email client, can become a web-based file browser or a terminal. To avoid these vulnerabilities, applications need to rigorously check for malicious data provided by users or any other untrustworthy source. Taint analysis helps to detect when data is not sanitize before it is used on security critical operations. In Section 2.2, we show how to harden `email.py` in order to reject the vulnerabilities shown in Figure 1.

The paper is organized as follows. Section 2 outlines the library API. Section 3 describes the most important implementation details of our approach. Section 4 covers related work. Section 5 provides some concluding remarks.

2 A library for taint analysis

On most situations, taint analysis propagates taint information on assignments. Intuitively, when the right-hand side of an assignment uses a tainted value, the variable appearing on the left-hand side becomes tainted. Taint analysis can be seen as an information-flow tracking mechanism for integrity [27]. In fact, taint analysis is just a mechanism to track explicit flows, i.e. direct flows of information from one variable to another. Taint analysis tends to ignore implicit flows [11], i.e. flows through the control-flow constructs of the language. Figure 3 presents an implicit flow. Variables `t` and `u` are tainted and untainted, respectively. Observe that variable `u` is untainted after the execution of the branch since an untainted value (`'a'` or `''`) is assigned to it. Yet, the value of the tainted variable `t` is copied into the untainted variable `u` when `t == 'a'`. It is not difficult to imagine programs that circumvent the taint analysis by copying the content of tainted strings into untainted ones by using implicit flows[26].

```
if t == 'a': u = 'a'
else: u = ''
```

Fig. 3. An implicit flow

In scenarios where attackers has full control over the code (e.g. when the code is potentially malicious), implicit flows present an effective way to circumvent the taint analysis. In this case, the attackers' goal is to craft the code and input data in order to circumvent security mechanisms. There is a large body of literature on the area of language-based security regarding how to track implicit flows [27].

```
1  v = taint(d)
2
3  web.input = untrusted(web.input)
4
5  @untrusted
6  def f(...) :
7      ...
8
9  class MyProtocol(LineOnlyReceiver):
10     @untrusted_args([1])
11     def lineReceived(self, line):
12     ...
13 eval = ssink(T)(eval)
14
15 @ssink(T)
16 def f(...) :
17     ...
18
19 w = cleaner(T)(wash)
20
21 @cleaner(T)
22 def f(...) :
23     ...
```

Fig. 4. API for taint analysis

There exists scenarios where the code is non-malicious, i.e. written without malice. Despite the good intentions and experience of programmers, the code might still contain vulnerabilities as the ones described in Section 1.1. The attackers' goal consists on craft input data in order to exploit vulnerabilities and/or corrupt data. In this scenario, taint analysis certainly helps to discover vulnerabilities. How dangerous are implicit flows in non-malicious code? We argue that they are frequently harmless [26]. The reason for that relies on that non-malicious programmers need to write a more involved, and rather unnatural, code in order to, for instance, copy tainted strings into untainted ones. In contrast, to produce explicit flows, programmers simply need to forget a call to some sanitization function. For the rest of the paper, we consider scenarios where the analyzed code is non-malicious.

2.1 Using the library

The library is essentially a series of functions to mark what are the sources of untrustworthy data, sensitive sinks, and sanitation functions. Figure 4 illustrates how the API works. Symbol `...` is a place holder for code that is not relevant to explain the purpose of the API. We assume that `v` is a variable, `d` is an string or integer, and `f` is a user-defined function. Symbol `T` represents a tag. By default, tags can take values `XSS`, `SQLI`, `OSI` (Operating System Injection), and `II` (Interpreter Injection). These values are used to indicate specific vulnerabilities that could be exploited by tainted data. For instance, tainted data associated with tag `SQLI` is likely to exploit SQL injection vulnerabilities. Function `taint` is used to taint values. For example, line 1 taints variable `d`. The call to `untrusted(web.input)` establishes that the results produced by `web.input` are tainted. Line 5 shows how `untrusted` can be used to mark the values returned by function `f` as untrustworthy. Observe the use of the decorator syntax (`@untrusted`). Function `untrusted_args` is used to indicate which functions' arguments must be tainted. This primitive is particularly useful when programming frameworks require to redefine some methods in order to get information from external sources. As an example, Twisted[3], a framework to develop network applications, calls

method `lineReceived` from the class `LineOnlyReceiver` every time that an string is received from the network. Lines 9–12 extend the class `LineOnlyReceiver` and implement the method `lineReceived`. Line 10 taints the data that Twisted takes from the network. Functions `taint`, `untrusted`, and `untrusted_args` associate all the tags to the tainted values. After all, untrustworthy data might exploit any kind of vulnerability. Line 13 marks `eval` as a sensitive sink. If eval receives a tainted data with the tag `T`, a possible vulnerability T is reported. Line 15 shows how to use `ssink` with the decorator syntax. Line 19 shows how `cleaner` establishes that function `wash` sanitizes data with tag `T`. As a result of that, function `w` removes tag `T` from tainted values. Line 21 shows the use of `cleaner` with the decorator syntax. Sensitive sinks and sanitization functions can be associated with more than one kind of vulnerabilities by just nesting decorators, i.e. `ssink(OSI)(ssink(II)(critical_operation))`.

2.2 Hardening `email.py`

We revise the example in Section 1.1. Figure 5 shows the secure version of the code given in Figure 1. Line 3 imports the library API. Line 4 imports some sanitization functions. Line 6 marks command `os.system` (capable to run arbitrary shell instructions) as a sensitive sink to `OSI` attacks. Tainted values reaching that sink must not contain the tag `OSI`. Lines 7 and 8 establish that functions `s_usermail` and `s_file` sanitize data in order to avoid `OSI` attacks. Lines 10 and 11 mark user input as untrustworthy. When executing the program, the taint analysis raises an alarm on line 16. The reason for that is that variable `cmd` is tainted with the tag `OSI`. Indeed, `cmd` is constructed from the untrustworthy values `usermail` and `file`. If we uncomment the lines where sanitization takes place (lines 12 and 13), the program runs normally, i.e. no alarms are reported. Observe that the main part of the code (lines 14–16) are the same than in Figure 1.

```
1  import sys
2  import os
3  from taintmode import *
4  from sanitize  import *
5
6  os.system = ssink(OSI)(os.system)
7  s_usermail = cleaner(OSI)(s_usermail)
8  s_file = cleaner(OSI)(s_file)
9
10 usermail = taint(sys.argv[1])
11 file = taint(sys.argv[2])
12 #usermail = s_usermail(usermail)
13 #file = s_file(file)
14 cmd = 'mail -s "Requested file" '
15       + usermail + ' < ' + file
16 os.system(cmd)
```

Fig. 5. Secure version of module `email.py`

3 Implementation

In this section we present the details of our implementation. Due to lack of space, we show the most interesting parts. The full implementation of the library is publicly available at [10].

```
1  def taint_class(klass, methods):
2      class tklass(klass):
3        def __new__(cls, *args, **kwargs):
4            self = super(tklass, cls).__new__(cls, *args, **kwargs)
5            self.taints = set()
6            return self
7      d = klass.__dict__
8      for name, attr in [(m, d[m]) for m in methods]:
9          if inspect.ismethod(attr) or
10            inspect.ismethoddescriptor(attr):
11               setattr(tklass, name, propagate_method(attr))
12     if '__add__' in methods and '__radd__' not in methods:
13         setattr(tklass, '__radd__',
14                 lambda self, other: tklass.__add__(tklass(other),
15                 self))
16     return tklass
```

Fig. 6. Function to generate taint-aware classes

One of the core part of the library deals with how to keep track of taint information for built-in classes. The library defines subclasses of built-in classes in order to indicate if values are tainted or not. An object of these subclasses posses an attribute to indicate a set of tags associated to it. Objects are considered untainted when the set of tags is empty. We refer to these subclasses as *taint-aware classes*. In addition, the methods inherited from the built-in classes are redefined in order to propagate taint information. More specifically, methods that belong to taint-aware classes return objects with the union of tags found in their arguments and the object calling the method. In Python, the dynamic dispatch mechanism guarantees that, for instance, the concatenations of untainted and tainted strings is performed with calls to methods of taint-aware classes, which properly propagates taint information.

3.1 Generating taint-aware classes

Figure 6 presents a function to generate taint-aware classes. The function takes a built-in class (`klass`) and a list of its methods (`methods`) where taint propagation must be performed. Line 2 defines the name of the taint-aware class `tklass`. Objects of `tklass` are associated to the empty set of tags when created (lines 3–6). Attribute `taints` is introduced to indicate the tags re-

```
def propagate_method(method):                          1
    def inner(self, *args, **kwargs):                  2
        r = method(self, *args, **kwargs)              3
        t = set()                                      4
        for a in args:                                 5
            collect_tags(a, t)                         6
        for v in kwargs.values():                      7
            collect_tags(v, t)                         8
        t.update(self.taints)                          9
        return taint_aware(r,t)                        10
    return inner                                       11
```

Fig. 7. Propagation of taint information

lated to tainted values. Using Python's introspection features, variable `d` contains, among other things, the list of methods for the built-in class (line 7). For each method in the built-in class and in `methods` (lines 8–10), the code adds to `tklass` a method that has the same name and computes the same results but also propagates taint information (line 11). Function `propagate_method` is explained below. Lines 12–15 set method `__radd__` to taint-aware classes when built-in classes do not include that method but `__add__`. Method `__radd__` is called to implement the binary operations with reflected (swapped) operands[3]. For instance, to evaluate the expression `x+y`, where `x` is a built-in string and `y` is a taint-aware string, Python calls `__radd__` from `y` and thus executing `y.__radd__(x)`. In that manner, the taint information of `y` is propagated to the expression. Otherwise, the method `x.__add__(y)` is called instead, which results in an untainted string. Finally, the taint-aware class is returned (line 16).

The implementation of `propagate_method` is shown in Figure 7. The function takes a method and returns another method that computes the same results but propagates taint information. Line 3 calls the method received as argument and stores the results in `r`. Lines 4–9 collect the tags from the current object and the method's arguments into `t`. Variable `r` might refer to an object of a built-in class and therefore not include the attribute `taints`. For that reason, function `taint_aware` is designed to transform objects from built-in classes into taint-aware ones. For example, if `r` refers to a list of objects of the class `str`, function `taint_aware` returns a list of objects of the taint-aware class derived from `str`. Function `taint_aware` is essentially implemented as a structural mapping on list, tuples, sets, and dictionaries. The library does not taint built-in containers, but rather their elements. This is a design decision based on the assumption that non-malicious code does not exploit containers to circumvent the taint analysis (e.g. by encoding the value of tainted integers into the length of lists). Otherwise, the implementation of the library can be easily adapted. Line 11 returns the taint-aware version of `r` with the tags collected in `t`.

```
STR = taint_class(str, str_methods)
INT = taint_class(int, int_methods)
```

Fig. 8. Taint-aware classes for strings and integers

To illustrate how to use function `taint_class`, Figure 8 produces taint-aware classes for strings and integers, where `str_methods` and `int_methods` are lists of methods for the classes `str` and `int`, respectively. Observe how the code presented in Figures 6 and 7 is general enough to be applied to several built-in classes.

3.2 Decorators

Except for `taint`, the rest of the API is implemented as decorators. In our library, decorators are high order functions [7], i.e. functions that take functions as arguments and return functions. Figure 9 shows

```
def untrusted(f):                                  1
    def inner(*args, **kwargs):                    2
        r = f(*args, **kwargs)                     3
        return taint_aware(r, TAGS)                4
    return inner                                   5
```

Fig. 9. Code for `untrusted`

[3] The built-in class for strings implements all the reflected versions of its operators but `__add__`.

the code for `untrusted`. Function f, given as an argument, is the function that returns untrustworthy results (line 1). Intuitively, function `untrusted` returns a function (`inner`) that calls function `f` (line 3) and taints the values returned by it (line 4). Symbol `TAGS` is the set of all the tags used by the library. Readers should refer to [10] for the implementation details about the rest of the API.

3.3 Taint-aware functions

Several dynamic taint analysis [23, 22, 16, 17, 13, 29] do not propagate taint information when results different from strings are computed from tainted values. (e.g. the length of a tainted string is usually an untainted integer). This design decision might affect the abilities of taint analysis to detect vulnerabilities. For instance, taint analysis might miss dangerous patterns when programs encode strings as lists of numbers. A common workaround to this problem is to mark functions that perform encodings of strings as sensitive sinks. In that manner, sanitization must occur before strings are represented in another format. Nevertheless, this approach is unsatisfactory: the intrinsic meaning of sensitive sinks may be lost. Sensitive sinks are security critical operations rather than functions that perform encodings of strings. Our library provides means to start breaching this gap.

```
1  def propagate_func(original):
2      def inner(*args, **kwargs):
3          t = set()
4          for a in args:
5              collect_tags(a,t)
6          for v in kwargs.values():
7              collect_tags(v,t)
8          r = original(*args,**kwargs)
9          if t == set([]):
10             return r
11         return taint_aware(r,t)
12     return inner
```

Fig. 10. Propagation of taint information among possibly different taint-aware objects

Figure 10 presents a general function that allows to define operations that return tainted values when their arguments involve taint-aware objects. As a result, it is possible to define functions that, for instance, take tainted strings and return tainted integers. We classify this kind of functions as *taint-aware*.

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

Fig. 11. Taint-aware functions for strings and integers

Similar to the code shown in Figure 7, `propagate_func` is a high order function. It takes function `f` and returns another function (`inner`) able to propagate taint information from the arguments to the results. Lines 3–7 collect tags from the arguments. If the set of collected tags is empty, there are no tainted values involved and therefore no taint propagation is performed (lines 9–10). Otherwise, a taint-aware version of the results is returned with the tags collected in the arguments (line 11).

To illustrate the use of `propagate_func`, Figure 11 shows some taint-aware functions for strings and integers. We redefine the standard functions to compute lengths of lists (`len`), the ASCII code of a character (`chr`), and its inverse (`ord`). As a result, `len(taint('string'))` returns the tainted integer 6. It is up to the users of the

library to decide which functions must be taint-aware depending on the scenario. The library only provides redefinition of standard functions like the ones shown in Figure 11.

3.4 Scope of the library

In Figure 6, the method to automatically produce taint-aware classes does not work with booleans. The reason for that is that class `bool` cannot be subclassed in Python [4]. Consequently, our library cannot handle tainted boolean values. We argue that this shortcoming does not restrict the usability of the library for two reasons. Firstly, different from previous approaches [23, 22, 16, 17, 13, 29], the library can provide taint analysis for several built-in types rather than just strings. Secondly, we consider that booleans are typically used on guards. Since the library already ignores implicit flows, the possibilities to find vulnerabilities are not drastically reduced by disregarding taint information on booleans.

When generating the taint-aware class `STR` (Figure 8), we found some problems when dealing with some methods from the class `str`. Python interpreter raises exceptions when methods `__nonzero__`, `__reduce__`, and `__reduce_ex__` are redefined. Moreover, when methods `__new__`, `__init__`, `__getattribute__`, and `__repr__` are redefined by function `taint_class`, an infinite recursion is produced when calling any of them. As for `STR`, the generation of the taint-aware class `INT` exposes the same behavior, i.e. the methods mentioned before produce the same problems. We argue that this restriction does not drastically impact on the capabilities to detect vulnerabilities. Methods `__new__` is called when creating objects. In Figure 6, taint-aware classes define this method on line 3. Method `__init__` is called when initializing objects. Python invokes this method after an object is created and programs do not usually called it explicitly. Method `__getattribute__` is used to access any attribute on a class. This method is automatically inherited from `klass` and it works as expected for taint-aware classes. Method `__nonzero__` is called when objects need to be converted into a boolean value. As mentioned before, the analysis ignores taint information of data that is typically used on guards. Method `__repr__` pretty prints objects on the screen. In principle, developers should be careful to not use calls to `__repr__` in order to convert tainted objects into untainted ones. However, this method is typically used for debugging [5]. Methods `__reduce__` and `__reduce_ex__` are used by Pickle [6] to serialize strings. Given these facts, the argument `method` in function `taint_class` establishes the methods to be redefined on taint-aware classes (Figure 6). This argument is also useful when the built-in classes might vary among different Python interpreters. It is future work to automatically determine the lists of methods to be redefined for different built-in classes and different versions of Python.

It is up to the users of the library to decide which built-in classes and functions must be taint-aware. This attitude comes from the need of being flexible and not affecting

[4] `http://docs.python.org/library/functions.html#bool`

[5] `http://docs.python.org/reference/datamodel.html`

[6] An special Python module

performance unless it is necessary. Why users interested on taint analysis for strings should accept run-time overheads due to tainted integers?

It is important to remark that the library only tracks taint information in the source code being developed. As a consequence, taint information could be lost if, for example, taint values are given to external libraries (or libraries written in other languages) that are not taint-aware. One way to tackle this problem is to augment the library functions to be taint-aware by applying `propagate_func` to them.

As a future work, we will explore if it is possible to automatically define taint-aware functions based on the built-in functions (found in the interpreter) and taint-aware classes in order to increase the number of taint-aware functions provided by the library. At the moment, the library provides taint-aware classes for strings, integers, floats, and unicode as well as some taint-aware functions (e.g. `len`, `chr`, and `ord`).

4 Related Work

A considerable amount of literature has been published on taint analysis. Readers can refer to [8] for a description of how this technique has been applied on different research areas. In this section, we focus on analyses developed for popular web scripting languages.

Perl [23] was the first scripting language to include taint analysis as a native feature of the interpreter. Perl taint mode marks strings originated from outside a program as tainted (i.e. inputs from users, environment variables, and files). Sanitization is done by using regular expressions. Writing to files, executing shell commands, and sending information over the network are considered sensitive sinks. Differently, our library gives freedom to developers to classify the sources of tainted data, sanitization functions, and sensitive sinks. Similar to Perl, Ruby [30] provides support for taint analysis. Ruby's taint mode, however, performs analysis at the level of objects rather than only strings. Both, Perl and Ruby, utilize dynamic techniques for their analyses.

Several taint analysis have been developed for the popular scripting language PHP. Aiming to avoid any user intervention, authors in [15] combine static and dynamic techniques to automatically repair vulnerabilities in PHP code. They propose to use static analysis (type-system) in order to insert some predetermined sanitization functions when tainted values reach sensitive sinks. Observe that the semantic of programs might be changed when inserting calls to sanitization functions, which constitutes the dynamic part of the analysis in [15]. Our approach, on the other hand, does not implement a type-system and only reports vulnerabilities, i.e. it is up to developers to decide where, and how, sanitization procedures must be called. In [22], Nguyen-Toung et al. adapt the PHP interpreter to provide a dynamic taint analysis at the level of characters, which the authors call *precise tainting*. They argue that precise tainting gains precision over traditional taint analyses for strings. Authors need to manually exploit, when feasible, semantics definitions of functions in order to accurately keep track of tainted characters. Our approach, on the other hand, uses the same mechanism to handle tainted values independently of the nature of a given function. Consequently, we are able to automatically extend our analysis to different set of data types but without being as precise as Nguyen-Toung et al.' work. It is worth seeing studies indicating how much

precision (i.e. less false alarms) it is obtained with *precise tainting* in practice. Similarly to Nguyen-Toung et al.'s work, Futoransky [13] et al. provide a precise dynamic taint analysis for PHP. Pietraszek and Berghe [24] modify the PHP runtime environment to assign *metadata* to user-provided input as well as to provide metadata-preserving string operations. Security critical operations are also instrumented to evaluate, when taken strings as input, the risk of executing such operations based on the assigned metadata. Jovanovic et al. [16] propose to combine a traditional data flow and alias analysis to increase the precision of their static taint analysis for PHP. They observe a 50% rate of false alarms (i.e. one false alarm for each vulnerability). The works in [5, 21] combine static and dynamic techniques. The static techniques are used to reduce the number of program variables where taint information must be tracked at run-time.

A taint analysis for Java [14] instruments the class `java.lang.String` as well as classes that present untrustworthy sources and sensitive sinks. The instrumentation of `java.lang.String` is done offline, while other classes are instrumented online. The authors mention that a custom class loader in the JVM is needed in order to perform online instrumentation. Another taint analysis for Java [31], called TAJ, focus on scalability and performance requirements for industry-level applications. To achieve industrial demands, TAJ uses static tecniques for pointer analysis, call-graph construction, and slicing. Similarly, the authors in [19] propose an static analysis for Java that focus on achieving precision and scalability.

A series of work [18, 9, 25] propose to provide information-flow security via a library in Haskell. These libraries handle explicit and implicit flows and programmers need to write programs with an special-purpose API. Similar to other taint analysis, our library does not contemplate implicit flows and programs do not need to be written with an special-purpose API.

Among the closest related work, we can mention [17] and [29]. In [17], authors modify the Python interpreter to provide a dynamic taint analysis. More specifically, the representation of the class `str` is extended to include a boolean flag to indicate if a string is tainted. We provide a similar analysis but without modifying the interpreter. The work by Seo and Lam [29], called InvisiType, aims to enforce safety checks without modifying the analyzed code. Similar to our assumptions, their approach is designed to work with non-malicious code. InvisiType is more general than our approach. In fact, authors show how InvisiType can provide taint analysis and access control checks for Python programs. However, InvisiType relies on several modifications in the Python interpreter in order to perform the security checks at the right places. For example, when native methods are called, the run-time environment firstly calls the special purpose method `__nativecall__`. As a manner to specifying policies, the approach provides the class `InvisiType` that defines special purposes methods to get support from the run-time system (e.g. `__nativecall__` is one of those methods). Subclasses of this class represent security policies. The approach relies on multiple inheritance to extend existing classes with security checks. To include or remove security checks from objects, programs need to explicitly call functions *promote* and *demote*. Being less invasive, our library uses decorators instead of explicit function calls to taint and untaint data. Our approach does not require multiple inheritance.

5 Conclusions

We propose a taint mode for Python via a library entirely written in Python. We show that no modifications in the interpreter are needed. Different from traditional taint analysis, our library is able to keep track of tainted values for several built-in classes. Additionally, the library provide means to define functions that propagate taint information (e.g. the length of a tainted string produces a tainted integer). The library consists on around 300 LOC. To apply taint analysis in programs, it is only needed to indicate the sources of untrustworthy data, sensitive sinks, and sanitization functions. The library uses decorators as a noninvasive approach to mark source code. Python's object classes and dynamic dispatch mechanism allow the analysis to be executed with almost no modifications in the code. As a future work, we plan to use the library to harden frameworks for web development and evaluate the capabilities of our library to detect vulnerabilities in popular web applications.

Acknowledgments Thanks are due to Arnar Birgisson for interesting discussions. This work was funded by the Swedish research agencies VR and SSF, and the scholarship program for graduated students from the Universidad Tecnológica Nacional, Facultad Regional Santa Fe.

References

[1] List of Python software. `http://en.wikipedia.org/wiki/List_of_Python_software`.

[2] The Ruby programming language. `http://www.ruby-lang.org`.

[3] The Twisted programming framework. `http://twistedmatrix.com`.

[4] M. Andrews. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.

[5] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2008. IEEE Computer Society.

[6] S. Bekman and E. Cholet. *Practical mod_perl*. O'Reilly and Associates, 2003.

[7] R. Bird and P. Wadler. *An introduction to functional programming*. Prentice Hall International (UK) Ltd., 1988.

[8] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications security*, New York, NY, USA, 2008. ACM.

[9] T. chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. *Computer Security Foundations Symposium, IEEE*, 0:187–202, 2007.

[10] J. J. Conti and A. Russo. A Taint Mode for Python via a Library. Software release. `http://www.cse.chalmers.se/~russo/juanjo.htm`, Apr. 2010.

[11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[12] Federal Aviation Administration (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems. `http://www.oig.dot.gov/sites/dot/files/pdfdocs/ATC_Web_Report.pdf`, June 2009. Note: thousands of vulnerabilities were discovered.

[13] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA Briefings*, Aug. 2007.

[14] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[15] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, pages 40–52. ACM, 2004.

[16] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.

[17] D. Kozlov and A. Petukhov. Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In *Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, June 2007.

[18] P. Li and S. Zdancewic. Encoding information flow in Haskell. *Computer Security Foundations Workshop, IEEE*, 0:16, 2006.

[19] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[20] M. Lutz and D. Ascher. *Learning Python*. O'Reilly & Associates, Inc., 1999.

[21] M. Monga, R. Paleari, and E. Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *IWSESS '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 25–32, Washington, DC, USA, 2009. IEEE Computer Society.

[22] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[23] Perl. The Perl programming language. `http://www.perl.org/`.

[24] T. Pietraszek, C. V. Berghe, C. V, and E. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[25] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM, 2008.

[26] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)*, 2009.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[28] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[29] J. Seo and M. S. Lam. InvisiType: Object-Oriented Security Policies. In *17th Annual Network and Distributed System Security Symposium*. Internet Society (ISOC), Feb. 2010.

[30] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.

[31] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In M. Hind and A. Diwan, editors, *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 87–97. ACM Press, 2009.

[32] A. van der Stock, J. Williams, and D. Wichers. OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`, 2007.

Implementing Erasure Policies Using Taint Analysis

Filippo Del Tedesco, Alejandro Russo, and David Sands

Chalmers University of Technology, Göteborg, Sweden
{tedesco,russo,dave}@chalmers.se

Abstract. Security or privacy-critical applications often require access to sensitive information in order to function. But in accordance with the principle of least privilege – or perhaps simply for legal compliance – such applications should not retain said information once it has served its purpose. In such scenarios, the timely disposal of data is known as an *information erasure policy*. This paper studies software-level information erasure policies for the data manipulated by programs. The paper presents a new approach to the enforcement of such policies. We adapt ideas from dynamic taint analysis to track how sensitive data sources propagate through a program and erase them on demand. The method is implemented for Python as a library, with no modifications to the runtime system. The library is easy to use, and allows programmers to indicate information-erasure policies with only minor modifications to their code.

1 Introduction

Sensitive or personal information is routinely required by computer systems for various legitimate tasks: online credit card transaction may handle a card number and related verification data, or a biometric-based authentication system may process a fingerprint. Such systems often operate under informal constraints concerning the handling of sensitive data: once the data has served its purpose, it must not be retained by the system.

The notion of erasure studied in this paper is much higher-level than the system-level and physical notions of data erasure which might involve, e.g. ensuring that caches are flushed and that hard-drives are overwritten sufficiently often to eradicate magnetic traces of data. The approach to program-based high-level erasure stems from the work of Chong and Myers [3]. That work and its subsequent developments deal with a notion of erasure which is relative to a multilevel security lattice [7]. For the purpose of this paper, we will not consider this extra dimension – so we view data as either *available* or *erased*.

In this paper, we present a new approach to the enforcement of information-erasure policies on programs which adapts concepts from dynamic taint analysis.

Language-based Erasure Our approach for information-erasure has several key features: it is a purely dynamic mechanism, it is based on taint analysis, and it is realised completely as a Python library. To see the benefits of these features, it is useful to consider previous work on erasure in the context of a simple erasure scenario (one which we will further elaborate upon in Section 3): a fingerprint-activated left-luggage locker of the kind that is increasingly common at US airports and amusement parks. When

depositing a bag, a fingerprint scan is recorded. The locker can only be opened with the same fingerprint that locked it. From a privacy perspective, there is a clear motivation for an erasure policy: the fingerprint (and any information derived from it) should be erased once a locker has been reopened.

Hunt and Sands [12] described the first approach to the enforcement of Chong-Myers-style erasure properties, reemphasizing two key features missing from [3]: the ability to associate erasure policies with IO (clearly needed in our erasure scenario), and a way to verify that a program correctly erases data by a purely static analysis (a type system). There are two key limitations in Hunt and Sands's approach. Firstly, in order to obtain a clean semantic model, the authors consider a restricted form of erasure policy which is specified in the code in the form: "the value received at this input statement must be erased by the end of the code block which follows it". This is suitable for the simple locker scenario (which is problematic for other reasons) but unsuitable for more complex conditional policies of the kind discussed by Chong and Myers. Secondly, the idea is only elaborated for a toy language. Scaling up to a real language is a nontrivial task for such a static analysis, and would require, among other things, a full alias analysis.

Chong and Myers [5] independently considered the problem of enforcing erasure policies and developed a hybrid static-dynamic approach. In their approach, data is associated with conditional erasure properties which state that data must be erased at the point when some (in principle arbitrary) condition becomes true. An implementation extending the Jif system uses a simple form of condition variables for this purpose [4]. To support such rich policies, they assume a combination of a static analysis and a runtime monitor. The static analysis ensures that all program variables are labeled with consistent policies. For example, if variable x is copied to y and x's policy says that it should be erased at some condition c, then the policy for y should be at least as demanding. It is then the job of the run-time system to detect when conditions become true, and implement the erasure on the behalf of the programmer (by overwriting all variables with a dummy value).

Neither of these approaches can satisfactorily handle the simple locker scenario (and certainly not the more complex variants we will consider later in this paper). The approach described in [5, 4] does not consider input at all, but only one-time erasure of variables – although this is arguably not a fundamental limitation. More fundamentally, both approaches use a semantic notion of erasure which is based on a strict information-flow property. In the locker scenario described previously, there is a small amount of information which is inevitably "retained" by the system, namely the fact that the fingerprint used to unlock the container matches the one used to lock it. This requirement cannot be easily captured by [5, 12] since no retention of information is allowed. Observe that the retained information is not enough to recover the fingerprint which produced it, and therefore we can consider the system as an erasing one. It is not difficult to imagine more complex scenarios (e.g. billing services) that need to retain portions of sensitive information to complete their task, but the amount of retained data is not enough to consider their behavior as a violation of some required erasure policies. Although the Chong-Myers approach includes declassification, declassification is not what is needed here, but an erasure dual: we need the ability to selectively ignore that some informa-

tion is remembered by the system. This feature might be called *delimited retention*, as it resembles the *delimited release* [21] property that some non-interferent system may exhibit.

Overview In the remainder of this paper we outline our alternative approach. We adopt the idea of *dynamic taint tracking* which is familiar from languages like Perl [1] and a number of recent pragmatic information-flow analysis tools [10, 13, 14, 8, 22]: a specific piece of data which is scheduled for erasure is labeled ("tainted"). As computation proceeds, the labels are tracked through the system ("taint propagation"). When it is time to erase the data, we can locate all the places to which the data has propagated and thereby erase all of them.

By performing a dynamic analysis, we obtain a system that is able to deal with complex conditional-erasure conditions. Taint analysis does not track all information flows; in particular the information flows which result purely from control-flow are not captured. This makes the approach unsuitable for malicious code (the approach presented in [16] could be integrated with our library in order to tackle such flows). However, when implicit information flows [7] are ignored, then the need for yet-more-complex *delimited retention* policies used at branching instructions seems to be unnecessary. In principle, it could be possible to encode any delimited retention policy using implicit flows at the price of writing complex and unnatural code, which supports the idea to explicitly include mechanisms for delimited retention.

We are able to implement erasure enforcement for Python, an existing widely-used programming language, simply by providing a library, with no modification of the language runtime system and no special purpose compiler needed. Python's dynamic dispatch mechanism is mainly responsible to facilitate the implementation of our approach as a library. It could be then possible to implement our enforcement for other programming languages with similar dynamic features as Python.

The programmer interface to the library does not require the program to be written in a particular style or using particular data structures, so in principle, it can be applied to existing code with minimal modifications.

The API for the library is particularly simple (Section 2) and its implementation builds on two well-known techniques from object-oriented programming and security, namely *delegation* [15] (Section 2.1) and *taint analysis* (Section 2.3). To use the library the programmer must identify *erasure sources* – in the case of the simple locker example, it is the input function which returns the fingerprint. Then, the programmer must mark in the code the point at which a given value must be erased. This allows the library to trace the origins of a given value and erase all its destination values (Section 2.4).

Section 3 illustrates the use of the library with an extended example based on the locker scenario, but with more involved policies.

In addition, we explore a new *lazy* form of erasure (Section 4). This form of erasure is triggered "just in time" at the points where data would otherwise escape the system and observably break the intended erasure policy. The advantage of lazy erasure is that it is able to easily express rich conditional-erasure policies, including those involving time constraints (e.g. "erase credit card numbers more than one week old"). Additional related work is described in Section 5.

2 The Erasure Library

This section presents the library to introduce information erasure policies into programs. Both its source code and the examples we are using in this paper are publicly available at `http://www.cse.chalmers.se/~russo/erasure`.

The library API essentially consists of three functions:

`erasure_source(f)` is used to mark that values produced by function `f` might be erased. Henceforth, we will say that such values are *erasure-aware*. In the locker scenario, suppose that the function responsible to perform the scan of a fingerprint and return its value is `getFingerprint`. Then, the programmer might declare (prior to any computation): `getFingerprint=erasure_source(getFingerprint)`. The instruction above can be interpreted as `t=erasure_source(getFingerprint); getFingerprint=t`, where `t` is a temporal variable. As an alternative, if the code for the definition of `getFingerprint()` is available, Python's decorator syntax can be used to obtain the same effect:

```
@erasure_source
def getFingerprint() :
  # body of definition ...
```

`erasure(v)` erases all erasure-aware data which was directly used in the computation of value `v`. The effect is to overwrite the data with a default value. For example, if a locker is locked with a fingerprint stored in a variable of the same name, then the code for the locked state might be:

```
while locked
  tryprint=getFingerprint()                  # get attempt
  locked=not(match(tryprint,fingerprint))    # unlock?
  erasure(tryprint)                          # erase attempt
erasure(fingerprint)                         # now unlock
```

A simple variant `erasure()` erases all erasure-aware data (strings and numbers), and any data computed from them.

`retain(f)` provides an escape-hatch for erasure. It declares that the result of function `f` does not need to be erased. We say that `f` is a *retainer*. It corresponds to declaring an escape hatch in delimited release, or a sanitisation function in a taint analysis. In the example above, we might declare `match` as a retainer: `match=retain(match)`.

We illustrate the effect of the library with the Python commands executed in an interactive session[1] in the left hand display of Figure 1. Here, the symbol `>>>` is the interpreter prompt, and `raw_input` is the built-in function that reads a line from the standard input.

In the session to the left, line 1 gets the string `'A'` as an input and stores it in variable `x`. Then, variable `x` is used in two elements of list `y`. Naturally, when printing the list, we can observe that the second element is `x` and the third one is some data derived from `x`, i.e. `x` concatenated with itself.

Now, let us consider a replay of this session in which the programmer wants to delete the information related to the input `x` after list `y` is printed once, which consti-

[1] We use Python 2.7 for this work. However, our techniques can also be applied to previous releases.

```
>>> x=raw_input()
A
>>> y=['E',x, x+x]
>>> y
['E','A','AA']
```

```
>>> from erasure import *
>>> raw_input=erasure_source(raw_input)
>>> x=raw_input()
A
>>> y=['E',x,x+x]
>>> y
['E','A','AA']
>>> erasure(x)
>>> y
['E','','']
```

Fig. 1. Examples of interactive sessions, without and with erasure

tutes an information erasure policy. To achieve that, the programmer needs to import our library, indicate that function `raw_input` returns erase-aware values, and call function `erasure` before printing the list for the second time. This revised session is illustrated on the right of Figure 1. Observe that erasure removes data related to `x`. It is worth noting that the core part of the program has not drastically changed in order to introduce an information erasure policy. The next subsections provide some insights into the implementation of the library.

2.1 Delegation

```
>>> x=Erasure('A')
>>> type(x)
<type 'instance'>
>>> x
'A'
>>> y=x+x
>>> type(y)
<type 'instance'>
>>> y
'AA'
>>> x.erase()
>>> y.erase()
>>> (x,y)
('', '')
```

Fig. 2. Mutable strings

In Python, numbers and strings are immutable. Consequently, values of that type cannot be changed in-place after their creation. For instance, every string operation is defined to produce a new string as a result. Having immutable strings goes against the nature of erasure, since removing information stored in a string implies in-place overwriting of its contents by, for instance, the empty string. By using a coding pattern usually known as *delegation*, the library carefully implements a mechanisms that allows the value of a string to be changed as shown by lines 7 and 10 in Figure 1.

Delegation is a composite-based structure that manages a wrapped object and propagates method calls to it. In our library, it is implemented by the class `Erasure`, which wraps an immutable object. Most of the method calls on that class are forwarded to the wrapped object. The forwarding mechanism assures that the results of method calls are also wrapped by the class `Erasure`. By doing so, the only reference to the wrapped immutable object is by a field on the class `Erasure`. As a result, it is possible to encode mutable strings by simply using delegation. To illustrate how it works, we present an example in Figure 2. Line 1 creates an object of the class `Erasure` that contains the immutable string `'A'`. Observe that line 3 indicates that it is indeed an object and not a string. Line 6 calls the concatenation method on the object `x`, which is forwarded to the concatenation method of the string `'A'`. The result of that, the immutable string `'AA'`, is wrapped by a new object of the class `Erasure` and stored in `y`. Class `Erasure` provides the method `erase`

to perform the concrete action of overwriting, with a default value, the class field where the immutable object is stored (see Lines 11–12). Consequently, the wrapped objects have now become the empty strings. The previous immutable objects, `'A'` and `'AA'`, are no longer referenced and thus will be garbage collected on due course. Programmers are not supposed to deal with the class `Erasure` directly (observe that it is not in the interface of the library). Determining what data must be wrapped by the class `Erasure` is tightly connected to what information must be erasure-aware. The next two subsections describe the internal use of `Erasure` by the different mechanisms of the library.

2.2 The primitive `erasure_source`

Erasure policies are expected to be only applied on a data source (i.e. an input) [12]. In fact, it does not make too much sense to erase information known at compile-time (e.g. global constants, function declarations, etc). In this light, the library provides the primitive `erasure_source` to indicate those sources of erase-aware values. More technically, the argument of `erasure_source` is a function, and the effect is to wrap, by using the class `Erasure`, the immutable values returned by it. As an example, we have the sequence of commands in Figure 3.

```
1 >>> from erasure import *
2 >>> raw_input=erasure_source(raw_input)
3 >>> x=raw_input()
4 A
5 >>> type(x)
6 <type 'instance'>
```

Fig. 3. Example of using `erasure_source`

Note that lines 1–3 are the same as the ones in Figure 1. In this case, the string `'A'`, returned by calling `raw_input`, is wrapped into an object of the class `Erasure`. As shown in Figure 1, users might want to delete a given input value as well as information computed from it. Therefore, the library must be able to automatically call the method `erase` on a given input as well as any piece of data computed from it. In order to do that, the library keeps track of how erasure-aware values flow inside programs by using taint analysis.

2.3 Taint analysis

Taint analysis is an automatic approach to find vulnerabilities in applications. Intuitively, taint analysis keeps track how tainted (untrustworthy) data flow inside programs in order to constrain data to be untainted (trustworthy), or sanitised, when reaching sensitive sinks (i.e. security critical operations). Perl was the first scripting language to provide taint analysis as a special mode of the interpreter called *taint mode* [2]. Similar to Perl, some interpreters for Ruby [24], PHP [17], and Python [14] have been carefully modified to provide taint modes. Rather than modifying the interpreter, Conti and Russo [6] show how to provide a taint mode via a library in Python.

There is a clear connection between the use of taint analysis for finding vulnerabilities and the problem of implementing an erasure policy. In taint analysis, data computed from untrustworthy values is tainted. In our library, data that is computed from erasure-aware values is erasure-aware. With this in mind, and inspired by Conti and Russo's work, we implement a mechanism to perform taint propagation, i.e. how to mark as

erasure-aware data that is computed from other erasure-aware values. From now on, we use taint and erasure-aware as interchangeable terms.

Let us consider tainting and taint propagation in the following example, which is an extended version of the listing in Figure 1:

```
1  >>> raw_input=erasure_source(raw_input)
2  >>> x=raw_input()
3  A
4  >>> x.tstamps
5  set([datetime.datetime(2010, 7, 3, 14, 13, 49, 21585)])
6  >>> y=raw_input()
7  B
8  >>> y.tstamps
9  set([datetime.datetime(2010, 7, 3, 14, 13, 56, 324137)])
10 >>> z=x+y
11 >>> z.tstamps
12 set([datetime.datetime(2010, 7, 3, 14, 13, 49, 21585), datetime.
     datetime(2010, 7, 3, 14, 13, 56, 324137)])
```

As mentioned previously, erasure policies intrinsically refer to some input in the program. Consequently, to enforce erasure policies, it is necessary to identify specific inputs. Our library associates a timestamp to each input, representing the date and time at which the data was provided. Timestamps are stored in the attribute `tstamps` of the class `Erasure`. Thus, the assignment `f=erasure_source(f)` makes the result of `f` erasure aware, and in addition it ensures that each value produced by `f` is (uniquely) timestamped. Line 5 shows the timestamps corresponding to the input that variable `x` depends on. The content of `x.tstamps` is the date and time when the input in line 3 was provided (2010–7–3 at 14:13:49 and some microseconds).

When erasure-aware values are involved in computations, taint information (i.e. timestamps) gets propagated. More specifically, newly created erasure-aware objects are associated to the set of timestamps obtained by merging the timestamps found in the different objects involved in the computation. Taint propagation is implemented inside the delegation mechanism of the class `Erasure` and it is performed after forwarding method calls for a given object.

Line 10 combines two inputs (`x` and `y`) in order to create a new value, which is stored in `z`. Line 11–12 illustrates that taint propagation is performed and that the set of timestamps associated to `z` are those corresponding to the inputs `x` and `y`. At this point, the reader might wonder why timestamps are used rather than a simple input-event counter. By using timestamps, we will be able to program temporal erasure policies (Section 4).

Explicit and implicit flows On most situations, taint analysis propagates taint information on assignments. Intuitively, when the right-hand side of an assignment uses tainted values, the variable appearing on the left-hand side becomes tainted. In fact, taint analysis is just a mechanism to track explicit flows, i.e. direct flows of information from one variable to another. Taint analysis tends to ignore implicit flows [7], i.e. flows through the control-flow constructs of the language.

```
1 if x == 'A': isA=True
2 else: isA=False
3 erasure(x)
```

Fig. 4. An implicit flow

Figure 4 presents an implicit flow where variable `x` is erasure-aware. Observe that variable `isA` is not erasure-aware. In fact, it is built from untainted Boolean constants. Although the value of `x` is erased (Line 3), information about `x` is still present in the program, i.e. the program knows if `x` referred to `'A'`. It is not difficult to imagine programs that circumvent the taint analysis by copying the content of erasure-aware strings into regular strings by using implicit flows[19]. In scenarios where attackers have full control over the code (e.g. when the code is potentially malicious), implicit flows present an effective way to circumvent the taint analysis. There is a large body of literature on the area of language-based security regarding how to track implicit flows [20]. In this work, we only track explicit flows, and thus our method is only useful for code which is written without malice. Despite the good intentions and experience of programmers, some pieces of code might not perform erasure of information as expected. For example, a programmer might forget to overwrite a variable that is used to temporarily store some sensitive information. In this case, taint analysis certainly helps to repair such errors or omissions. How much information are implicit flows able to retain in non-malicious code? As it has been argued for taint analysis [19], we argue that implicit flows are unlikely to account for a large volume of unintended data retention. The reason is that data retention relies on the non-malicious programmer writing more involved and rather unnatural code in order to, for instance, copy tainted (erasure-aware) strings into untainted ones [19]. In contrast, to produce explicit flows, programmers simply need to forget to remove the content of a variable.

2.4 Erasing data

The taint analysis described above allows the library to determine, given a value, which erasure-aware inputs were used to create it. These inputs are identified by a set of timestamps. To perform erasure, however, the library must take these timestamps and track down all primitive values which are built from those inputs (c.f. line 8 in Figure 1). To track which erasure-aware values depend on which inputs, the library internally maintains a dependency table. It is the interaction of taint analysis and this table what determines one of the differences between our approach and [6]. The table maps each timestamp to the set of (references to) erasure-aware values – i.e. objects of the class `Erasure`. If timestamp t is mapped to objects a and b, it means that the only values in the program created by the input value provided at time t are a and b. The dependency table is extended each time an erasure-aware input value is generated. It is updated when erasure-aware values are formed from already existing ones. Primitive `erasure_source` and the taint propagation mechanism are responsible for properly updating the dependency table. Primitive `erasure(v)`, which performs the actual erasure of data, can be then easily implemented. More precisely, calling `erasure(v)` triggers the method `erase` (recall Figure 2) on all the objects which depend on the timestamps associated to `v`. As a result, erasure-aware values derived from the same inputs as `v` are erased from the program. Similarly, calling `erasure()` triggers the method `erase` on *every* object in the dependency table.

```
1  def lockerSystem():
2    while(True):
3      print 'Welcome to the locker system'
4      fingerprint=getFingerprint()
5      ts=datetime.today()
6      if fingerprint in ADM:
7        log.add('MEMORY DUMP -->'+fingerprint+': '+str(ts))
8        dump(log.getLog())
9      else:
10       suspect=local_police.check(fingerprint)
11       h = hash(fingerprint)
12       if locker.isFree():
13         key = h
14         locker.occupied()
15         print 'Please, do not forget to retrive your goods'
16         log.add('LOCKED -->'+fingerprint+': '+str(ts))
17       else:
18         if key == h:
19           locker.free()
20           print 'Thanks for using the service'
21           log.add('UNLOCKED -->'+fingerprint+': '+str(ts))
22         else:
23           print 'You are not the right owner'
24           log.add('INVALID ACCESS -->'+fingerprint+': '+str(ts))
```

Fig. 5. Locker system

3 Extended Example

To give a fuller illustration of the capabilities of our approach, we add some extra functionalities to the locker system described previously that are likely to be found in a real implementation. Firstly, the system is able to keep track of events in a log that a group of special users, called *administrators*, can fetch using their fingerprints. Secondly, since such lockers are typically found in security-critical public infrastructures, we anticipate that there will be communication with some external authority in order to cross-check the input fingerprints with the ones contained in special records (terrorist suspects, wanted criminals etc.). For simplicity, and without loosing generality, we consider a system connected to just a single locker rather than several ones.

The code in Figure 5 shows an implementation of the locker system. As before, function `getFingerprint` reads a fingerprint. Function `datetime.today` returns a timestamp representing the current date and time. Object `log` implements logging facilities. Method `log.add` inserts a line into the log and method `log.getLog` provides the log back inside a container.

When the fingerprint matches one of the administrator's fingerprints stored in the container `ADM`, the `dump` function is executed using `log.getLog` as argument, and the log is output (lines 7-8). Object `local_police` represents a connection to the external authority. Method `local_police.check` cross-checks the fingerprint given as an argument against a database of suspects.

In all other cases (i.e. for locking and opening purposes), the locker only needs a `hash` of the fingerprint, which is assigned to variable `h`. The object `locker` represents the state of the locker, which is initially "free" and could become "occupied" during the execution. If the fingerprint does not belong to an administrator, the locker is tested with the `isFree` method. If the answer is positive, the user can store luggage; the hash is then saved in `key` and the locker state is set to `occupied` (lines 13-16). Otherwise, the locker is full and it is released only if the current hash matches with the one used to lock it. In this case the method `free` makes the locker available for the next user (lines 19-21).

When it comes to logging, it is crucial to define what we want and is allowed to log. The program logs four different responses corresponding to the system usage: *'LOCKED'*, *'UNLOCKED'*, *'INVALID ACCESS'*, and *'MEMORY DUMP'*. Naturally, it is important to register the actions performed by the systems as well as the time when they occur. Clearly, information erasure emerges as a desirable property when it comes to handle fingerprints. On one hand, *fingerprints corresponding to regular users must be removed from the system (including from its log) after they are used for the intended purpose*, which constitutes the information erasure policy of the locker system (observe the hash of the fingerprint is stored in the system for the authentication purpose, and for the purposes of this example is considered to be OK to store). Fingerprints corresponding to suspects, on the other hand, can be logged as evidence in case of a police investigation. In order to give credit for his or her work, fingerprints from administrators can also be logged. In other words, fingerprints from regular users must be erased after using them, while fingerprints from suspects and administrators can remain in the system. The code shown in Figure 5 does not fulfill the information erasure policy described before. It actually logs the fingerprints of any user, which violates citizens privacy. Although it is relatively simple to detect the violation of the information erasure policy in this example, the same task could be very challenging in a more complex system where there could be multiple sources of sensitive information in several thousands lines of codes.

```
1  from erasure import erasure_source, retain, erasure
2
3  # Erasure-aware sources
4  getFingerprint=erasure_source(getFingerprint)
5
6  # Retention statement
7  hash=retain(hash)
8
9  def lockerSystem():
10       ...
11       suspect=local_police.check(fingerprint)
12       h = hash(fingerprint)
13       if not(suspect):
14         erasure(fingerprint)
15       ...
```

Fig. 6. Locker system patched to fulfill the erasure policy regarding fingerprints

Figure 6 shows how programmers can use the library to make the code fulfill the erasure policy regarding fingerprints. Line 1 imports our library. Line 4 identifies that

fingerprints are subjected to erasure policies, i.e. they are erasure-aware values. Line 7 states that `hash` is properly written, namely its outputs cannot be related to its input, and therefore they are not considered to violate any erasure policy. Then, the implementation of function `lockerSystem` is only changed to call `erasure` when the user of the locker is not a suspect (lines 13-14). The rest of the code remains unchanged.

4 Lazy Erasure

The notion of erasure presented in the previous section is very intuitive. To remove all erasure-aware inputs used to compute a given value `v`, it is enough to call `erasure(v)`. When calling `erasure`, the library immediately triggers the mechanism to perform erasure over the current state of the program. Due to that fact, we call the mechanism implemented by the API in Section 2 *eager erasure*[2].

Eager erasure does not easily capture some classes of erasure policies without major encoding overhead, which might drastically modify the code of the program. In particular, let us consider conditional policies that cannot be immediately decided, e.g. a certain value can only remain in the system for a period of time, after which it has to be erased. Clearly, it is not possible to trigger the erasure mechanism straight away, but the need for erasure has to be remembered in the system and triggered at the right time. To deal with such policies without any additional major runtime infrastructure, the library provides *lazy erasure* as a mechanism to perform erasure at the latest possible moment, i.e. when needed.

Lazy erasure deletes information "just in time" at the points where data would otherwise escape the system and observably break the intended erasure policy. Programmers only need to state what is supposed to be erased and it is up to the library to trigger the erasure mechanisms at certain output points, i.e. when information is leaving the system.

4.1 The Lazy Erasure API

Lazy erasure adds some additional functions to the API of the library. The other primitives such as `erasure_source` have the same semantics as before.

`erasure_escape(f)` This function is used syntactically in the same manner as `erasure_source` – i.e. as a function wrapper. It is used to identify the functions which are to be considered as "outputs" for the system. These are the functions where an erasure policy could be observable violated – for example writing to a file or communicating with the outside world in some other manner. The lazy erasure policies are enforced by inspecting the arguments to the functions which have been wrapped by the primitive `erasure_escape`.

`lazy_erasure(v,p)` Primitive `lazy_erasure` introduces an erasure policy into the program, but does not perform any actual erasure of information. It receives as arguments a value `v` and a policy function `p`. The policy function (henceforth an *erasure policy*) is a function from timestamps (i.e. timestamps of inputs) to Boolean values.

[2] In functional languages, eager and lazy evaluation are commonly used terms to indicate when evaluation is performed. We use the same terminology for erasure of data rather than evaluation of terms.

```
1  from datetime import datetime, timedelta
2  from erasure import *
3
4  getFingerprint=erasure_source(getFingerprint)
5
6  hash=retain(hash)
7
8  dump=erasure_escape(dump)
9
10 lazy_erasure(fivedays_policy)
11
12 def lockerSystem():
13       ...
```

Fig. 7. Locker system with a lazy erasure policy

Internally, a policy can use any of the program state, together with the timestamp argument (representing the timestamp of the value to be erased) to make judgment on whether the value should be erased or not. Thus, declaring `lazy_erasure(v,p)` indicates that any input values (and values computed from them) which were used in the creation of `v` should be erased if policy `p` holds for their timestamps. Erasure is then enforced at the output functions indicated by `erasure_escape`.

Two abbreviations are supported: `lazy_erasure(v)`, which is equivalent to `lazy_erasure(v,(lambda t:True))` and thus unconditionally enforces erasure at the erasure-escape points, and `lazy_erasure(p)`, which is an abbreviation for calling `lazy_erasure` with the policy `p` applied to every erasure-aware value in the system.

4.2 Lazy Erasure Examples

To illustrate how lazy erasure works, we start by encoding a temporal erasure policy that allows to only keep fingerprints (administrators and suspects' ones) for a limited time of five days. The following piece of code implements the condition for such a policy:

```
1 def fivedays_policy(time):
2     return (datetime.today()-time)>timedelta(days=5)
```

Policy `fivedays_policy` takes a timestamp as input and returns whether the timestamp is more than five days old. In Figure 7, we show how to apply the policy in our locker system. Line 8 indicates that before extracting the log from the system, erasure must be performed. Line 10 introduces the erasure policy `fivedays_policy` into the system. As a result, dumping the log triggers erasure on each of its entries which are older than 5 days.

Lazy erasure is particularly useful to express policies that cannot be immediately decided when input data enters the system. To illustrate this, we extend the locker scenario a bit further. A common experience with network connections is the loss of connectivity. To handle this situation properly, we introduce the constant `'no_connection'` to be returned by method `local_police.check` when the connection with the police department cannot be established. Enforcing an erasure policy that depends on the connection to the police department is not as simple as the policies considered previously. On one hand, we would like to have in the log the fingerprints which got the

```
1  def lockerSystem():
2      global police_mode
3      police_mode=False
4      ...
5      if fingerprint in ADM:
6        log.add('MEMORY DUMP -->'+fingerprint+': '+str(ts))
7        if fingerprint=='police':
8          police_mode=True
9        dump(log.getLog())
10       police_mode=False
11     else:
12       suspect=local_police.check(fingerprint)
13       h = hash(fingerprint)
14       if suspect==False:
15         lazy_erasure(fingerprint)
16       elif r=='no_connection':
17         lazy_erasure(fingerprint,role_policy)
18       else:
19         pass
20     ...
```

Fig. 8. `lockerSystem` reimplemented for lazy erasure

`'no_connection'` answer since they could belong to suspects. On the other hand, fingerprints that got the `'no_connection'` answer and do not belong to suspects must be erased in order to avoid violating users privacy when administrators dump the log.

As a trade-off between preserving fingerprints of suspects and privacy of regular citizens is represented by the enforcement of an erasure policy which depends on the person doing the dumping of the log. If a police agent is included in the set of administrators, then he or she can dump the log if necessary. Since a police agent represents the public authority, the agent has full access to the fingerprints stored in the log. Therefore, all the entries are included in the log, including those ones with the `'no_connection'` answer. In contrast, if the dumper is a regular administrator, the entries with `'no_connection'` are removed from the log. In this way, suspect-related data may get lost but privacy is not compromised. Clearly, the erasure policy is more involved than the ones that we have been considered so far. However, we show that it can be easily encoded by our library.

```
1  def role_policy(time):
2    global police_mode
3    return not(police_mode)
```

Fig. 9. Example of a lazy policy based on roles.

We start by introducing the Boolean global variable `police_mode` to represent when a police agent is dumping the log. Then, the function `lockerSystem` has to signal whether the person dumping the log is the police agent. Figure 8 shows an extension to `lockerSystem`. In line 3, `police_mode` is initially set to `False`. Immediately before dumping the log (line 9), the administrator identity is checked. If it is a police agent, `police_mode` is set to `True` (line 8). The state is then reset at line 10. If the person dumping the log is a regular administrator, the value of `police_mode` does not change. Observe that line 17 associates the erasure policy `role_policy` to

those fingerprints received when the connection to the policy department cannot be established. Consequently, the erasure of the fingerprint depends on the value returned by the policy *at the time of dumping the log*. Figure 9 defines `role_policy`. This policy only returns true when the dumping is done by a regular administrator (line 3). As a consequence, those fingerprints associated with `'no_connection'` are erased immediately before dumping the log provided that `police_mode` is false.

5 Related Work

As we have already explained in the introduction, application level erasure has been studied in [3] and [12]. A simpler form of erasure for Java bytecode is discussed in [11]. In [23], the counterpart of erasing systems (according to the definition given in [12]) has been explored, providing some insights into the obligations of a user who interacts with a system which promises erasure. These works all deal with an attacker model where an attacker can in the worst case inject arbitrary code into the system at a point in time at which erasure is supposed to have occurred. At lower levels of abstraction, for example [9], conditions and techniques to guarantee physical erasure on storage devices are considered. The need for physical erasure comes from a much stronger attacker model where the attacker is not hindered by any abstraction layers. An end-to-end view linking the high-level application level and the low level physical views should be possible, but it has not been previously considered.

To the best of our knowledge, Jif$_E$ [4] is the only system currently implementing application-level erasure. This is based on the Jif compiler which deals with a subset of Java extended with security labels. Unlike the very general model on which it is based [5], the only conditions allowed in Jif$_E$'s conditional erasure policies are a special class of Boolean condition variables. The implementation ensures that whenever such a condition variable changes, any necessary erasures are triggered. It would be simple to mimic this style of implementation (modulo implicit flows) using our primitives.

Erasure can be also related to *usage control*, since it is based on the idea of changing the way data is handled in the system after a certain moment. In [18], the authors present a model to reason on usage control, based on *obligations* the data receiver has to enforce through some *mechanisms*. The model is very general, and erasure can be described as an obligation (actually it is explicitly mentioned as a data owner requirement), but its purpose does not correspond to our approach, which deals with techniques to implement that obligation. The work in [26] extends access control with temporal and times-consuming features, leading to what they call TUCON (Times-based Usage Control) model. This approach allows to reason with policies that deal with the period of time in which a given object is available. Although it would not be very natural (policies here seem to be more user-oriented), it should be also possible to reason about erasure in this framework as well; similar considerations about implementation holds in this case as well. However, concepts from the usage control literature could provide inspiration for a study of the enforcement of a wider class of usage policies at code level.

6 Conclusions and Future Work

We have presented a library-approach to enforce erasure policies. The library transparently adds taint tracking to data sources, making it easy to use and permitting programmers to indicate information-erasure policies with only minor modifications to their

code. To the best of our knowledge, this is the first implementation of a library that connects taint analysis and information-erasure policies. From our limited experience, the imperfections of taint analysis (the inability to track implicit flows) serve to keep the policy specifications simple, and enable us to handle examples for which existing approaches would not be sufficiently expressive. We have also introduced the concept of lazy erasure – an observational form of erasure which supports richer erasure policies, including temporal policies, with a simple implementation.

There are a number of directions for further work. One challenge ahead is how to deal with permanent storage like databases or file systems when specifying erasure policies. Policies like "user information must be erased when his or her account is closed" are out of scope in the existing approaches [3, 12], where erasure is performed on internal data structures. User information, on the other hand, is usually placed in databases (e.g. web application) or file systems (e.g. Unix-like operating systems). We believe that it is possible to extend the interfaces for accessing files and databases in order to store data as well as erasure information (timestamps). Those interfaces usually involve handling objects and thus the library needs to be extended to consider them. To achieve that, we could threat objects as just mere containers and apply similar tainting techniques as the ones used for dictionaries. Another important aspect is the evaluation of the overheads caused by the library – in particular, how taint propagation and updates in the dependency table impact on performance. It would also be interesting to evaluate how *precise tainting* [17, 8] could be exploited to obtain more precision when erasing data. Precise tainting associates taint information to characters rather than to whole strings. In our library, if an small part of an string contains some information that should be erased, then the whole string is deleted. By using *precise tainting*, it would be possible, in principle, to only delete those pieces of the string containing the information to erase. Precise tainting usually requires to fully understand the semantics of each function that manipulates erasure-aware values. As for most approaches to dynamic taint analysis, our approach ignores implicit flows. As a consequence programs might retain information indirectly via their control constructs. Rather than fixing this problem, a reasonable alternative might be to bound it. Inspired by preserving confidentiality, the work in [16] develops a mechanism to obtain bounds on the information leaked by implicit-flows. We believe that it is feasible to adapt such mechanism to obtain bounds on the information retained by control constructs. On the theoretical side, it could be important to describe precisely the security condition that taint analysis is enforcing in the presence of delimited retention policies. In fact, to the best of our knowledge, the work by [25] is the only one that presents a security condition for taint analysis using formal semantics.

References

1. The Perl programming language. `http://www.perl.org/`
2. Bekman, S., Cholet, E.: Practical mod_perl. O'Reilly and Associates (2003)
3. Chong, S., Myers, A.C.: Language-based information erasure. In: Proc. IEEE Computer Security Foundations Workshop. pp. 241–254 (Jun 2005)
4. Chong, S.: Expressive and Enforceable Information Security Policies. Ph.D. thesis, Cornell University (Aug 2008)
5. Chong, S., Myers, A.C.: End-to-end enforcement of erasure and declassification. In: CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium. pp. 98–111. IEEE Computer Society, Washington, DC, USA (2008)

6. Conti, J.J., Russo, A.: A taint mode for python via a library. OWASP AppSec Research 2010 (2010)
7. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Comm. of the ACM 20(7), 504–513 (Jul 1977)
8. Futoransky, A., Gutesman, E., Waissbein, A.: A dynamic technique for enhancing the security and privacy of web applications. In: Black Hat USA Briefings (Aug 2007)
9. Gutmann, P.: Data remanence in semiconductor devices. In: SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium. pp. 4–4. USENIX Association, Berkeley, CA, USA (2001)
10. Haldar, V., Chandra, D., Franz, M.: Dynamic Taint Propagation for Java. In: Proceedings of the 21st Annual Computer Security Applications Conference. pp. 303–311 (2005)
11. Hansen, R.R., Probst, C.W.: Non-interference and erasure policies for java card bytecode. In: 6th International Workshop on Issues in the Theory of Security (WITS '06) (2006)
12. Hunt, S., Sands, D.: Just forget it – the semantics and enforcement of information erasure. In: Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008. pp. 239–253. No. 4960 in LNCS, Springer Verlag (2008)
13. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In: 2006 IEEE Symposium on Security and Privacy. pp. 258–263. IEEE Computer Society (2006)
14. Kozlov, D., Petukhov, A.: Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology. In: Proc. of Young Researchers' Colloquium on Software Engineering (SYRCoSE) (Jun 2007)
15. Lutz, M.: Learning Python. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2003)
16. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security. pp. 73–85. ACM (2009)
17. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically Hardening Web Applications Using Precise Tainting. In: In 20th IFIP International Information Security Conference. pp. 372–382 (2005)
18. Pretschner, A., Hilty, M., Basin, D., Schaefer, C., Walter, T.: Mechanisms for usage control. In: ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security. pp. 240–244. ACM, New York, NY, USA (2008)
19. Russo, A., Sabelfeld, A., Li, K.: Implicit flows in malicious and nonmalicious code. 2009 Marktoberdorf Summer School (IOS Press) (2009)
20. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (Jan 2003)
21. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Proc. International Symp. on Software Security (ISSS'03). LNCS, vol. 3233, pp. 174–191. Springer-Verlag (Oct 2004)
22. Seo, J., Lam, M.S.: InvisiType: Object-Oriented Security Policies. In: 17th Annual Network and Distributed System Security Symposium. Internet Society (ISOC) (Feb 2010)
23. Tedesco, F.D., Sands, D.: A user model for information erasure. In: SecCo'09, 7th International Workshop on Security Issues in Concurrency. Electronic Proceedings in Theoretical Computer Science (2009), to appear
24. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby. The Pragmatic Programmer's Guide. Pragmatic Programmers (2004)
25. Volpano, D.: Safety versus secrecy. In: Proc. Symp. on Static Analysis. LNCS, vol. 1694, pp. 303–311. Springer-Verlag (Sep 1999)
26. Zhao, B., Sandhu, R., Zhang, X., Qin, X.: Towards a times-based usage control model. In: Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security. pp. 227–242. Springer-Verlag, Berlin, Heidelberg (2007)

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)

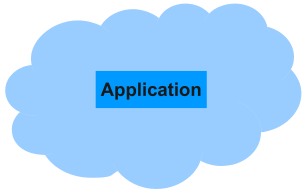
Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

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


Alice



Bob


Charlie

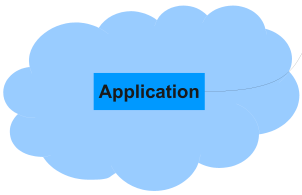
Motivaton: Confidentiality

(private data-leak)


Alice

What is Charlie up to?

Bob

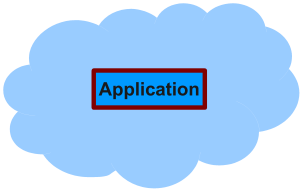



Charlie

Motivaton: Confidentiality

(private data-leak)


Alice



Bob

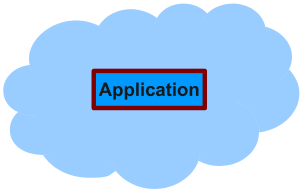

Charlie

Charlie is the owner of that information, therefore he **decides** where it goes. The system should respect that decision.

Motivaton: Confidentiality

(private data-leak)


Alice



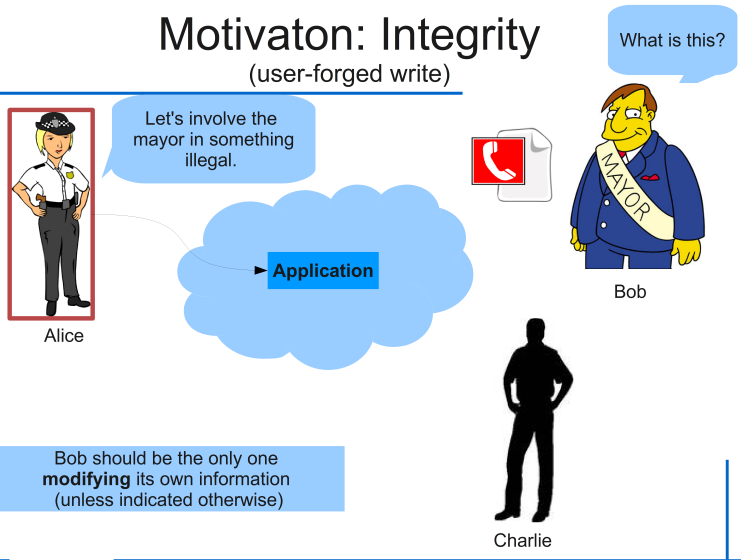
Bob


Charlie

Alice and Bob **collaborate in creating aggregated data**

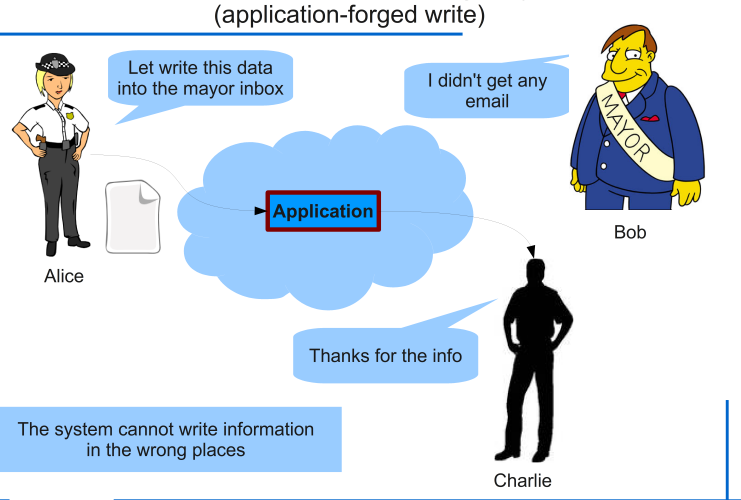
The system must not send that data to Charlie unless Alice and Bob **agree** on that!

Motivaton: Integrity
(user-forged write)



Bob should be the only one **modifying** its own information (unless indicated otherwise)

Motivaton: Integrity
(application-forged write)



The system cannot write information in the wrong places

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

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

Either principal can read the data

- Integrity

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

Either principal can modify the data

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

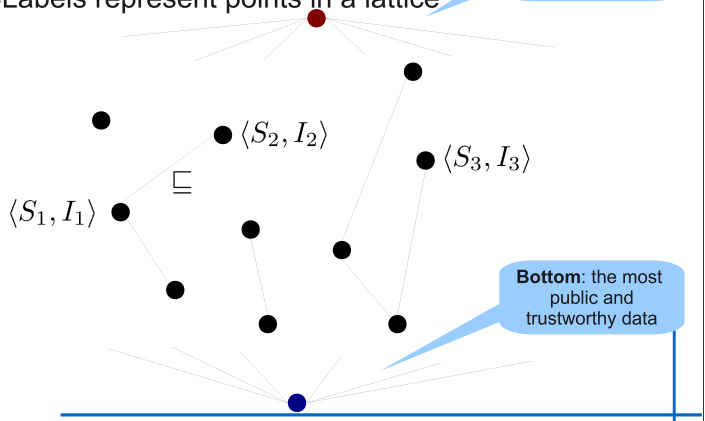
$$\frac{\forall c_1 \in S_1.\exists c_2 \in S_2 : c_2 \Rightarrow c_1 \qquad \forall c_2 \in I_2.\exists c_1 \in I_1 : c_1 \Rightarrow c_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

$$\Leftrightarrow$$

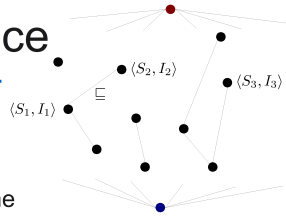
$$\frac{S_2 \Rightarrow S_1 \wedge I_1 \Rightarrow I_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle}$$

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

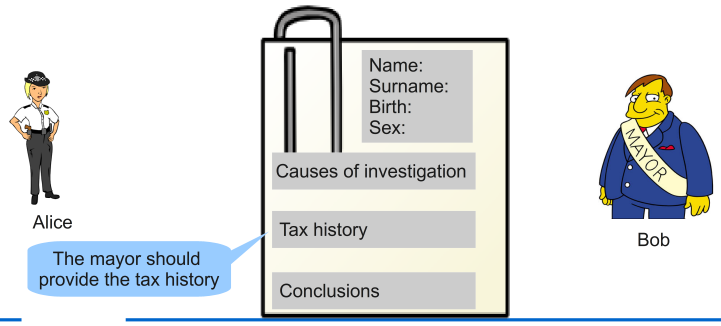
- Any system have some sort of intended release of information
- In a mutual distrust environment, it is necessary to declassify data after some collaborative effort

$$\langle [Alice] \wedge [Bob], [] \rangle \rightarrow \langle [Alice], [] \rangle$$

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

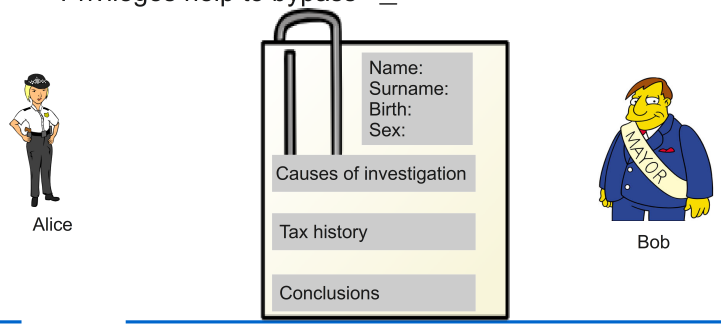
Declassification

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

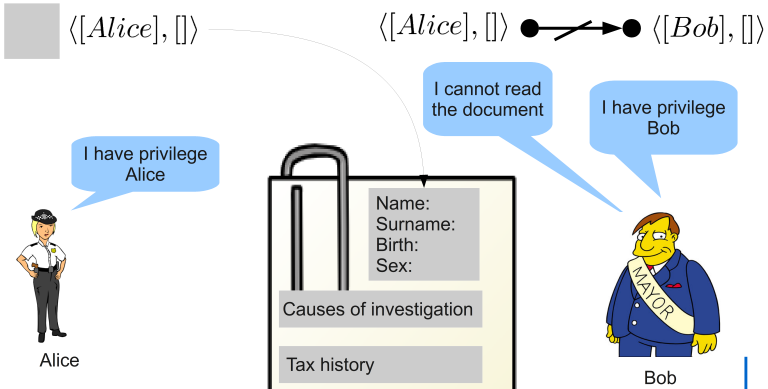


Declassification

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



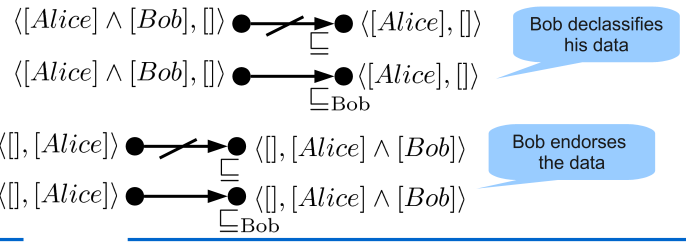
Declassification



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:

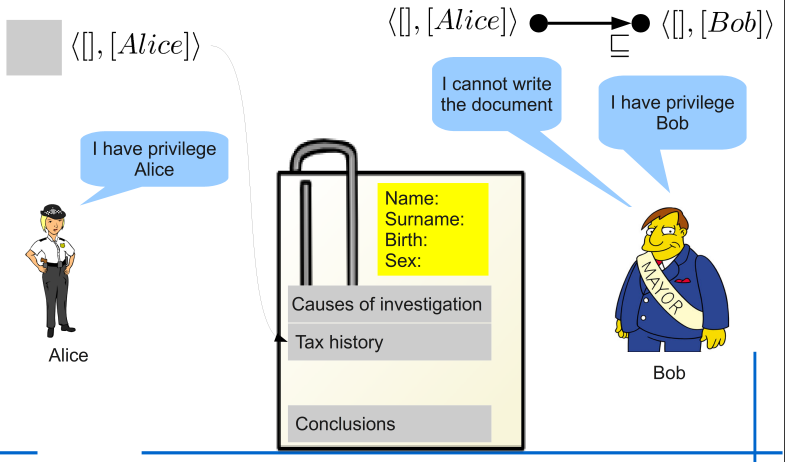
$$\frac{\langle S_1, I_1 \wedge P \rangle \sqsubseteq \langle S_2 \wedge P, I_2 \rangle}{\langle S_1, I_1 \rangle \sqsubseteq_P \langle S_2, I_2 \rangle}$$



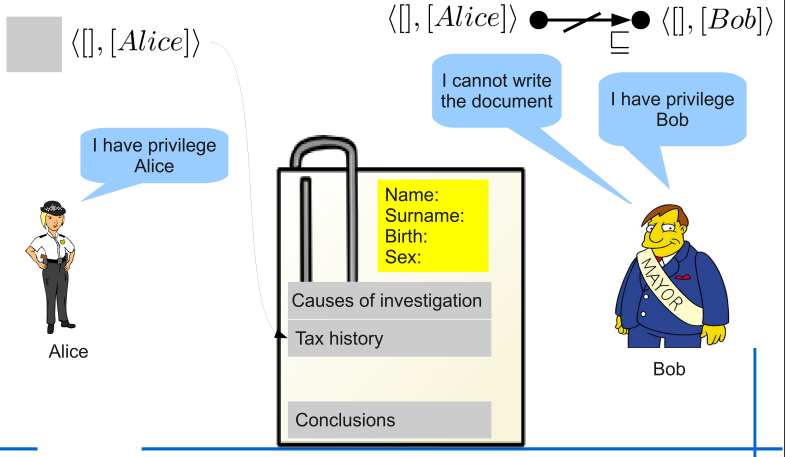
Declassification

Declassification

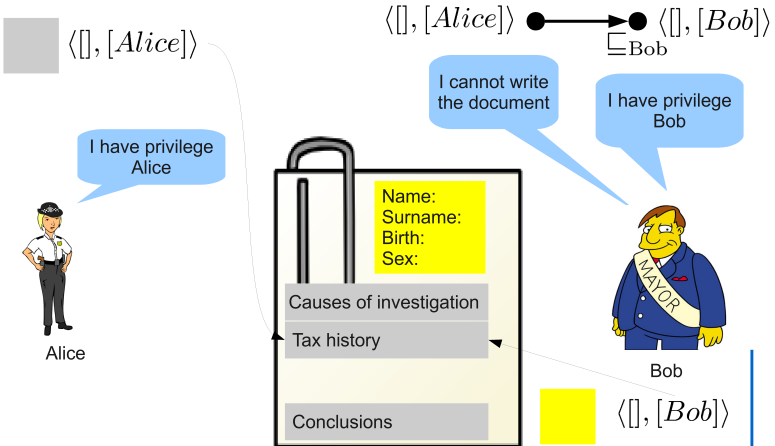
Endorsement



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

```
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

```
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

Floating Current Label

Program written using LIO

There is a current label at any point of the computation

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

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

It is low

It is high

It continues low


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

It cannot write to public data

Discretionary Access Control

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

Clearance

Program written using LIO

There is a current clearance at any point of the computation

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

The program finishes its execution here!

It is low

It is high

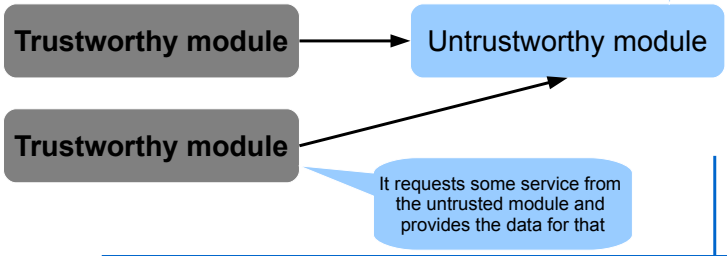
It is low, i.e. the piece of code cannot access secret data

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

Architecture

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

It export some services that required security policies



API: `label` and `unlabel`

It does not modify the current label and clearance!

We ignore this parameter

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

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

`lbot` is `bottom` in DCLabels

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

`ltop` is `top` in DCLabels

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

Using DCLabels!

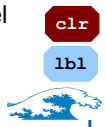
```
bob :: LIO DCLabel () (Labeled DCLabel String)
bob = label (newDC ("Alice" .\/. "Bob") "Bob") "BobData"
```

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!



:: 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
```


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)

Only to be imported by trustworthy code!

```
module ExampleUnLabelT where

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

It imports the service from the untrustworthy code

```
import ExampleUnLabelU (computation)

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

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

It provides some data to the service and executes it!

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


Example (trustworthy code)

```
module ExampleToLabeledT where

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

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

```
import ExampleToLabeledU (computation')

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

Remember that this executes `label`

```
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
```


Example (untrustworthy code)

To be imported by 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
```

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

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
```

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

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

Example (untrustworthy code)

```
module ExampleToLabeledU where

import LIO.DCLabel
import LIO.LIO

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

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

It is not raised when executing `toLabeled`

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

API: `labelOf`

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

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

Example (trustworthy code)

```
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)

```
module ExampleLabelOfU where

import LIO.DCLabel
import LIO.LIO

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


API: References

We ignore this parameter

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

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

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

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

API: References

We ignore this parameter

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

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

Example (trustworthy code)

```
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

Flexible Dynamic Information Flow Control in Haskell

Deian Stefan[1] Alejandro Russo[2] John C. Mitchell[1] David Mazières[1]

(1) Stanford University, Stanford, CA, USA
(2) Chalmers University of Technology, Gothenburg, Sweden
{deian,mitchell}@cs.stanford.edu russo@chalmers.se

Abstract

We describe a new, dynamic, floating-label approach to language-based information flow control, and present an implementation in Haskell. A labeled IO monad, `LIO`, keeps track of a *current label* and permits restricted access to IO functionality, while ensuring that the current label exceeds the labels of all data observed and restricts what can be modified. Unlike other language-based work, `LIO` also bounds the current label with a *current clearance* that provides a form of discretionary access control. In addition, programs may encapsulate and pass around the results of computations with different labels. We give precise semantics and prove confidentiality and integrity properties of the system.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Security, Languages, Design

Keywords Information flow control, Monad, Library

1. Introduction

Complex software systems are often composed of modules with different provenance, trustworthiness, and functional requirements. A central security design principle is the *principle of least privilege,* which says that each component should be given only the privileges it needs for its intended purpose. In particular, it is important to differentially regulate access to sensitive data in each section of code. This minimizes the trusted computing base for each overall function of the system and limits the downside risk if any component is either maliciously designed or compromised.

Information flow control (IFC) tracks the flow of sensitive data through a system and prohibits code from operating on data in violation of security policy. Significant research, development, and experimental effort has been devoted to static information flow mechanisms. Static analysis has a number of benefits, including reduced run-time overhead, fewer run-time failures, and robustness against implicit flows [10]. However, static analysis does not work well in environments where new classes of users and new kinds of data are encountered at run-time. In order to address the needs of such systems, we describe a new, dynamic, floating-label approach to language-based information flow control and present an implementation in Haskell.

Haskell'11, September 22, 2011, Tokyo, Japan.

Our approach uses a `Labeled` type constructor to protect values by associating them with labels. However, the labels themselves are typed values manipulated at run-time, and can thus be created dynamically based on other data such as a username. Conceptually, at each point in the computation, the evaluation context has a *current label.* We use a labeled IO monad, `LIO`, to keep track of the current label and permit restricted access to IO functionality (such as a labeled file system), while ensuring that the current label accurately represents an upper bound the labels of all data observed or modified. Unlike other language-based work, `LIO` also bounds the current label with a *current clearance.* The clearance of a region of code may be set in advance to impose an upper bound on the floating current label within that region. This restricts data access, limits the amount of code that could manipulate sensitive data, and reduces opportunities to exploit covert channels. Finally, we introduce an operator, `toLabeled`, that allows the result of a computation that would have raised the current label instead to be encapsulated within the `Labeled` type.

The main features of our system can be understood using the example of an online conference review system, called λChair. In this system, which we describe more fully later in the paper, authenticated users can read any paper and can normally read any review. This reflects the normal practice in conference reviewing, for example, where every member of the program committee can see submissions, their reviews, and participate in related discussion. Users can be added dynamically and assigned to review specific papers. In addition, as an illustration of the power of the labeling system, integrity labels are used to make sure that only assigned reviewers can write reviews for any given paper. Conversely, confidentiality labels are used to manage conflicts of interest. Users with a conflict of interest on a specific paper lack the privileges, represented by confidentiality labels, to read a review. As conflicts of interest are identified, confidentiality labels on the papers may change dynamically and become more restrictive. It is also possible to remove conflicts of interest dynamically, if desired. A subtlety that we have found advantageous is that reviewers with a conflict of interest can potentially refer to reviews (by having a name that is bound to a review) but cannot perform specific operations simply because they can refer to them. As we have structured our online conference review system, the actual display of a conflict-of-interest review is a prohibited operation.

The main contributions of this paper include:

- We propose a new design point for IFC systems in which most values in lexical scope are protected by a single, mutable, *current label*, yet one can also encapsulate and pass around the results of computations with different labels. Label encapsulation is explicitly reflected by types in a way that prevents implicit flows.
- We prove information flow and integrity properties of our design and describe LIO, an implementation of the new model in

Haskell. LIO, which can be implemented entirely as a library (based on type safety), demonstrating both the applicability and simplicity of the approach.

- Unlike other language-based work, our model provides a notion of *clearance* that imposes an upper bound on the program label, thus providing a form of discretionary access control on portions of the code.

IFC originated with military applications [5, 11] that *label* data and processes with sensitivity security levels. The associated label-checking algorithms then prevent a Trojan horse reading classified data, for example, from leaking the data into less classified files. In operating systems, IFC is generally enforced at the kernel boundary, allowing a small amount of trusted code to impose a flexible security policy on a much larger body of supporting software. Extending the core concepts of IFC to a broader range of situations involving mutually distrustful parties that mix their code and data, Myers and Liskov [28] subsequently introduced a decentralized label model (DLM) that has been the basis of much subsequent OS and language-based work. Unfortunately, despite its attractiveness, the DLM is not widely used to protect data in web applications, for example. In the operating systems domain, most of the past DLM-inspired work has relied exclusively on dynamic enforcement [21, 37, 39]. This is due to the dynamic nature of operating systems, which must support a changing set of users, evolving policies, and dynamically loaded code. But it is often inconvenient to establish security domains by arranging software according to course-grained kernel abstractions like processes and files. Moreover, adopting a new OS presents an even bigger barrier to deployment than adopting a new compiler. LIO uses the type system to enforce abstraction statically, but checks the values of labels dynamically. Thanks to the flexibility of dynamic checking, the library implements an IFC mechanism that is more permissive than previous static approaches [26, 30, 32] but providing similar security guarantees [34]. Though purely language-based, LIO explores a new design point centered on floating labels that draw on past OS work. Both the code and technical details omitted in this paper can be found at `http://www.scs.stanford.edu/~deian/lio`.

2. Security Library

In this section, we give an overview of the information flow control approach used in our dynamic enforcement library for Haskell.

2.1 Labels and IFC

The goal of information flow control is to track and control the propagation of information according to a security policy. A well-known policy addressed in almost every IFC system is *non-interference:* publicly-readable program results must not depend on secret inputs. This policy preserves confidentiality of sensitive data [15].

To enforce information flow restrictions corresponding to security policies such as non-interference, every piece of data is associated with a *label*, including the labels themselves. Labels form a lattice [9] with partial order $\sqsubseteq$ (pronounced "can flow to") governing the allowed flows. A lattice can be as simple as a few security levels. For instance, the three labels `L`, `M`, and `H`, respectively denoting unclassified, secret and top secret levels, form the lattice $\texttt{L} \sqsubseteq \texttt{M} \sqsubseteq \texttt{H}$. An IFC system such as our LIO library prohibits a computation running with security level `M` from reading top secret data (labeled `H`) or writing to public channels (labeled `L`). Dual to such confidentially policies are integrity policies [6], which use the partial order on labels to enforce restrictions on writes.

Our library is polymorphic in the label type, allowing different types of labels to be used. Custom label formats can be created by defining basic label operations: the can flow to label comparison ($\sqsubseteq$), a function computing the *join* of two labels ($\sqcup$), and a function computing the *meet* of two labels ($\sqcap$). Concretely, label types are instances of the `Label` type class:

```
class (Eq l) ⇒ Label l where
  leq :: l → l → Bool -- Can flow to (⊑)
  lub :: l → l → l    -- Join (⊔)
  glb :: l → l → l    -- Meet (⊓)
```

For any two labels L_1 and L_2, the join has the property that $L_i \sqsubseteq (L_1 \sqcup L_2), i = 1, 2$, while the meet has the property that $(L_1 \sqcap L_2) \sqsubseteq L_i, i = 1, 2$. In this section we present examples using the simple three-point lattice introduced above, or a generic/abstract label format; Section 3 details DC labels, a new, practical label format used to implement λChair.

Compared to existing systems, LIO is a language-based *floating-label* system, inspired by IFC operating systems [39, 40]. In a floating-label system, the label of a computation can rise to accommodate reading sensitive data (similar to the *program counter* (`pc`) of more traditional language-based systems [33]). Specifically, in a floating-label system, a computation C with label L_C wishing to observe an object labeled L_R must raise its label to the join, $L_C \sqcup L_R$, of the two labels. Consider, for instance, a simple λChair review system computation executing on behalf of a user, Clarice, with label L_C that retrieves and prints a review labeled L_R, as identified by R:

```
readReview R = do              -- L_C
  rv ← retrieveReview R        -- L_C ⊔ L_R
  printLabeledCh rv            -- L_C ⊔ L_R
```

The computation label (initially L_C) is shown in the comments, on the right. Internally, the `retrieveReview` function is used to retrieve the review contents `rv`, raising the computation label to $L_C \sqcup L_R$ to reflect the observation of confidential data. This directly highlights the notion of a "floating-label": a computation's label effectively "floats above" the labels of all objects it observes. Moreover, this implies that a computation cannot write below its label; doing so could potentially result in writing secret data to public channels.

To illustrate the way floating labels restrict data writes, consider the action following the review retrieval: `printLabeledCh rv`. The trusted `printLabeledCh` function returns an action that writes the review content `rv` to an output channel, permitting the output channel label L_O. The output channel label L_O is dynamically set according to the user executing the computation. Specifically, L_O is carefully set as to allow for printing out all but the conflicting reviews. For example, if Clarice is in conflict with review R then L_O is set such that $L_R \not\sqsubseteq L_O$. Since the computation label directly corresponds to the labels of the data it has observed, `printLabeledCh` simply checks that the computation label flows to the output channel. In the example above, the trusted function checks that $L_C \sqcup L_R \sqsubseteq L_O$ before printing to (standard) output channel O.

As already mentioned, in contrast to other language-based systems, LIO also associates a *clearance* with each computation. This clearance sets an upper bound on the current floating label within some region of code. For example, code executing with a secret (`M`) clearance can never raise its label to read top secret data (labeled `H`). The notion of clearance can also prevent Clarice from retrieving (and not just printing) the contents of a conflicting review R by setting the computation's clearance to L_P such that $L_R \not\sqsubseteq L_P$. When the action `retrieveReview` R attempts to raise the current label to $L_C \sqcup L_R$ to retrieve the review contents, the dynamic check will fail because $L_C \sqcup L_R \not\sqsubseteq L_P$. For flexibility, the output channel label can simply be $L_O = \top$, allowing any information that can be retrieved to be written to the output channel.

Additionally, clearance can be used to prevent Clarice from using termination as a covert channel. For example, the following code can be used to leak conflicting-review information:

```
leakingRetriveReview r = do
  rv ← retrieveReview r
  if rv == "Paper..."
    then forever (return rv)
    else return rv
```

However, using clearance, we prevent such leaks by setting the clearance and review labels in such manner that `retrieveReview` fails when raising the computation label to retrieve conflicting reviews.

2.2 Library Interface

LIO is a *termination-insensitive* [2] and *flow-sensitive* [20] IFC library that *dynamically* enforces information flow restrictions. At a high level, LIO defines a monad called `LIO`, intended to be used in place of `IO`. The library furthermore contains a collection of `LIO` actions, many of them similar to `IO` actions from standard Haskell libraries, except that they contain label checks that enforce IFC. For instance, LIO provides file operations like those of the standard library, but confining the application to a dedicated portion of the file system where a label is stored along with each file.

To implement the notion of floating label bounded by a clearance, our library provides `LIO` as a state monad that uses `IO` as the underlying base monad and it is parametrized by the type of labels. The state consists of a *current label* L_{cur}, i.e., the computation's floating label, and a *current clearance* C_{cur}, which is an upper bound on L_{cur}, i.e., $L_{\text{cur}} \sqsubseteq C_{\text{cur}}$. Specifically, the (slightly simplified) `LIO` monad can be defined as:

```
newtype Label l ⇒ LIO l a = LIO (StateT (l, l) IO a)
```

where the state corresponds to the $(L_{\text{cur}}, C_{\text{cur}})$ pair. To allow for the execution of `LIO` actions, our library provides a function (`evalLIO`) that takes an `LIO` action and returns an `IO` action which, when executed, will return the result of the IFC-secured computation. It is important to note that untrusted LIO code cannot execute `IO` computations by binding `IO` actions with `LIO` ones (to bypass IFC restrictions), and thus effectively limits `evalLIO` to trusted code. Additionally, using `evalLIO`, (trusted) programmers can easily, though cautiously, enforce IFC in parts of an otherwise IFC-unaware program.

The current label provides means for associating a label with every piece of data. Hence, rather than individually labeling definitions and bindings, all symbols in scope are protected by L_{cur} (when a single `LIO` action is executed). Moreover, the only way to read or modify differently labeled data is to execute actions that internally access restricted symbols and appropriately validate and adjust the current label (or clearance).

However, in many practical situations, it is essential to be able to manipulate differently-labeled data without monotonically increasing the current label. For this purpose, the library additionally provides a `Labeled` type for labeling values with a label other than L_{cur}. A `Labeled`, polymorphic in the label type, protects an immutable value with a specified label (irrespective of the current label). This is particularly useful as it allows a computation to delay raising its current label until necessary. For example, an alternative approach to the above `retrieveReview` (called `retrieveReviewAlt`) retrieves the review, encapsulates it as a `Labeled` value, and returns the `Labeled` review, leaving the current label unmodified. This approach delays the creeping of current label until the review content, as encapsulated by `Labeled`, is actually *needed*, for instance, by `printLabeledCh`.

We note that `LIO` can be used to protect pure values in a similar fashion as `Labeled`. However, the protection provided by `Labeled` allows for serializing labeled values and straight forward inspection by trusted code (which should be allowed to ignore the protecting label). Unlike `LIO`, `Labeled` is not a monad. Otherwise, the monad instance would allow a computation to arbitrarily manipulate labeled values without any notion of the current label or clearance, and thus possibly violate the restriction that `LIO` computations should not handle values below their label and above their clearance. Moreover, such instance would require a definition for a default label necessary when lifting a value with `return`. Instead, our library provides several functions that allows for the creation and usage of labeled values within `LIO`. Specifically, we provide (among other) the following functions:

- `label :: Label l ⇒l → a → LIO l (Labeled l a)`
  Given a label l such that $L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and a value v, the action `label` l v returns a labeled value that protects v with l.
- `unlabel :: Label l ⇒Labeled l a →LIO l a`
  Conversely, the action `unlabel lv` raises the current label (clearance permitting) to the join of `lv`'s label and the current label, returning the value with the label removed. Note that the new current label is at least as high as `lv`'s label, thus protecting the confidentiality of the value.
- `toLabeled :: Label l ⇒l → LIO l a →LIO l (Labeled l a)`
  Given a label l such that $L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and an `LIO` action m, `toLabeled` l m executes m without raising L_{cur}. However, instead of returning the result directly, the function returns the result of m encapsulated in a `Labeled` with label l. To preserve confidentiality (see Section 4 for further details), action m must not read any values with a label above l. We can implement `toLabeled` as follows:

  ```
  toLabeled l m = do
    (L'cur, C'cur) ← get      -- Save context
    res ← m                   -- Execute action
    (Lcur, _) ← get           -- Get inner context
    unless (Lcur ⊑ l) fail    -- Check IFC violation
    put (L'cur, C'cur)        -- Restore context
    lRes ← label l res        -- Encapsulate result
    return lRes               -- Return result
  ```

  In monadic terms, `toLabeled` is an environment-oriented action that provides a different context for a temporary bind thread, while `unlabel` is a state-oriented action that affects the current bind thread.
- `labelOf :: Label l ⇒Labeled l a →l`
  If `lv` is a labeled value with label l and value v, `labelOf lv` returns l.

The formal semantics of these functions are given in Section 4 (see Figure 4); in this section, we illustrate their functionality and use through examples.

Consider the previous example of `readReview`. The internal function `retrieveReview` takes a review identifier R and returns the review contents. This implies that, internally, `retrieveReview` has access to a list of reviews. These reviews are individually protected by a label, where the addition of a new review to the system can be implemented as:

```
addReview R L_R rv = do
  r ← label L_R rv
  addToReviewList R r
```

where the `addToReviewList` simply adds the `Labeled` review to the internal list. The implementation of `retrieveReview` directly follows:

```
retrieveReview R = do       -- Lcur = L_C
  r ← getFromReviewList R   -- Lcur = L_C
  rv ← unlabel r            -- Lcur = L_C ⊔ L_R
  return rv                 -- Lcur = L_C ⊔ L_R
```

where the `getFromReviewList` retrieves the `Labeled` review from the internal list and `unlabel` removes the protecting label, and raises the current label to reflect the read.

We previously alluded to an alternative implementation of `retrieveReview` which, instead, returns the labeled review content while keeping the current label the same. As `getFromReviewList` is a trusted function and not directly available to untrusted users, such as Clarice, `retrieveReviewAlt` can be implemented in terms of `toLabeled` and `retrieveReview`:

```
retrieveReviewAlt R = do                    -- Lcur = LC
  r ← toLabeled (LC ⊔ LR) $ do              -- Lcur = LC
        rv ← retrieveReview R               -- Lcur = LC ⊔ LR
        return rv                           -- Lcur = LC ⊔ LR
  return r                                  -- Lcur = LC
```

Note that although the current label within the inner computation is raised, the outer computation's label does not change—instead the review content is protected by $(L_C \sqcup L_R)$. Hence, only when the review content is actually needed, `unlabel` can be used to retrieve the content and raise the computation's label accordingly:

```
readReviewAlt R = do                        --  Lcur = LC
   r ← retrieveReviewAlt R                  --  Lcur = LC
   -- Perform other computations            --  Lcur = LC'
   rv ← unlabel r                           --  Lcur = LC' ⊔ LR
   printLabeledCh rv                        --  Lcur = LC' ⊔ LR
```

Our library also provides labeled alternatives to `IORefs` and files. Specifically, we provide labeled references `Ref l a` that are created with `newRef`, read with `readRef`, and written to with `writeRef`. When creating or writing to a reference with label L_R, it must be the case that $L_{\text{cur}} \sqsubseteq L_R \sqsubseteq C_{\text{cur}}$, while reading raises L_{cur} to $L_{\text{cur}} \sqcup L_R \sqsubseteq C_{\text{cur}}$. The rules for file operations follow identically, however writing to a file also implies observation (since the write can fail) and so the current label is raised in both cases. Finally, though beyond the scope of this paper, the library provides support for *privileges*. Privileges allow `LIO` code to operate under a more permissive $\sqsubseteq$ relation, but still more restricting than simply allowing the execution of arbitrary `IO` actions.

3. λChair

To demonstrate the flexibility of our dynamic information flow library, we present λChair, a simple API (built on the examples of Section 2) for implementing secure conference reviewing systems. In general, a conference reviewing system should support various features (and security policies) that a program committee can use in the review process; minimally, it should support:

- *Paper submission*: ability to add new papers to the system.
- *User creation*: ability to dynamically add new reviewers.
- *User login*: a means for authenticating users.
- *Review delegation*: ability to assign reviewers to papers.
- *Paper reading*: means for reading papers.
- *Review writing*: means for writing reviews.
- *Review reading*: means for reading reviews.
- *Conflict establishment*: ability to restrict specific users from reading conflicting reviews.

Even for such a minimal system, a number of security concerns must be addressed. First, only users assigned to a paper may write the corresponding reviews. Second, information from the review of one paper should not leak into a different paper's review. Third, a reviewer should not be permitted to modify the review of a paper that she/he is not assigned to review. And, fourth, users should not received any information regarding papers for which they have conflicts. We establish these four policies as non-interference policies for the confidentiality and integrity of reviews. We note that, although enforcing additional security properties is desirable, these four policies are sufficient when implementing a minimalistic and fair review system.

λChair's API provides the aforementioned security policies by applying information flow control. Following the examples of Section 2, we take the approach of enforcing IFC when writing to output channels, and thus the security for the above policies correspond to that of non-interference, i.e., secret data is not leaked into less secret channels/reviews. We do, however, note that the alternative, clearance restricting approach of Section 2, can be implemented and thus enforce the security policies by confinement rather than non-interference (see Section 5). Before delving into the details of the λChair, we first introduce the specific label format used in the implementation.

3.1 DC Labels

λChair is implemented using *Disjunction-Category (DC) labels*, a new label format especially suitable for systems with mutually distrusting parties. DC labels can be used to express a conjunction of restrictions on data, which allows for the construction of policies that reflect the concern of multiple parties. Such policies are expressed by leveraging the notions of *principals* and *Disjunction Categories* (henceforth just categories).

A principal is a string representation of a source of authority such as a user, a group, a role, etc. To ensure egalitarian protection mechanisms, *any* code is free to create principals.

A category is an information-flow restriction specifying the set of principals that *own* it. Each category is denoted as a disjunction of its owners; for example, the category owned by principals P_1 and P_2 is written as $[P_1 \vee P_2]$. Additionally, categories are qualified to be *secrecy* or *integrity* categories. A secrecy category restricts who can read, receive, or propagate information; an integrity category specifies who can modify a piece of data.

A *DC label* $L = \langle S, I \rangle$ is a set S of secrecy categories and a set I of integrity categories. All categories must be satisfied in order to allow information to flow and thus we write each set as a conjunction of categories. For example, the DC label $\langle \{[P_1 \vee P_2] \wedge [P_2 \vee P_3]\}, \{[P_4]\} \rangle$ has two secrecy categories and a single integrity category. Data with a DC label L_1 can be propagated to an endpoint having a DC label L_2 if the restrictions imposed by L_1 are uphold by L_2. We formalize this notion using the $\sqsubseteq$-relationship as follows.

Definition 1 (Can flow to)**.** Given any two DC labels $L_1 = \langle S_1, I_1 \rangle$ and $L_2 = \langle S_2, I_2 \rangle$, and interpreting each principal as a Boolean variable named according to the content of the string itself, we have

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

where $\Rightarrow$ denotes Boolean implication.

From now on, when we refer to a principal P, it can be interpreted as a string or Boolean variable depending on the context. As an example of the use of $\sqsubseteq$-relationship, the DC label $\langle \{[P_1 \vee P_2] \wedge [P_2 \vee P_3]\}, \{[P_4]\} \rangle \sqsubseteq \langle \{[P_1] \wedge [P_3]\}, \{[P_4 \vee P_6]\} \rangle$ since $P_1 \Rightarrow P_1 \vee P_2$, $P_3 \Rightarrow P_2 \vee P_3$, and $P_4 \Rightarrow P_4 \vee P_6$. Intuitively, the higher we move in the $\sqsubseteq$-relationship, the more restrictive the secrecy category becomes, while the integrity category, on the other hand, changes into a more permissive one. Additionally, we note that if a label contains a category that is implied by another, the latter is extraneous, as it has no effect on the value of the label, and can be safely removed.

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

$$L_1 \sqcup L_2 = \langle \mathsf{reduce}(S_1 \wedge S_2), \mathsf{reduce}(I_1 \vee I_2) \rangle$$
$$L_1 \sqcap L_2 = \langle \mathsf{reduce}(S_1 \vee S_2), \mathsf{reduce}(I_1 \wedge I_2) \rangle$$

Here, reduce removes any extraneous categories from a given set and $\wedge$ and $\vee$ denote the conjunction and disjunction of two category sets viewed as Boolean formulas of principals in conjunctive normal form.

In the context of the well-known DLM [28], a DC label secrecy category of the form $[P_1 \vee P_2 \vee \cdots \vee P_n]$ can be interpreted as the (slightly modified) DLM label component/policy $\{P_1, P_2, \ldots, P_n : P_1, P_2, \ldots, P_n\}$, where principals $P_1, \ldots, P_n$ are both the owners and readers. Although a DLM component consists of a single owner, which does not need to be part of the reader list, a DC label component (category) consists of multiple owners which are also the (only) readers. Using this slightly modified notion of a label component, a DLM label (set of components) loosely corresponds to our notion of a label (conjunction of disjunctions). Readers interested in the formal semantics of DC labels and the comparison with DLM can refer to `http://www.scs.stanford.edu/~deian/dclabel` for further details.

3.2 DC Labels in λChair

In this section we describe the data structures and the role of DC labels (from now on just labels) in λChair.

λChair provides an API to build review systems for the functionalities described in Section 3. Intuitively, the API just supplies administrators and reviewers with functions for querying review entries and modifying user accounts. Technically speaking, λChair runs over an underlying state monad that stores information regarding reviews and users.

Review entries A review entry is defined as a record consisting of a paper id, a reference to the corresponding paper, and a reference to the shared review 'notebook'. Specifically, a review entry is defined as

```
data ReviewEnt =  ReviewEnt { paperId :: Id
                            , paper   :: DCRef Paper
                            , review  :: DCRef Review }
```

where `DCRef` is a labeled reference using DC labels. In other words, `type DCRef a = Ref DCLabel a`. Note that this differs from the examples of Section 2, in which the reviews were simply `Labeled` types.

Reading and writing papers Upon logging in, users are allowed to read and print out any paper by providing the paper id. The label of the reference `paper` in the ith-review entry is set to $\langle\{\}, \{[P_i]\}\rangle$. Observe that the secrecy categories is empty (we interpret the empty category as the true Boolean value), thus allowing any function (without other integrity categories in its label) to read the paper by reading the reference content, i.e., the paper. This label does, however, restrict the modification of the paper to code running in a process that owns the integrity category P_i and can therefore run with the category $[P_i]$ in the integrity set of its current label. Only a trusted administrator is allowed to own such principals. Consequently, reviewers' code cannot modify the paper because their current label (assigned by the trusted login procedure) never includes P_i in their integrity set.

Reading and writing reviews Similarly, reviewers' code is allowed to access any reviews written to any reference `review`. However, once a review has been read, its contents must not be written to another paper's review. We fulfill this requirement by identifying, using labels, when a given piece of code reads a certain review. More specifically, we label the reference `review` in the ith-review entry as $\langle\{[R_i]\}, \{[R_i]\}\rangle$. As a consequence, when a function wishes to read the review for entry i, it must raise its current label as to include category $[R_i]$ in its secrecy and integrity sets (clearance permitted). Once a process has been *tainted* as such, it will not be able to modify the contents of another paper's `review` since the integrity category $[R_i]$ will cause the current label's integrity set to contain R_i in every category and $(R_i \vee C) \not\Rightarrow R_j$ for any C and $i \neq j$. Consider, for instance, a reviewer's code that has the current label set (by the trusted login procedure) to $\langle\{[R_i]\}, \{[R_i]\}\rangle$, i.e., in the process of reviewing paper P_i. If the code reads another review with label $L_j = \langle\{[R_j]\}, \{[R_j]\}\rangle$, the current label is then updated to $L = \langle\{[R_i] \wedge [R_j]\}, \{[R_i \vee R_j]\}\rangle$, which clearly implies that $L \not\sqsubseteq L_j$. The integrity category $[R_i]$ restricts the modification of the review to processes that own R_i. In this case, however, the process running reviewers' code, assigned to review paper i, contains, at least initially, category $[R_i]$ in the integrity set of its current label.

Users A reviewer is defined as a record consisting of a unique user name, password (used for authentication), and two disjoint sets of paper ids (in our implementation these are simple lists). One set corresponds to the user's conflicting papers, the second set corresponds to the papers the user has been assigned to review. Concretely, we define a user as follows:

```
data User = User { name        :: Name
                 , password    :: Password
                 , conflicts   :: [Id]
                 , assignments :: [Id] }
```

A user is authenticated using the name and password credentials. Upon logging in, the code of the reviewer assigned to papers $1, \ldots, n$ is executed with the current label initially set to $\langle\{\}, \{[R_1] \wedge \cdots \wedge [R_n]\}\rangle$, where R_i is the principal corresponding to review entry i. The current clearance is set to $\langle$`ALL`$, \{\}\rangle$. The special category set `ALL` (denoting the conjunction of all possible categories) in the clearance allows the executing code to (raise its current label and) read any data, while the integrity categories in the current label allow the process to only write to assigned reviews. Note, however, that in our case all reviewers append their review to the same review "notebook" and thus a write implies a read. Hence, to allow a reviewer to effectively perform a write-only operation, the process must execute the append function using `toLabeled`. We note the user is exposed to a function that appends to the review rather than directly writing to it, because multiple users are assigned to review the same paper and one should not be allowed to overwrite the work of another (using privileges a more elegant solution can easily be implemented).

Conflicts Following the `readReview` examples of Section 2, we restrict the reading, or more specifically, printing of a review to those reviewers in conflict with the paper. Although every user is allowed to retrieve a review, they cannot observe the result unless they write it to an output channel (in our simple example this corresponds to the standard output). Hence, code running on behalf of a user (determined after logging in) can only write to the output channel (using `printLabelCh`) if the current label L can flow to the output channel label L_o. Using the set of conflicting paper ids, for every user, we dynamically assign the output channel label $L_o = \langle S_o, \{\}\rangle$, where $S_o = \{[R_1] \wedge \cdots \wedge [R_n] \wedge [R_{n+1} \vee$ `CONFLICT`$] \wedge \cdots \wedge [R_N \vee$ `CONFLICT`$]\}$ and R_i where $i = n + 1, \ldots, N$ are the principals corresponding to *all* the review entries in the system (at the point of the print) that the authenticated user conflicts with. Here, `CONFLICT` corresponds to a principal that none of the users own (similar to P_i used in the labels of paper references). For each conflicting paper i, we create a category $[R_i \vee$ `CONFLICT`$]$. To observe the properties of this label, consider the case when executing code reads a conflicting paper R_i. In this situation, the current label is raised to $L = \langle\{[R_i] \wedge \cdots\}, \{\cdots\}\rangle$, and subsequently when attempting to write to the output channel, it is the case that $L \not\sqsubseteq L_o$. For $L \sqsubseteq L_o$ to hold true, there must be a category in L_o that implies $[R_i]$. However, due to the conflict, the only category containing R_i in the channel label's secrecy category is $[R_i \vee$ `CONFLICT`$]$ (and clearly $[R_i \vee$ `CONFLICT`$] \not\Rightarrow [R_i]$), which

asserts that conflicting data cannot flow to the output channel. We further highlight that the channel label permits non-conflicting reviews j to be observed by including the corresponding category $[R_j]$ without principal `CONFLICT`.

3.3 Implementation

In this section we present the API provided by λChair. As the main goal of λChair is to demonstrate the flexibility and power of our dynamic information flow library, we do not extend our example to a full-fledged system; the API can, however, be used to build relatively complex review systems. Below, we present the details of the λChair functions, which return actions in the `RevLIO` monad. This monad is a State monad defined using the State monad transformer with `LIO` as the base monad, and a state consisting of the system users, review entries, and name of the user that the executing code is running on behalf of.

System administrator interface A λChair administrator is provided with several functions that dynamically change the system state. Of these, we detail the most interesting cases below.

- `addPaper` :: `Paper` → `RevLIO Id`
  Given a paper, it creates a new review entry for the paper and return the paper id. Internally, `addPaper` uses a function similar to `addReview` of Section 2.
- `addUser` :: `Name` → `Password` → `RevLIO ()`
  Given a unique user name and password, it adds the new user.
- `addAssignment` :: `Name` → `Id` → `RevLIO ()`
  Given a user name and paper id, it assigns the user to review the corresponding paper. The user must not already be in conflict with the paper.
- `addConflict` :: `Name` → `Id` → `RevLIO ()`
  Given a user name and paper id, it marks the user as being in conflict with the paper. As above, it must be the case that the user is not already assigned to review the paper.
- `asUser` :: `Name` → `RevLIO ()` → `RevLIO ()`
  Given a user name, and user-constructed piece of code, it firsts authenticates the user and then executes the provided code with the current label and clearance of the user as described in Section 3.2. After the code is executed, the current label and clearance are restored and any information flow violations are reported.

Reviewer interface The reviewer, or user, composes an untrusted `RevLIO` computation (or action) that the trusted code executes using `asUser`. Such actions may be composed using the following interface:

- `findPaper` :: `String` → `RevLIO Id`
  Given a paper title, it returns its paper id, or fails if the paper is not found.
- `readPaper` :: `Id` → `RevLIO Paper`
  Given a paper id, the function returns an action which, when executed, returns the paper content.
- `readReview` :: `Id` → `RevLIO ()`
  Given a paper id, the function returns an action which, when executed, prints the review to the standard output. Its implementation is similar to the example of Section 2, except operating on references.
- `appendToReview` :: `Id` → `Content` → `RevLIO ()`
  Given a paper id and a review content, the function returns an action which, when executed, appends the supplied content to the review entry. Since there is no direct observation of the current review content, and to avoid label creep, the function, internally, uses `toLabeled`.

An IFC violation results in an exception (not-catchable by untrusted code) being thrown (in the semantics presented in Section 4, the program gets "stuck"). Figure 1 shows a simple example using the λChair API. In this example, Alice is assigned to review two papers. She does so by reading each paper (for the second, she also reads the existing reviews) and appending to the shared review. Bob, on the other hand, is added to the system after Alice's code is executed. Bob first writes a review for paper 2 and then attempts to violate IFC by trying to read (and write to the output channel) the reviews of paper 1. Though his review is appended to the correct paper, the review of the first paper is suppressed. We finally note that although the example is quite simple, it illustrates the use of the λChair primitives that may be used to implement a usable paper review system.

Figure 1 An example of code using λChair API.

```
module Admin where

import Alice
import Bob

main = evalRevLIO $ do
  -- Adding users to system
  addUser "Alice" "password"
  -- Adding papers to system
  p1 ← addPaper "Flexible Dynamic..."
  p2 ← addPaper "A Static..."
  -- Assign reviewers
  addAssignment "Alice" p1
  addAssignment "Alice" p2

  -- Executing Alice's code
  asUser "Alice" $ aliceCode

  -- Adding new users to system
  addUser "Bob" "password"
  -- Assign reviewers and conflicts
  addAssignment "Bob" p2
  addConflict "Bob"   p1

  -- Executing Bob's code
  asUser "Bob" $ bobCode

module AliceCode where

aliceCode = do
  p1 ← findPaper "Flexible Dynamic..."
  p2 ← findPaper "A Static..."
  readPaper p1
  appendToReview p1 "Interesting work!"
  readPaper p2
  readReview p2
  appendToReview p2 "What about adding new users?"
  return ()

module BobCode where

bobCode = do
  p1 ← findPaper "Flexible Dynamic..."
  p2 ← findPaper "A Static..."
  appendToReview p2 "Hmm, IFC.."
  readReview p1 -- IFC violation attempt
  return ()
```

Figure 2 Formal syntax for terms, expressions, and types.

Label:		l
Address:		a
Term:	$v ::=$	$\texttt{true} \mid \texttt{false} \mid \texttt{()} \mid l \mid a \mid x \mid \lambda x.e \mid (e, e)$
		$\mid \texttt{fix}\ e \mid \texttt{Lb}\ v\ e \mid (e)^{\texttt{LIO}} \mid \bullet$
Expression:	$e ::=$	$v \mid e\ e \mid \pi_i\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ e$
		$\mid \texttt{label}\ e\ e \mid \texttt{unlabel}\ e \mid \texttt{toLabeled}\ e\ e$
		$\mid \texttt{newRef}\ e\ e \mid \texttt{readRef}\ e \mid \texttt{writeRef}\ e\ e$
		$\mid \texttt{lowerClr}\ e \mid \texttt{getLabel} \mid \texttt{getClearance}$
		$\mid \texttt{labelOf}\ e \mid \texttt{labelOfRef}\ e$
Type:	$\tau ::=$	$\texttt{Bool} \mid \texttt{()} \mid \tau \to \tau \mid (\tau, \tau)$
		$\mid \ell \mid \texttt{Labeled}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau \mid \texttt{Ref}\ \ell\ \tau$
Store:		ϕ :Address $\to \texttt{Labeled}\ \ell\ \tau$

4. Formal Semantics for LIO

This section formalizes our library for a simple call-by-name[1] λ-calculus extended with Booleans, unit values, pairs, recursion, references, and the monadic operations for LIO. Figure 2 provides the formal syntax of the considered language. Syntactic categories v, e, and τ represent terms, expressions, and types, respectively. Terms are side-effect free while expressions denote (possible) side-effecting computations.

In the syntax category v, symbol `true` and `false` represent Boolean values. Symbol `()` represents the unit value. Symbol ℓ denotes security labels. Symbol a represent memory addresses in a given store. Terms include variables (x), functions ($\lambda x.e$), tuples (e, e), and recursive functions ($\texttt{fix}\ e$). Three special syntax nodes are added to this category: $\texttt{Lb}\ v\ e$, $(e)^{\texttt{LIO}}$, and $\bullet$. Node $\texttt{Lb}\ v\ e$ denotes the run-time representation of a labeled value. Similarly, node $(e)^{\texttt{LIO}}$ denotes the run-time representation of a monadic LIO computation. Node $\bullet$ represents an erased term (explained in Section 5). None of these special nodes appear in programs written by users and they are merely introduced for technical reasons.

Expressions are composed of values (v), function applications ($e\ e$), pair projections ($\pi_i\ e$), conditional branches ($\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$), and local definitions ($\texttt{let}\ x = e\ \texttt{in}\ e$). Additionally, expressions may involve operations related to monadic computations in the LIO monad. More precisely, $\texttt{return}\ e$ and $e\ \texttt{>>=}\ e$ represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the LIO monad are given by `label`, `unlabel`, and `toLabeled`. Expression $\texttt{label}\ e_1\ e_2$ creates a labeled value that guards e_2 with label e_1. Expression $\texttt{unlabel}\ e$ acquires the content of the labeled value e while in a LIO computation. Expression $\texttt{toLabeled}\ e_1\ e_2$ creates a labeled value, with label e_1, of the result obtained by evaluating the LIO computation e_2. Non-proper morphisms related to creating, reading, and writing of references are respectively captured by expressions `newRef`, `readRef`, and `writeRef`. Expression $\texttt{lowerClr}\ e$ allows lowering of the current clearance to e. Expressions `getLabel` and `getClearance` return the current label and current clearance of an LIO computation. Finally, expressions $\texttt{labelOf}\ e$ and $\texttt{labelOfRef}\ e$ respectively obtain the security label of labeled values and references.

We consider standard types for Booleans (`Bool`), unit (`()`), pairs (τ, τ), and function $(\tau \to \tau)$ values. Type ℓ describes security labels. Type $\texttt{Labeled}\ \ell\ \tau$ describes labeled values of type τ where the label is of type ℓ. Type $\texttt{LIO}\ \ell\ \tau$ represents monadic LIO computations, with a result type τ and the security labels of type ℓ. Type $\texttt{Ref}\ \ell\ \tau$ describes labeled references, with labels of type ℓ, to values of type τ.

[1] For clarity, we use a call-by-name instead of call-by-need calculus; extension to the latter is straight forward, as shown in [23].

Figure 3 Typing rules for terms.

$$\vdash l : \ell \qquad \frac{\Gamma(a) = \texttt{Labeled}\ \ell\ \tau}{\Gamma \vdash a : \texttt{Ref}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{Lb}\ e_1\ e_2 : \texttt{Labeled}\ \ell\ \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e)^{\texttt{LIO}} : \texttt{LIO}\ \ell\ \tau} \qquad \Gamma \vdash \bullet : \tau$$

The typing judgments have standard form $\Gamma \vdash e : \tau$, such that expression e has type τ assuming the typing environment Γ; we use Γ for both variable and store typings. The typing rules for several terms are shown in Figure 3; the typing for the remaining terms and expressions are standard and we therefore do not describe them any further. We, however, note that, different from previous work [12, 32], we do not require to use any of the sophisticated features of Haskell's type-system, a direct consequence of our dynamic approach.

The LIO monad presented in Section 2 is implemented as a State monad. To simplify the formalization and description of expressions, without loss of generality, we make the state of the monad part of a run-time environment. More precisely, for a given LIO computation, the symbol Σ denotes a run-time environment that contains the current label, written $\Sigma.\texttt{lbl}$, the current clearance, written $\Sigma.\texttt{clr}$, and store, written $\Sigma.\phi$. A run-time environment Σ and LIO computation form a *configuration* $\langle \Sigma, e \rangle$. Given a configuration $\langle \Sigma, e \rangle$, the current label, clearance, and store when starting evaluation e is given by $\Sigma.\texttt{lbl}$, $\Sigma.\texttt{clr}$, and $\Sigma.\phi$, respectively.

The relation $\langle \Sigma, e \rangle \longrightarrow \langle \Sigma', e' \rangle$ represents a single evaluation step from expression e, under the run-time environment Σ, to expression e' and run-time environment Σ'. We say that e reduces to e' in one step. We define such relation in terms of a structured operational semantics via evaluation contexts [14].

The reduction rules for the core simply-typed λ-calculus are standard and therefore omitted. We note that substitution $([e_1/x]\ e_2)$ is defined in the usual way: homomorphic on all operators and renaming bound names to avoid captures. Figure 4 presents the evaluation contexts and non-standard reduction rules for our language. These rules guarantee that programs written using our approach fulfill non-interference, i.e., confidential information is not leaked, and confinement, i.e., a computation cannot access data above the clearance.

The main contribution of our language are the primitives `label`, `unlabel`, and `toLabeled`. Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the LIO computation and thus guaranteeing containment properties (see Section 5). Rule (UNLAB) provides a method for accessing the content e of a labeled value $\texttt{Lb}\ l\ e$ in LIO computations. When the content of a labeled value is "retrieved" and used in a LIO computation, the current label is raised ($\Sigma' = \Sigma[\texttt{lbl} \mapsto l']$, where $l' = \Sigma.\texttt{lbl} \sqcup l$), capturing the fact that the remaining computation might depend on e. Rule (TOLAB) deserves some attention. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Expression $\texttt{toLabeled}\ l\ e$ is used to execute the provided LIO computation e until completion ($\langle \Sigma, e \rangle \longrightarrow^* \langle \Sigma', (v)^{\texttt{LIO}} \rangle$) and wraps its result v into a labeled value with label l. Observe that the label l needs to be an upper bound on the current label for the evaluation of computation e ($\Sigma'.\texttt{lbl} \sqsubseteq l$). Specifying label l is responsibility of the programmer. The reason for this is due to the fact that security labels are protected by the current

Figure 4 Semantics for non-standard constructs.

$$
\begin{aligned}
E \quad ::= \; & [\cdot] \mid \texttt{Lb}\ E\ e \mid E\ e \mid \pi_i\ E \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{return}\ E \mid E\ \texttt{>>=}\ e \\
& \mid \texttt{label}\ E\ e \mid \texttt{unlabel}\ E \mid \texttt{toLabeled}\ E\ e \\
& \mid \texttt{newRef}\ E\ e \mid \texttt{readRef}\ E \mid \texttt{writeRef}\ E\ e \\
& \mid \texttt{lowerClr}\ E \mid \texttt{labelOf}\ E \mid \texttt{labelOfRef}\ E
\end{aligned}
$$

$$\langle \Sigma, E[\texttt{return}\ v] \rangle \longrightarrow \langle \Sigma, E[(v)^{\texttt{LIO}}] \rangle$$

$$\langle \Sigma, E[(v)^{\texttt{LIO}}\ \texttt{>>=}\ e_2] \rangle \longrightarrow \langle \Sigma, E[e_2\ v] \rangle$$

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

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

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

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

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

(WREF)
$$\frac{\Sigma.\phi(a) = \texttt{Lb}\ l\ e \qquad \Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e']}{\langle \Sigma, E[\texttt{writeRef}\ a\ e'] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ ()] \rangle}$$

(LWCLR)
$$\frac{\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\texttt{clr} \mapsto l]}{\langle \Sigma, E[\texttt{lowerClr}\ l] \rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ ()] \rangle}$$

(CLAB)
$$\frac{l = \Sigma.\texttt{lbl}}{\langle \Sigma, E[\texttt{getLabel}] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ l] \rangle}$$

(CCLR)
$$\frac{l = \Sigma.\texttt{clr}}{\langle \Sigma, E[\texttt{getClearance}] \rangle \longrightarrow \langle \Sigma, E[\texttt{return}\ l] \rangle}$$

(GLAB)
$$\langle \Sigma, E[\texttt{labelOf}\ (\texttt{Lb}\ l\ e)] \rangle \longrightarrow \langle \Sigma, E[l] \rangle$$

(GLABR)
$$\frac{e = \Sigma.\phi(a)}{\langle \Sigma, E[\texttt{labelOfRef}\ a] \rangle \longrightarrow \langle \Sigma, E[\texttt{labelOf}\ e] \rangle}$$

label, effectively making them public information accessible to any computation within scope (see rules (GLAB) and (GLABR)). As a consequence, if `toLabeled` did not require an upper bound on the data to be observed within e, labels can be used to leak information. Recall that the current label and clearance of a given `LIO` computation can be changed dynamically. To illustrate this point, consider a computation whose current label is l_0, taking two (confidential) labeled values as arguments, with respective labels l_1 and l_2 such that $l_i \not\sqsubseteq l_0, i = 1, 2$.

```
leak lV1 lV2 = do
  lV3 ← toLabeled' $ do
    v1 ← unlabel lV1       -- L_cur = l1
    if v1 then return True
          else unlabel lV2 -- L_cur = l2
  return (labelOf lV3)
```

Therefore, if the returned value is l_1 or l_2 (remember that labels are public information), information is directly leaked! To close this channel, programmers should provide an upper bound of the current label obtained when e finishes computing. Since our approach is dynamic, flow-sensitive, and sound, this may require non-trivial, and possibly complicated, static analysis in order to automatically determine the label for each call of `toLabeled` [31].

By using big-step semantics instead of an evaluation context of the form `toLabeled` $l\ E$, rule (TOLAB) does not need to rely on the use of trusted primitives or a stack for (saving and) restoring the current label and clearance when executing `toLabeled`.

When creating a reference, `newRef` $l\ e$ produces a labeled value that guards e with label l (`Lb` $l\ e$) and stores it in the memory store ($\Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e]$). The result of this operation is the memory address a (`return` a). Observe that references are created only if the reference's label (l) is between the current label and clearance label of the `LIO` monad ($\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}$). The restriction $l \sqsubseteq \Sigma.\texttt{clr}$ is to assure that programs cannot manipulate or access data beyond their clearance (see Section 5). Rule (RREF) obtains the content e of a labeled value `Lb` $l\ e$ stored in the address a. This rule raises the current label to the security level l' ($\Sigma' = \Sigma[\texttt{lbl} \mapsto l']$ where $l' = \Sigma.\texttt{lbl} \sqcup l$). As in the previous rule, (RREF) enforces that the result of reading a reference is below the clearance ($l' \sqsubseteq \Sigma.\texttt{clr}$). Rule (WREF) updates the memory store with a new value for the reference ($\Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e']$) as long as the label of the reference is above the current label and it does not exceed the clearance ($\Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr}$). If considering $\Sigma.\texttt{lbl}$ as a dynamic version of the `pc` the restriction that the label of the reference must be above the current label ($\Sigma.\texttt{lbl} \sqsubseteq l$) is similar to the one imposed by [30].

Rule (LWCLR) allows a computation to lower the current clearance to l. This operation is particularly useful when trying to contain the access to some data as well as the effects produced by computations executed by `toLabeled`. Rules (CLAB) and (CCLR) obtain the current label and clearance from the run-time environment. Finally, rules (GLAB) and (GLABR) return the labels of labeled values and references. Observe that, regardless of the current label and clearance of the run-time environment, these two rules always succeed, effectively making labels public data.

5. Soundness

In this section we show that `LIO` computations satisfy two security policies: non-interference and containment. Non-interference shows that secrets are not leaked, while containment establishes that certain piece of code cannot manipulate or have access to certain data. The latter policy is similar to the containment policies presented in [4, 24].

5.1 Non-interference

As in [26, 32], we prove the non-interference property by using the technique of *term erasure*. Intuitively, data at security levels where the attacker cannot observe information can be safely rewritten to the syntax node $\bullet$. For the rest of the paper, we assume that the attacker can observe data up to security level L. The syntactic term $\bullet$, denoting an erased expression, may be associated to any type (recall Figure 3). Function ε_L is responsible for performing the rewriting for data at security level not lower than L. In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(\texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e') = \texttt{if}\ \varepsilon_L(E)\ \texttt{then}\ \varepsilon_L(e)\ \texttt{else}\ \varepsilon_L(e')$). In the case of data constructors it is simply the identity function.

Figure 5 Erasure function for several terms, memory store and configurations.

$$\varepsilon_L(l) = l \qquad \varepsilon_L(a) = a$$

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

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

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

The two interesting cases for this function are when ε_L is applied to a labeled value or a given configuration. In such cases, term erasing could indeed modify the behavior of the program. Figure 5 shows the definition of ε_L for several terms and configurations. A labeled value is erased if the label assigned to it is above L ($\varepsilon_L(\texttt{Lb}\ l\ e) = \texttt{Lb}\ l\ \bullet$, if $l \not\sqsubseteq L$). Similarly, the computation performed in a certain configuration is erased if the current label is above L ($\varepsilon_L(\langle \Sigma, e\rangle) = \langle \varepsilon_L(\Sigma), \bullet\rangle$ if $\Sigma.\texttt{lbl} \not\sqsubseteq L$).

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

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

Expressions under this relationship are evaluated in the same way as before, with the exception that, after one evaluation step, the erasure function is applied to the resulting configuration, i.e., run-time environment and expression. In that manner, the relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$.

Most results that prove non-interference pursue the goal of establishing a relationship between $\longrightarrow^*$ and $\longrightarrow_L^*$ through the erasure function, as highlighted in Figure 1. Informally, the diagram establishes that erasing all secret data, i.e., data not below L, and then taking evaluation steps in $\longrightarrow_L$ is the same as taking steps in $\longrightarrow$ and then erasing all the secret values in the resulting configuration. Observe that if information from some level above L is leaked by e, then erasing all secret data and then taking evaluation steps in $\longrightarrow_L$ might not be the same as taking steps in $\longrightarrow$ and then erasing all the secret values in the resulting configuration.

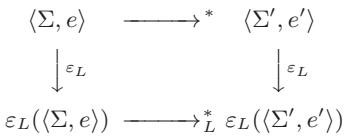


Figure 1. Simulation between $\longrightarrow^*$ and $\longrightarrow_L^*$.

For simplicity, we assume that the space address of the memory store is split into different security levels and that allocation is deterministic. In that manner, the address returned when creating a reference with level l depends only on the references with level l already in the store. These assumptions are valid in our language since, similar to traditional references in Haskell, we do not provide any mechanisms for deallocation or inspection of addresses in the API. However, when memory allocation is an observable channel, the library could be adapted in order to deal with non-opaque pointers [17].

We start by showing that the evaluation relationship $\longrightarrow$ and $\longrightarrow_L$ are deterministic. We note that $e = e'$ means syntactic equality between expressions e and e'. Equality between run-time environments, written $\Sigma = \Sigma'$, is defined as the point-wise equality between mappings Σ and Σ'.

Proposition 1 (Determinacy of $\longrightarrow$)**.**

- *For any expression e and run-time environment Σ such that $\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e''\rangle$, there is a unique term e' and unique evaluation context E such that $e = E[e']$.*
- *If $\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle$ and $\langle \Sigma, e\rangle \longrightarrow \langle \Sigma'', e''\rangle$, then $e' = e''$ and $\Sigma' = \Sigma''$.*

Proof. By induction on expressions and evaluation contexts. □

Proposition 2 (Determinacy of $\longrightarrow_L$)**.** *If $\langle \Sigma, e\rangle \longrightarrow_L \langle \Sigma', e'\rangle$ and $\langle \Sigma, e\rangle \longrightarrow_L \langle \Sigma'', e''\rangle$, then $e' = e''$ and $\Sigma' = \Sigma''$.*

Proof. From Proposition 1 and definition of ε_L. □

The following proposition shows that the erasure function is homomorphic to the application of evaluation contexts and substitution as well as that it is idempotent.

Proposition 3 (Properties of erasure function)**.**

1. $\varepsilon_L(E[e]) = \varepsilon_L(E)[\varepsilon_L(e)]$
2. $\varepsilon_L([e_2/x]e_1) = [\varepsilon_L(e_2)/x]\varepsilon_L(e_1)$
3. $\varepsilon_L(\varepsilon_L(e)) = \varepsilon_L(e)$
4. $\varepsilon_L(\varepsilon_L(E)) = \varepsilon_L(E)$
5. $\varepsilon_L(\varepsilon_L(\Sigma)) = \varepsilon_L(\Sigma)$
6. $\varepsilon_L(\varepsilon_L(\langle \Sigma, e\rangle)) = \varepsilon_L(\langle \Sigma, e\rangle)$

Proof. From the definition of ε_L and by induction on expressions and evaluation contexts. □

The next lemma establishes a simulation between $\longrightarrow$ and $\longrightarrow_L$ for expressions that do not execute `toLabeled`.

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 $\Gamma \vdash e' : \tau$ and $\varepsilon_L(\langle \Sigma, e\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', e'\rangle)$.*

Proof. Subject reduction holds by showing that a reduction step does not change the types of references in the store $\Sigma.\phi$ and then applying induction on the typing derivations. The simulation holds by simple case analysis on e. □

Using this lemma, we then show that the simulation is preserved when performing several evaluation steps.

Lemma 2 (Simulation for expressions not executing `toLabeled`)**.** *If $\Gamma \vdash e : \tau$, $\langle \Sigma, e\rangle \longrightarrow^* \langle \Sigma', e'\rangle$ where there are no executions of `toLabeled`, then $\Gamma \vdash e' : \tau$ and $\varepsilon_L(\langle \Sigma, e\rangle) \longrightarrow_L^* \varepsilon_L(\langle \Sigma', e'\rangle)$.*

Proof. By induction on $\longrightarrow$ and applying Lemma 1. □

The reason for highlighting the distinction between expressions executing `toLabeled` and those not executing it is due to the fact that the evaluation of `toLabeled` involves big-step semantics (recall rule (TOLAB) in Figure 4). However, the next lemma shows the simulation between $\longrightarrow^*$ and $\longrightarrow_L^*$ for any expression e, and it is proved by simple induction on the number of executed `toLabeled`.

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

Proof. By induction on the number of executed `toLabeled` and applying Lemma 2 for the base case. □

Figure 6 L-equivalence for expressions.

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

We define L-equivalence between expressions. Intuitively, two expressions are L-equivalent if they are syntactically equal, modulo

labeled values whose labels are above L. We use $\approx_L$ to represent L-equivalence for expressions. Figure 6 shows the definition for labeled values. Considering the simple lattice: $\texttt{L} \sqsubseteq \texttt{M} \sqsubseteq \texttt{H}$ and an attacker at level L, it holds that $\texttt{Lb H } 8 \approx_{\texttt{L}} \texttt{Lb H } 9$, but it does not hold that $\texttt{Lb L } 2 \approx_{\texttt{L}} \texttt{Lb L } 3$ or $\texttt{Lb H } 8 \approx_{\texttt{L}} \texttt{Lb M } 8$. Recall that labels are protected by the current label, and thus (usually) observable by an attacker — unlike the expressions they protect, labels must be the same even if they are above L. The rest of $\approx_L$ is defined as syntactic equality between constants (e.g., $\texttt{true} \approx_L \texttt{true}$) or homomorphisms (e.g., $\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \approx_L \texttt{if } e' \texttt{ then } e_1' \texttt{ else } e_2'$ if $e \approx_L e'$, $e_1 \approx_L e_1'$, and $e_2 \approx_L e_2'$).

Since our language encompasses side-effecting expressions, it is also necessary to define L-equivalence between memory stores. Specifically, we say that two run-time environments are L-equivalent if an attacker at level L cannot distinguish them:

Definition 2 (L-equivalence for stores)**.**

$$\frac{l \sqsubseteq L \vee l' \sqsubseteq L \qquad \forall a.\Sigma.\phi(a) = \texttt{Lb } l\ e \approx_L \Sigma'.\phi(a) = \texttt{Lb } l'\ e'}{\Sigma.\phi \approx_L \Sigma'.\phi}$$

Note that the L-equivalence ignores the store references with labels above L. Similarly, we define L-equivalence for configurations.

Definition 3 (L-equivalence for configurations)**.**

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

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

In the above definition, it is worth remarking that we do not require $\approx_L$ for expressions when the current label is not below L. This omission comes from the fact that e and e' would be reduced to $\bullet$ when applying our simulation between $\longrightarrow^*$ and $\longrightarrow_L^*$ (recall Figure 5).

The next theorem shows the non-interference policy. It essentially states that given two inputs with possibly secret information, the result of the computation is indistinguishable to an attacker. In other words, there is no information-flow from confidential data to outputs observable by the attacker.

Theorem 1 (Non-interference)**.** *Given a computation e (with no $\bullet$, $(\ )^{\texttt{LIO}}$, or $\texttt{Lb}$) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \to \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau')$, 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*

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

Observe that even though we assume that the input labeled values e_1 and e_2 are observable by the attacker ($l \sqsubseteq L$), they might contain confidential data. For instance, e_1 could be of the form $\texttt{Lb}\ l\ (\texttt{Lb}\ l'\ \texttt{true})$ where $l' \not\sqsubseteq L$.

Proof. The L-equivalence (and, thus, the proof) directly follows by Lemma 3 and determinacy of $\longrightarrow_L$. □

5.2 Confinement

In this section we present the formal guarantees that LIO computations cannot modify data below their current label and manipulate information above their current clearance. These kind of properties are similar to the ones found in [4, 24].

We start by proving that the current label of a LIO computation does not decrease.

Proposition 4 (Monotonicity of the current label)**.** *If $\Gamma \vdash e : \tau$ and $\langle \Sigma, e\rangle \longrightarrow^* \langle \Sigma', e'\rangle$, then $\Sigma.\texttt{lbl} \sqsubseteq \Sigma'.\texttt{lbl}$.*

Similarly, we show that the current clearance of a LIO computation never increases.

Proposition 5 (Monotonicity of the current clearance)**.** *If $\Gamma \vdash e : \tau$ and $\langle \Sigma, e\rangle \longrightarrow^* \langle \Sigma', e'\rangle$, then $\Sigma'.\texttt{clr} \sqsubseteq \Sigma.\texttt{clr}$.*

Proposition 4 and 5 are crucial to assert that once a LIO computation reads confidential data, it cannot lower its current label to leak it. Similarly, a computation should not be able to arbitrarily increase its clearance; doing so would allow it to read any data with no access restrictions.

Before delving into the containment theorems, we first define a store modifier that removes all store elements with a label above a given label l:

$$\frac{(\Sigma.\phi)_{\downarrow l} = \Sigma.\phi \setminus \{(a, \texttt{Lb}\ l'\ e) : a \in \mathrm{dom}\,(\Sigma.\phi) \wedge l \sqsubseteq l'\}}{(\Sigma.\phi)_{\downarrow l}}$$

In other words, this retains all the labeled references with a label below l, usually the current label.

The first theorem states that LIO computations cannot create labeled values, new locations or modify memory cells below their current label (*no-write down*).

Theorem 2 (Containment imposed by the current label)**.** *Given labels l, l_c, and l_v, a computation e (with no $\bullet$, $(\ )^{\texttt{LIO}}$, or $\texttt{Lb}$) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \to \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau)$, environment $\Sigma[\texttt{lbl} \mapsto l, \texttt{clr} \mapsto l_c]$ such that $l \sqsubseteq l_c$, and $l \not\sqsubseteq l_v$.*

$$\begin{aligned}&\Gamma \vdash e_1 : \texttt{Labeled}\ \ell\ \tau\\ &\wedge e_1 = \texttt{Lb}\ l_v\ e_1' \wedge \langle \Sigma, e\ e_1\rangle \longrightarrow^* \langle \Sigma', (\texttt{Lb}\ l_v\ e_1'')^{\texttt{LIO}}\rangle\\ &\Rightarrow (\Sigma.\phi)_{\downarrow l} = (\Sigma'.\phi)_{\downarrow l} \wedge e_1' = e_1''\end{aligned}$$

Proof. The proof follows directly from Proposition 4, definition of the store modifier and induction on $\longrightarrow^*$. □

The theorem simply states that the computation cannot allocate or modify the store below l. Moreover the computation should only be able to return a labeled value below its current label that was provided as input, or by capture.

Dual to Theorem 2, the next theorem captures the fact that LIO computations cannot compute on labeled values above their clearance. In other words, LIO computations cannot create, read, and write references or read and create contents for labeled values above the clearance (recall that references store labeled values). Again, we first define a store modifier; in this case, one that removes all store elements below a given clearance as follows:

$$\frac{(\Sigma.\phi)_{\uparrow l} = \Sigma.\phi \setminus \{(a, \texttt{Lb}\ l'\ e) : a \in \mathrm{dom}\,(\Sigma.\phi) \wedge l' \sqsubseteq l\}}{(\Sigma.\phi)_{\uparrow l}}$$

In other words, this retains all the labeled references with a label above l, usually the current clearance.

Theorem 3 (Containment imposed by clearance)**.** *Given labels l, l_c, and l_v, a computation e (with no $\bullet$, $(\ )^{\texttt{LIO}}$, or $\texttt{Lb}$) where $\Gamma \vdash e : \texttt{Labeled}\ \ell\ \tau \to \texttt{LIO}\ \ell\ (\texttt{Labeled}\ \ell\ \tau)$, environment $\Sigma[\texttt{lbl} \mapsto l, \texttt{clr} \mapsto l_c]$ such that $l \sqsubseteq l_c$ and $l_v \not\sqsubseteq l_c$,*

$$\begin{aligned}&\Gamma \vdash e_1 : \texttt{Labeled}\ \ell\ \tau\\ &\wedge e_1 = \texttt{Lb}\ l_v\ e_1' \wedge \langle \Sigma, e\ e_1\rangle \longrightarrow^* \langle \Sigma', (\texttt{Lb}\ l_v\ e_1'')^{\texttt{LIO}}\rangle\\ &\Rightarrow (\Sigma.\phi)_{\uparrow l_c} = (\Sigma'.\phi)_{\uparrow l_c} \wedge e_1' = e_1''\end{aligned}$$

Proof. Directly from Proposition 5, definition of the store modifier and induction on $\longrightarrow^*$. □

6. Related Work

Heintze and Riecke [18] consider security for lambda-calculus where lambda-terms are explicitly annotated with security labels, for a type-system that guarantees non-interference. One of the key aspects of their work consists of an operator which raises the security annotation of a term in a similar manner to our raise of the current label when manipulating labeled values. Similar ideas of floating labels have been used by many operating systems, dating back to the High-Water-Mark security model [22] of the ADEPT-50 in the late 1960s. Asbestos [13] first combined floating labels with the Decentralized label model [28].

Abadi et al. [1] develop the dependency core calculus (DCC) based on a hierarchy of monads to guarantee non-interference. In their calculus, they define a monadic type that "protects" (the confidentiality of) side-effect-free values at different security levels. Though not a monad, our `Labeled` type similarly protects pure values at various security levels. To manipulate such values, DCC uses a non-standard typing rule for the bind operator; the essence of this operator, in a dynamic setting with side-effectful computations, is captured in our library through the interaction of of `Labeled`, `unlabel`, and `LIO`.

Tse and Zdancewic [36] translate DCC to System F and show that non-interference can be stated using the parametricity theorem for System F. The authors also provide a Haskell implementation for a two-point lattice. Their implementation encodes each security level as an abstract data type constructed from functions and binding operations to compose computations with permitted flows. Since they consider the same non-standard features for the `bind` operation as in DCC, they provide as many definitions for `bind` as different type of values produced by it. Moreover, their implementation needs to be compiled with the flag `-fallow-undecidable-instances`, in GHC. Our work, in contrast, defines only one bind operation for `LIO`, without the need for such compiler extensions.

Harrison and Hook show how to implement an abstract operating system called *separation kernel* [16]. Programs running under this multi-threading operating system satisfy non-interference. To achieve this, the authors rely on the state monad to represent threads, monad transformers to present parallel composition, and the resumption monad to achieve communication between threads. Non-interference is then enforced by the scheduler implementation, which only allow signaling threads at the same, or higher, security level as the thread that issued the signal. The authors use monads differently from us; their goal is to construct secure kernels rather than provide information-flow security as a library. Our library is simpler and more suitable for writing sequential programs in Haskell. Extending our library to include concurrency is stated as a future work.

Crary et al. [8] design a monadic calculus for non-interference for programs with mutable state. Similar to our work, their language distinguishes between term and expressions, where terms are pure and expressions are (possibly) effectful computations. Their calculus mainly tracks information flow by statically approximating the security levels of effects produced by expressions. Compared to their work, we only need to make approximations of the side-effects of a given computation when using `toLabeled`; the state of `LIO` keeps track of the dynamic security level upper bound of observed data. Overall, our dynamic approach is more flexible and permissive than their proposed type-system.

Pottier and Simonet [30, 35] designed FlowCaml, a compiler to enforce non-interference for OCaml programs. Rather than implementing a compiler from scratch, and more similar to our approach, the seminal work by Li and Zdancewic [25] presents an implementation of information-flow security as a library, in Haskell, using a generalization of monads called Arrows [19]. Extending their work, Tsai et al. [7] further consider side-effects and concurrency. Contributing to library-based approaches, Russo et al. [32] eliminate the need for Arrows by showing an IFC library based solely on monads. Their library defines monadic types to track information-flow in pure and side-effectful computations. Compared to our dynamic IFC library, Russo et al.'s library is slightly less permissive and leverages Haskell's type-system to statically enforce non-interference. However, we note that our library has similar (though dynamic) functions provided by their SecIO library; similar to `unlabel`, they provide a function that maps pure labeled values into side-effectful computations; similar to `toLabeled`, they provide a function that allows reading/writing secret files into computations related to public data.

Recently, Morgenstern et al. [27] encoded an authorization- and IFC-aware programming language in Agda. Their encoding, however, does not consider computations with side-effects. More closely related, Devriese and Piessens [12] used monad transformers and parametrized monads [3] to enforce non-interference, both dynamically and statically. However, their work focuses on modularity (separating IFC enforcement from underlying user API), using type-class level tricks that make it difficult to understand errors triggered by insecurities. Moreover, compared to our work, where programmers write standard Haskell code, their work requires one to firstly encode programs as values of a specific type.

Compared to other language-based works, LIO uses the notion of clearance. Bell and La Padula [5] formalized clearance as a bound on the current label of a particular users' processes. In the 1980s, clearance became a requirement for high-assurance secure systems purchased by the US Department of Defense [11]. More recently, HiStar [39] re-cast clearance as a bound on the label of any resource created by the process (where raising a process's label is but one means of creating a something with a higher label). We adopt HiStar's more stringent notion of clearance, which prevents software from copying data it cannot read and facilitates bounding the time during which possibly untrustworthy software can exploit covert channels.

7. Conclusion

We propose a new design point for IFC systems in which most values in lexical scope are protected by a single, mutable, *current label*, yet one can also encapsulate and pass around the results of computations with different labels. Unlike other language-based work, our model provides a notion of *clearance* that imposes an upper bound on the program label, thus providing a form of discretionary access control on portions of the code.

We prove information flow and integrity properties of our design and describe LIO, an implementation of the new model in Haskell. LIO, which can be implemented entirely as a library (based on type safety), demonstrates both the applicability and simplicity of the approach. Our non-interference theorem proves the conventional property that lower-level results do not depend on higher-level inputs – the label system prevents inappropriate flow of information. We also prove containment theorems that show the effect of clearance on the behavior of code. In effect, lowering the clearance imposes a discretionary form of access control by preventing subsequent code (within that scope) from accessing higher-level information.

As an illustration of the benefits and expressive power of this system, we describe a reviewing system that uses LIO labels to manage integrity and confidentiality in an environment where users and labels are added dynamically. Although we have use LIO for the λChair API and even built a relatively large web-framework that securely integrates untrusted third-party applications, we believe that changes in the constructs are likely to occur as the lan-

guage matures. This further supports our library-based approach to language-based security.

An interesting future work consists on extending our library to handle concurrency. Enforcement mechanisms for sequential programs do not generalize naturally to multithreaded programs. In this light, it is hardly surprising that Jif [29], the mainstream IFC compiler, lack support for multithreading. Due to the monadic structure of LIO programs, we believe it is possible to extend our library to consider concurrency, that addresses termination [2] and internal-timing leaks [38].

Acknowledgments

We thank Alex Aiken and the anonymous reviewers for their insightful comments. This work was funded by DARPA (CRASH and PROCEED), NSF (including a Cybertrust award and the TRUST Center), the Air Force Office of Scientific Research, the Office of Naval Research, and the Swedish research agency VR.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the 13th European Symp. on Research in Computer Security*, pages 333–348. Springer-Verlag, 2008.

[3] R. Atkey. Parameterised notions of computation. In *Workshop on mathematically structured functional programming, ed. Conor McBride and Tarmo Uustalu. Electronic Workshops in Computing, British Computer Society*, pages 31–45, 2006.

[4] A. Banerjee and D. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(02):131–177, 2005.

[5] D. E. Bell and L. L. Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[6] K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.

[7] T. chung Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell, July 2007.

[8] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15:249–291, March 2005.

[9] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[11] *Trusted Computer System Evaluation Criteria (Orange Book)*. Department of Defense, DoD 5200.28-STD edition, December 1985.

[12] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New York, NY, USA, 2011. ACM.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system, October 2005.

[14] M. Felleisen. The theory and practice of first-class prompts. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 180–190. ACM, 1988.

[15] J. Goguen and J. Meseguer. Security policies and security models, April 1982.

[16] W. L. Harrison. Achieving information flow security through precise control of effects. In *In 18th IEEE Computer Security Foundations Workshop*, pages 16–30. IEEE Computer Society, 2005.

[17] D. Hedin and D. Sands. Noninterference in the presence of non-opaque pointers. In *Proc. of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.

[18] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[19] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[20] S. Hunt and D. Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, POPL '06, pages 79–90. ACM, 2006.

[21] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions, October 2007.

[22] C. E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[23] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 144–154. ACM, 1993.

[24] X. Leroy and F. Rouaix. Security properties of typed applets. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 391–403. ACM, 1998.

[25] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[26] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[27] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10. ACM, 2010.

[28] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

[29] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

[30] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[31] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.

[32] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell, 2008.

[33] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[34] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, June 2009.

[35] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[36] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2004.

[37] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. on Computer Systems*, 25(4):11:1–43, December 2007. A version appeared in *Proc. of the 20th ACM Symp. on Operating System Principles*, 2005.

[38] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar, November 2006.

[40] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control, April 2008.

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]

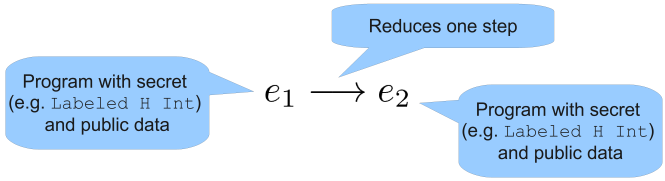
- Formalizes the non-interference guarantee provided by LIO
- For the proof, we consider a core and simple and functional language
  - Why not full Haskell?
  - λ-calculus extended with boolean values, pairs, recursion, monadic operations, references
- *We formally prove that the concept of monads works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

Proof Technique

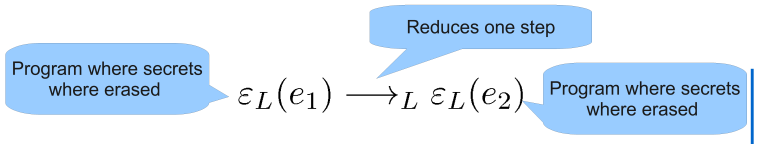
- Similar technique as the one used by Li and Zdancewic [Li, Zdancewic 10]
- Programs are expressions
- Main idea is simple:
  - If a program, that **involves secret and public information**, computes a public result, then the same public result can be obtained by a program that consists on the original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

Proof Technique

- More technically, we build a simulation between



and



The Language

The language

- The language and types

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

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*

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

return and >>=

- Every monad has two operations: return and bind

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

$\texttt{>>=} :: Ma \to (a \to Mb) \to Mb$

- 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

Expression to evaluate

$$e ::= \texttt{true} \mid \texttt{false} \mid (e,e) \mid \pi_i\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{el}$$
$$E ::= [\cdot] \mid \pi_i\ E \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \ldots$$
$$M = \texttt{if}\ \pi_1\ (\texttt{true}, \texttt{false})\ \texttt{then true else}\ (\texttt{false}, \texttt{true})$$
$$E_1 = \texttt{if}\ [\cdot]\ \texttt{then true else}\ (\texttt{false}, \texttt{true})$$
$$M = E_1[\pi_1\ (\texttt{true}, \texttt{false})]$$

Expressed in terms of evaluation contexts

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

Reduction step

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

Reduction rules

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

Operational Semantics for LIO

$$v ::= \ldots \mid \texttt{Lb}\ v\ e \mid \ldots$$
$$e ::= \ldots \mid \texttt{label}\ e\ e \mid \texttt{unlabel}\ e \mid \texttt{toLabeled}\ e\ e \mid \texttt{newRef}\ e\ e \mid \ldots$$
$$E ::= [\cdot] \mid \ldots$$

- 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

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

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

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

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

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

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

$$\text{(NREF)}\quad \frac{a\ \textit{fresh} \qquad \Sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \Sigma.\texttt{clr} \qquad \Sigma' = \Sigma.\phi[a \mapsto \texttt{Lb}\ l\ e]}{\langle \Sigma, E[\texttt{newRef}\ l\ e]\rangle \longrightarrow \langle \Sigma', E[\texttt{return}\ a]\rangle}$$

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

Assumptions about the type of free variables in e

Expression to type

Type

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

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

$$\Gamma \vdash x + x : ?$$

$$\Gamma = [x : Int] \longrightarrow \Gamma \vdash x + x : Int$$

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!

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

$$\Gamma \vdash ((\texttt{true}, 1), \texttt{true}) : ((\texttt{Bool}, \texttt{Int}), \texttt{Bool})$$

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

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

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

- 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

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

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

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

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:

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

- $l \not\sqsubseteq L$:

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

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

Case:

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

- $l \sqsubseteq L$:

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

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 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

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

no `toLabeled`

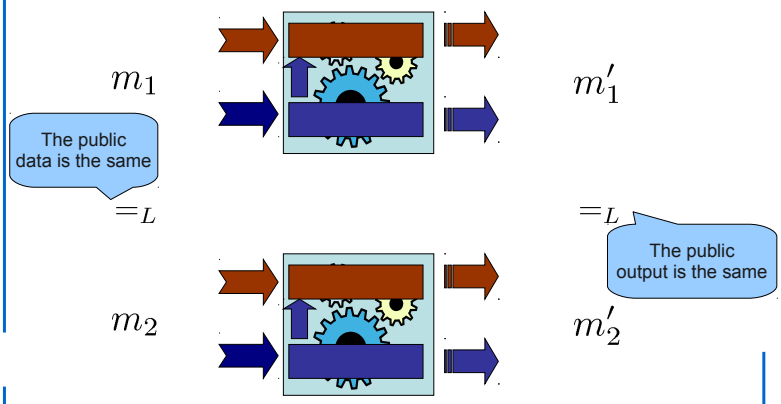
≤ k `toLabeled`

≤ 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

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

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

Non-interference

Theorem 1 (Non-interference). *Given a computation e (with no $\bullet$, $(\,)^{LIO}$, 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*

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

Non-interference (specialized)

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

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

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

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

- By our simulation, we know that

By the simulation

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

Proof Sketch III

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

Erase function goes inside the configuration

- We expand it

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

- A little bit more

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

Proof Sketch IV

These are the same configurations

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

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

By equality and definition of erasure function

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

- Which means,

By definition of erasure function

Remember what we assume in the begining

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

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

Proof Sketch V

- Then,

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

- 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

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

- Which implies that

$$\Sigma_1'.\texttt{lbl} = \Sigma_2'.\texttt{lbl} \wedge \Sigma_1'.\texttt{clr} = \Sigma_2'.\texttt{clr}$$

Proof Sketch VII

- So, having

$$\Sigma_1'.\phi \approx_L \Sigma_2'.\phi \qquad \texttt{Lb}\ l_1\ e_1'' \approx_L \texttt{Lb}\ l_2\ e_2''$$
$$\Sigma_1'.\texttt{lbl} = \Sigma_2'.\texttt{lbl} \quad \Sigma_1'.\texttt{clr} = \Sigma_2'.\texttt{clr}$$

- 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

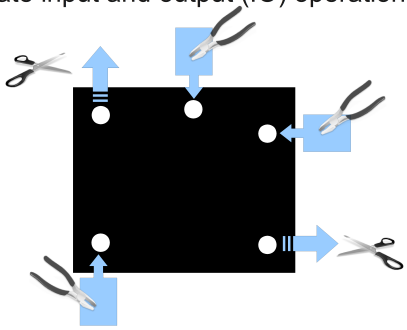
Enforcement for non-interference

- 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]

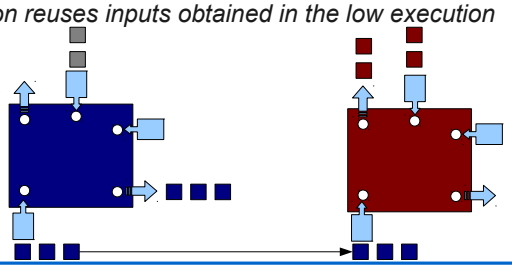
- Black box approach to enforce non-interference
  - No need to inspect the code
  - Manipulate input and output (IO) operations



Secure Multi-Execution

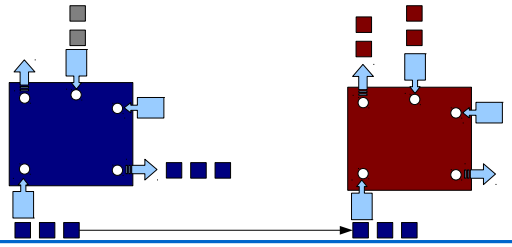
[Devriese, Piessens 10]

- 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?

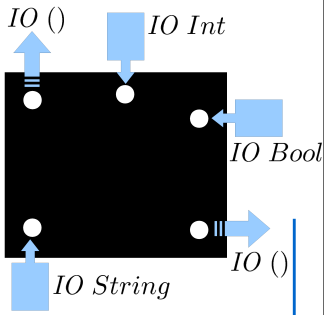
- Executed program satisfies non-interference
  - No explicit and implicit flows
- The secure multi-execution produces the same results
- Otherwise, the semantics changes to preserve security



Secure Multi-Execution in Haskell

[Jaskelioff, Russo 11]

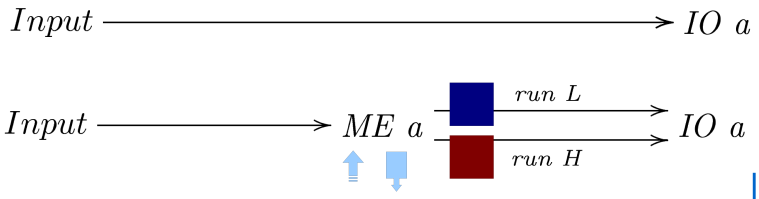
- Clear separation of pure computations with those with side-effects
- Every computation with side-effects is encapsulated into the monad *IO*
- Identify where IO is performed



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



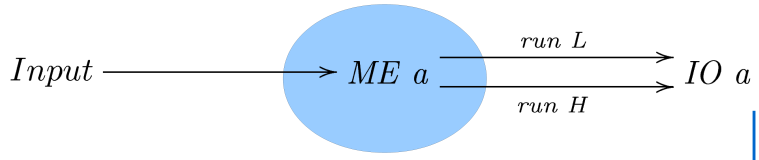
Monad ME

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

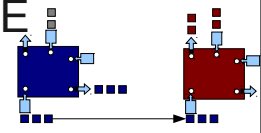
```
data ME a    =  Return a
             |  Write FilePath String (ME a)
             |  Read FilePath (String -> ME a)


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

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


Monad ME

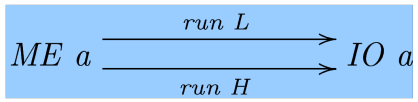
```
data ME a    =  Return a
             |  Write FilePath String (ME a)
             |  Read FilePath (String -> ME a)

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 $\mathit{discount}/\mathit{discount\ period}\ \mathbf{net}/\mathit{credit\ period}$
- 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
    - $\mathit{loan} - \mathit{loan} \times (1 - \mathit{discount}/100)$
    - $\$1000 - \$1000 \times (1 - .2) = \$20$
  - Cost of credit
    - $\dfrac{\mathit{discount}}{100 - \mathit{discount}} \times \dfrac{360}{\mathit{credit\ period} - \mathit{discount\ period}}$
    - $\dfrac{2}{98} \times \dfrac{360}{20} = .3673$

Example Scenario

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

Security Policy

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


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


Example: Code

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

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

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

Example: Malicious Code

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

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

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

Future Work

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

Final Remarks

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

Secure Multi-Execution in Haskell

Mauro Jaskelioff and Alejandro Russo

[1] CIFASIS-CONICET/Universidad Nacional de Rosario.
[2] Dept. of Computer Science and Engineering, Chalmers University of Technology

Abstract. Language-based information-flow security has emerged as a promising technology to guarantee confidentiality in on-line systems, where enforcement mechanisms are typically presented as run-time monitors, code transformations, or type-systems. Recently, an alternative technique, called *secure multi-execution*, has been proposed. The main idea behind this novel approach consists on running a program multiple times, once for each security level, using special rules for I/O operations. Compared to run-time monitors and type-systems, secure multi-execution does not require to inspect the full code of the application (only its I/O actions). In this paper, we propose the core of a library to provide *non-interference* through secure-multi execution. We present the code of the library as well as a running example for Haskell. To the best of our knowledge, this paper is the first work to consider secure-multi execution in a functional setting and provide this technology as a library.

1 Introduction

Over the past years, there has been a significant increase in the number of online activities. Users can do almost everything using a web browser. Even though web applications are probably among the most used pieces of software, they suffer from vulnerabilities that permit attackers to steal confidential data, break the integrity of systems, and affect the availability of services. Web-based vulnerabilities have already outplaced those of all other platforms [1] and there are no reasons to think that this situation is going to change [9].

In this work, we focus on preserving confidentiality of data through the security policy known as non-interference [3, 10] (i.e. not leaking secrets into public channels). Confidentiality policies are getting more and more relevant for widely open connected systems as the web, where compromised confidential data can be used to impersonate users in Facebook, Twitter, Flickr, and other social networks.

Language-based information-flow security [27] has developed approaches to analyze applications' code, leading to special-purpose languages, interpreters or compilers [19, 24] that guarantee security policies like non-interference. Rather than producing new languages from scratch, security can also be provided by libraries [14]. The potential of this approach has been shown across a range of programming languages and security policies [32, 25, 23, 4, 15, 6].

Traditionally, information-flow analysis on a program is done statically (e.g. using a type-system), dynamically (e.g. using an execution monitor), or with a combination of both. Recently, authors in [7] devised an alternative approach, called *secure multi-execution*, based on the idea of executing the same program several times, once for

each security level. As opposed to previous enforcement mechanisms, this novel approach does not demand to design type-systems or deploy heavy-weight monitoring of programs; it only requires modifying the semantics of I/O operations.

In this paper, we present the main ideas of a library based on monads [17, 16] to provide *non-interference* through secure-multi execution. The ideas can be easily applied to any pure language and are illustrated with an implementation for the programming language Haskell. To the best of our knowledge, this paper is the first one to consider secure-multi execution as library in a pure functional setting.

2 Secure multi-execution

Devriese and Piessens [7] propose the novel approach of secure multi-execution to enforce non-interference. We organize security levels in a security lattice $\mathcal{L}$, where security levels are ordered by a partial order $\sqsubseteq$, with the intention to only allow leaks from data at level ℓ_1 to data at level ℓ_2 when $\ell_1 \sqsubseteq \ell_2$. Secure multi-execution runs a program multiple times, once for each security level. In order to enforce security, the I/O operations of those multiple copies of the program are interpreted differently. Outputs on a given channel at security level ℓ is performed only in the execution of the program linked to that security level. Inputs coming from a channel at security level ℓ are replaced by a default value if the execution of the program is linked to a security level ℓ_e such that $\ell \not\sqsubseteq \ell_e$. In that manner, the execution of the program linked to level ℓ_e never obtains information higher than its security level. In the case that $\ell_e = \ell$, the input operation is performed normally. Finally, if $\ell \sqsubseteq \ell_e$, the execution of the program linked to level ℓ_e reuses the inputs obtained by the execution linked to level ℓ.

Devriese and Piessens show that secure multi-execution is *sound* and *precise*. Soundness states that each execution linked to a given level cannot get any information from higher levels and consequently, all of its output will have to be generated from information at its level or below, guaranteeing non-interference. Precision establishes that if a program satisfies non-interference under normal execution, then its behavior is the same as the one obtained by secure multi-execution on terminating runs.

3 Secure multi-execution in Haskell

In most pure functional programming languages, computations with side-effects such as inputs and outputs can be distinguished by its type. For instance, in Haskell every computation performing side-effects must be encoded as a value of the monad (or abstract data type) IO [22]. Specifically, a value of type $IO\ \ a$ is an action (i.e., a computation which may have side-effects) which produces a value of type a when executed. This manner in which monads identify computations with side-effects fits particularly well with the idea of secure multi-execution of giving different interpretations to I/O operations as specified by the execution level.

For simplicity, we consider a two-point security lattice with elements L and H, where $L \sqsubseteq H$ and $H \not\sqsubseteq L$. Levels L and H represent public and secret confidentiality levels, respectively. The implementation shown here, however, works for an arbitrary

finite security lattice. In Fig. 1 we show the implementation of the lattice as elements of the datatype $Level$ and define the order relationship $\sqsubseteq$ and the non-reflexive $\sqsubset$.

$$\begin{aligned}
&\mathbf{data}\ Level = L \mid H\ \mathbf{deriving}\ (Eq, Enum)\\
&\cdot \sqsubseteq \cdot, \cdot \sqsubset \cdot :: Level \to Level \to Bool\\
&H \sqsubseteq L = False\\
&_ \sqsubseteq\ _ = True\\
&p \sqsubset q\ = p \sqsubseteq q \wedge p \not\equiv q
\end{aligned}$$

Fig. 1. Security lattice

We propose a library that works by replacing I/O actions (i.e., values of the IO monad) by a pure description of them [31]. Haskell programs which perform some I/O actions have type $a \to IO\ b$. That is, given some argument of type a, the program performs some I/O actions and then returns a value of type b as the result. In secure multi-execution, the I/O actions performed by such program must be interpreted differently depending on the security level linked to a given execution (see Section 2). Hence, programs to be run under secure multi-execution do not return I/O actions, but rather a pure description of them. With this in mind, secure programs have the type $a \to ME\ b$, where monad ME describes the side-effects produced during the computation. When the program is executed, those I/O descriptions are interpreted according to the specification of secure multi-execution. For security levels L and H, the program is run twice, where the I/O actions are interpreted differently on the execution linked at level L, and on the one linked at level H. Figure 2 summarizes the ideas behind our library. Function run executes and links the program to the security level given as argument. Observe that function run is also responsible for the interpretation of the I/O actions described in the monad ME.

For simplicity, we only consider reading and writing files as the possible I/O actions. It is easy to generalize our approach to consider other I/O operations. Function $level :: FilePath \to Level$ assigns security levels to files indicating the confidentiality of their contents. We assume that, when a file is read, its access time gets updated as a side-effect of the operation. An attacker, or public observer, is able to learn the content of public files as well as their access time.

a) $a \longrightarrow IO\ b$

b) $a \longrightarrow ME\ b \xrightarrow[run\ H]{run\ L} IO\ b$

c)
$$\begin{aligned}
&\mathbf{data}\ ME\ a = Return\ a\\
&\quad \mid Write\ FilePath\ String\ (ME\ a)\\
&\quad \mid Read\ FilePath\ (String \to ME\ a)
\end{aligned}$$

Fig. 2. Type for a typical program with side-effects (a) and a secure multi-execution program (b), and definition of ME (c).

Monad ME describes the I/O actions performed by programs and is defined in Fig. 2(c). Constructors $Return$, $Write$, and $Read$ model programs performing different actions. Program $Return\ x$ simply returns value x without performing any I/O operations. Program $Write\ file\ x\ p$ models a program that writes string x into file *file* and then behaves as program p. Program $Read\ file\ g$ models a program that reads the contents x of file *file* and then behaves as program $g\ x$. Technically, ME is an intermediate monad that provides a pure model of the reading and writing of files in the IO monad.

Users of the library do not write programs using the constructors of ME directly. Instead, they use the interface provided by the monad: $return :: a \to ME\ a$ and

$(\gg\!=) :: ME\ a \to (a \to ME\ b) \to ME\ b$. The $return$ function lifts a pure value into the ME monad. The operator $\gg\!=$, called *bind*, is used to sequence computations. A bind expression $(m \gg\!= f)$ takes a computation m and function f which will be applied to the *value* produced by m and yields the resulting computation. These are the only primitive operations for monad ME, and consequently, programmers must sequence individual computations explicitly using the bind operator. Fig. 3 shows the implementation of $return$ and $\gg\!=$. The expression $return\ x$ builds a trivial computation, i.e., a computation which does not perform any $Write$/$Read$ actions and just returns x. Values in ME are introduced with $Return$, so this is the only case where f is applied. In the other two cases, the $Write$/$Read$ operations are preserved and bind with f $(\gg\!= f)$ is recursively applied. Besides $return$ and $(\gg\!=)$, the monad ME has operations to denote I/O actions on files. These operations model the equivalent operations on the IO monad and are given by the following functions.

$$
\begin{array}{l}
\textbf{instance}\ Monad\ ME\ \textbf{where} \\
\quad return\ x \qquad\qquad\ = Return\ x \\
\quad (Return\ x) \quad \gg\!= f \quad = f\ x \\
\quad (Write\ file\ s\ p) \gg\!= f = Write\ file\ s\ (p \gg\!= f) \\
\quad (Read\ file\ g) \gg\!= f \quad = Read\ file\ (\lambda i \to g\ i \gg\!= f)
\end{array}
$$

Fig. 3. Definitions for $return$ and $\gg\!=$

$$
\begin{array}{l}
writeFile :: FilePath \to String \to ME\ () \\
writeFile\ file\ s = Write\ file\ s\ (return\ ())
\end{array}
\qquad
\begin{array}{l}
readFile :: FilePath \to ME\ String \\
readFile\ file = Read\ file\ return
\end{array}
$$

4 An interpreter for the monad ME

$$
\begin{array}{l}
run :: Level \to ChanMatrix \to ME\ a \to IO\ a \\
run\ l\ \_\ (Return\ a) = return\ a \\
run\ l\ c\ (Write\ file\ o\ t) \\
\quad \mid level\ file \equiv l \quad = \textbf{do}\ IO.writeFile\ file\ o \\
\qquad\qquad\qquad\qquad\ \ run\ l\ c\ t \\
\quad \mid otherwise \qquad = run\ l\ c\ t \\
run\ l\ c\ (Read\ file\ f) \\
\quad \mid level\ file \equiv l \quad = \textbf{do}\ x \leftarrow IO.readFile\ file \\
\qquad\qquad\qquad\qquad\ \ broadcast\ c\ l\ file\ x \\
\qquad\qquad\qquad\qquad\ \ run\ l\ c\ (f\ x) \\
\quad \mid level\ file \sqsubset l \quad = \textbf{do}\ x \leftarrow reuseInput\ c\ l\ file \\
\qquad\qquad\qquad\qquad\ \ run\ l\ c\ (f\ x) \\
\quad \mid otherwise \qquad = run\ l\ c\ (f\ (defvalue\ file))
\end{array}
$$

Fig. 4. Interpreter for monad ME

Fig. 4 shows the interpreter for programs of type $ME\ a$. Intuitively, $run\ l\ c\ p$ executes p and links the execution to security level l. Argument c is used when inputs from executions linked to lower levels need to be reused (explained below). The implementation is pleasantly close to the informal description of secure multi-execution in Section 2. Outputs are only performed ($IO.writeFile\ file\ o$) when the confidentiality level of the output file is the same as the security level linked to the execution ($level\ file \equiv l$). Inputs are obtained ($IO.readFile\ file$) when files' confidentiality level is the same as the security level linked to the execution ($level\ file \equiv l$). Data from those inputs is broadcasted to executions linked to higher security levels in order to be properly reused when needed ($broadcast\ c\ l\ file\ x$). If the current execution level is higher than the file's confiden-

tiality level ($level\ file \sqsubset l$)), the content of the file is obtained from the execution linked to the same security level as the file ($reuseInput\ \ c\ \ l\ \ file$). Otherwise, the input data is replaced by a default value. Function $defvalue :: FilePath \rightarrow String$ sets default values for different files. Unlike [7], and to avoid introducing runtime errors, we adopt a default value for each file (i.e., input point) in the program. Observe that inputs can be used differently inside programs. For instance, the contents of some files could be parsed as numbers, while others as plain strings. Therefore, choosing a constant default value, e.g. the empty string, could trigger runtime errors when trying to parse a number out of it.

An execution linked to security level ℓ reuses inputs obtained in executions linked to lower levels. Hence, we implement communication channels between executions, from a security level ℓ' to a security level ℓ, if $\ell' \sqsubset \ell$. In the interpreter, the argument of type $ChanMatrix$ consists of a matrix of communication channels indexed by security levels. An element $c_{\ell',\ell}$ of the matrix denotes a communication channel from security level ℓ' to ℓ where $\ell' \sqsubset \ell$; otherwise $c_{\ell',\ell}$ is undefined. In this manner, execution linked at level ℓ' can send its inputs to the execution linked at level ℓ, where $\ell' \sqsubset \ell$. Messages transmitted on these channels have type $(FilePath, String)$, i.e., pairs of a filename and its contents. Function $broadcast\ \ c\ \ l\ \ file\ \ x$ broadcasts the pair $(file, x)$ on the channels linked to executions at higher security levels, i.e., channels $c_{l,\ell}$ such that $l \sqsubset \ell$. Function $reuseInput\ \ c\ \ l\ \ file$ matches the filename $file$ as the first component of the pairs in channel $c_{level\ file,l}$ and returns the second component, i.e., the contents of the file.

```
sme :: ME a → IO ()
sme t = do
  c ← newChanMatrix
  l ← newEmptyMVar
  h ← newEmptyMVar
  forkIO (do run L c t; putMVar l ())
  forkIO (do run H c t; putMVar h ())
  takeMVar l; takeMVar h
```

Fig. 5. Secure multi-execution

Multithreaded secure multi-execution is orchestrated by the function sme. This function is responsible for creating communication channels to implement the reuse of inputs, creating synchronization variables to wait for the different threads to finish, and, for each security level, forking a new thread that runs the interpreter at that level. Fig. 5 shows a specialized version of sme for the two-point security lattice. However, in the library implementation, the function sme works for an arbitrary finite lattice. Function $newChanMatrix$ creates the communication channels. Synchronization variables are just simple empty $MVar$s. When a thread tries to read ($takeMVar$) from an empty $MVar$ it will block until another thread writes to it ($putMVar$) [21]. Function $forkIO$ spawns threads that respectively execute the interpreter run at levels L and H, and then signal termination by writing ($putMVar\ \ l\ \ (); putMVar\ \ h\ \ ()$) to the thread's synchronization variable. The main thread locks on these variables by trying to read from them ($takeMVar\ \ l; takeMVar\ \ h$).

Unlike [7], function sme does not require the scheduler to keep the execution at level L ahead of the execution at level H. In [7], this requirement helps to avoid timing leaks at the price of probably modifying the runtime system (i.e, the scheduler). As mainstream information-flow compilers, monad ME also ignores timing leaks.

```
data CreditTerms = CT {discount :: Rational, ddays :: Rational, net :: Rational}
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)
                  -- writeFile "Client-Statistics" (show ccost ++ loanStr)
```

Fig. 6. Financial calculator

5 A motivating example

We present a small example of how to build programs using monad ME. We consider the scenario of a financial company who wants to preserve the confidentiality of their clients but, at the same time, compute statistics by hiring a third-party consultant company. Given certain loan, the company wants to write code to compute the *cost of credit* [5] and the total amount of interest that it will receive as income. When taking a loan, credit terms usually indicate a due date as well as a cash discount if the credit is canceled before an expiration date. We consider credit terms of the form "*discount* / *discount period* net / *credit period*", which indicates that if payment is made within *discount period* days, a *discount* percent cash discount is allowed. Otherwise, the entire amount is due in *credit period* days. Given a credit term, the amount of money paid when the credit is due is $loan - loan \times (1 - discount/100)$. The yearly cost of credit, i.e., the cost of borrowing money under certain terms is $\frac{discount}{100-discount} \times \frac{360}{credit\ period - discount\ period}$. For instance, in an invoice of \$1000 with terms 2 /10 net 30, the total interest payed at the due date is $\$1000 - \$1000 \times (1 - .2) =$ \$20, and the cost of credit becomes $\frac{2}{98} \times \frac{360}{20} = .3673$, i.e., 37%.

In this setting, we consider the amount of every loan to be confidential (secret) information, while cost of credit is public and thus available for statistics. By writing our program using monad ME, we can be certain that confidential information is never given for statistics. In other words, the third-party consultant company does not learn anything about the amount in the loans provided by the financial company. Figure 6 shows one possible implementation of the program to compute interests and cost of credit. Files `"Client"` and `"Client-Interest"` are considered secret (level H), while `"Client-Terms"` and `"Client-Statistics"` are considered public (level L). The code is self-explanatory.

If a programmer writes, by mistake or malice, $show\ ccost +\!\!+ loanStr$ as the information to be written into the public file (see commented line), then secure multi-execution avoids leaking the sensitive information in $loanStr$ by given the empty string to the execution linked to security level L.

6 Related work

Previous work addresses non-interference and functional languages [11, 33, 24, 29]. The seminal work by Li and Zdancewic [14] shows that information-flow security can also be provided as a library for real programming languages. Morgenstern et al. [18] encode a programming language aware of authorization and information-flow policies in Agda. Devriese and Piessens [8] enforce non-interference, either dynamically or statically, using monad transformers in Haskell. Different from that work, the monad ME does not encode static checks in the Haskell's type-system or monitor every step of programs' executions. Moreover, Devriese and Piessens' work requires to encode programs as values of a certain data type, while our approach only models I/O operations.

Russo et al. [26] outline the ground idea for secure multi-execution as a naive transformation. A transformed program runs twice: one execution computes the public results, where secret inputs were removed, and the second execution computes the secret outputs of the program. Devriese and Piessens [7] propose secure multi-execution as a novel approach to enforce non-interference. Devriese and Piessens implement secure multi-execution for the Spider-monkey JavaScript engine. The implementation presented in this work is clean and short (approximately 130 lines of code), and thus making it easy to understand how multi-execution works concretely. Unlike [7], our approach does not consider termination and timing covert channels. We argue that dealing with termination and timing covert channels in a complex language, without being too conservative, is a difficult task. In this light, it is not surprising that the main information-flow compilers (Jif [20] –based on Java–, and FlowCaml [30] –based on Caml–) ignore those channels.

Close to the notion of secure multi-execution, Jif/split [34] automatically partitions a program to run securely on heterogenously trusted hosts. Different from secure multi-execution, the partition of the code is done to guarantee that if host h is subverted, hosts trusting h are the only ones being compromised. Swift [2] uses Jif/split technology to partition the program into JavaScript code running on the browser, and JavaScript code running on the web server.

7 Concluding remarks

We propose a monad and an interpreter for secure multi-execution. To the best of our knowledge, we are the first ones to describe secure multi-execution in a functional language. We implement our core ideas in a small Haskell library of around 130 lines of code and present a running example. The implementation is compact and clear, which makes it easy to understand how secure multi-execution works concretely. Broadcasting input values to executions at higher levels is a novelty of our implementation if compared with Devriese and Piessens' work. This design decision is not tied to the Haskell implementation, and the idea can be used to implement the reuse of inputs in any secure multi-execution approach for any given language. The library is publicly available [13].

Future work Our long-term goal is to provide a fully-fledged library for secure multi-execution in Haskell. The IO monad can perform a wide range of input and output

operations. It is then interesting to design a mechanism capable to lift, as automatically as possible, IO operations into the monad ME [12]. Another direction for future work is related with declassification, or deliberate release of confidential information [28]. Declassification in secure multi-execution is still an open challenge. Due to the structure of monadic programs, we believe that it is possible to identify, and restrict, possible synchronization points where declassification might occur. Then, declassification cannot happen arbitrarily inside programs but only on those places where we can give some guarantees about the security of programs. To evaluate the capabilities of our library, we plan to use it to implement a medium-size web application. Web applications are good candidates for case studies due to their demand on confidentiality as well as frequent input and output operations (i.e. server requests and responses). It is also our intention to perform benchmarks to determine the overhead introduced by our library. The library seems to multiply execution time by the number of levels, but since file operations are only done once, the reality could be better if the broadcast mechanism is not expensive.

Acknowledgments Alejandro Russo was partially funded by the Swedish research agency VR. We would like to thank Arnar Birgisson, Andrei Sabelfeld, Dante Zanarini and the anonymous reviewers for their helpful comments.

References

[1] M. Andrews. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.

[2] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, Oct. 2007.

[3] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[4] J. J. Conti and A. Russo. A taint mode for Python via a library. *NordSec 2010. Selected paper by OWASP AppSec Research 2010*, 2010.

[5] Credit Research Foundation. Ratios and formulas in customer financial analysis. `http://www.crfonline.org/orc/cro/cro-16.html`, 1999.

[6] F. Del Tedesco, A. Russo, and D. Sands. Implementing erasure policies using taint analysis. In T. Aura, editor, *The 15th Nordic Conf. in Secure IT Systems*. Springer Verlag, 2010.

[7] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[8] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '11, pages 59–72, New York, NY, USA, 2011. ACM.

[9] Federal Aviation Administration (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems. `http://www.oig.dot.gov/sites/dot/files/pdfdocs/ATC_Web_Report.pdf`, June 2009. Note: thousands of vulnerabilities were discovered.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[11] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[12] M. Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

[13] M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. software release. `http://www.cse.chalmers.se/~russo/sme/`, 2011.

[14] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[15] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In T. Aura, editor, *The 15th Nordic Conf. in Secure IT Systems*. Springer Verlag, 2010.

[16] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Edinburgh, Scotland, 1989.

[17] E. Moggi. Computational lambda-calculus and monads. In *Proc., Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society, 1989.

[18] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN Int. Conf. on Funct. Prog.*, ICFP '10, pages 169–180, New York, NY, USA, 2010. ACM.

[19] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[20] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[21] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL '96: Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, New York, NY, USA, 1996. ACM.

[22] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. of the ACM Conf. on Principles of Programming*, pages 71–84, 1993.

[23] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In R. Safavi-Naini and V. Varadharajan, editors, *ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)*, Sydney, Australia, March 2009. ACM Press.

[24] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[25] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell'08: Proc. of the 1st ACM SIGPLAN Symp. on Haskell*. ACM, 2008.

[26] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Comp. Science Conf. (ASIAN'06)*, LNCS. Springer-Verlag, 2007.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[28] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[29] V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proc. of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[30] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.

[31] W. Swierstra and T. Altenkirch. Beauty in the beast. In *Proc. of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 25–36, New York, NY, USA, 2007. ACM.

[32] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007.

[33] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, July 2002.

[34] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. ACM Symp. on Operating System Principles*, 2001.