Information flow Libraries

- privacy concerns in software systems
- code must not leak sensitive data to the internet

↳ Restrict data propogation (Information flow)

Haskell Library: Mandatory Access Control (MAC)

What is IFC?

- Specify how information may propogate in the system:

  "Sensitive inputs may not flow"
- Track data flows
- Detect data leaks (and suppress)

Building IFC is hard

We need DSLs, custom oracles (operating systems, browsers, etc.), compilers, ...

This is easier in Haskell!

- Haskell pure abstractions can directly express IFC analysis (no side-effects!)
- No need for custom compilers just an API for Haskell.

Several Approaches

- MAC : Static
- HLIO : Hybrid
- LIO : Dynamic

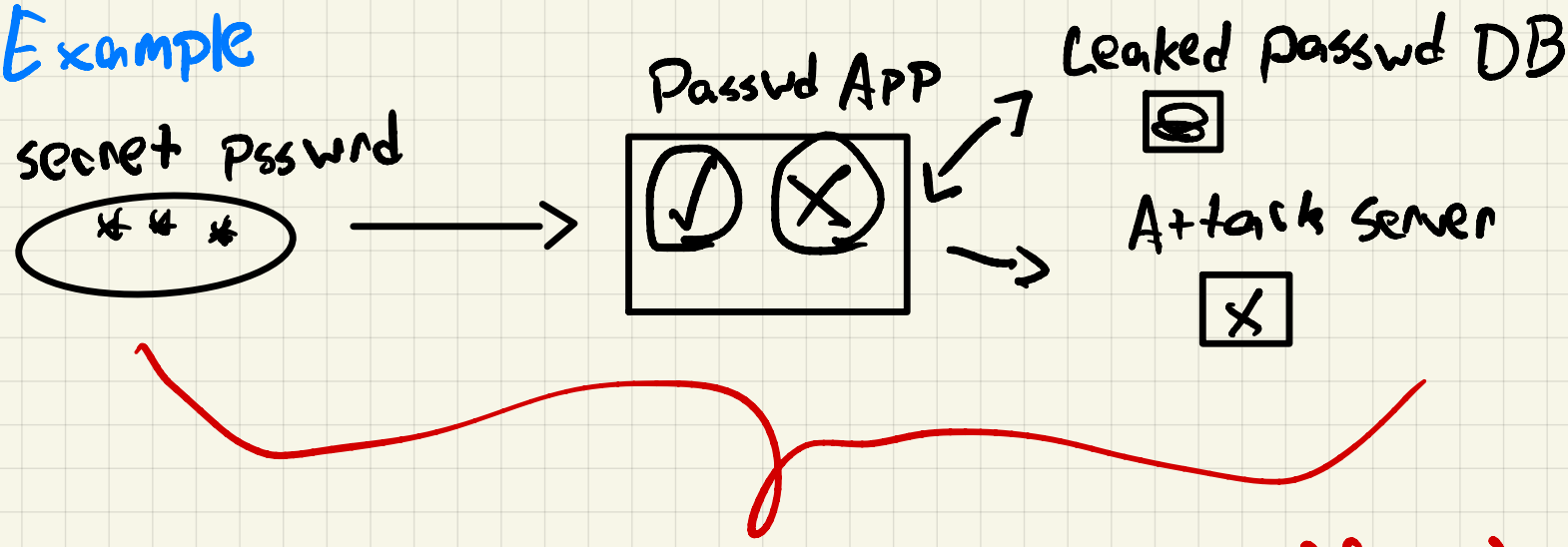
Challenge: Write a MAC program that leaks but still obeys MAC properties

Haskell's type system restricts where I/O is allowed

These libraries prevent leaks by:

- Untrusted code may perform I/O only through IFC library. There is an IO monad we can only do IO when wrapped in.
- Wrapped with security types.

Example



isWeakPwd :: String -> Bool (No side effects)

No queries!

isWeakPwd :: String -> IO Bool (Can leak!)

- Public observable outputs can leak password
- IO might be used incorrectly

Sol. IFC libs disallow public outputs when data is in scope.

MAC

- Multi-parameter type classes: Reuse type system
- Safe Haskell: untrusted code can't cheat the ts
- Small ~200 LOC

How to specify security policies?

- Security labels data L and 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
```

- ⊑ : flow is reflexive and transitive (antisymmetric)

Label data

- Define ADT for labeled data

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

- These types explicitly assign labels to data

```
password :: Labeled H String
```

- ADT for secure computations

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

- Encapsulate IO actions that don't leak
- Handle data at security level L

```
readPwdFile :: MAC H String
```

- Only trusted code can run secure computations

How do we not leak?

- Follows MAC principles:
  1) No read-up: IO actions may not read data at higher levels
  2) No write-down: IO actions may not write data at lower levels

We have safety from the module system in which for:

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

- You cannot call unlabel when it's unsafe and the password is safe behind a wrapper.

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