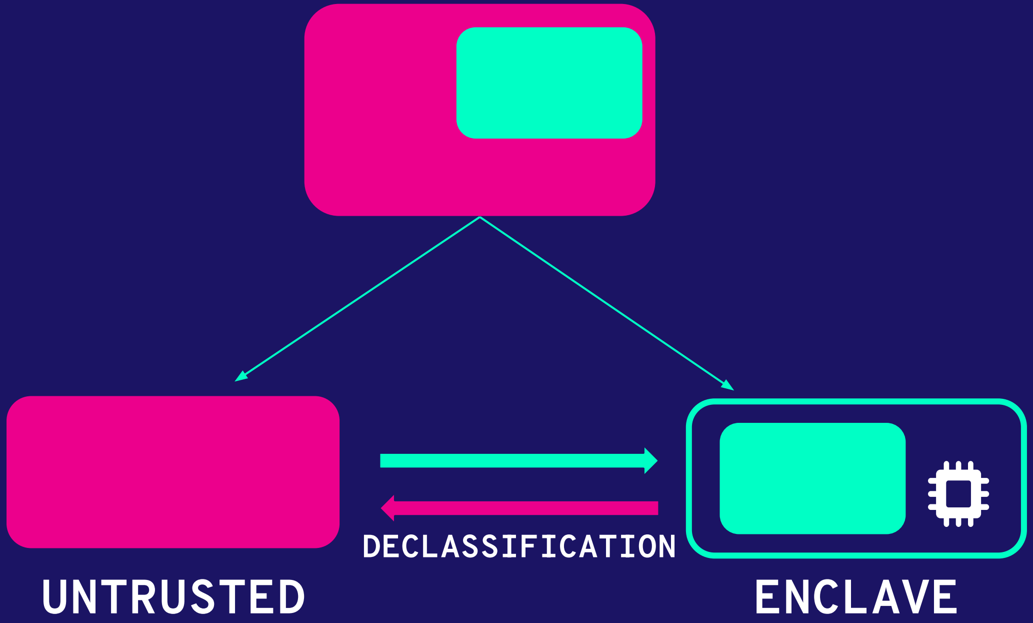
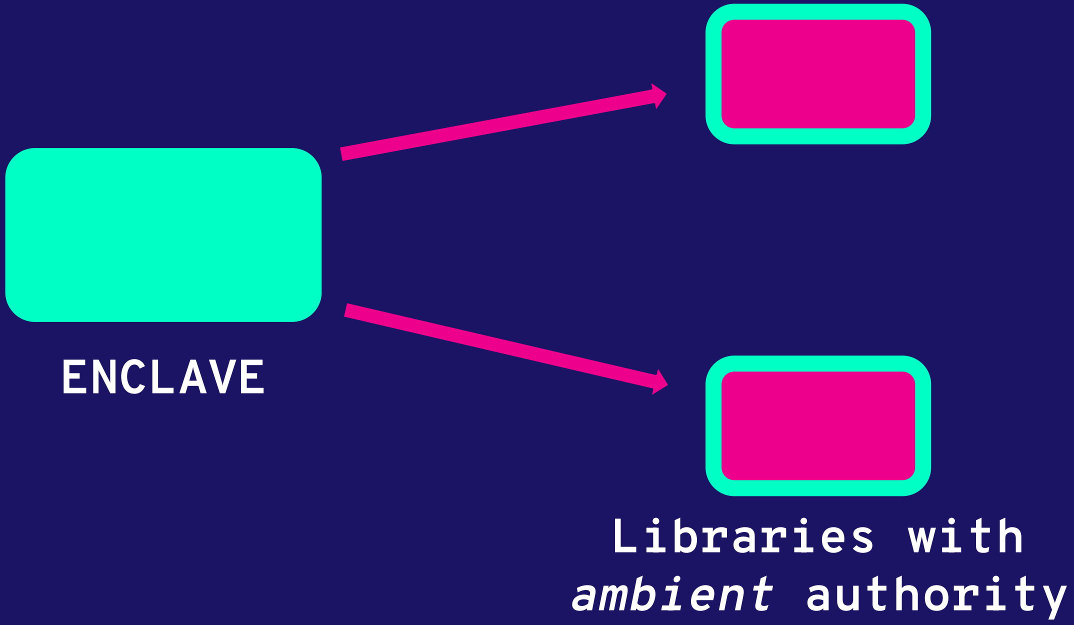
Confidential Computing with Haskell

Abhiroop Sarkar
Chalmers University, Gothenburg

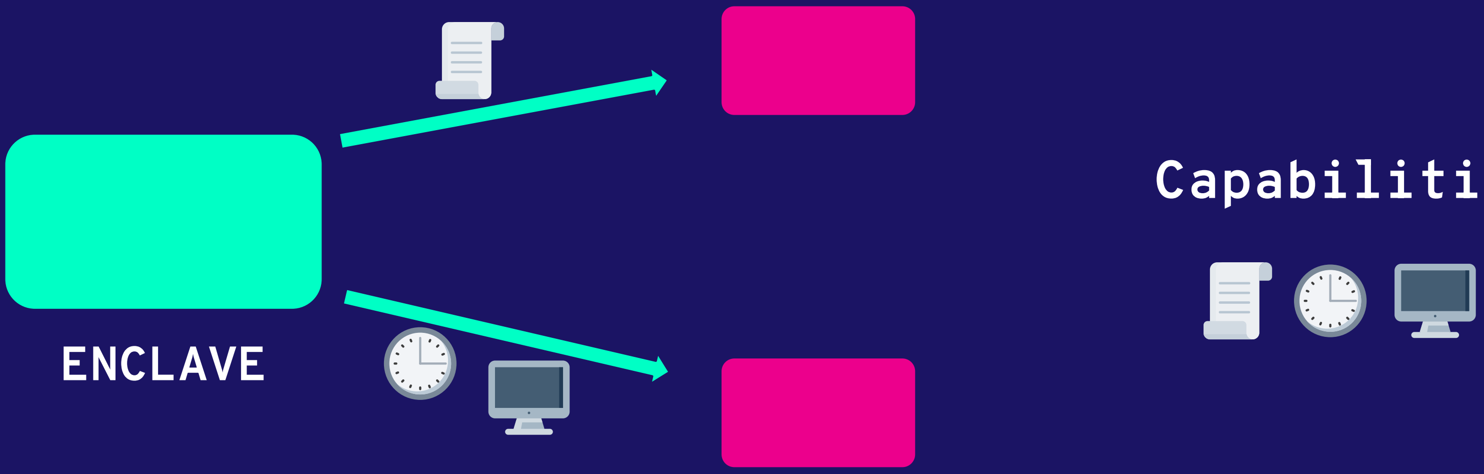
KEY IDEA 1



KEY IDEA 2



KEY IDEA 2



ENCLAVE

Capabilities

Libraries with
ambient authority

HasTEE

Haskell on Trusted Execution Environments

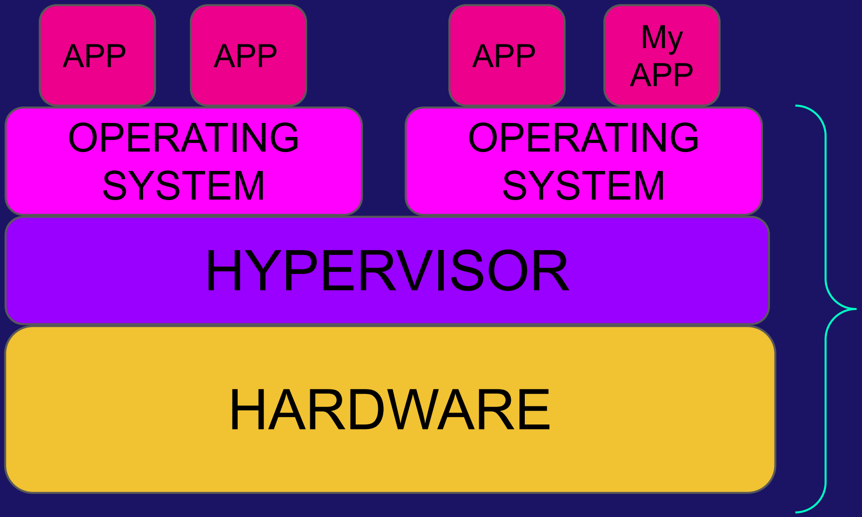
Authors

Abhiroop Sarkar

Robert Krook

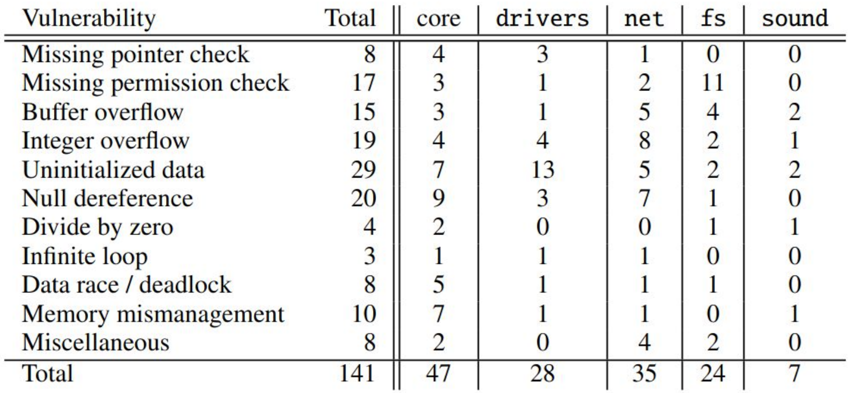
Koen Claessen

Cloud Deployments



Trusted Computing Base

OS Vulnerabilities



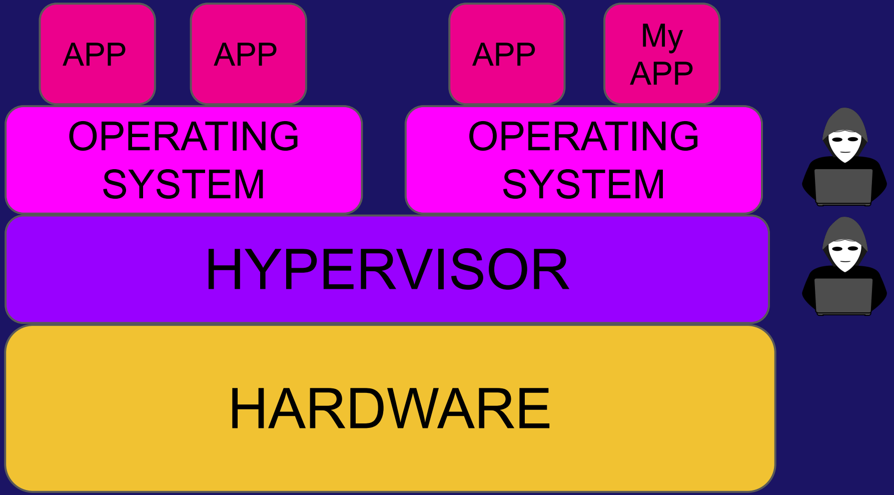
Vulnerability	Total	core	drivers	net	fs	sound
Missing pointer check	8	4	3	1	0	0
Missing permission check	17	3	1	2	11	0
Buffer overflow	15	3	1	5	4	2
Integer overflow	19	4	4	8	2	1
Uninitialized data	29	7	13	5	2	2
Null dereference	20	9	3	7	1	0
Divide by zero	4	2	0	0	1	1
Infinite loop	3	1	1	1	0	0
Data race / deadlock	8	5	1	1	1	0
Memory mismanagement	10	7	1	1	0	1
Miscellaneous	8	2	0	4	2	0
Total	141	47	28	35	24	7

Figure 2: Vulnerabilities (rows) vs. locations (columns).

Linux kernel vulnerabilities: State-of-the-art defenses and open problems. Mao et al. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (pp. 1-5).

Characterizing hypervisor vulnerabilities in cloud computing servers. Perez-Botero et al. In *Proceedings of the 2013 international workshop on Security in cloud computing*.

Cloud Deployments



APP

APP

APP

My APP

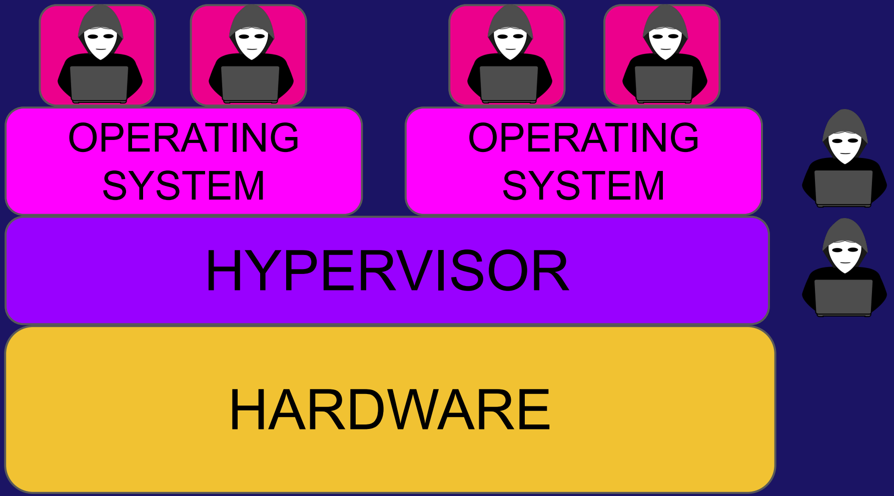
OPERATING SYSTEM

OPERATING SYSTEM

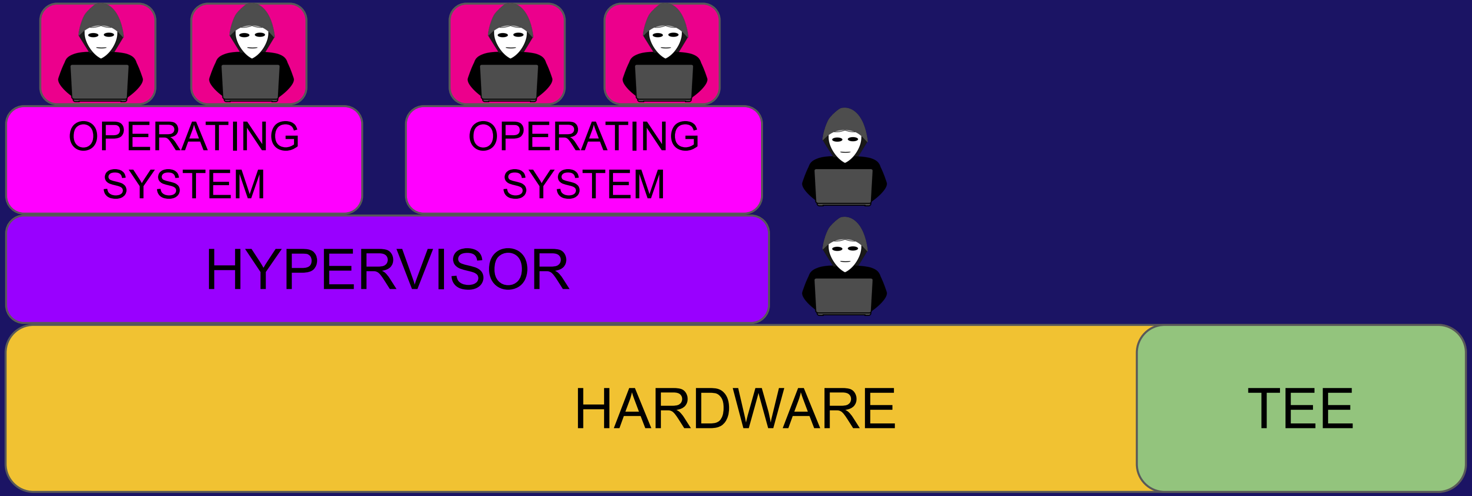
HYPERVISOR

HARDWARE

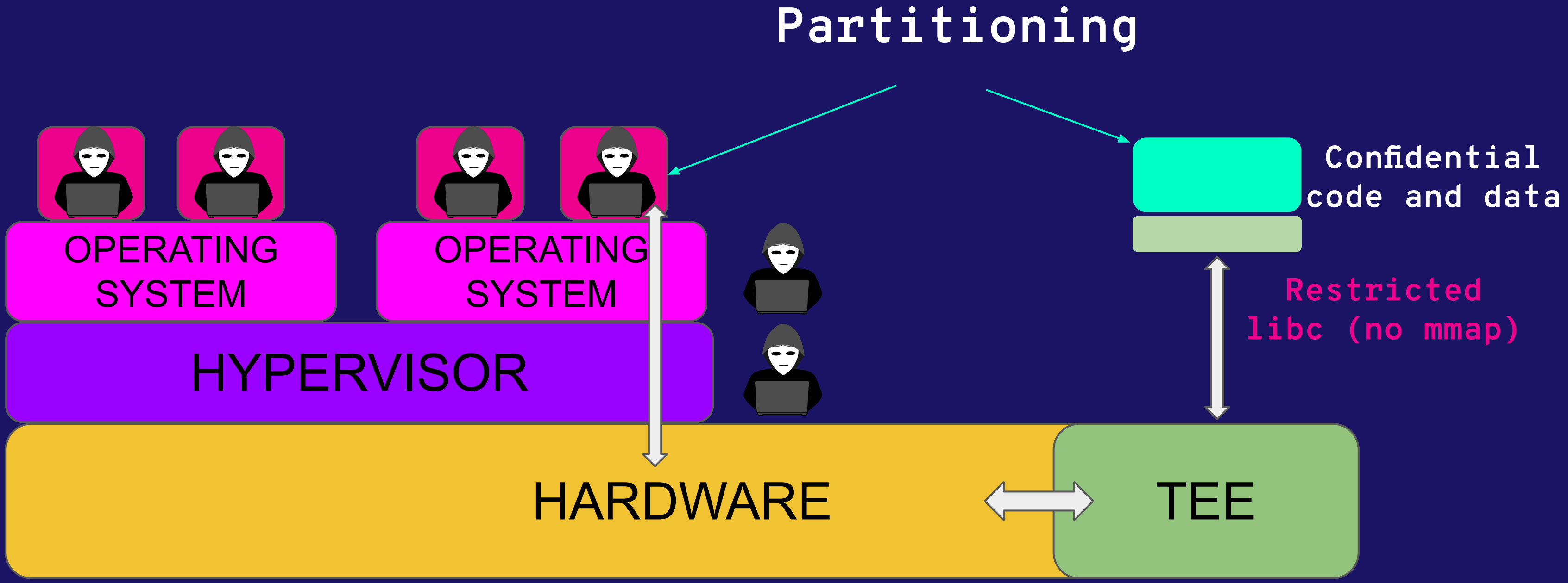
Cloud Deployments



Trusted Execution Environment (TEE)



Trusted Execution Environment (TEE)



Trusted Execution Environment (TEE)

intel SGX

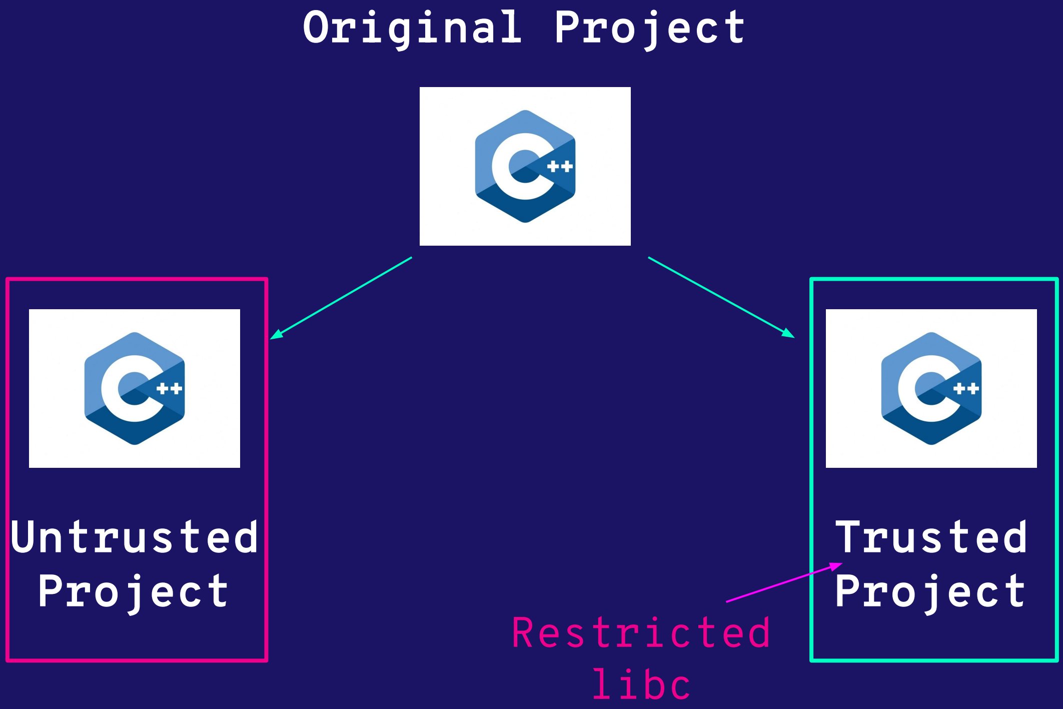
arm TRUSTZONE

AMD SEV-SNP

RISC-V

Physical Memory Protection

Programming TEEs



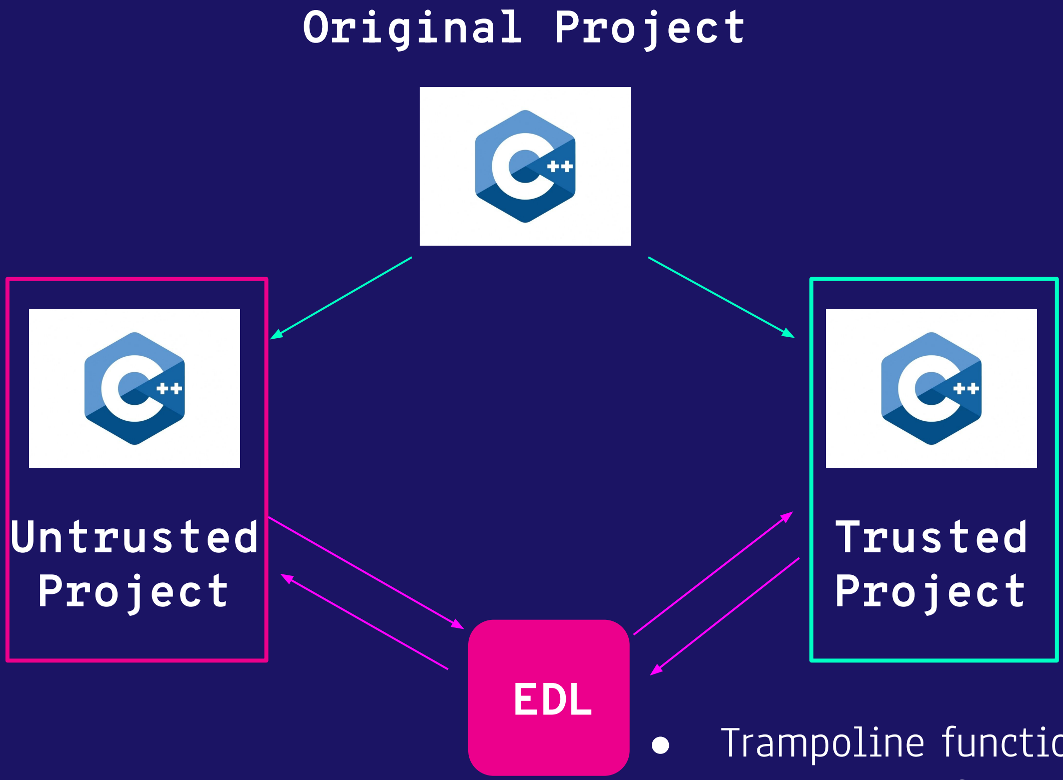
Original Project

Untrusted Project

Trusted Project

Restricted libc

Programming TEEs



Original Project

Untrusted Project

Trusted Project

EDL

- Trampoline functions
- Arcane Makefiles

HasTEE

- Type-driven Partitioning of a single program
- Program in a high-level language - Haskell
- Enforce Information Flow Control on data within enclaves

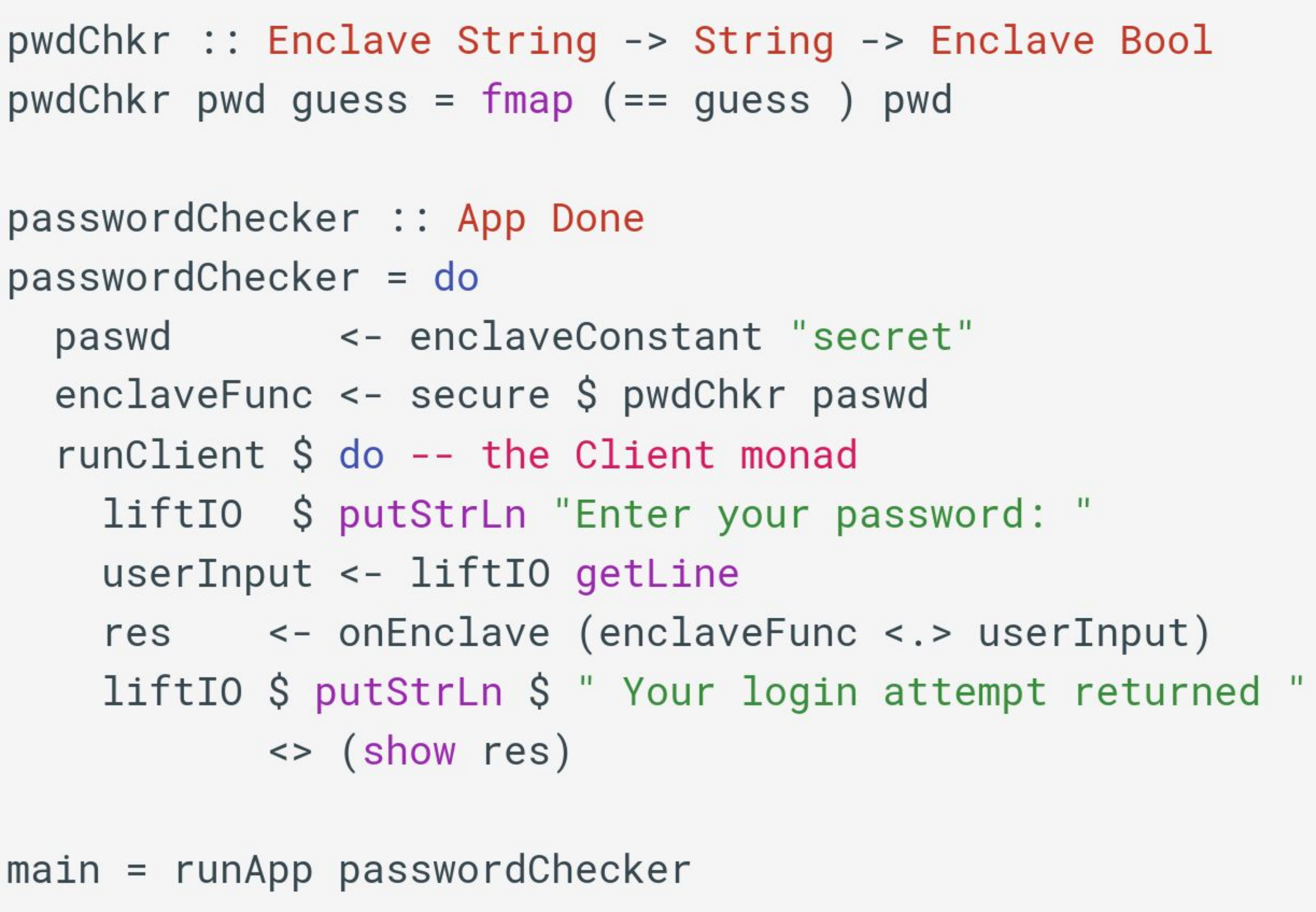
Haskell

```
add :: Int → Int → Int

add_with_IO :: Int → Int → IO Int
```

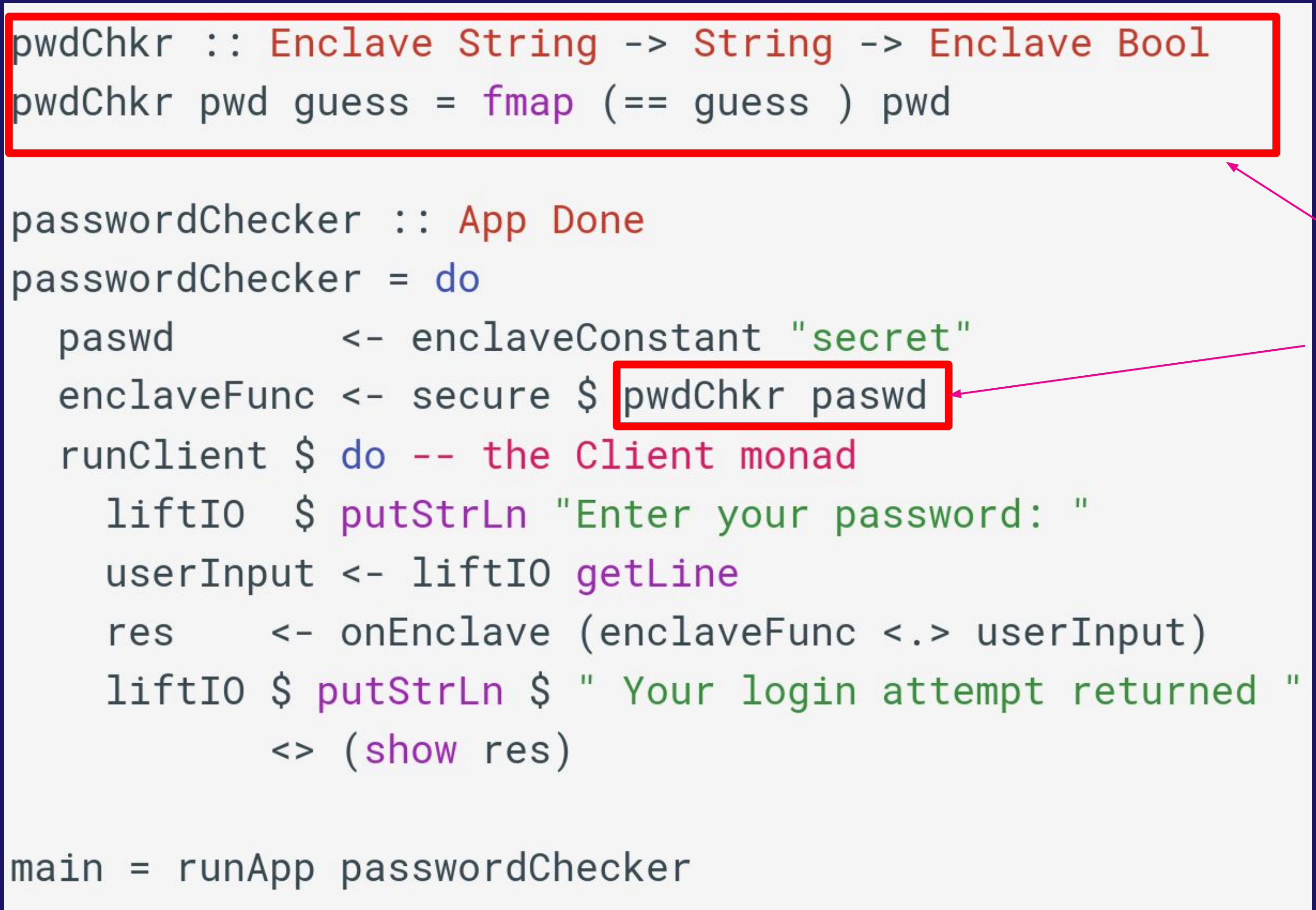
Monad

```
add_with_IO :: Int → Int → IO Int
add_with_IO x y = do
    name ← read “Enter your name”
    putStrLn (“Hello” ++ name)
    putStrLn (“Result = ” ++ (show (x + y)))
```

Illustration : Password Checker

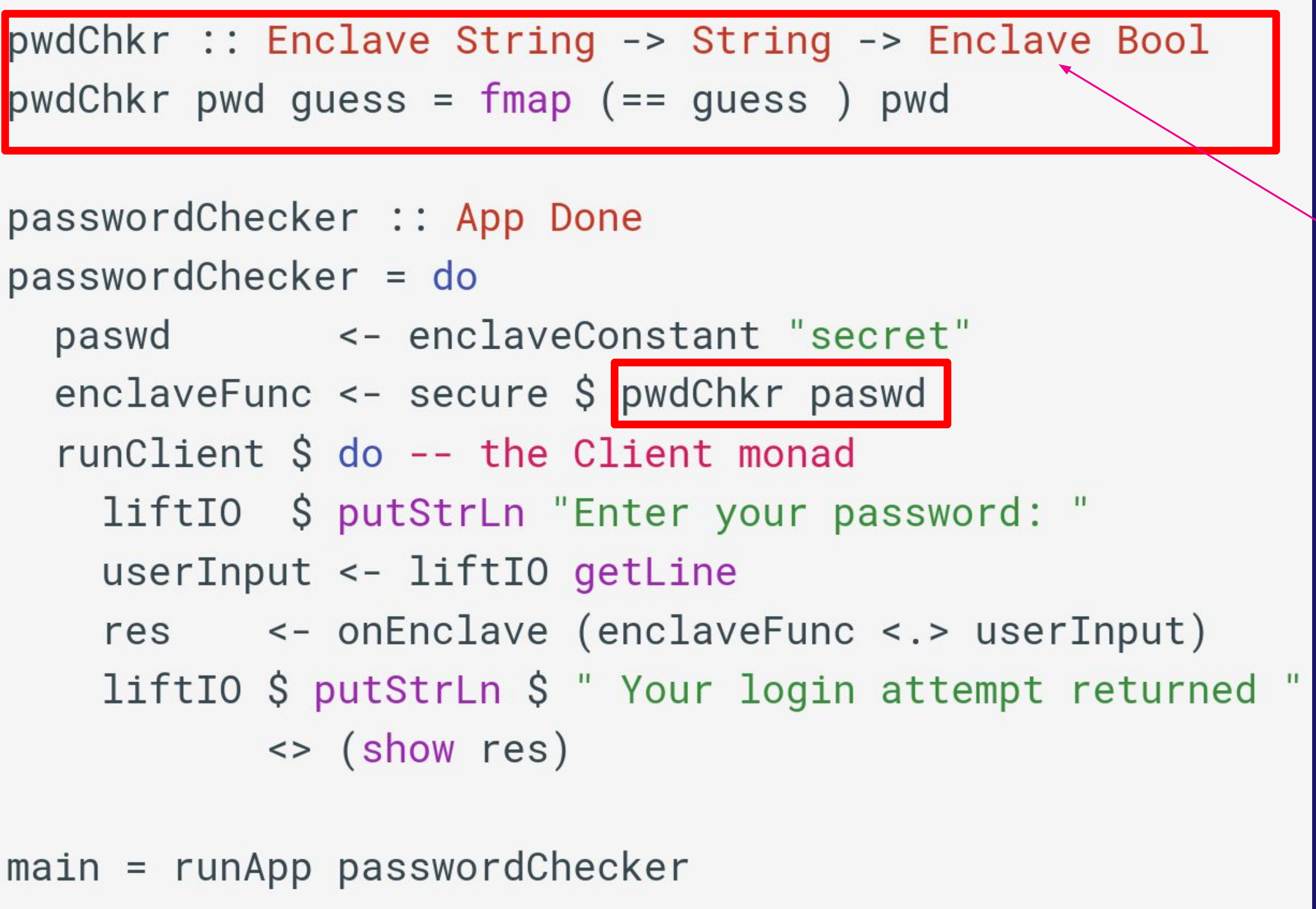


```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```



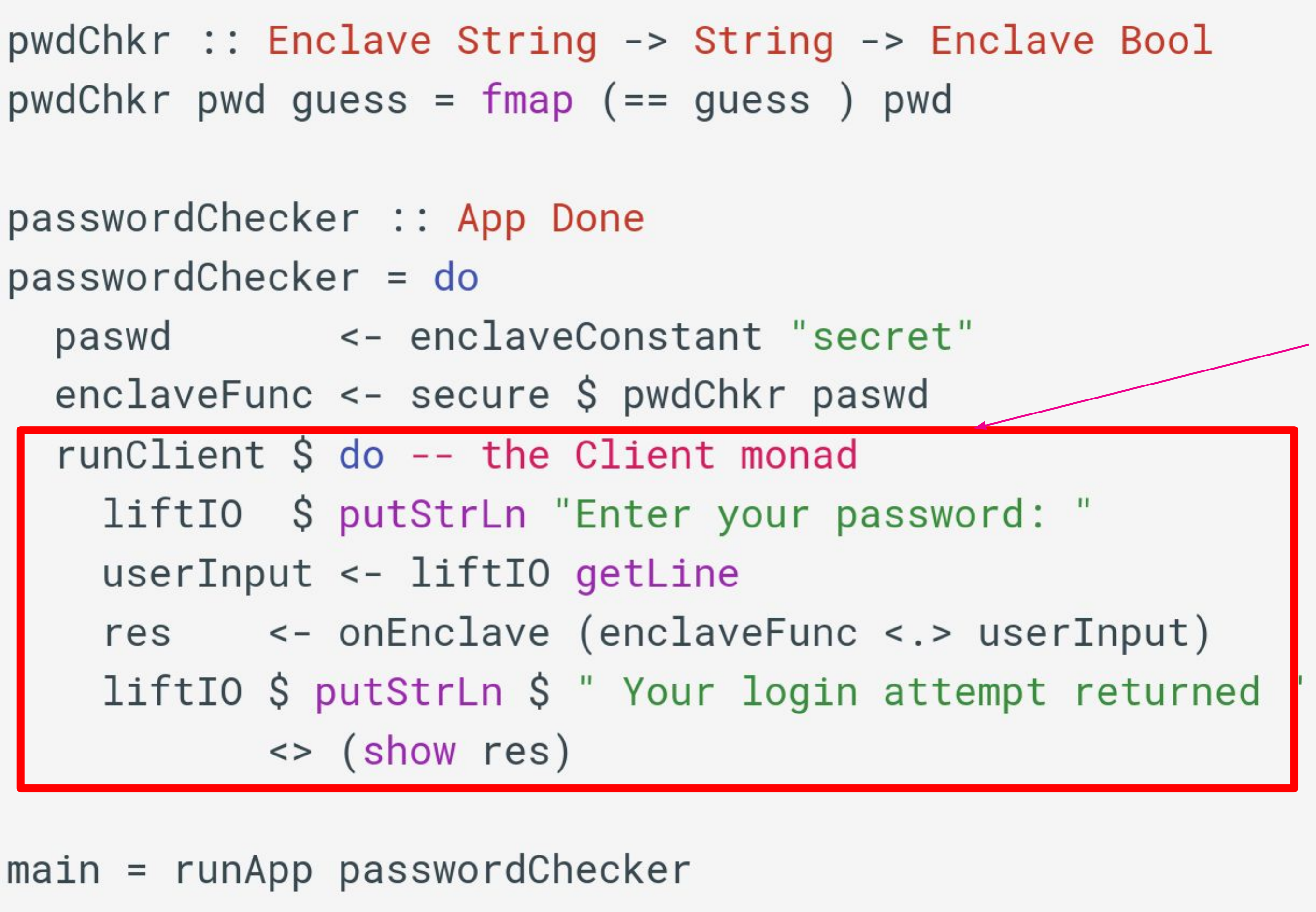
```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

The secure code and data



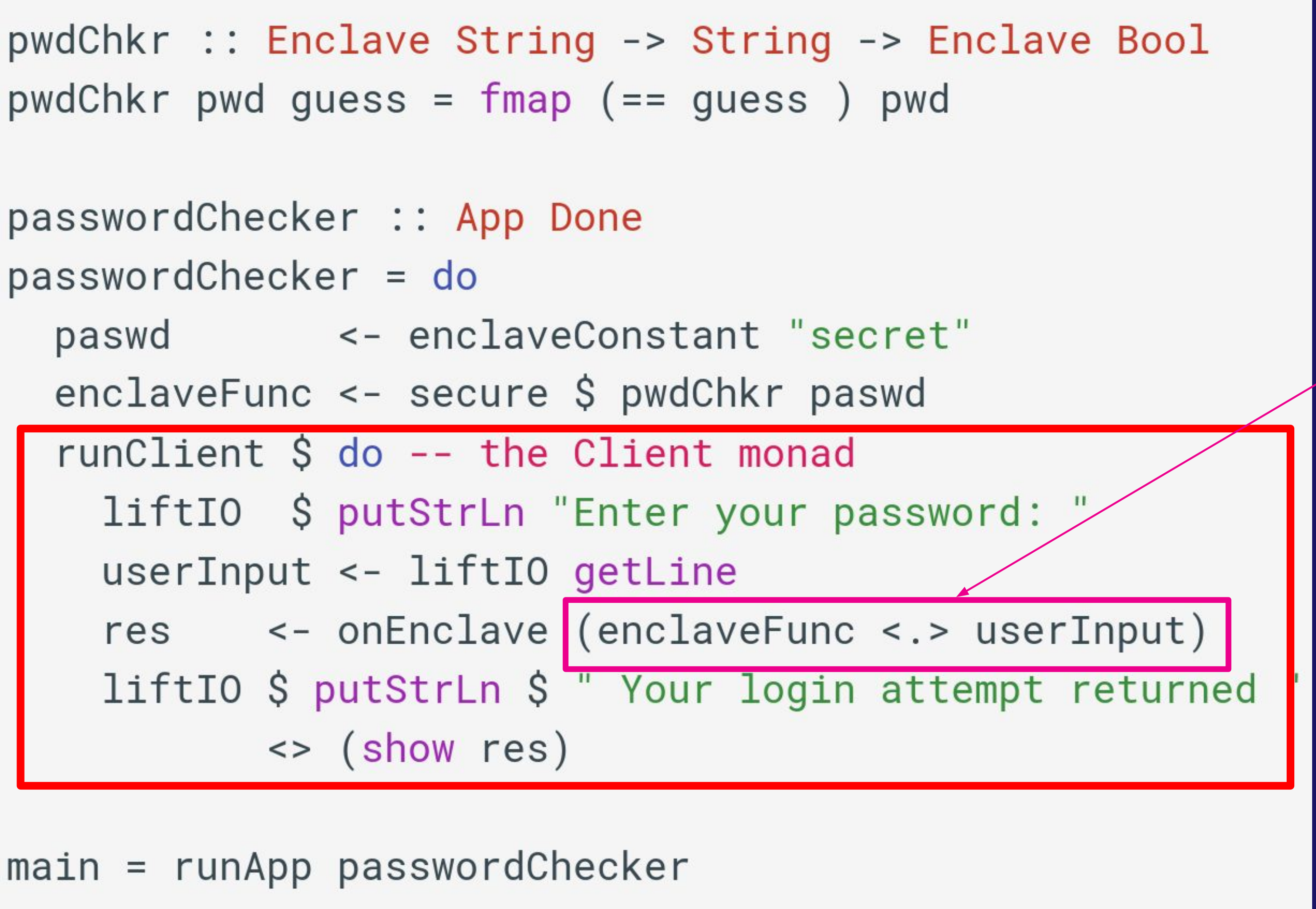
```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res     <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

The Enclave monad



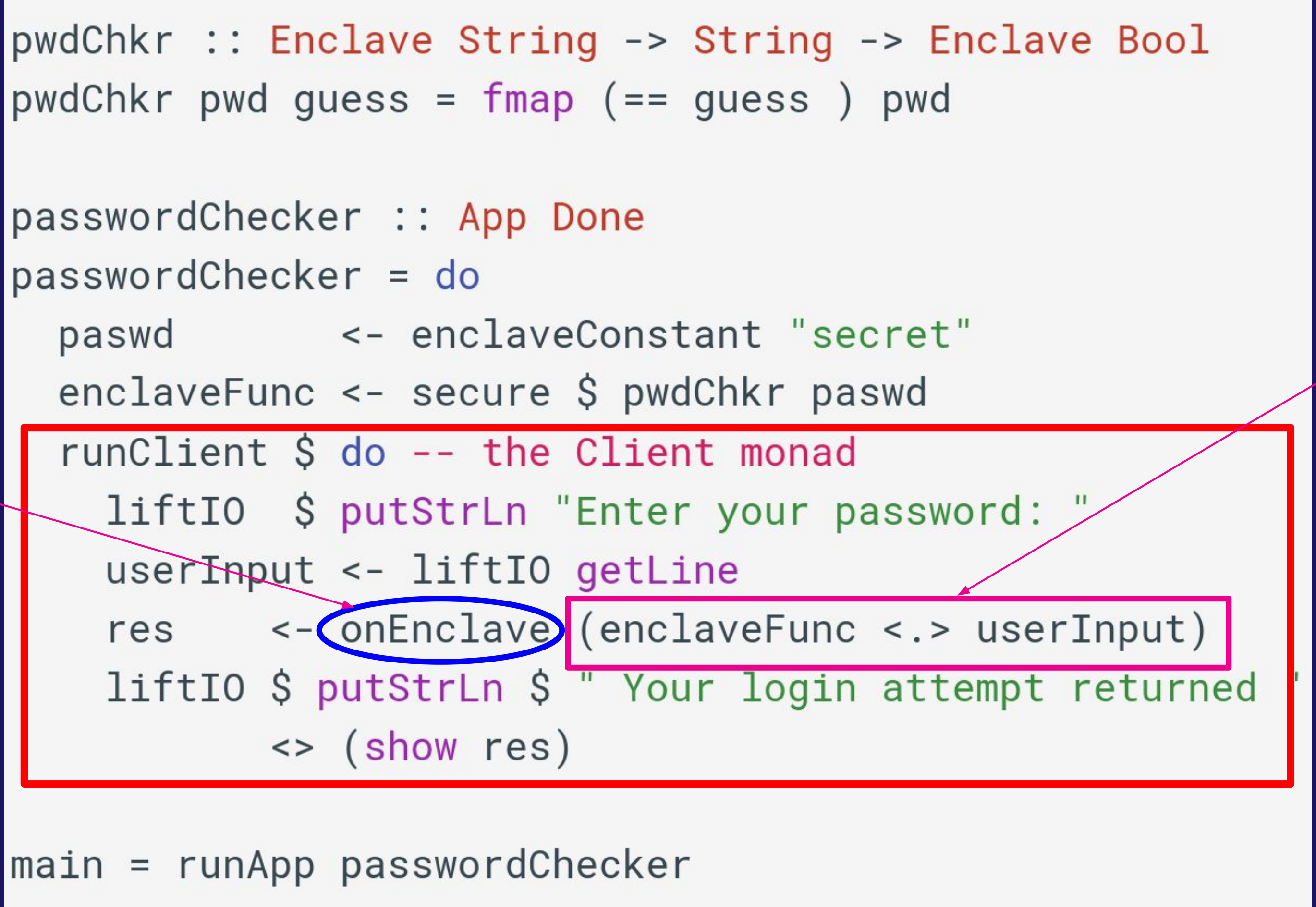
```
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

The untrusted part



```
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

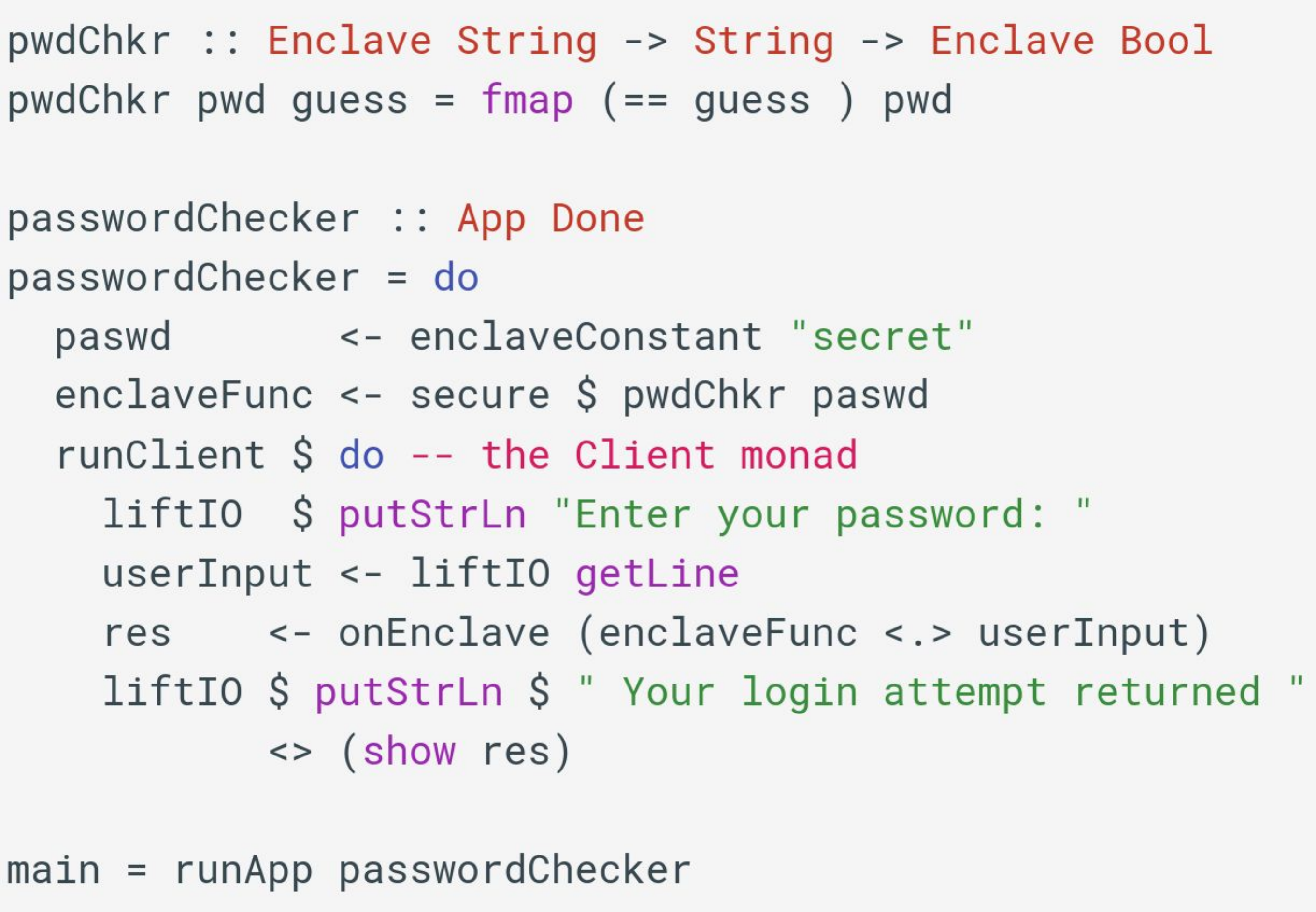
Enclave Function Application



```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

Mimics a remote procedure call

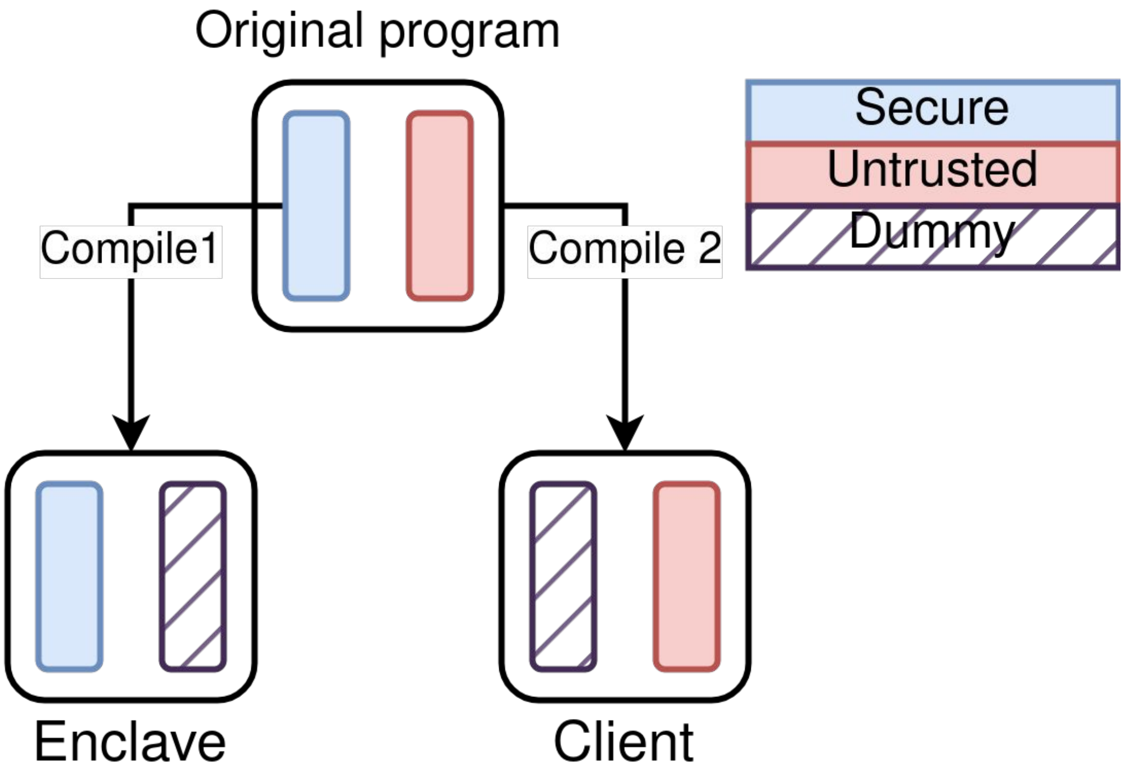
Enclave Function Application

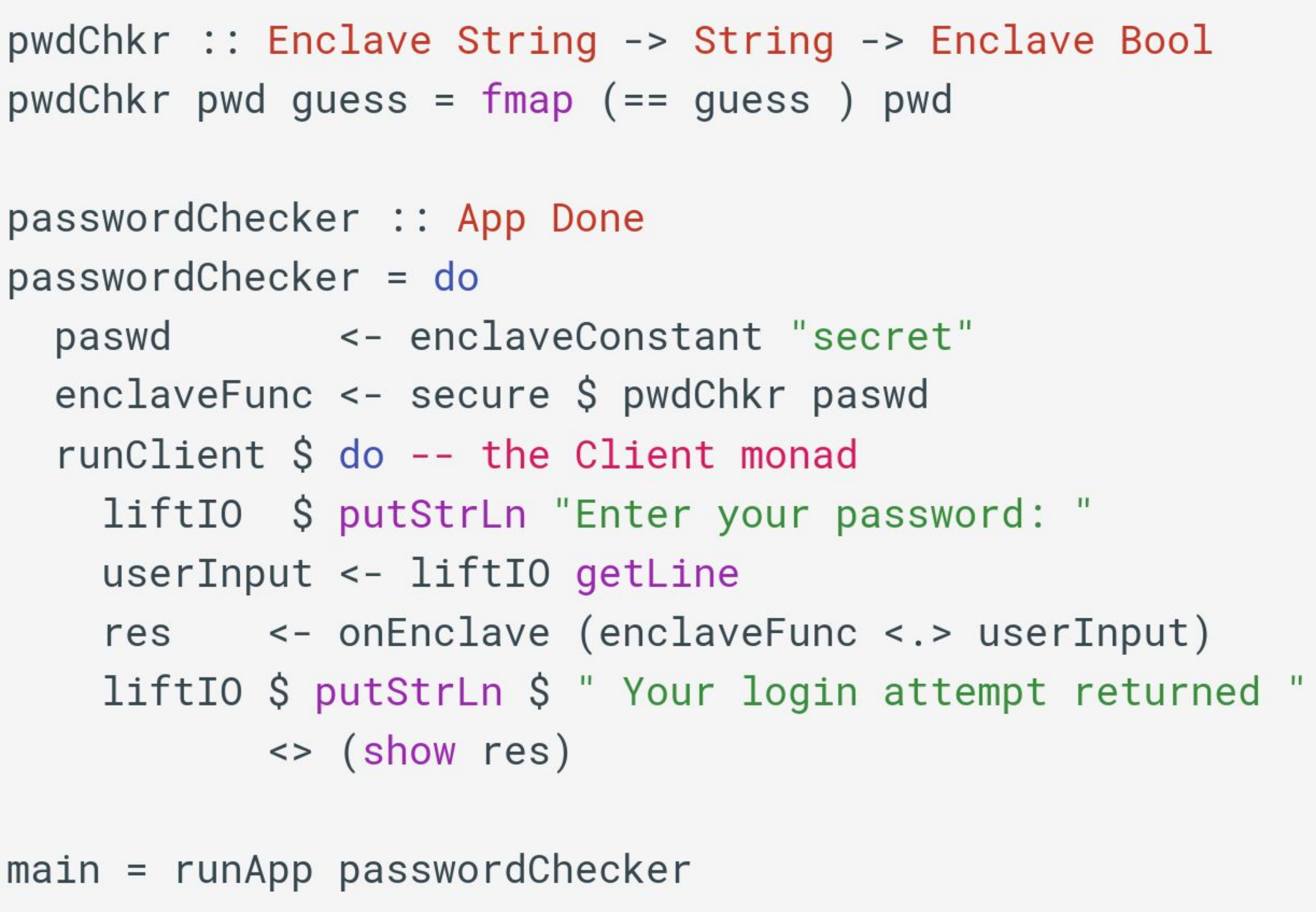


```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

```
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

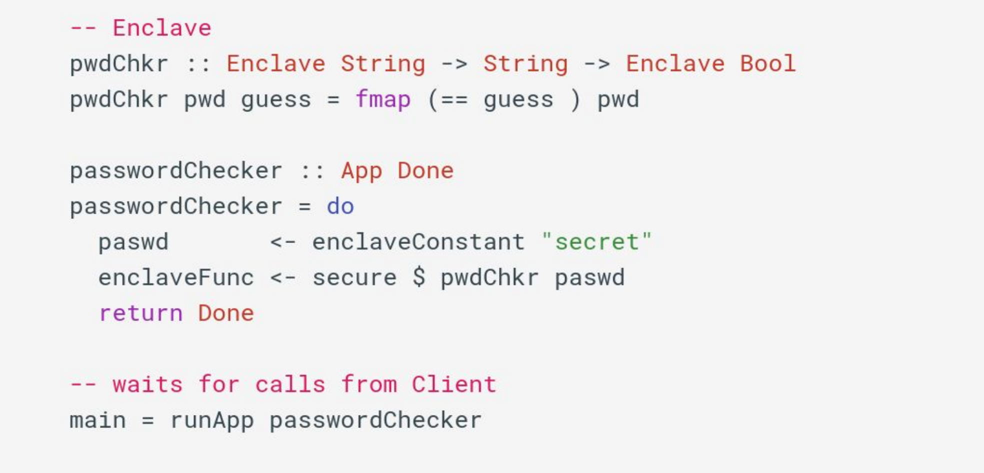
COMPILED TWICE


Original program
Secure
Untrusted
Dummy
Compile1
Compile 2
Enclave
Client



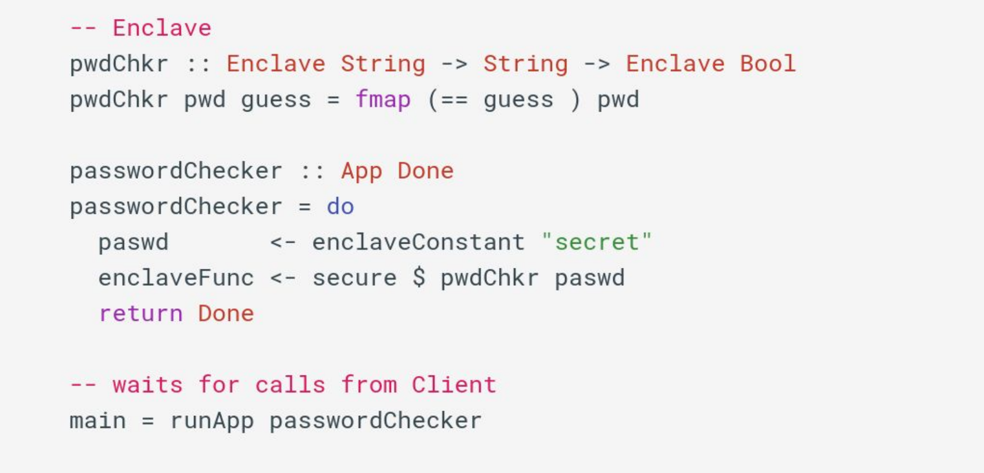
```
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

main = runApp passwordChecker
```

Compilation 1



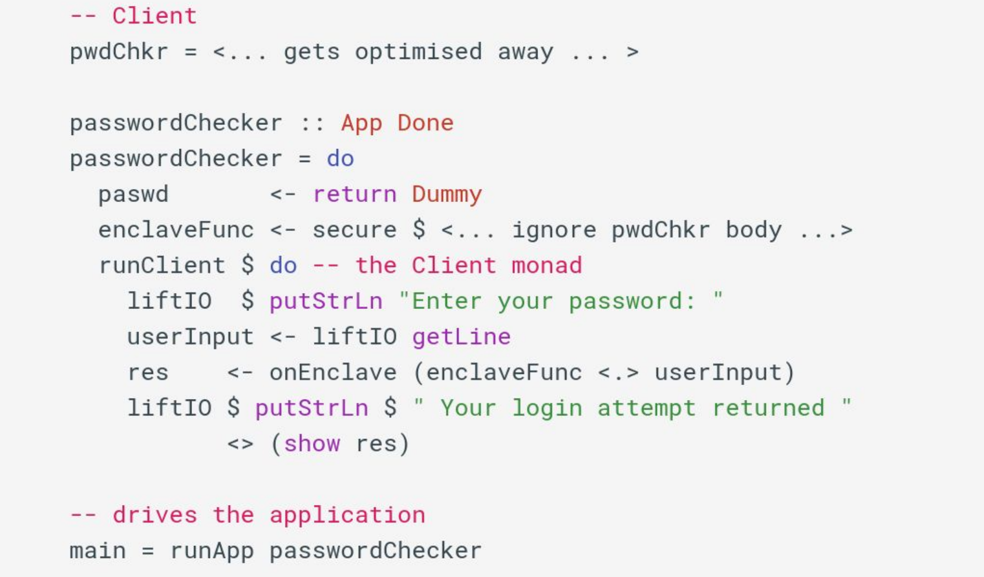
```
-- Enclave
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  return Done

-- waits for calls from Client
main = runApp passwordChecker
```

Compilation 1

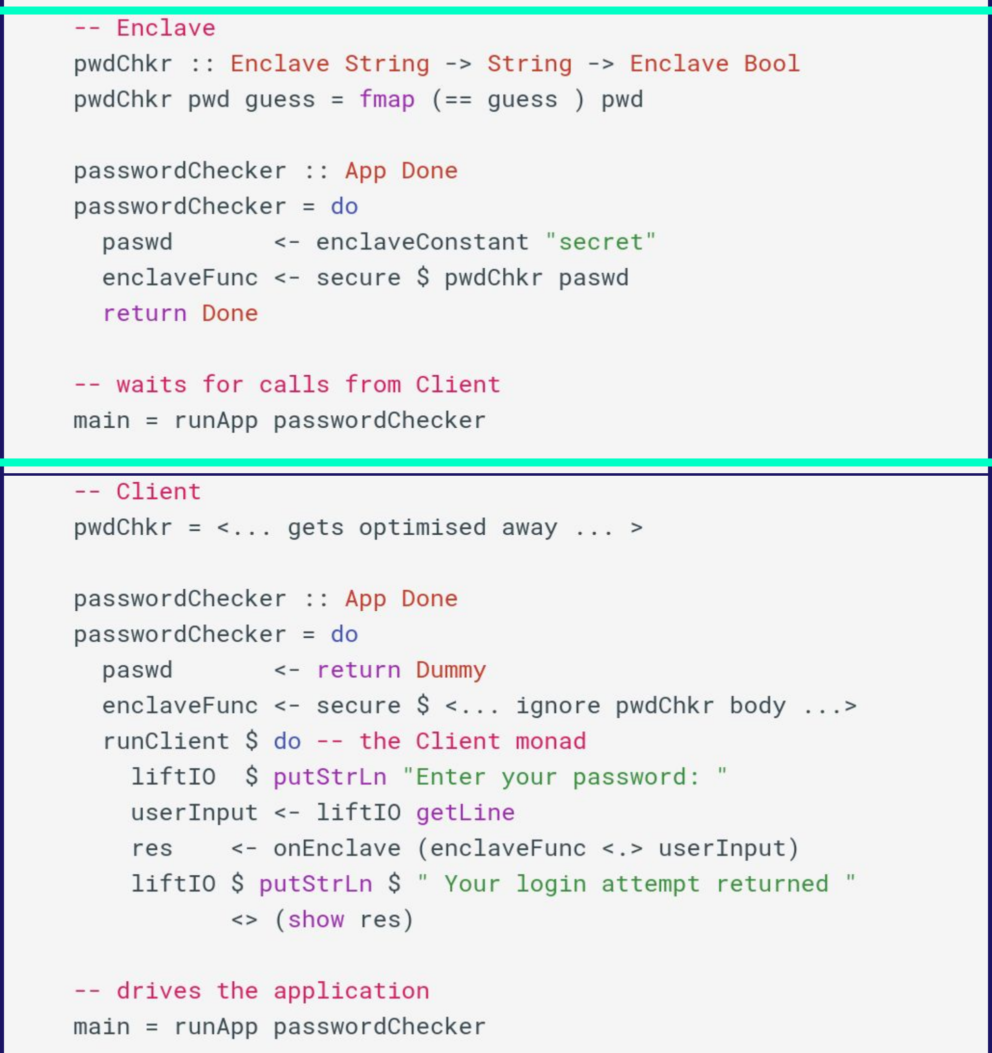


```haskell
-- Enclave
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd       <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  return Done

-- waits for calls from Client
main = runApp passwordChecker
```

Compilation 2



```haskell
-- Client
pwdChkr = <... gets optimised away ... >

passwordChecker :: App Done
passwordChecker = do
  paswd       <- return Dummy
  enclaveFunc <- secure $ <... ignore pwdChkr body ...>
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

-- drives the application
main = runApp passwordChecker
```



Compilation 1

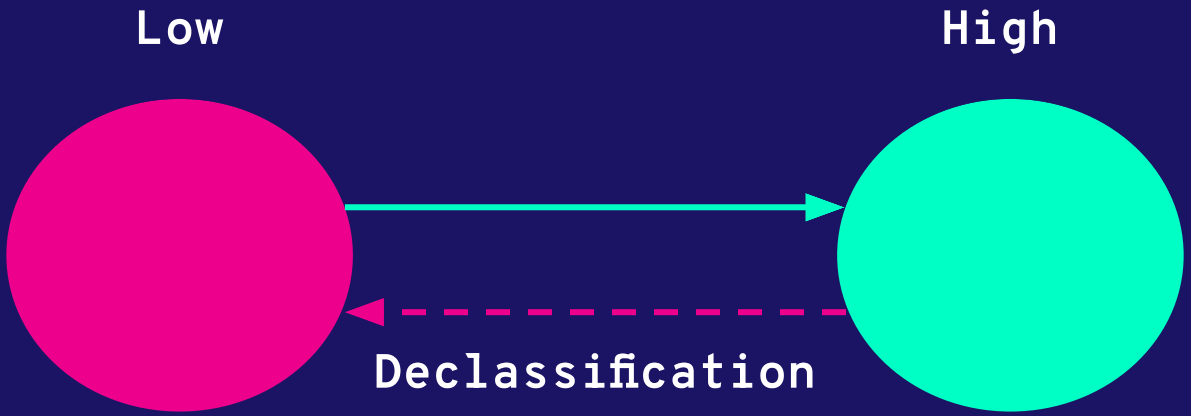
```haskell
-- Enclave
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess ) pwd

passwordChecker :: App Done
passwordChecker = do
  paswd      <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr paswd
  return Done

-- waits for calls from Client
main = runApp passwordChecker
```

Runs on a Trusted GHC Runtime using a subset of glibc

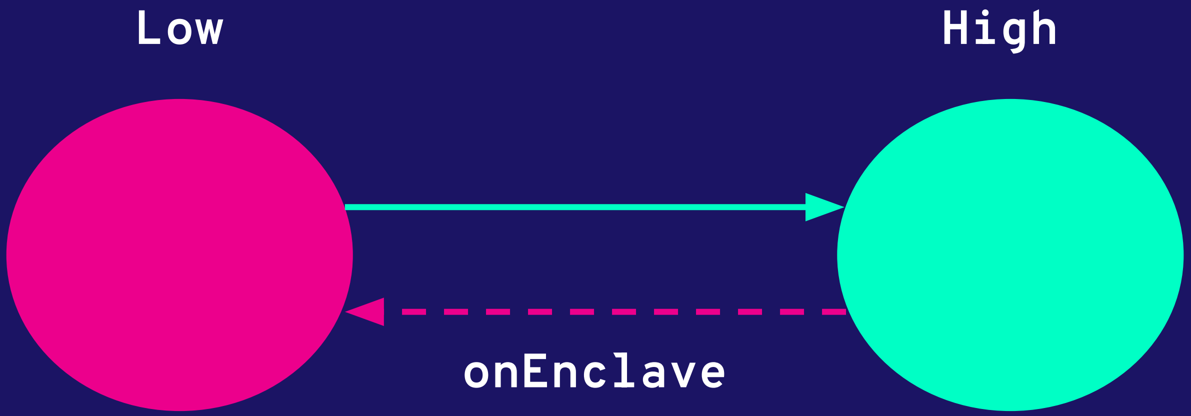
Compilation 2

```haskell
-- Client
pwdChkr = <... gets optimised away ... >

passwordChecker :: App Done
passwordChecker = do
  paswd      <- return Dummy
  enclaveFunc <- secure $ <... ignore pwdChkr body ...>
  runClient $ do -- the Client monad
    liftIO  $ putStrLn "Enter your password: "
    userInput <- liftIO getLine
    res    <- onEnclave (enclaveFunc <.> userInput)
    liftIO $ putStrLn $ " Your login attempt returned "
           <> (show res)

-- drives the application
main = runApp passwordChecker
```

Information Flow Control



Information Flow Control



Information Flow Control

```
onEnclave :: (Binary a) => Secure (Enclave a) → Client a
```

Information Flow Control

```
onEnclave :: (Binary a) => Secure (Enclave a) → Client a
```

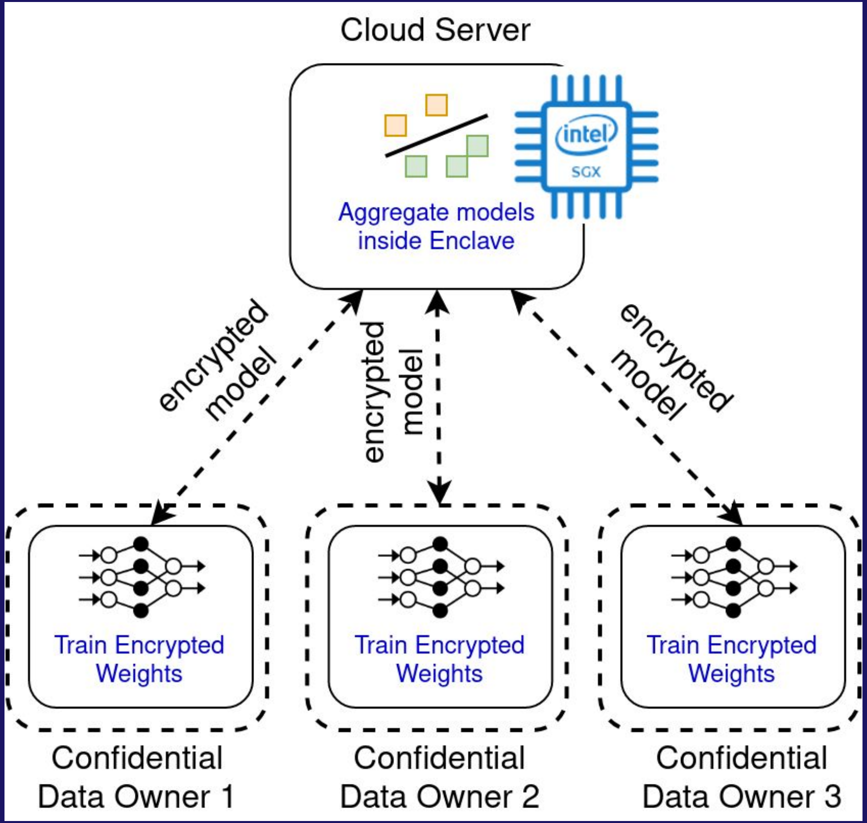
Lack of a Binary instance
prevents accidental leaks

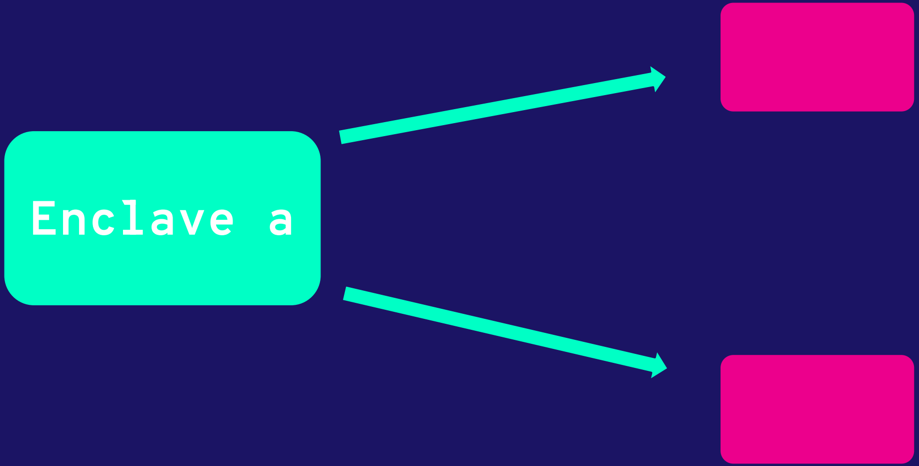
Information Flow Control

```
onEnclave :: (Binary a) => Secure (Enclave a) → Client a
```

Enclave monad restricted
using a RestrictedIO typeclass

Zero Trust Federated Learning



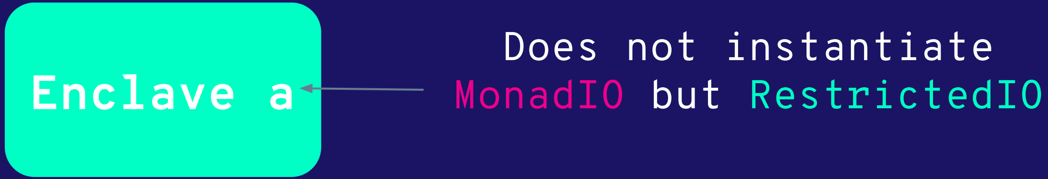


Enclave a

Possibly malicious libraries

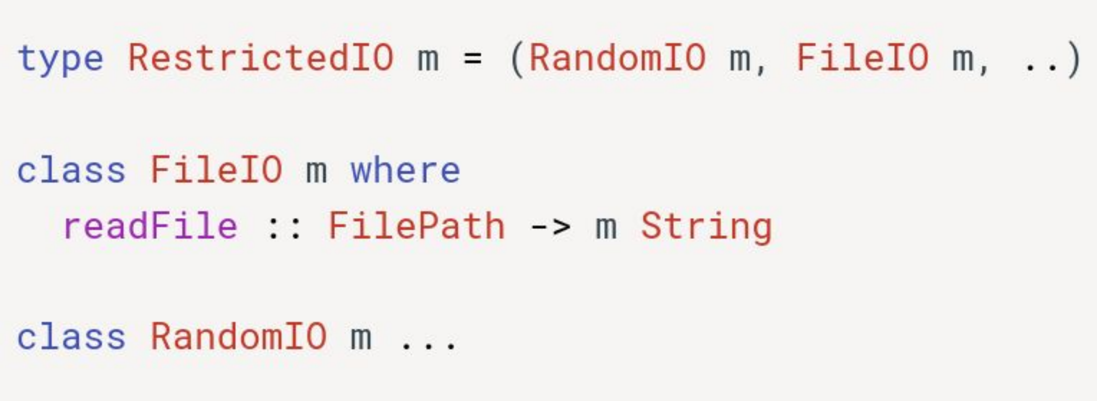
Haskell has a long history of using the type system to protect confidential data*

*MAC, LIO, HLIO [Haskell 2008], [ICFP 2012], [OSDI 2012], [CSF 2014], [ICFP 2015], [CCS 2017], [CSF 2019], [POPL 2019], [CSF 2020]

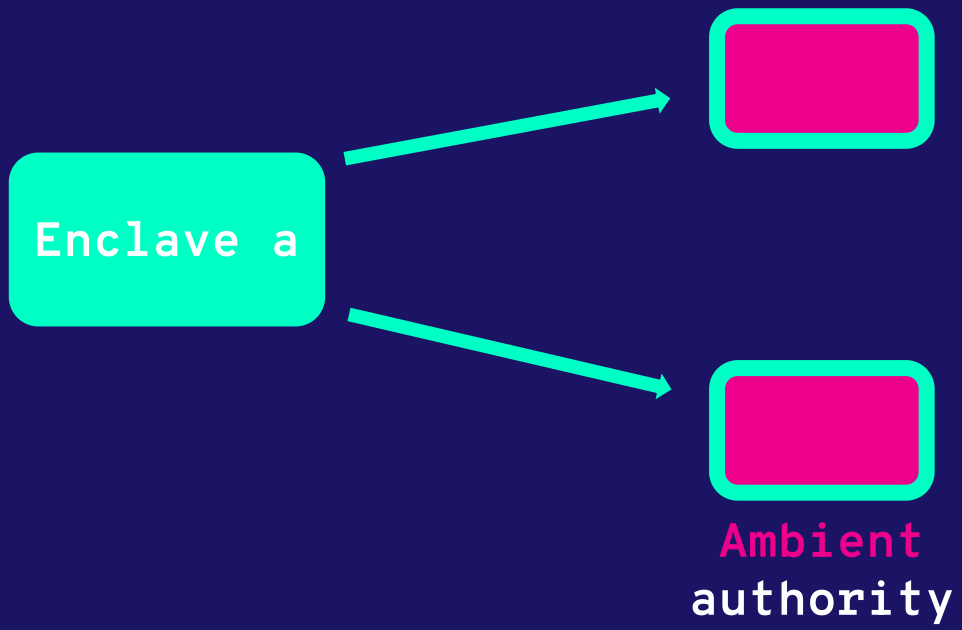


Enclave a

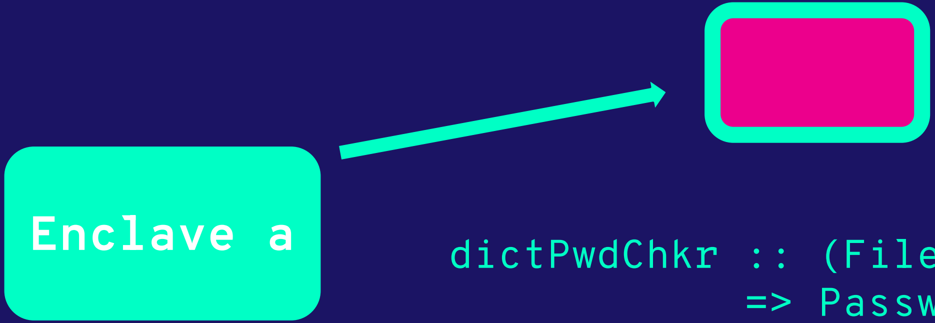
Does not instantiate
MonadIO but RestrictedIO



```haskell
type RestrictedIO m = (RandomIO m, FileIO m, ..)

class FileIO m where
  readFile :: FilePath -> m String

class RandomIO m ...
```

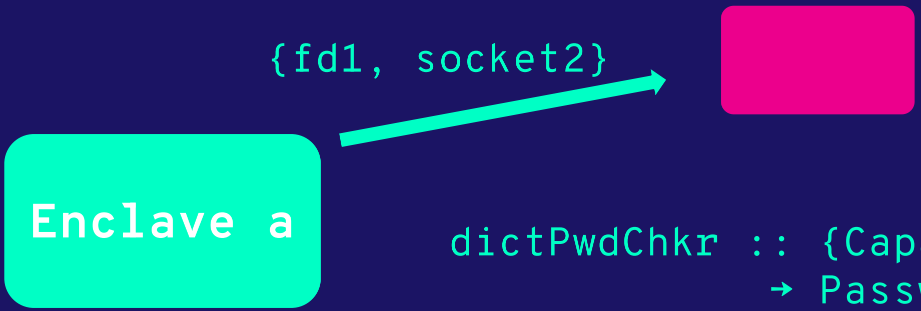


Enclave a
Ambient
authority



```
dictPwdChkr :: (FileIO m, NetworkIO m)
            => Password → m Bool
dictPwdChkr pwd = do
   localDict ← readFile “foo.txt”
   let b = any (== pwd) localDict
   res ← compareDictPwd socket2 pwd
   str ← readFile “/etc/passwd”
   send socket1 str
   return (res || b)
```

```
readFile :: FileDescriptor → IO String
```

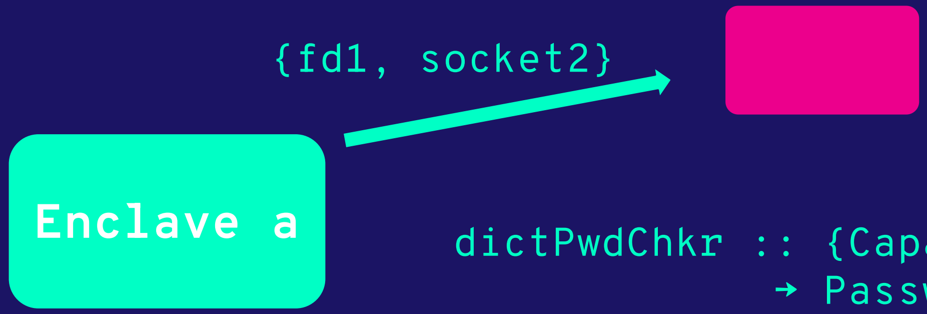


{fd1, socket2}

Enclave a

```
dictPwdChkr :: {Capability}
             → Password → IO Bool
dictPwdChkr pwd = do
   (fd1, socket2) ← getCaps
   localDict ← readFile fd1
   let b = any (== pwd) localDict
   res ← compareDictPwd socket2 pwd
   str ← readFile ???
   send ??? str
   return (res || b)
```

```
readFile :: FileDescriptor → IO String
```



{fd1, socket2}

Enclave a

```
dictPwdChkr :: {Capability}
            → Password → IO Bool
dictPwdChkr pwd = do
    (fd1, socket2) ← getCaps
    localDict ← readFile fd1
    let b = any (== pwd) localDict
    res ← compareDictPwd socket2 pwd
    str ← readFile 7
    send ??? str
    return (res || b)
```

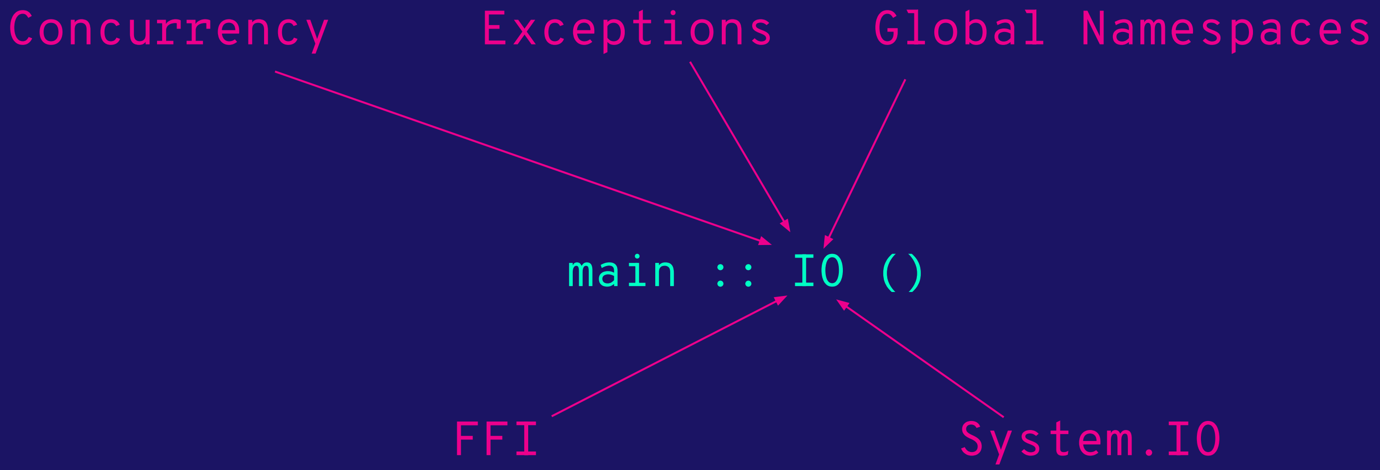
Attempts to forge will fail as the file table can be protected outside the library sandbox

Is Haskell’s purity and type system ideal for tracking capabilities?

```haskell
main :: IO ()
main = putStrLn “Hello World!”
```

Not capability-safe as
System.IO exposes “stdout”

```
main :: IO ()
main = putStrLn “Hello World!”
```



Concurrency

Exceptions

Global Namespaces

```
main :: IO ()
```

FFI

System.IO

Capability Languages

Joule

E

Caja

Joe-E

Secure EcmaScript

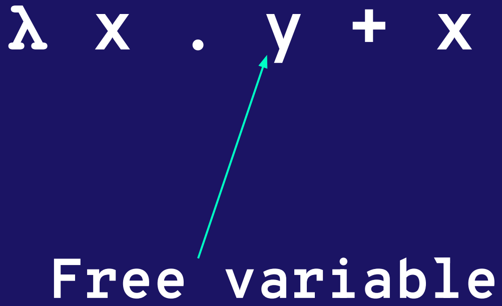
WebAssembly

Mostly dynamic languages

Capability Taming is tedious and error prone (see JoeE)

```
dictPwdChkr :: Password → IO Bool
```

Can we look at library interfaces and figure out what capabilities they require?



λ x . y + x

Free variable

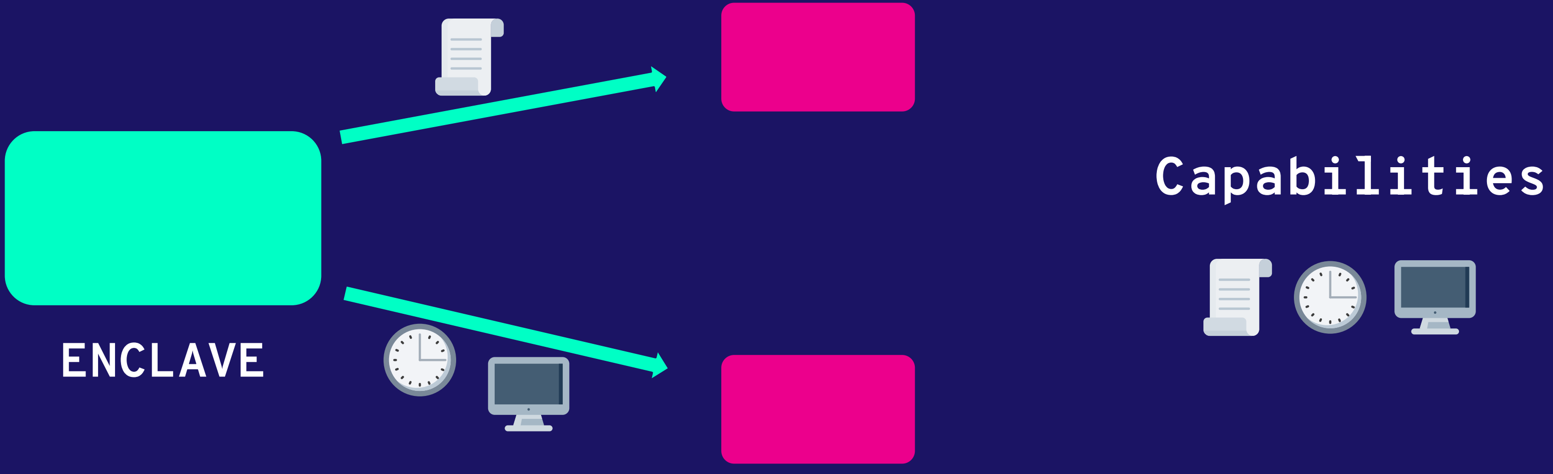
```
λ x . … putStrLn x
```

Free variable

```
dictPwdChkr :: □(Password → IO Bool)
```

Blocks all ambient capabilities

Recovering Purity with Comonads and Capabilities. Choudhury et al. ICFP ‘20
Practical Normalization by Evaluation for EDSLs. Valliappan et al. ICFP ‘21



ENCLAVE

Libraries with *ambient* authority

Capabilities

HasTEE

Secure Enclave Programming

THANKS!

Do you have any questions?

sarkara@chalmers.se