Confidential Computing with Haskell

Abhiroop Sarkar
Chalmers University, Gothenburg
KEY IDEA 1

UNTRUSTED

DECLASSIFICATION

ENCLAVE
KEY IDEA 2

ENCLAVE

Libraries with ambient authority
KEY IDEA 2

ENCLAVE

Libraries with *ambient* authority

Capabilities
Haskell on Trusted Execution Environments
Authors

Abhiroop Sarkar
Robert Krook
Koen Claessen
Cloud Deployments

- HARDWARE
- HYPERVISOR
- OPERATING SYSTEM
- APP
  - My APP

Trusted Computing Base
## OS Vulnerabilities

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

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


Cloud Deployments

HARDWARE

HYPERVISOR

OPERATING SYSTEM

APP

APP

APP

My APP
Cloud Deployments

- Hardware
- Hypervisor
- Operating System
- Operating System

Diagram showing layers of cloud deployments.
Trusted Execution Environment (TEE)
Trusted Execution Environment (TEE)

- HARDWARE
- OPERATING SYSTEM
- OPERATING SYSTEM
- PARTITIONING
- CONFEIDENTIAL CODE AND DATA
- RESTRICTED libc (no mmap)
Trusted Execution Environment (TEE)

- Intel SGX
- ARM TrustZone
- AMD SEV-SNP
- RISC-V Physical Memory Protection
Programming TEEs

Original Project

Untrusted Project

Restricted libc

Trusted Project
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
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
  passwd <- enclaveConstant "secret"
enclaveFunc <- secure $ pwdChkr passwd
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
  passwd <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr passwd
  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
    passwd <- enclaveConstant "secret"
    enclaveFunc <- secure $ pwdChkr passwd
    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
    passwd <- enclaveConstant "secret"
    enclaveFunc <- secure $ pwdChkr passwd
    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
  passwd <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr passwd
  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
  passwd <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr passwd
  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
  passwd <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr passwd
  runClient $ do -- the client module
    liftIO $ putStrLn "Enter your password:"
    use Input <- liftIO getLine
    res <- onEnclave (enclaveFunc <> user Input)
    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
  passwd <- enclaveConstant "secret"
  enclaveFunc <- secure $ pwdChkr passwd
  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

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

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

-- waits for calls from Client
main = runApp passwordChecker
-- Enclave
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

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

-- waits for calls from Client
main = runApp passwordChecker

-- Client
pwdChkr = (... gets optimised away ... )

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

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

Compilation 2

```haskell
-- Client

pwdChkr = <$> ... gets optimised away ... >

passwordChecker :: App Done
passwordChecker = do
  passwd <- 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
```

Runs on a Trusted GHC Runtime using a subset of glibc
Information Flow Control

- Low to High
- Declassification

Diagram showing the flow of information from low to high security levels with declassification as a possible path.
Information Flow Control

onEnclave :: (Binary a) => Secure (Enclave a) → Client a
Information Flow Control

onEnclave :: (Binary a) => Secure (Enclave a) \rightarrow Client a

Lack of a Binary instance prevents accidental leaks
Information Flow Control

\[ \text{onEnclave} :: (\text{Binary } a) \Rightarrow \text{Secure (Enclave } a) \rightarrow \text{Client } a \]

Enclave monad restricted using a RestrictedIO typeclass
Zero Trust Federated Learning

The diagram illustrates a federated learning system with confidentiality. Data from Confidential Data Owners 1, 2, and 3 are encrypted and sent to the Cloud Server. The Cloud Server aggregates models inside an Enclave. The encrypted models are then sent back to the respective Confidential Data Owners to train encrypted weights.
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

definition

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

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

class RandomIO m ...
Enclave a

Ambient authority
Enclave a

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)
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 socket2
    send socket2 str
    return (res || b)
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?
main :: IO ()
main = putStrLn "Hello World!"
Not capability-safe as System.IO exposes “stdout”

main :: IO ()
main = putStrLn "Hello World!"
main :: IO ()

Concurrency  Exceptions  Global Namespaces

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?
\( \lambda x . y + x \)

Free variable
\lambda \ x \ . \ ... \ \text{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