Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John Mitchell, and David Mazières
Motivation

Web framework for integrating 3rd party apps
Current Approach

- Platforms restrict what data apps can see

- No guarantee what app can do with your data

```haskell
sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
```
Current Approach

• Platforms restrict what data apps can see
  - [Image: Facebook permission dialogue]
  - "Your messages"

• No guarantee what app can do with your data

sendMessage user message = do
messages <- getUserMessages user
putUserMessages user (message:messages)
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data

sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data

sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data

sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data

sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data

sendMessage user message = do
messages <- getUserMessages user
when (messages `hasRecipient` "Julian Assange")
alertTSA
putUserMessages user (message:messages)
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data
Current Approach

- Platforms restrict what data apps can see

- No guarantee what app can do with your data
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data
Current Approach

• Platforms restrict what data apps can see

• No guarantee what app can do with your data
Fundamental Problem

- Problem:
  - Read sensitive data with `getUserMessages`
  - Wrote to remote host with `alertTSA`

- Solution:
  - Restrict who the app can communicate with depending on what data it has read
Alternative Approach
Information Flow Control with LIO

• Label every object with a security level/policy
  ➤ Label protects data by specifying who can read/write

• Example security label lattice:
LIO Monad

- Execute computations in \textbf{LIO} monad
  - Opaque monad records context “current” label
  - I.e., tracks taint of computation
  - Restricts side-effects an app can perform

- Example (sending Bob a message):
LIO Monad

• Execute computations in LIO monad
  ➢ Opaque monad records context “current” label
  ➢ I.e., tracks taint of computation
  ➢ Restricts side-effects an app can perform

• Example (sending Bob a message):

  ![Diagram of sending a message]
LIO Monad

• Execute computations in LIO monad
  ➢ Opaque monad records context “current” label
  ➢ I.e., tracks taint of computation
  ➢ Restricts side-effects an app can perform

• Example (sending Bob a message):
LIO Monad

- Execute computations in LIO monad
  - Opaque monad records context “current” label
  - I.e., tracks taint of computation
  - Restricts side-effects an app can perform

- Example (sending Bob a message):
LIO Monad

• Execute computations in LIO monad
  ➢ Opaque monad records context “current” label
  ➢ I.e., tracks taint of computation
  ➢ Restricts side-effects an app can perform

• Example (sending Bob a message):

Send app message
LIO Monad

- Execute computations in \texttt{LIO} monad
  - Opaque monad records context “current” label
  - I.e., tracks taint of computation
  - Restricts side-effects an app can perform

- Example (sending Bob a message):
  - Send app message
  - Write all new messages
LIO Monad

- Execute computations in **LIO** monad
  - Opaque monad records context “current” label
  - I.e., tracks taint of computation
  - Restricts side-effects an app can perform

- Example (sending Bob a message):
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` "Julian Assange")
    alertTSA
  putUserMessages user (message:messages)
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` "Julian Assange")
    alertTSA
    putUserMessages user (message:messages)
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` "Julian Assange")
    alertTSA
    putUserMessages user (message:messages)

App receives exception:
Trying to leak sensitive data.

Send app message
LIO Monad

Overly restrictive

• Messenger app wishes to send broadcast message

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> sendMessage user message
```

Send app message
LIO Monad

Overly restrictive

- Messenger app wishes to send broadcast message

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
    forM_ users $ \user -> sendMessage user message
```

Send app message
LIO Monad

Overly restrictive

- Messenger app wishes to send broadcast message

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> sendMessage user message
```
LIO Monad
Overly restrictive

• Messenger app wishes to send broadcast message

```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> sendMessage user message
```

Send app message
LIO Monad
Overly restrictive

- Messenger app wishes to send broadcast message

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> sendMessage user message
```

App receives exception:
May be leaking Bob’s data.
Practical Concerns

- Strawman: use `discard` to execute sensitive actions
  ➤ Do not observe result ➔ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ 
    λuser -> discard $ 
      sendMessage user message
```

```plaintext`
+---+---+---+---+
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
+---+---+---+---+
```
Practical Concerns

- Strawman: use `discard` to execute sensitive actions
  ➤ Do not observe result ➔ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
```

Practical Concerns
Practical Concerns

- Strawman: use `discard` to execute sensitive actions
  - Do not observe result ➔ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
```

Practical Concerns

- Strawman: use `discard` to execute sensitive actions
  - Do not observe result ⟷ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
```

sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
Practical Concerns

- Strawman: use discard to execute sensitive actions
  ➤ Do not observe result ➤ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
```
Practical Concerns

• Strawman: use `discard` to execute sensitive actions
  ➤ Do not observe result ➔ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \
    \user -> discard $ sendMessage user message
```

Practical Concerns

- Strawman: use `discard` to execute sensitive actions
  - Do not observe result ➤ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
```

sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
Practical Concerns

- Strawman: use `discard` to execute sensitive actions ➤ Do not observe result ➵ no leak!

```haskell
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
```

Practical Concerns
... discard covertly leaks termination information.
Termination Attack

- Leak secret bit through non-termination

```haskell
isConsiprator :: User -> Int -> LIO ()
isConsiprator victim n = do
discard $ do
    messages <- getUserMessages victim
    let user = recipient (message!!n)
    when (user == "Julian Assange") ⊥
    writeToPublicChannel "clean"
```

- If user matches: diverge in `discard` block
- Else: write “clean” to public channel
Termination Attack

• Address at the framework/system level

• Use different attacker model
  ➤ Termination-insensitive non-interference: *if a program terminates, then confidentiality and integrity of data is preserved*

• Don’t address: very low bandwidth channel
  ➤ Leaks 1 bit per run
Adding Fire

- Threads are crucial to modern web frameworks
  ➢ Need to concurrently serve requests, etc.

- Viability of covert channel attacks
  ➢ Termination attack leaks 1 bit per thread
  ➢ Can leak data within same program
  ➢ Permits attacks relying on internal timing
Internal Timing Attack

- Leak secret bit by affecting output ordering

```haskell
isConsiprator :: User -> Int -> LIO ()
isConsiprator victim n = do
  fork $ do
    delay 100
    writeToPublicChannel "y"
  fork $ do
    discard $ do
      messages <- getUserMessages victim
      let user = msgDestination (message!!n)
      when (user == "Julian Assange")$ delay 500
      writeToPublicChannel "es"
```

- If user matches: write “y” first, then “es”
  Else: write “es” then “y”

- Analyze output: “yes” contact with Assange
Solution: Threads
Fighting fire with fire

• Decoupling `discard` computations
  ➢ Spawn new thread to execute sub-computation
  ➢ Immediately return a labeled future to thread

• Making LIO safe:
  - `discard`
  + `lFork`: spawn new, labeled threads
  + `lWait`: force thread evaluation, first “raising” context label to read result and termination
Termination Attack

- Cannot leak bits through non-termination

```haskell
isConsiprator :: User -> Int -> LIO ()
isConsiprator victim n = do
  lFork $ do
    messages <- getUserMessages victim
    let user = recipient (message!!n)
    when (user == "Julian Assange") ⊥
    writeToPublicChannel "clean"
```

- If user matches: diverge in `discard` block
- Always write “clean” to public channel
Internal Timing Attack

• Cannot affect output ordering

```haskell
isConsiprator :: User -> Int -> LIO ()
isConsiprator victim n = do
  lFork $ do
delay 100
  writeToPublicChannel "y"
  lFork $ do
    lFork $ do
      messages <- getUserMessages victim
      let user = msgDestination (message!!n)
      when (user == "Julian Assange")$ delay 500
      writeToPublicChannel "es"
```

➤ Always write “es” first, then “y”
Status of LIO

- Used in production system
-Formalized as call-by-name $\lambda$-calculus
  ➤ Support for thread spawning and joining with $\texttt{Fork}$ and $\texttt{Wait}$
  ➤ Support for MVars

- Theorem: Termination-sensitive non-interference
  ➤ Informally: Confidentiality and integrity of data is preserved even if threads diverge.
A Practical Perspective

- Covert channels closed by LIO
  - Termination
  - Internal timing

- What about external timing channel?
A Practical Perspective

- Covert channels closed by LIO
  - Termination
  - Internal timing

- What about external timing channel?
A Practical Perspective

- Covert channels closed by LIO
  - Termination
  - Internal timing

- What about external timing channel?
A Practical Perspective

- Covert channels closed by LIO
  - Termination
  - Internal timing

- What about external timing channel?
A Practical Perspective

- Covert channels closed by LIO
  - Termination
  - Internal timing

- What about external timing channel?
A Practical Perspective

• Covert channels closed by LIO
  ➢ Termination
  ➢ Internal timing

• What about external timing channel?
A Practical Perspective

- Covert channels closed by LIO
  - Termination
  - Internal timing

- What about external timing channel?
Thank you

cabal install lio

http://gitstar.com/scs/lio