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 3\textsuperscript{rd} party apps
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

```haskell
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

```plaintext
sendMessage user message = do
    messages <- getUserMessages user
    when (messages `hasRecipient` "Brad Pitt")
        alertPaparazzi
    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 `alertPaparazzi`

• 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 LIO 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
  ➢ 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
  ➢ 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 **LIO** monad
  - Records context “current” label
  - I.e., tracks taint of computation
  - Restricts side-effects an app can perform

- Example (sending Bob a message):

  ![Diagram](image.png)
LIO Monad

- Execute computations in LIO 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
  - Get existing messages

Effectively let’s us reprogram ‘;’
LIO Monad

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

• Example (sending Bob a message):

\[\text{Effectively let’s us reprogram ‘;’}\]
LIO Monad

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

• Example (sending Bob a message):

  Effectively let’s us reprogram ‘;’
LIO Monad

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

• Example (sending Bob a message):

  ![Diagram](Send app message)
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` "Brad Pitt")
  alertPaparazzi
  putUserMessages user (message:messages)
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` "Brad Pitt")
  alertPaparazzi
  putUserMessages user (message:messages)
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` "Brad Pitt")
    alertPaparazzi
    putUserMessages user (message:messages)

Send app message

App receives exception:
Trying to leak sensitive data.
LIO Monad
Overly restrictive

• Messenger app wishes to send broadcast message

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

- Messenger app wishes to send broadcast message

\[
\text{sendMessages} :: [\text{User}] \rightarrow \text{Message} \rightarrow \text{LIO} ()
\]
\[
\text{sendMessages users message} = \text{do}
\quad \text{forM\_ users $\lambda$user $\rightarrow$ sendMessage user message}
\]
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

\[
\text{sendMessages} :: [\text{User}] \to \text{Message} \to \text{LIO} ()
\]
\[
\text{sendMessages} \text{ users message } = \text{do}
\quad \text{forM} \_ \text{ users } \& \lambda \text{user } \to \text{sendMessage user message}
\]
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 $ \user -> discard $ sendMessage user message
```

```haskell
sendMessage user message
```
Practical Concerns

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

\[
\text{sendMessages} :: \text{[User]} \rightarrow \text{Message} \rightarrow \text{LIO ()} \\
\text{sendMessages users message} = \text{do} \\
\text{forM\_ users } \rightarrow \text{ discard } \rightarrow \text{ discard } \\
\text{sendMessage user message}
\]
Practical Concerns

- Strawman: use \texttt{discard} to execute sensitive actions
  ➤ Do not observe result \(\Longrightarrow\) no leak!

\begin{verbatim}
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
  forM_ users $ \user -> discard $ sendMessage user message
\end{verbatim}
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
```

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
```

![Diagram showing a file being sent to multiple users](image-url)
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
```
... discard covertly leaks termination information.
Termination Attack

- Leak secret bit through non-termination

```haskell
isOfInterest :: User -> Int -> LIO ()
isOfInterest victim n = do
discard $ do
  messages <- getUserMessages victim
  let user = recipient (message!!n)
  when (user == "Brad Pitt") ⊥
  writeToPublicChannel "boring"
```

- If user matches: diverge in `discard` block
- Else: write “boring” 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
- 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

isOfInterest :: User -> Int -> LIO ()
isOfInterest victim n = do
  fork $ do delay 100
    writeToPublicChannel "y"
  fork $ do
    discard $ do
      messages <- getUserMessages victim
      let user = msgDestination (message!!n)
      when (user == "Brad Pitt")$ delay 500
      writeToPublicChannel "es"

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

• Analyze output: “yes” ➤ contact with Brad Pitt
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*
  + 1Fork: spawn new, labeled threads
  + 1Wait: force thread evaluation, first “raising” context label to read result and termination
Termination Attack

- Cannot leak bits through non-termination

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

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

- Cannot affect output ordering

```haskell
isOfInterest :: User -> Int -> LIO ()
isOfInterest victim n = do
  discard $ do
    delay 100
    writeToPublicChannel "y"
  discard $ do
    messages <- getUserMessages victim
    let user = msgDestination (message!!n)
    when (user == "Brad Pitt")$ 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 `fork` and `wait`
  ➢ Support for mutable single-place channels

• 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
• Predicative mitigation (Zhang, Askarov, Myers)
  ➤ *Black-box approach ➤ monad transformers!*
  ➤ *Predict output schedule for app*
  ➤ *Misprediction leaks 1 bit, total bounded in log-time*
Mitigation

• Predicative mitigation (Zhang, Askarov, Myers)
  ➢ Black-box approach ➔ monad transformers!
  ➢ Predict output schedule for app
  ➢ Misprediction leaks 1 bit, total bounded in log-time
Mitigation

• Predicative mitigation (Zhang, Askarov, Myers)
  ➤ Black-box approach ➔ monad transformers!
  ➤ Predict output schedule for app
  ➤ Misprediction leaks 1 bit, total bounded in log-time
Mitigation

- Predicative mitigation (Zhang, Askarov, Myers)
  - Black-box approach ➤ monad transformers!
  - Predict output schedule for app
  - Misprediction leaks 1 bit, total bounded in log-time
Mitigation

- Predicative mitigation (Zhang, Askarov, Myers)
  - *Black-box approach ➞ monad transformers!*
  - *Predict output schedule for app*
  - *Misprediction leaks 1 bit, total bounded in log-time*
Mitigation

• Predicative mitigation (Zhang, Askarov, Myers)
  ➤ Black-box approach $\rightarrow$ monad transformers!
  ➤ Predict output schedule for app
  ➤ Misprediction leaks 1 bit, total bounded in log-time