Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems

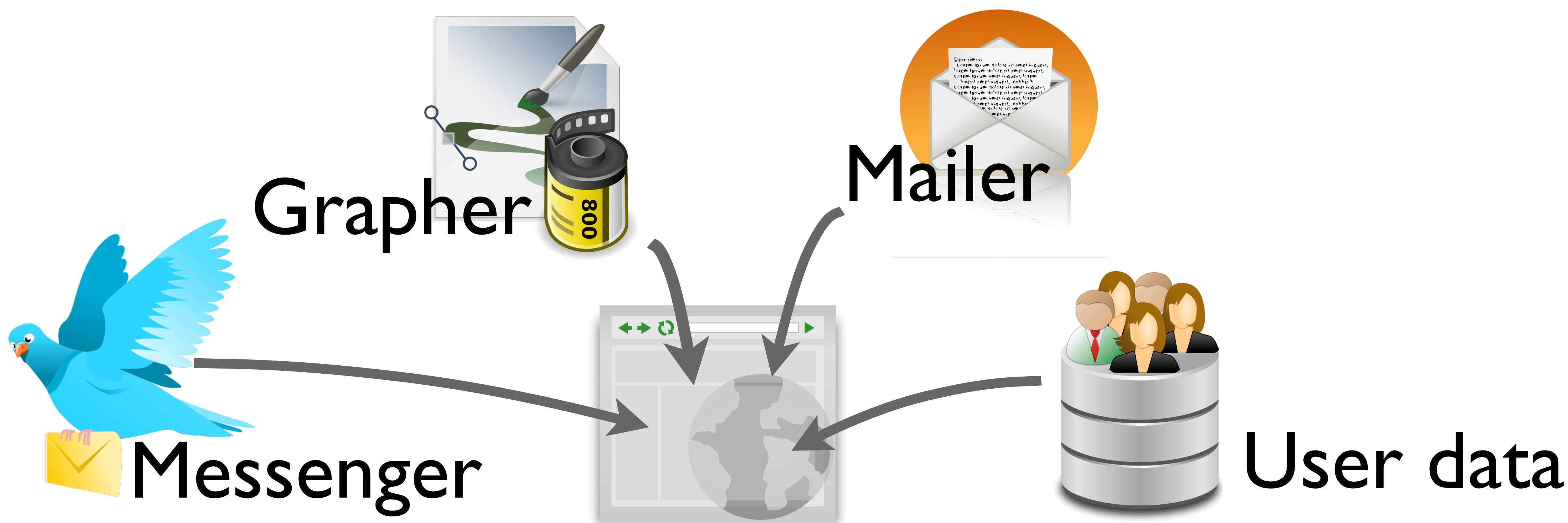
Deian Stefan, Alejandro Russo, Pablo Buiras,
Amit Levy, John Mitchell, and David Mazières

STANFORD UNIVERSITY

CHALMERS

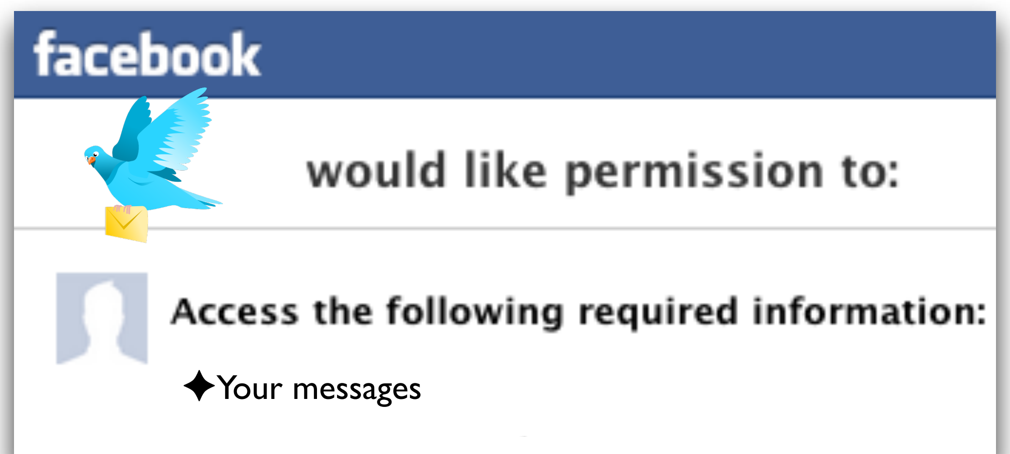
Motivation

Web framework for integrating 3rd party apps



Current Approach

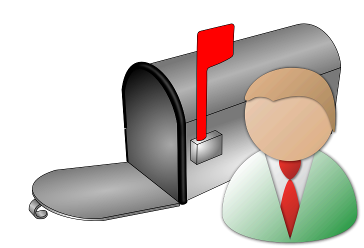
- Platforms restrict what data apps can see



- No guarantee what app can do with your data

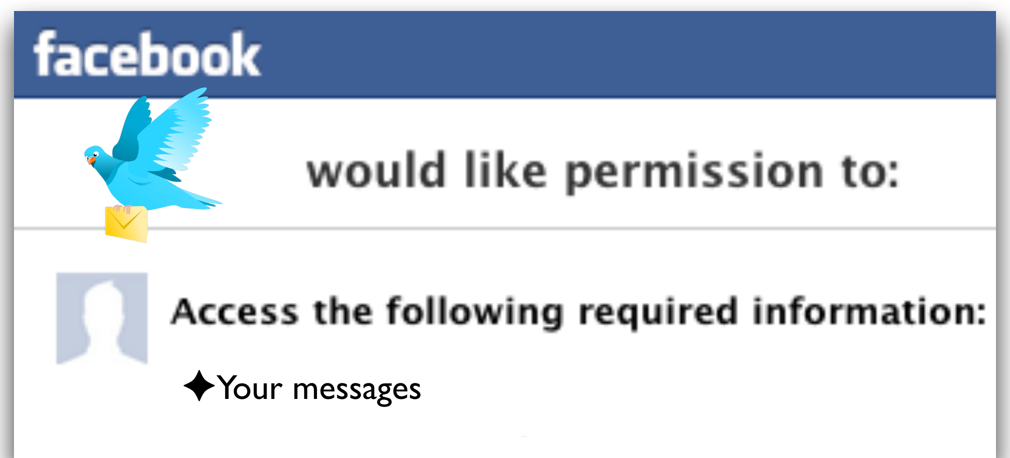
```
sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
```





Current Approach

- Platforms restrict what data apps can see

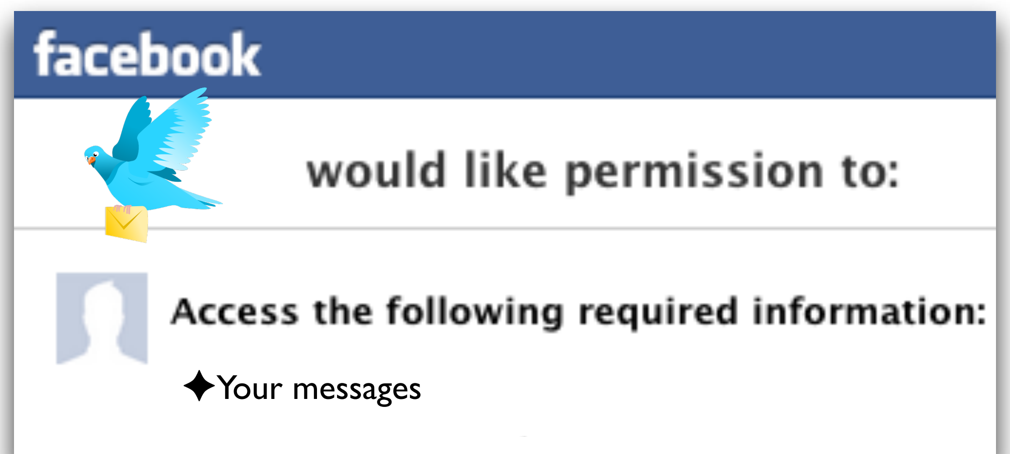


- No guarantee what app can do with your data

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

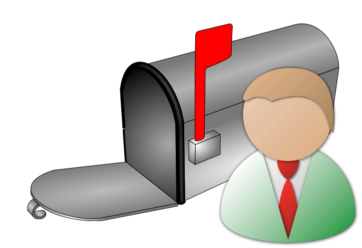
Current Approach

- Platforms restrict what data apps can see



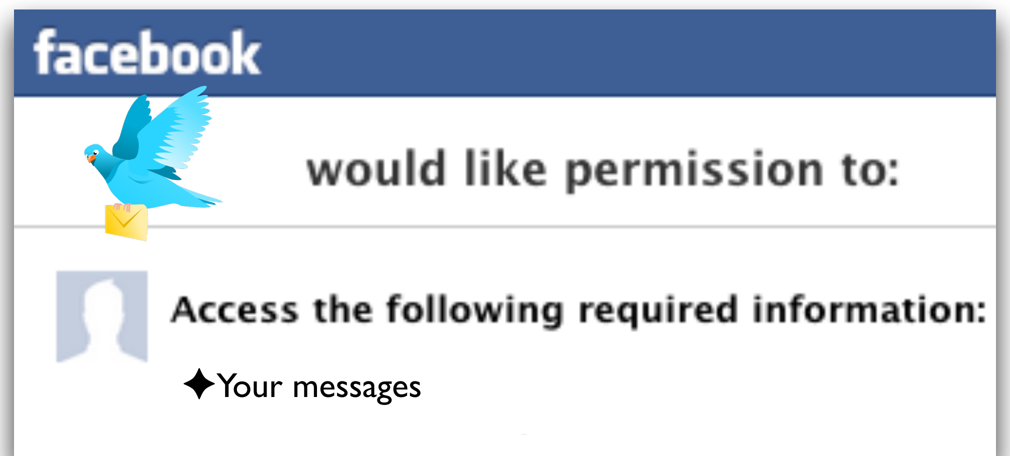
- No guarantee what app can do with your data

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



Current Approach

- Platforms restrict what data apps can see

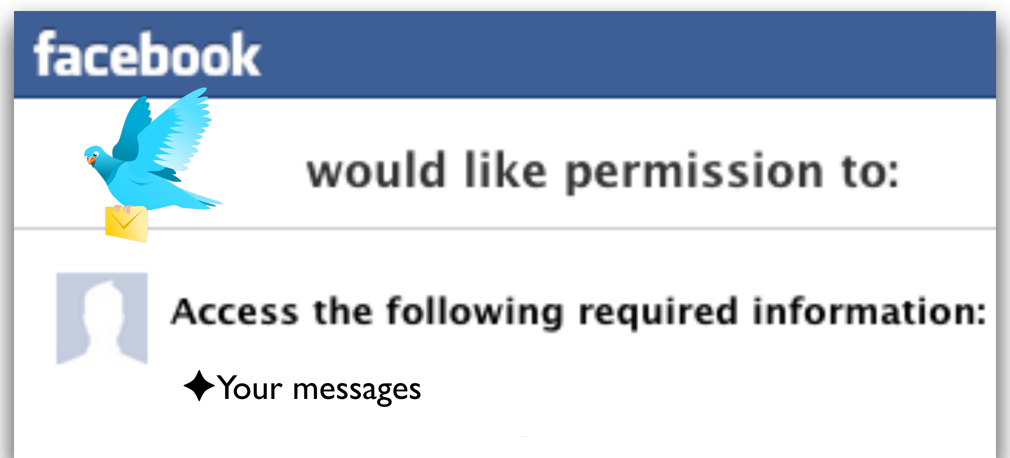


- No guarantee what app can do with your data

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

Current Approach

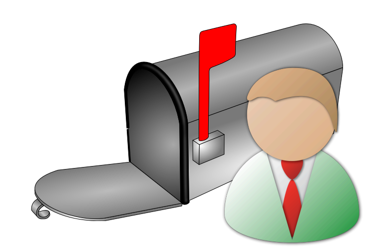
- Platforms restrict what data apps can see



- No guarantee what app can do with your data

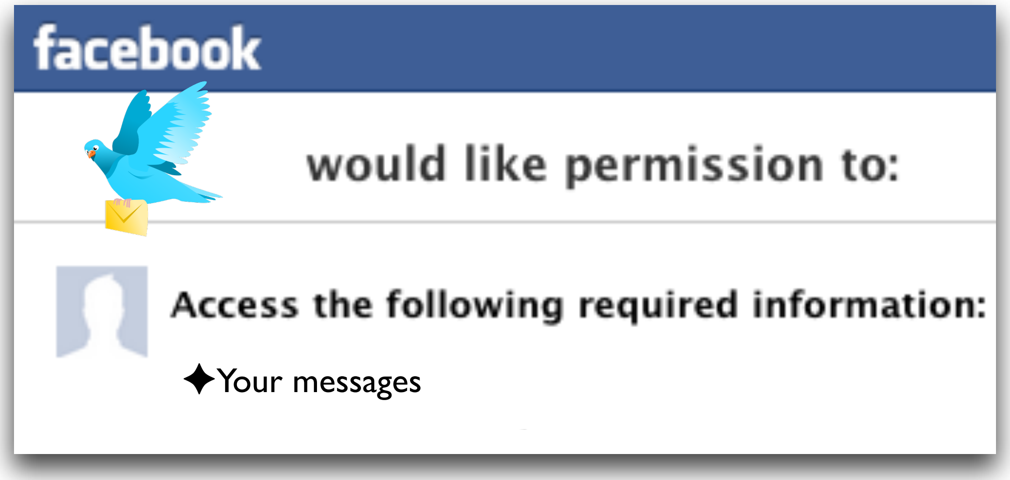
```
sendMessage user message = do
  messages <- getUserMessages user
  putUserMessages user (message:messages)
```





Current Approach

- Platforms restrict what data apps can see

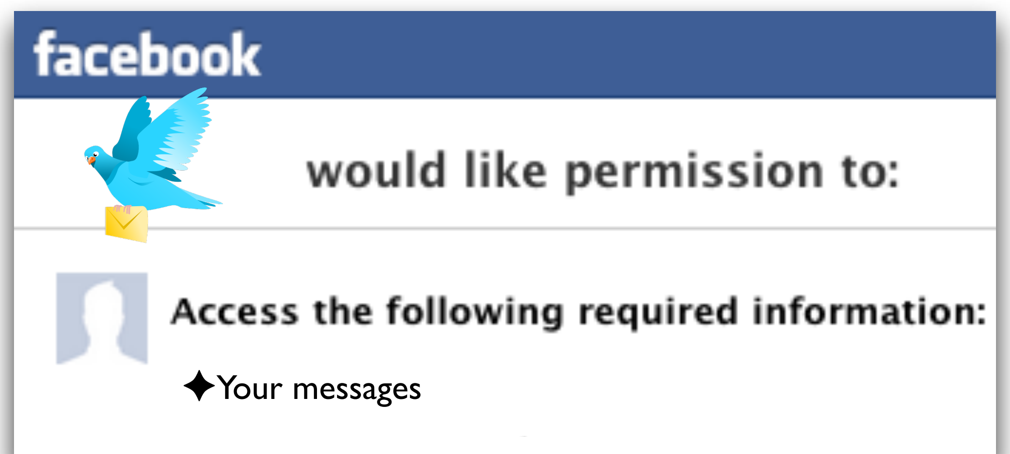


- No guarantee what app can do with your data

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

Current Approach

- Platforms restrict what data apps can see

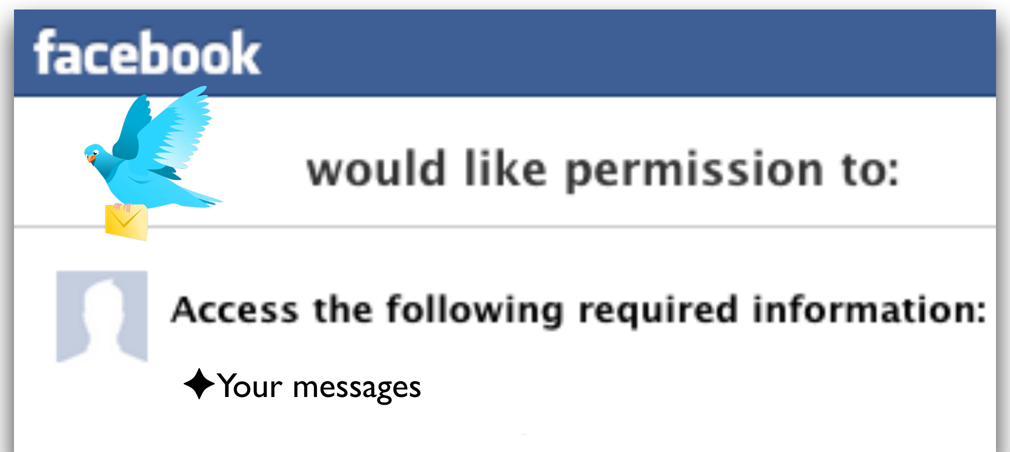


- No guarantee what app can do with your data

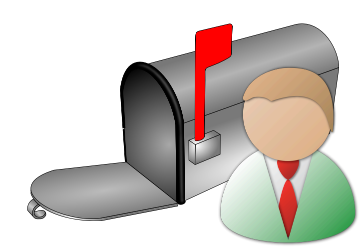
```
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` “Brad Pitt”)
        alertPaparazzi
  putUserMessages user (message:messages)
```

Current Approach

- Platforms restrict what data apps can see

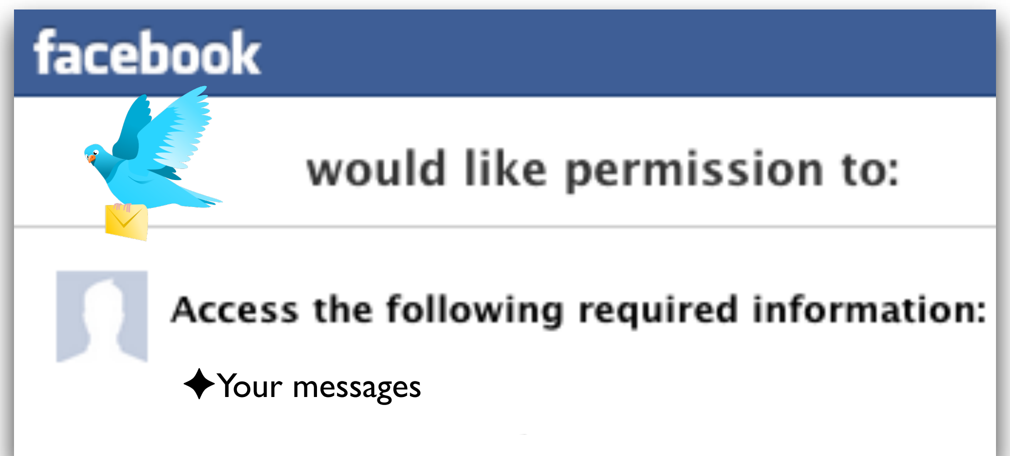


- No guarantee what app can do with your data



Current Approach

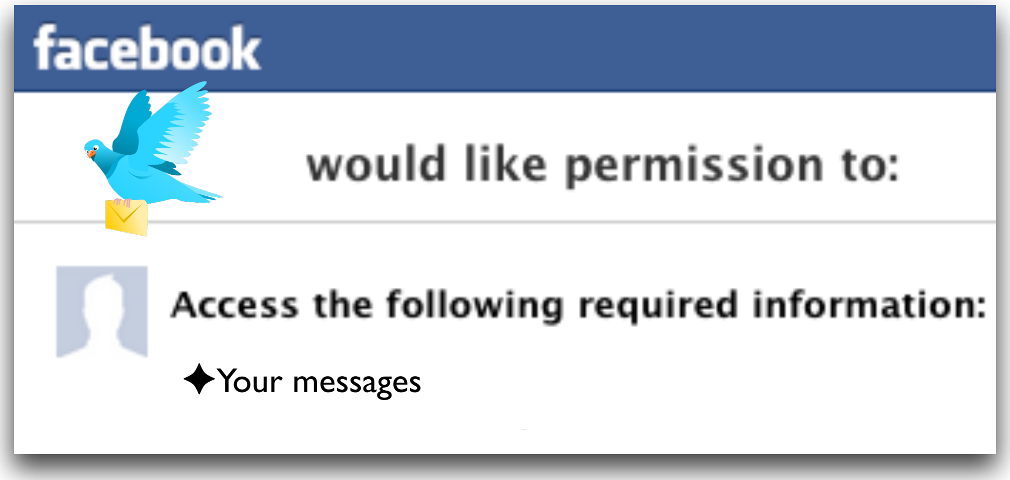
- Platforms restrict what data apps can see



- No guarantee what app can do with your data

Current Approach

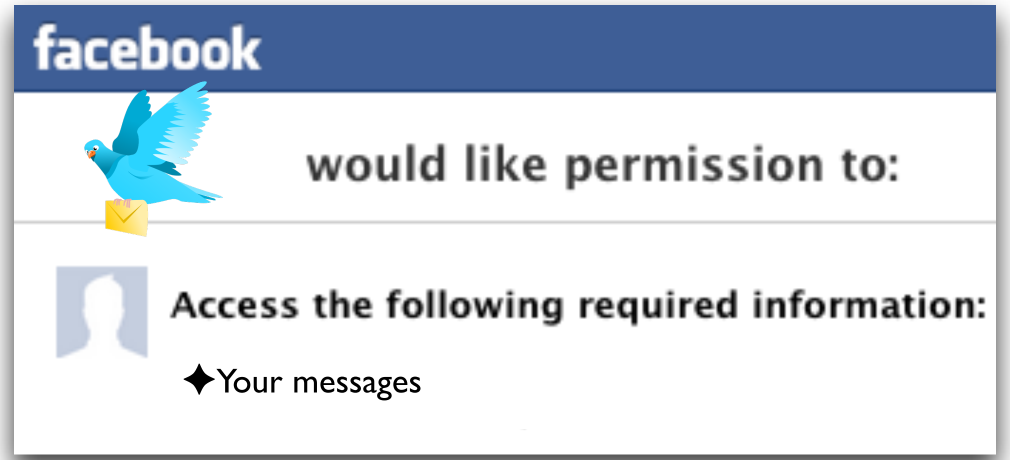
- Platforms restrict what data apps can see



- No guarantee what app can do with your data

Current Approach

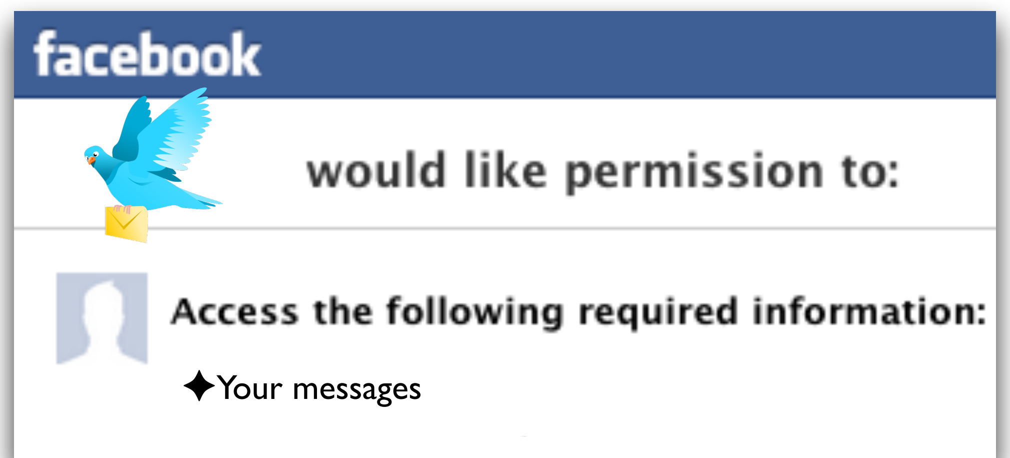
- Platforms restrict what data apps can see



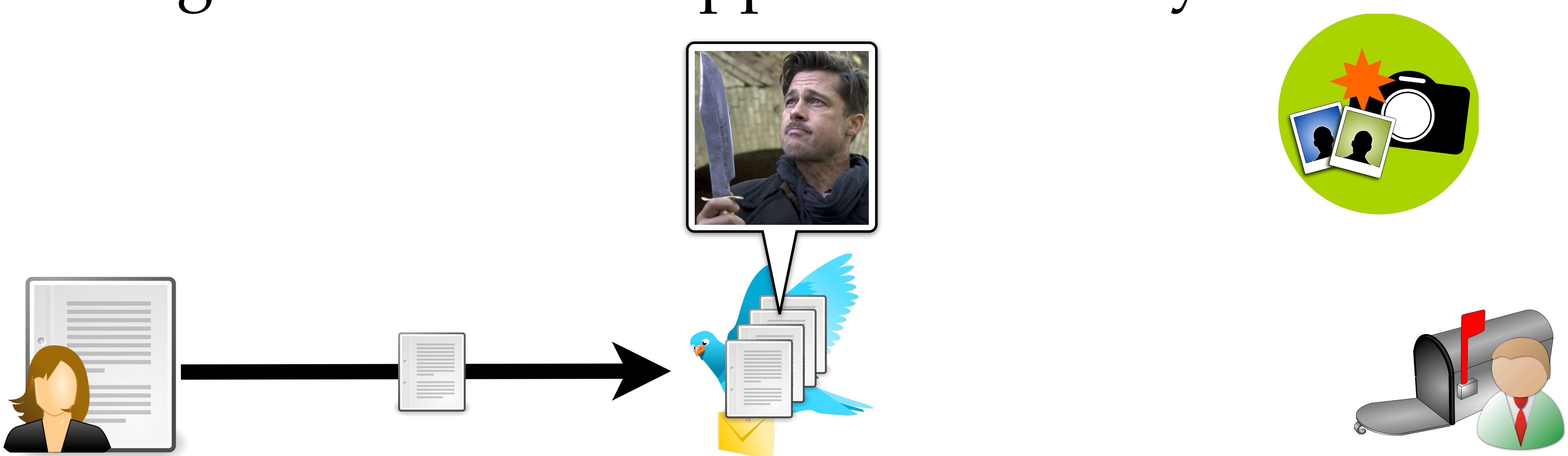
- No guarantee what app can do with your data

Current Approach

- Platforms restrict what data apps can see

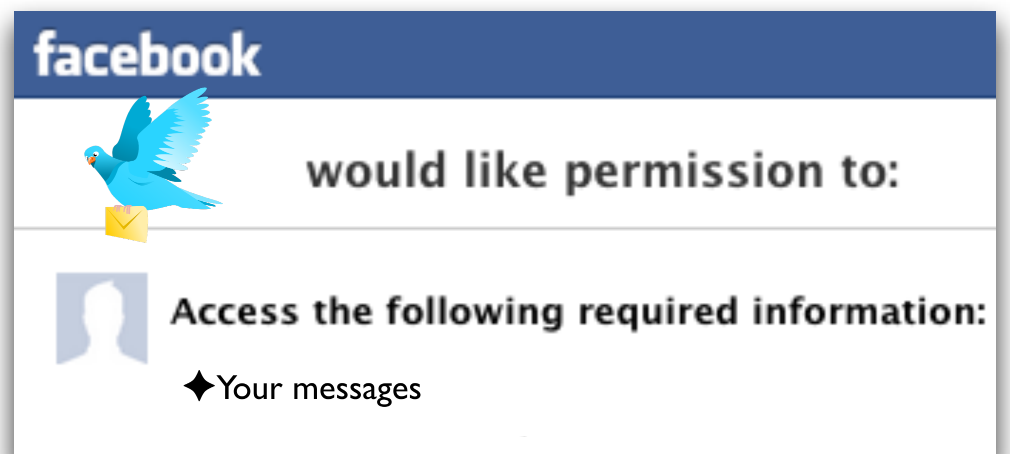


- No guarantee what app can do with your data



Current Approach

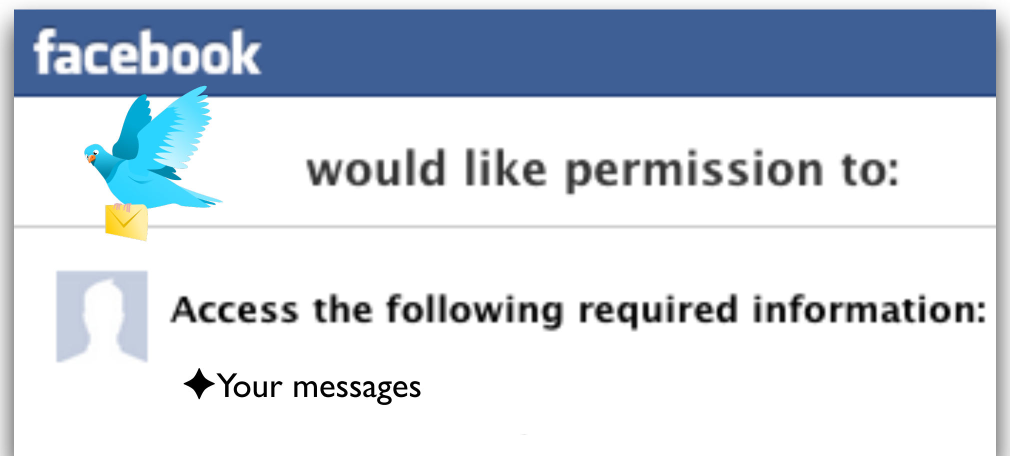
- Platforms restrict what data apps can see



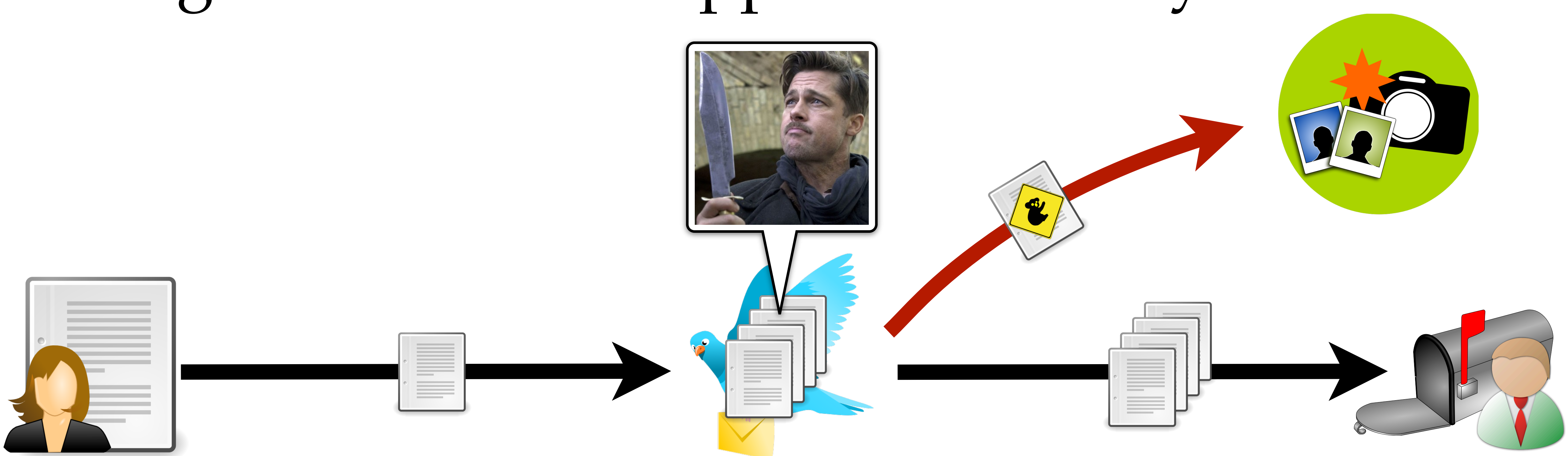
- No guarantee what app can do with your data

Current Approach

- Platforms restrict what data apps can see



- No guarantee what app can do with your data




PRAKTICA

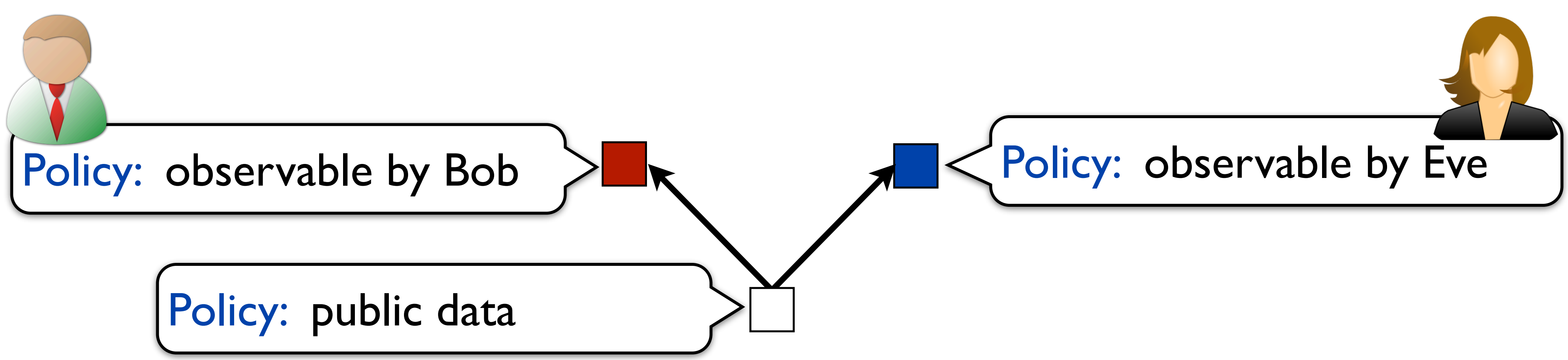
Fundamental Problem

- 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

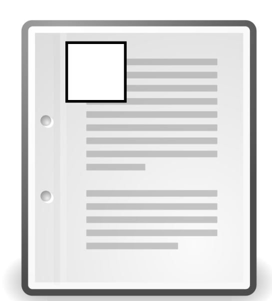
Information Flow Control with LIO

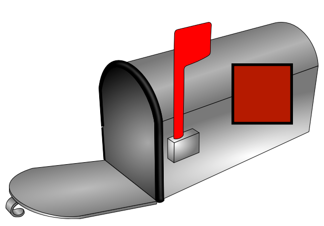
- Label every object with a security level/policy
  - *Label protects data by specifying who can read/write*
- Example security label lattice:



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):

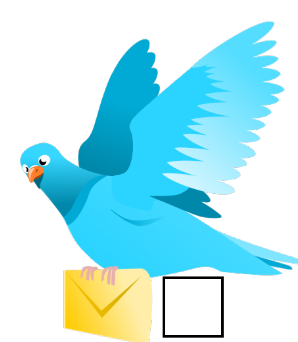


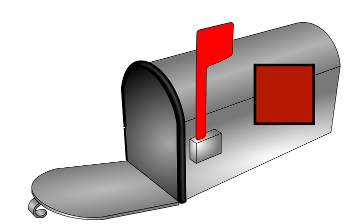


LIO Monad

Effectively let's us reprogram ';'

- 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):

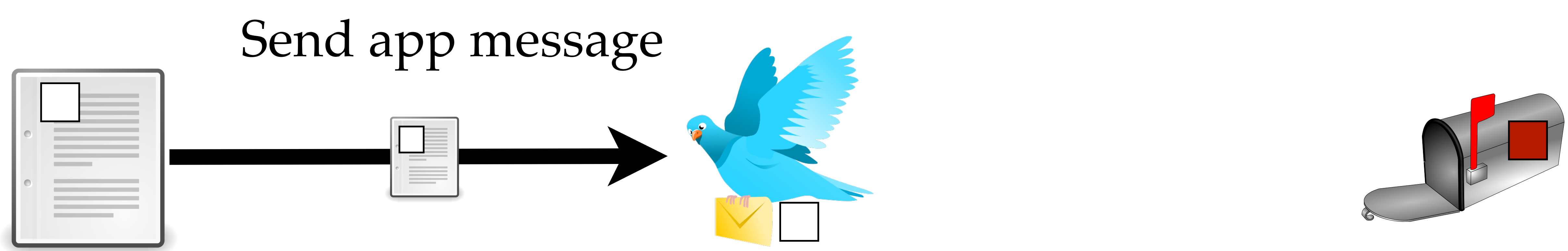




LIO Monad

Effectively let's us reprogram ';'

- 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

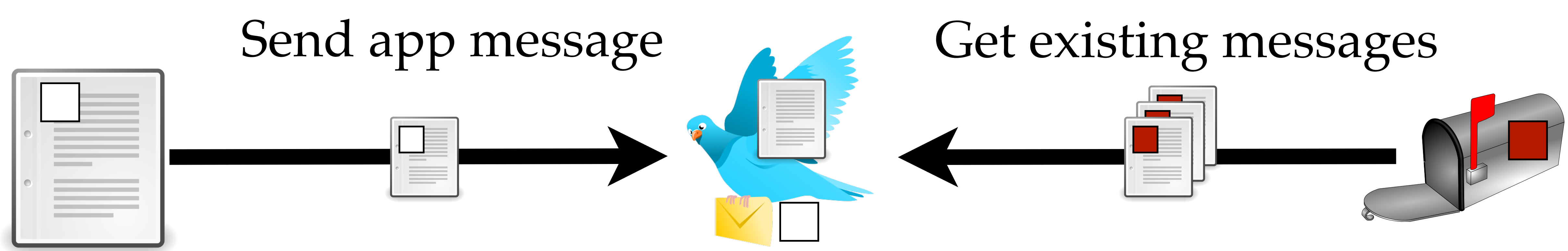
Effectively let's us reprogram ';'

- 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

Effectively let’s us reprogram ‘;’

- 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

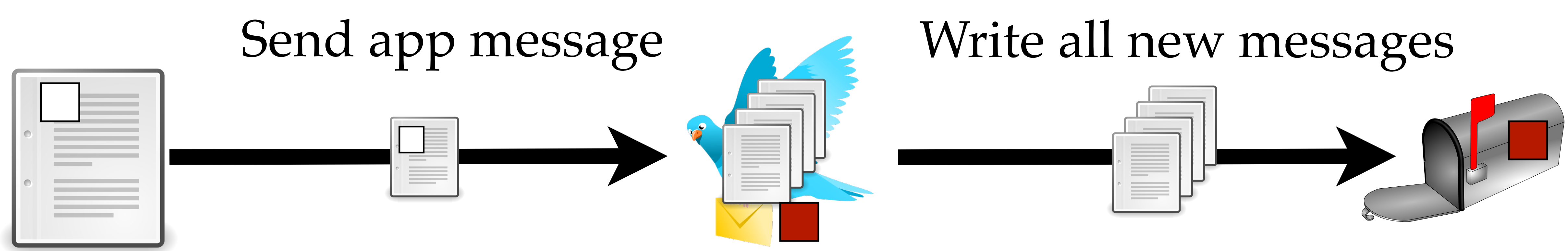
Effectively let's us reprogram ';'

- 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

Effectively let's us reprogram ';'

- 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

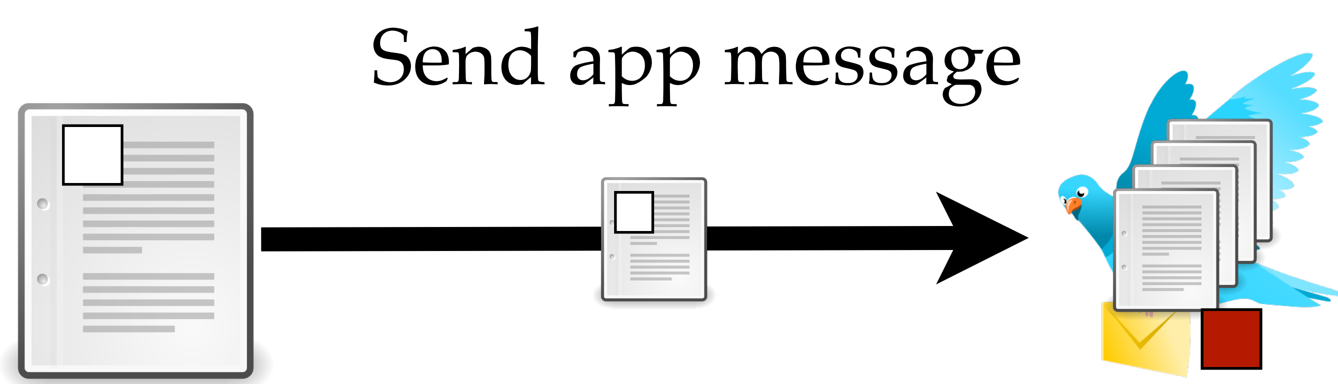
Effectively let's us reprogram ';'

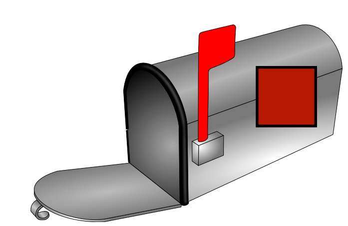
- Execute computations in `LIO` monad
  - Records context "current" label
  - I.e., tracks taint of computation
  - Restricts side-effects an app can perform
- Example (sending Bob a message):

LIO Monad

Preventing unwanted leaks

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

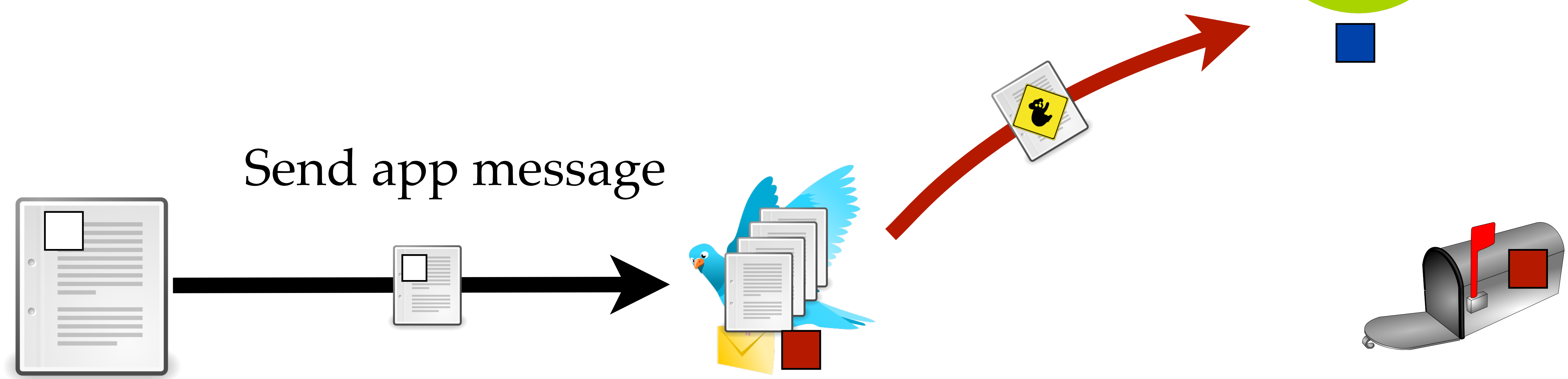




LIO Monad

Preventing unwanted leaks

```
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` “Brad Pitt”)
        alertPaparazzi
  putUserMessages user (message:messages)
```

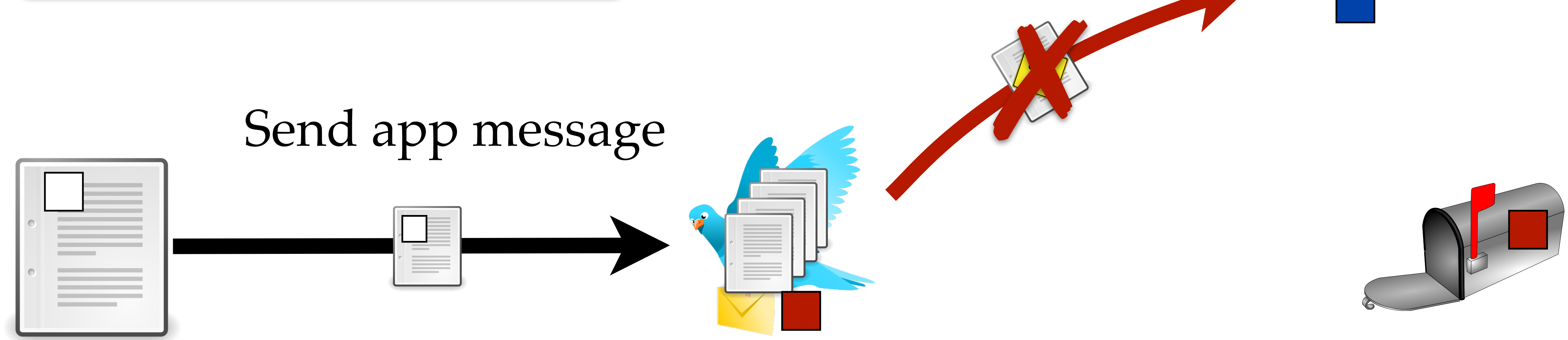


LIO Monad

Preventing unwanted leaks

```
sendMessage user message = do
  messages <- getUserMessages user
  when (messages `hasRecipient` “Brad Pitt”)
        alertPaparazzi
  putUserMessages user (message:messages)
```

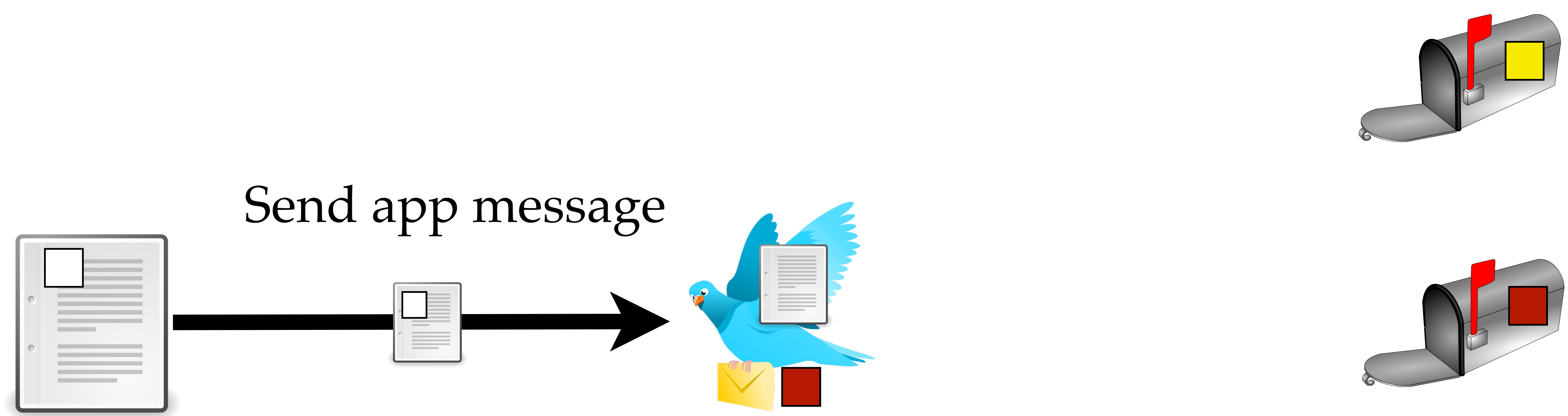
App receives exception:
Trying to leak sensitive data.



LIO Monad
Overly restrictive

- Messenger app wishes to send broadcast message

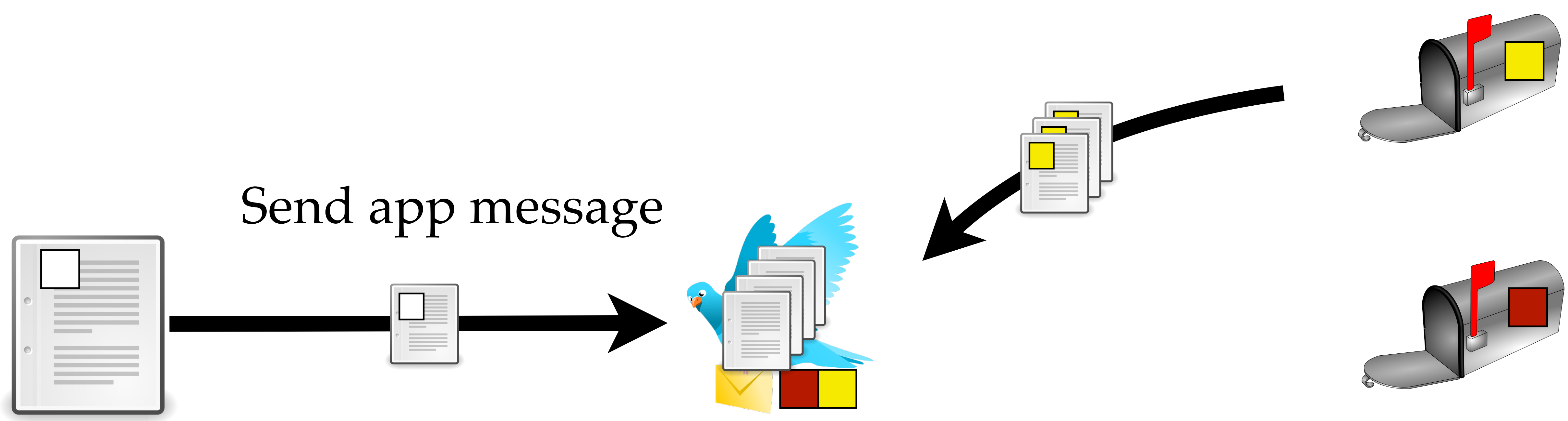
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message =  do
   forM_ users $ λuser -> sendMessage user message
```



LIO Monad
Overly restrictive

- Messenger app wishes to send broadcast message

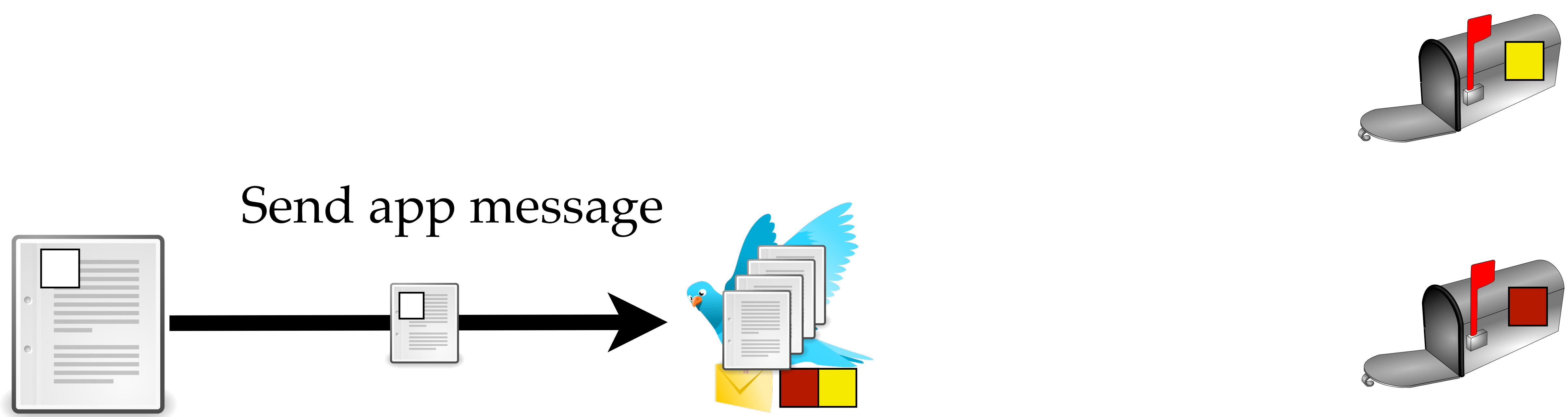
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message =  do
   forM_ users $ λuser -> sendMessage user message
```



LIO Monad
Overly restrictive

- Messenger app wishes to send broadcast message

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

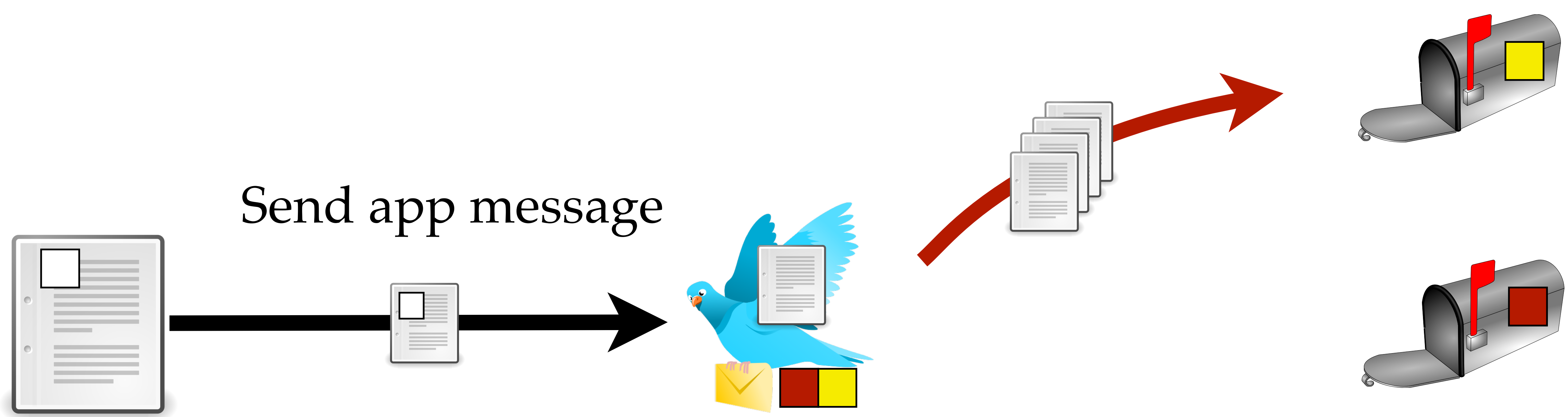


LIO Monad

Overly restrictive

- Messenger app wishes to send broadcast message

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

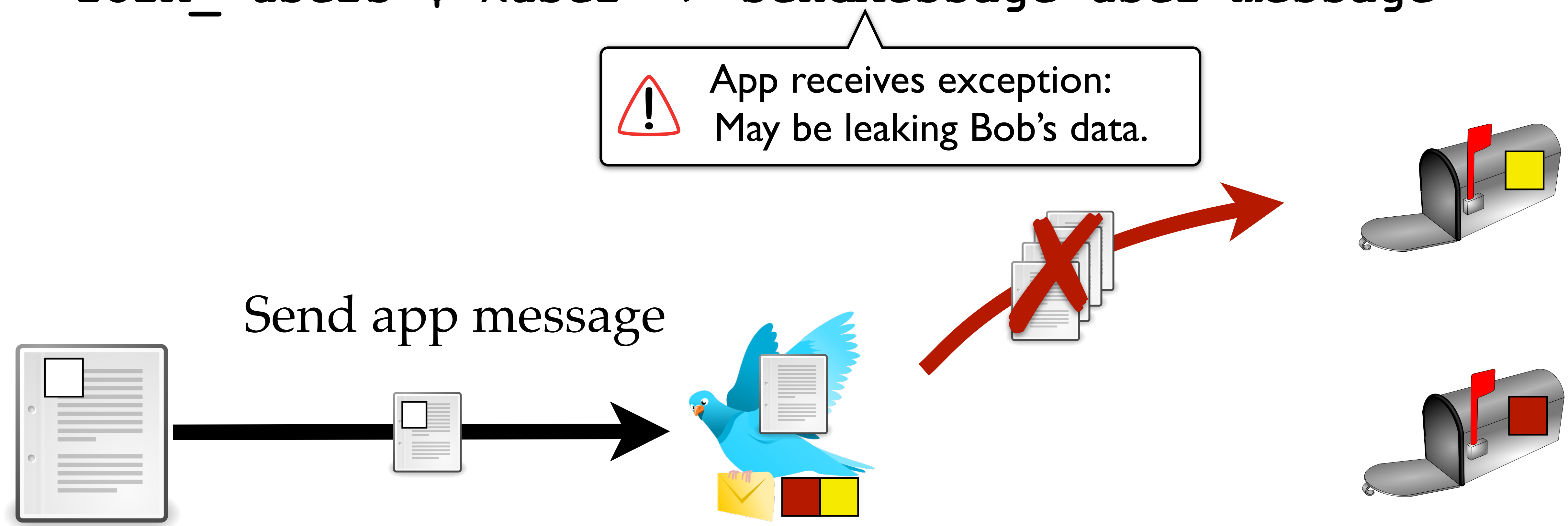


LIO Monad

Overly restrictive

- Messenger app wishes to send broadcast message

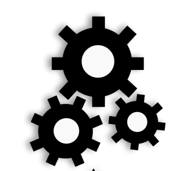
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
   forM_ users $ λuser -> sendMessage user message
```

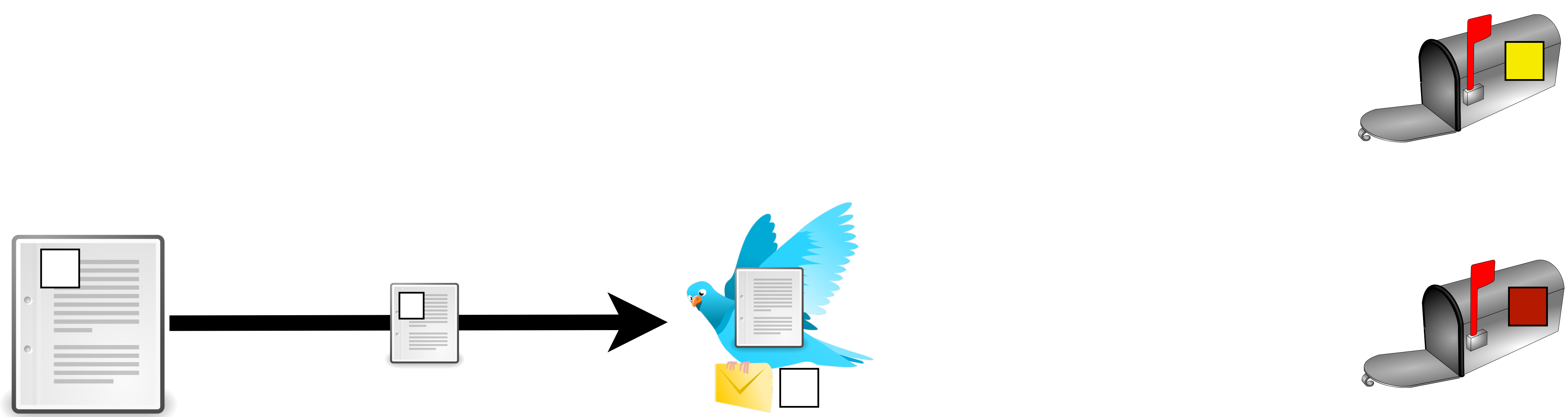


Practical Concerns

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

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

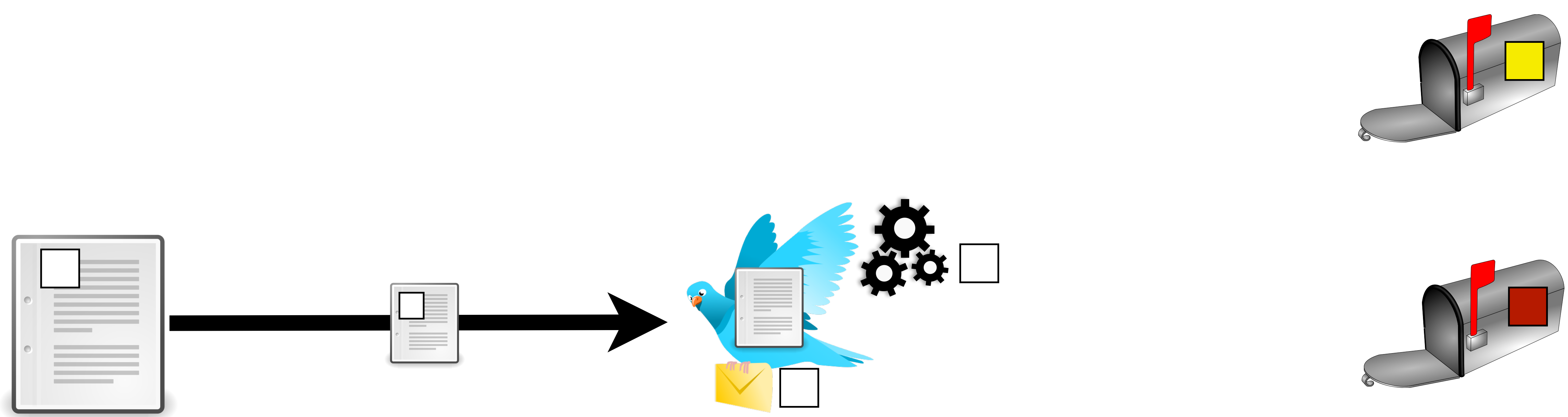




Practical Concerns

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

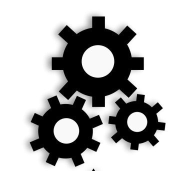
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message =  do
   forM_ users $ λuser -> discard $
                          sendMessage user message
```

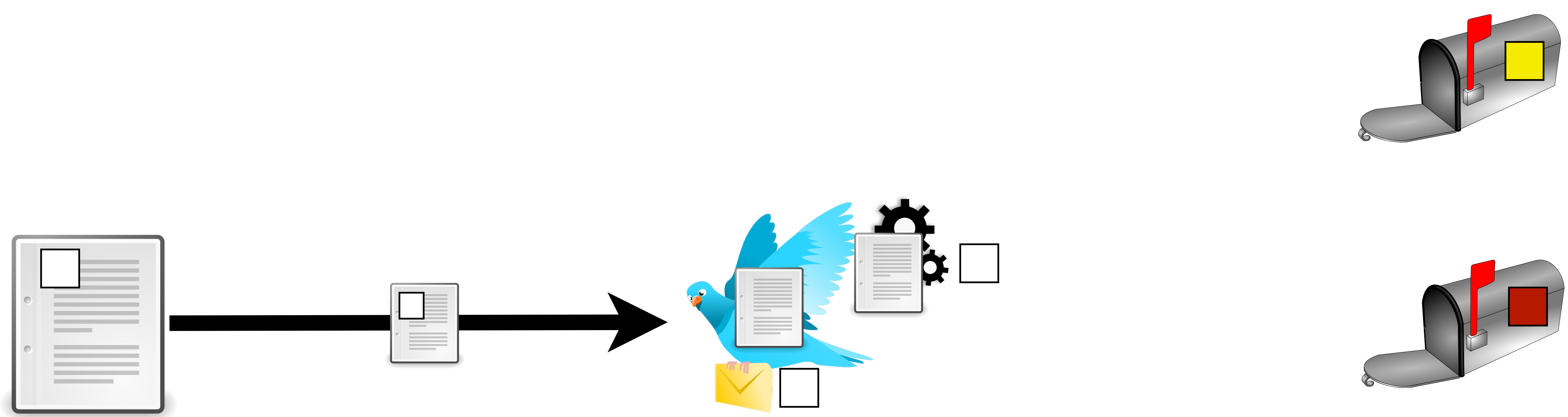


Practical Concerns

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

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

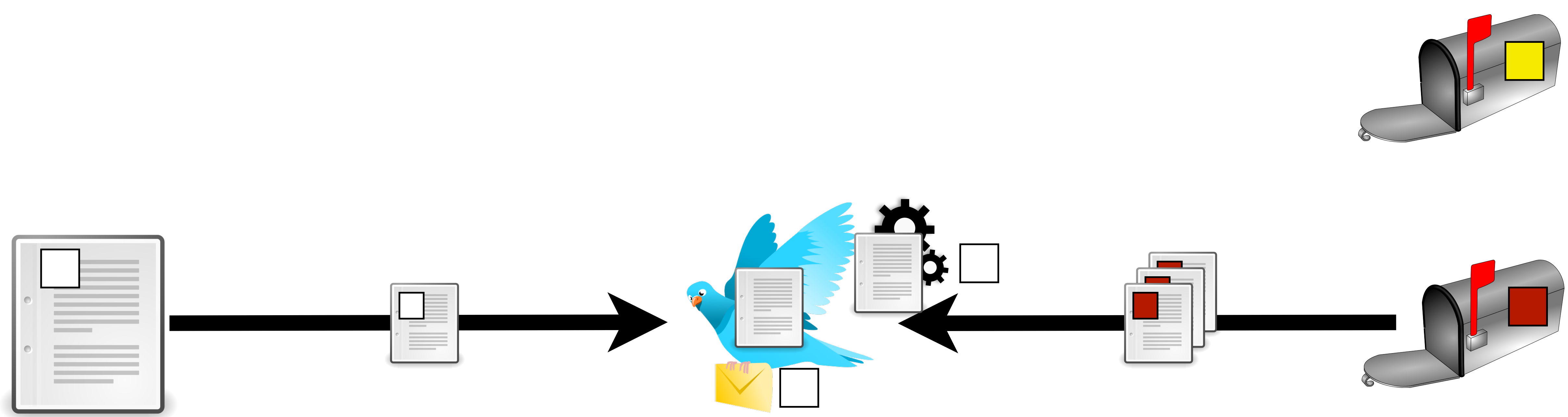




Practical Concerns

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

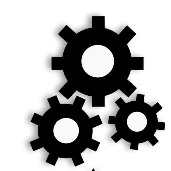
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message = do
   forM_ users $ λuser -> discard $
                           sendMessage user message
```

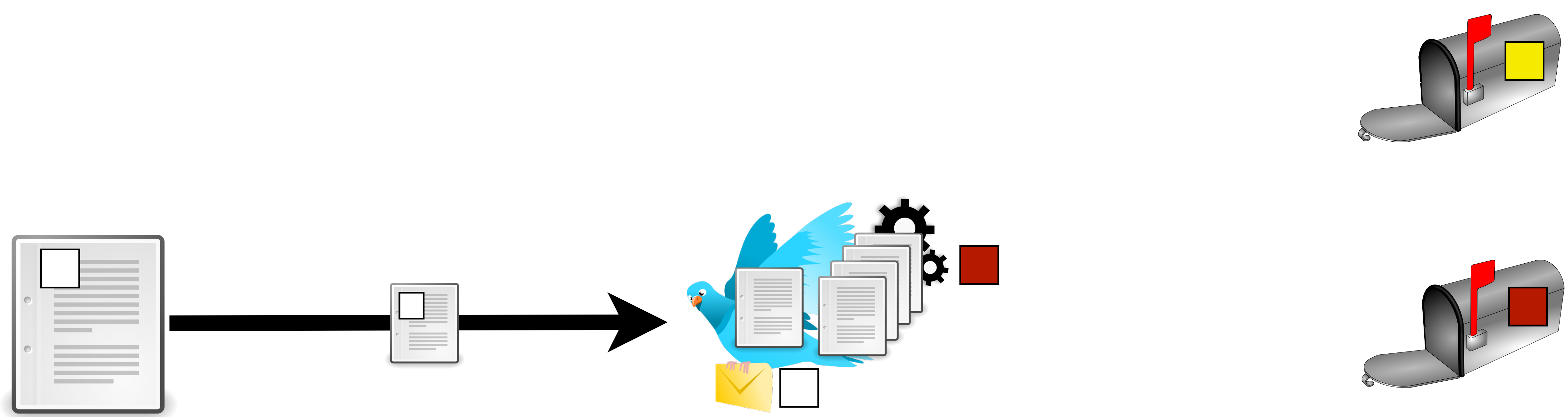


Practical Concerns

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

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

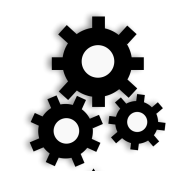


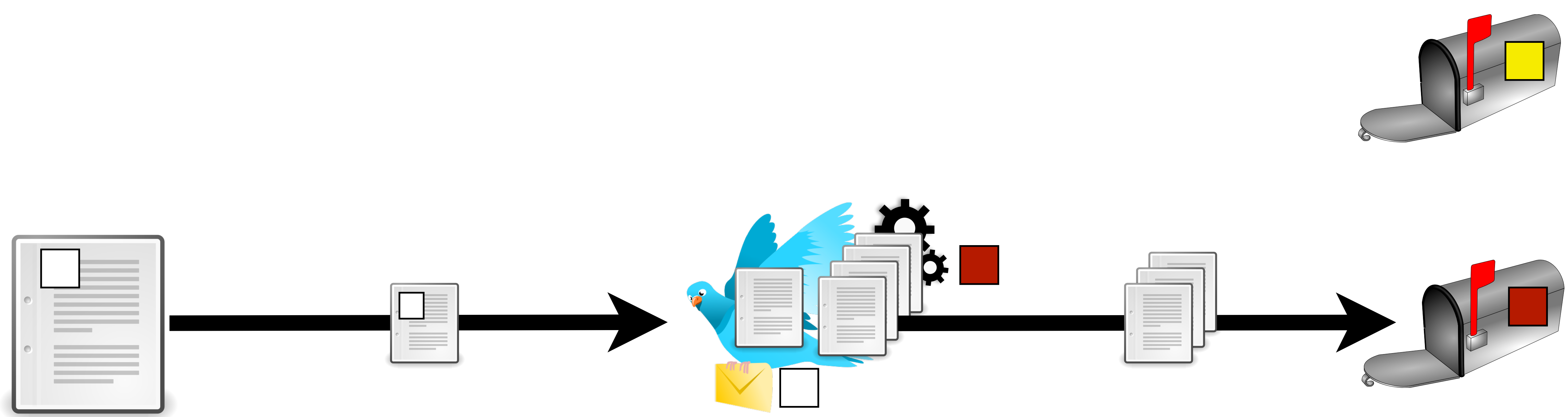


Practical Concerns

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

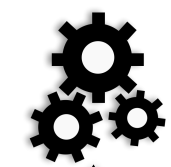
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message =  do
   forM_ users $ λuser -> discard $
                          sendMessage user message
```



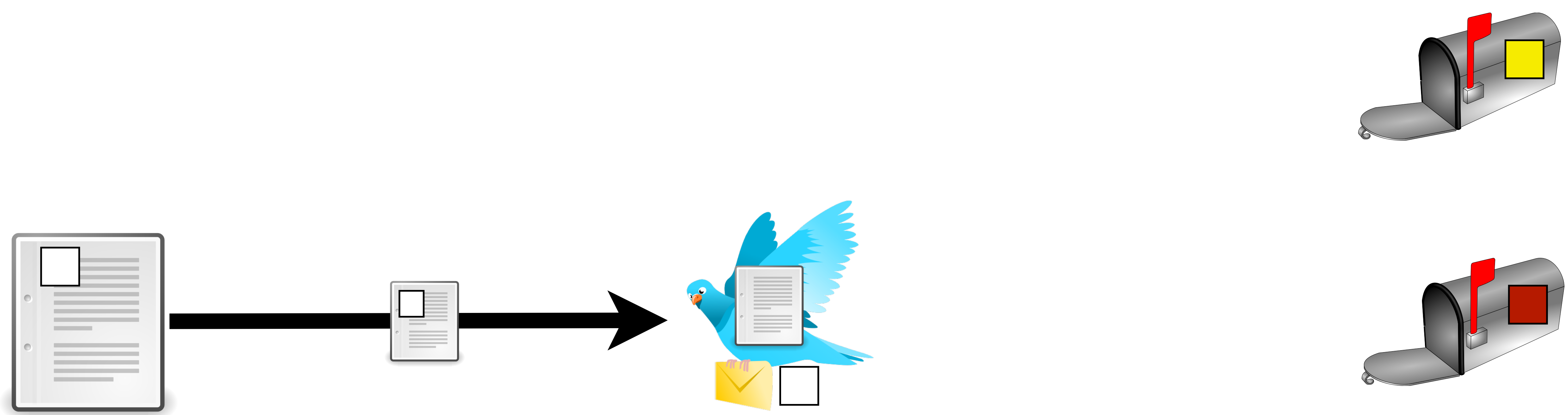


Practical Concerns

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



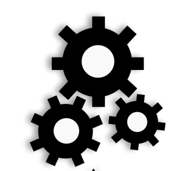
```
sendMessages :: [User] -> Message -> LIO ()
sendMessages users message =  do
   forM_ users $ λuser -> discard $
                          sendMessage user message
```

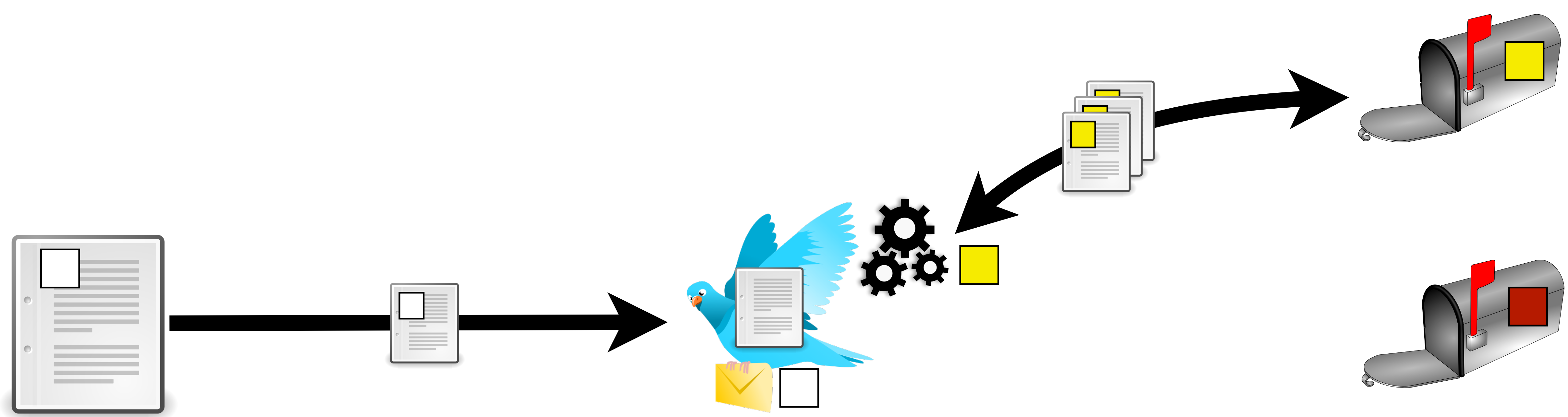


Practical Concerns

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

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





… `discard` *covertly leaks termination information.*

Termination Attack

- Leak secret bit through non-termination

```
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"
```

Write race to public channel

  - ➤ 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`**
  - + **`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

```
isOfInterest :: User -> Int -> LIO ()
isOfInterest victim n = do
  discard lFork $ 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

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

NO race to public channel

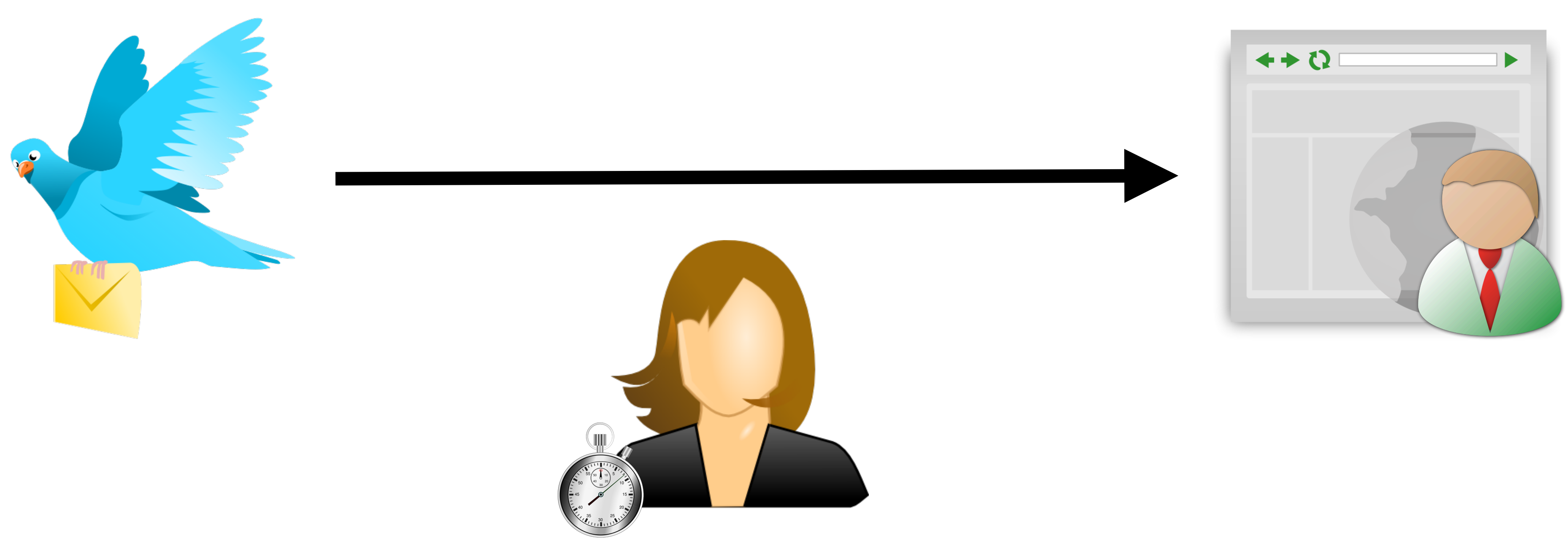
➤ Always write "es" first, then "y"

Status of LIO

- Used in production system
- Formalized as call-by-name λ-calculus
  - Support for thread spawning and joining with **`lFork`** and **`lWait`**
  - Support for mutable single-place channels
- Theorem: Termination-sensitive non-interference
  - Informally: *Confidentiality and integrity of data is preserved even if threads diverge.*

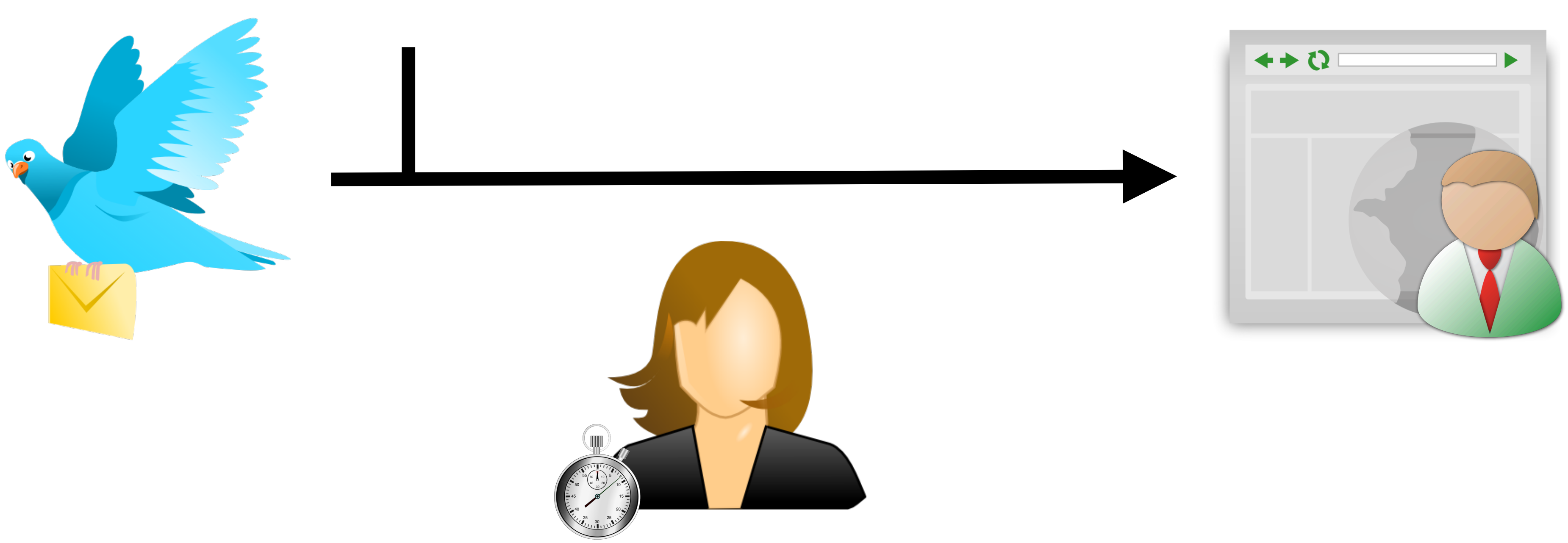
A Practical Perspective

- Covert channels closed by LIO
  - *Termination*
  - *Internal timing*
- What about external timing channel?



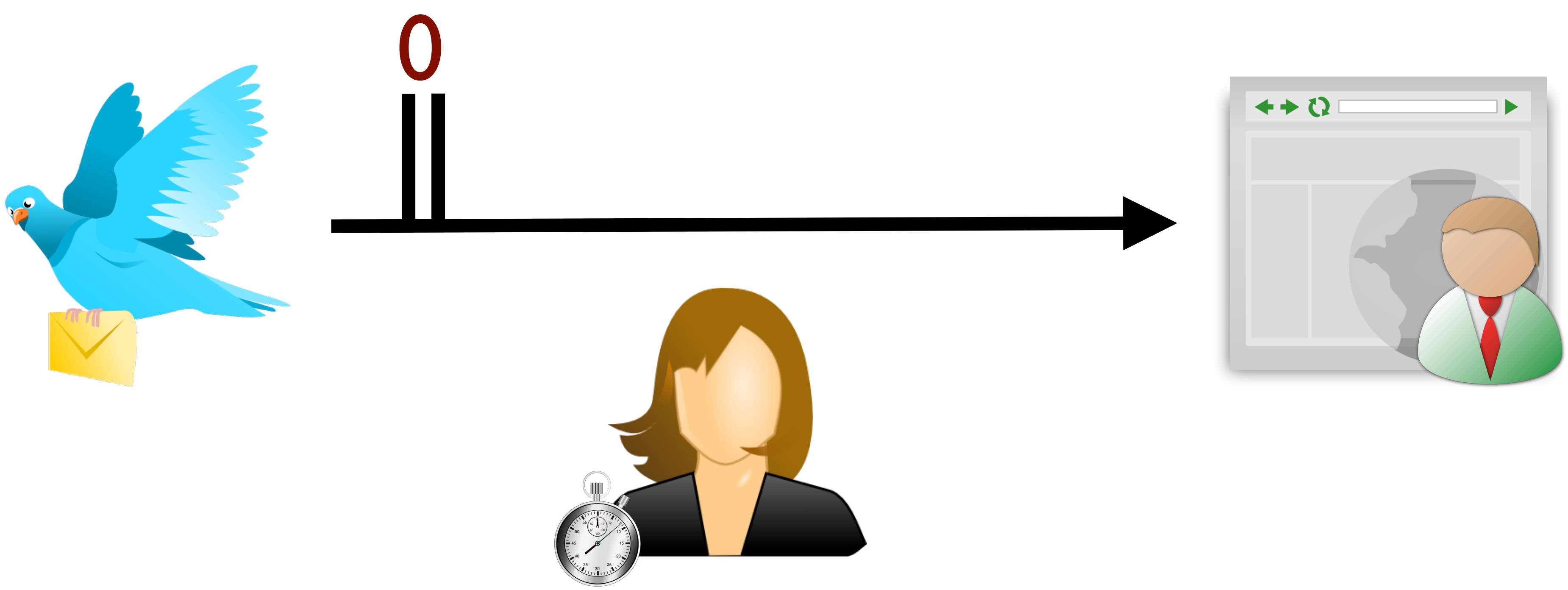
A Practical Perspective

- Covert channels closed by LIO
  - *Termination*
  - *Internal timing*
- What about external timing channel?



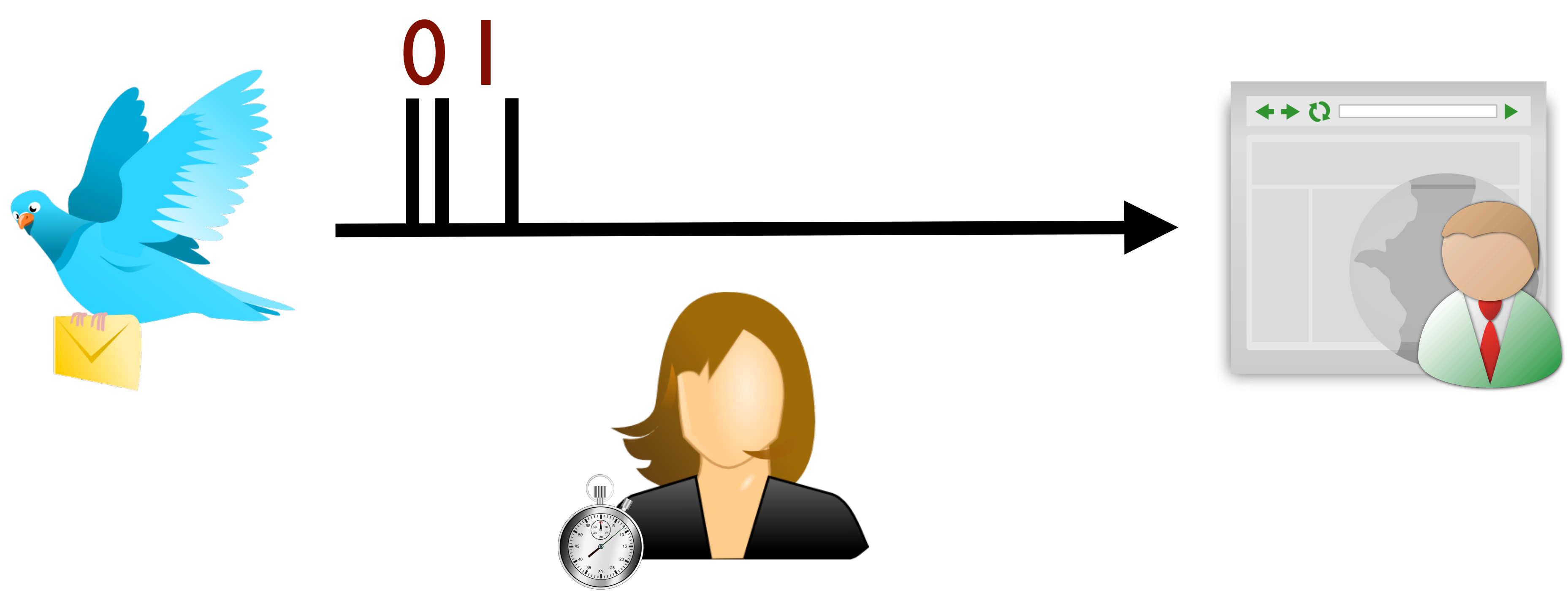
A Practical Perspective

- Covert channels closed by LIO
  - *Termination*
  - *Internal timing*
- What about external timing channel?



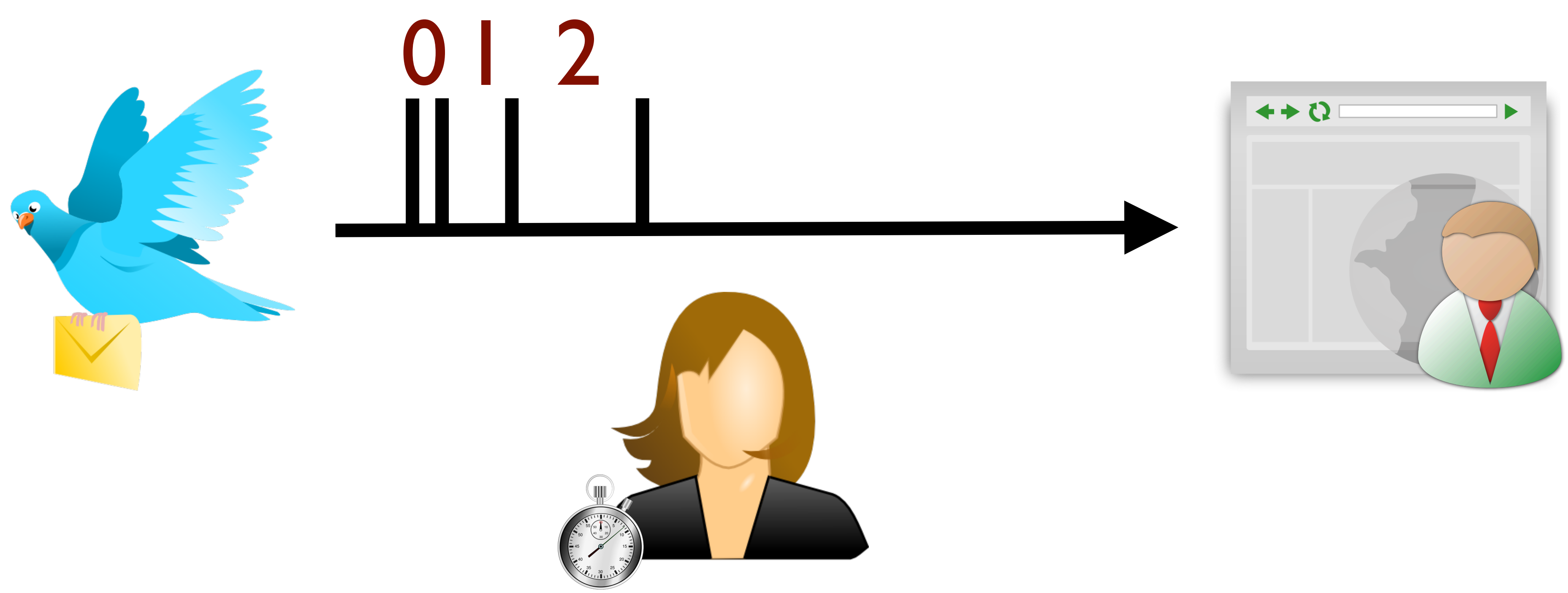
A Practical Perspective

- Covert channels closed by LIO
  - *Termination*
  - *Internal timing*
- What about external timing channel?



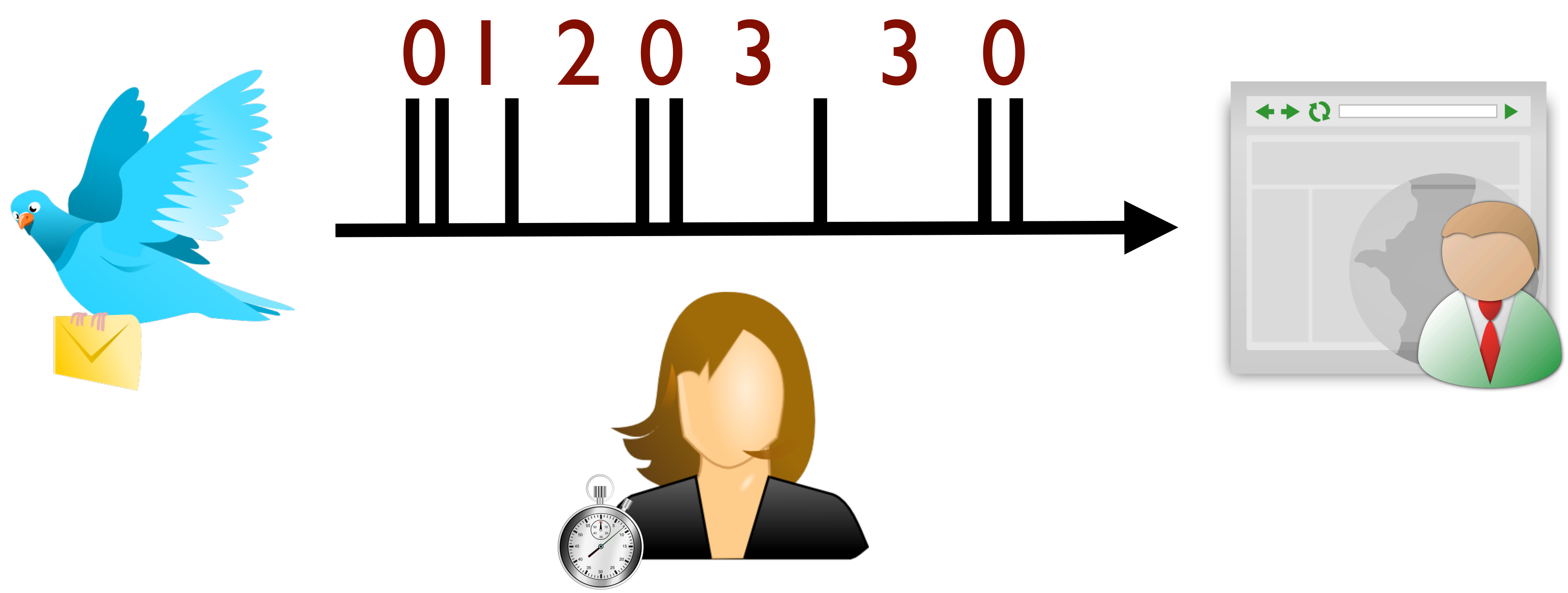
A Practical Perspective

- Covert channels closed by LIO
  - *Termination*
  - *Internal timing*
- What about external timing channel?



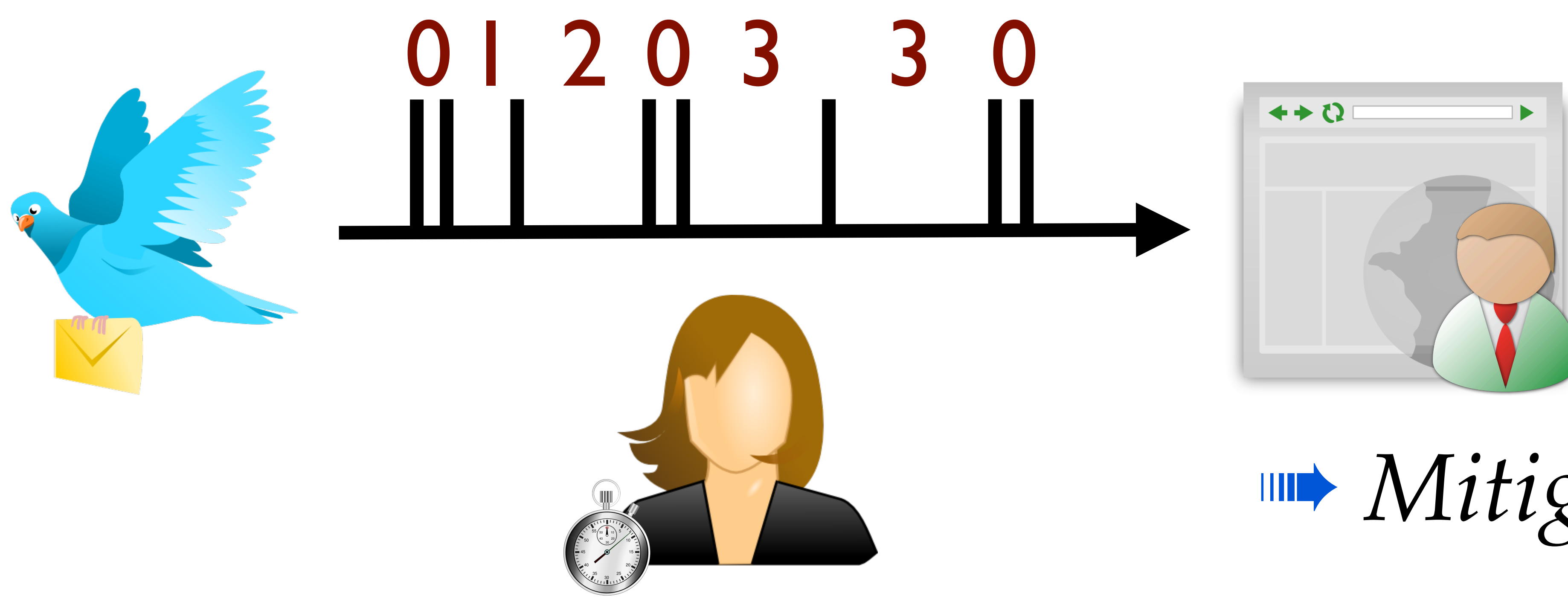
A Practical Perspective

- 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?



➠ *Mitigation*

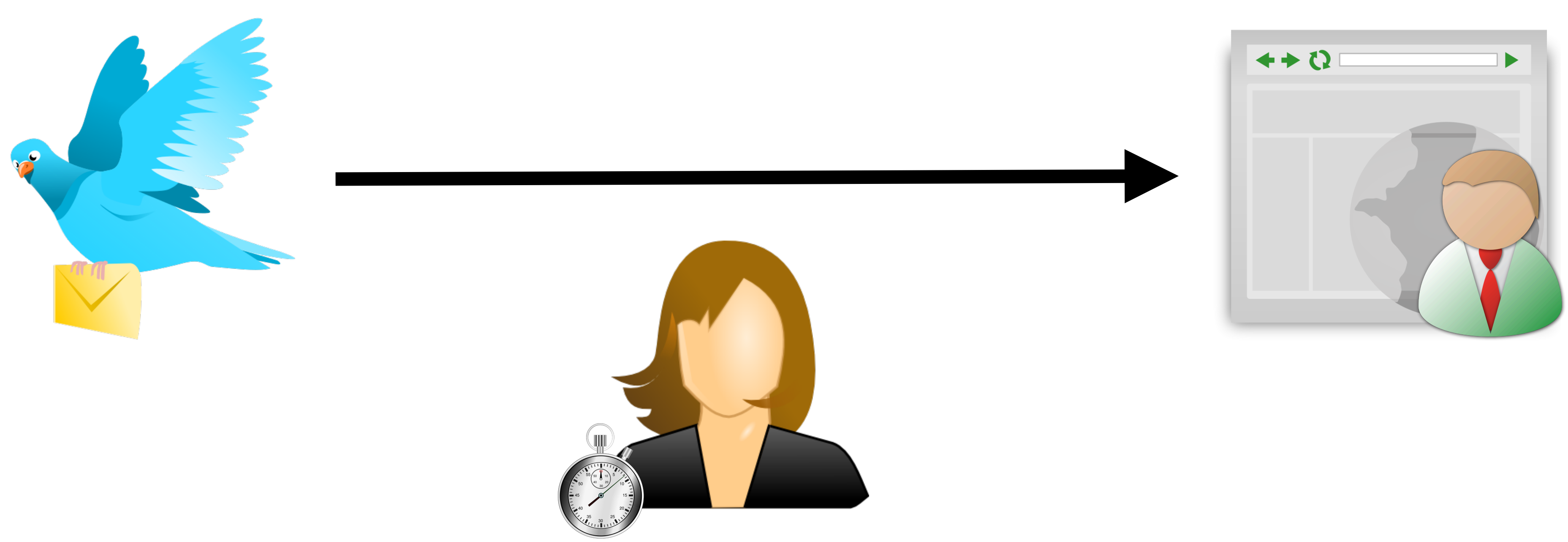
Thank you

`cabal install lio`

`http://gitstar.com/scs/lio`

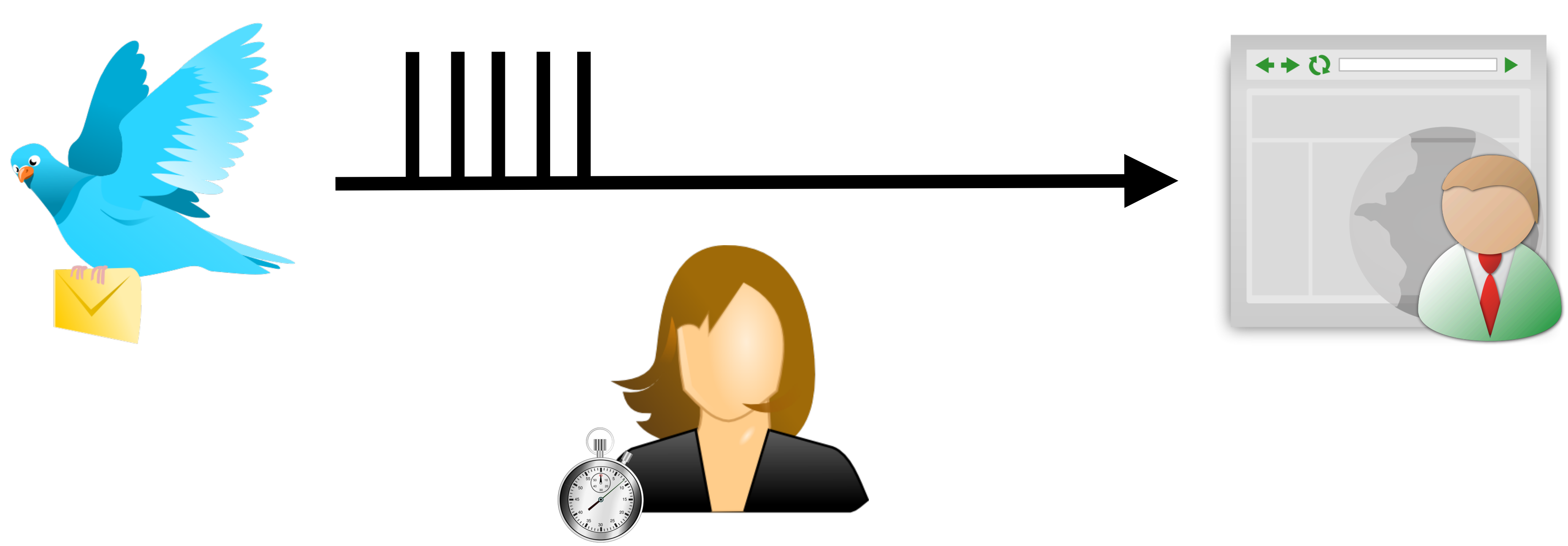
Mitigation

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



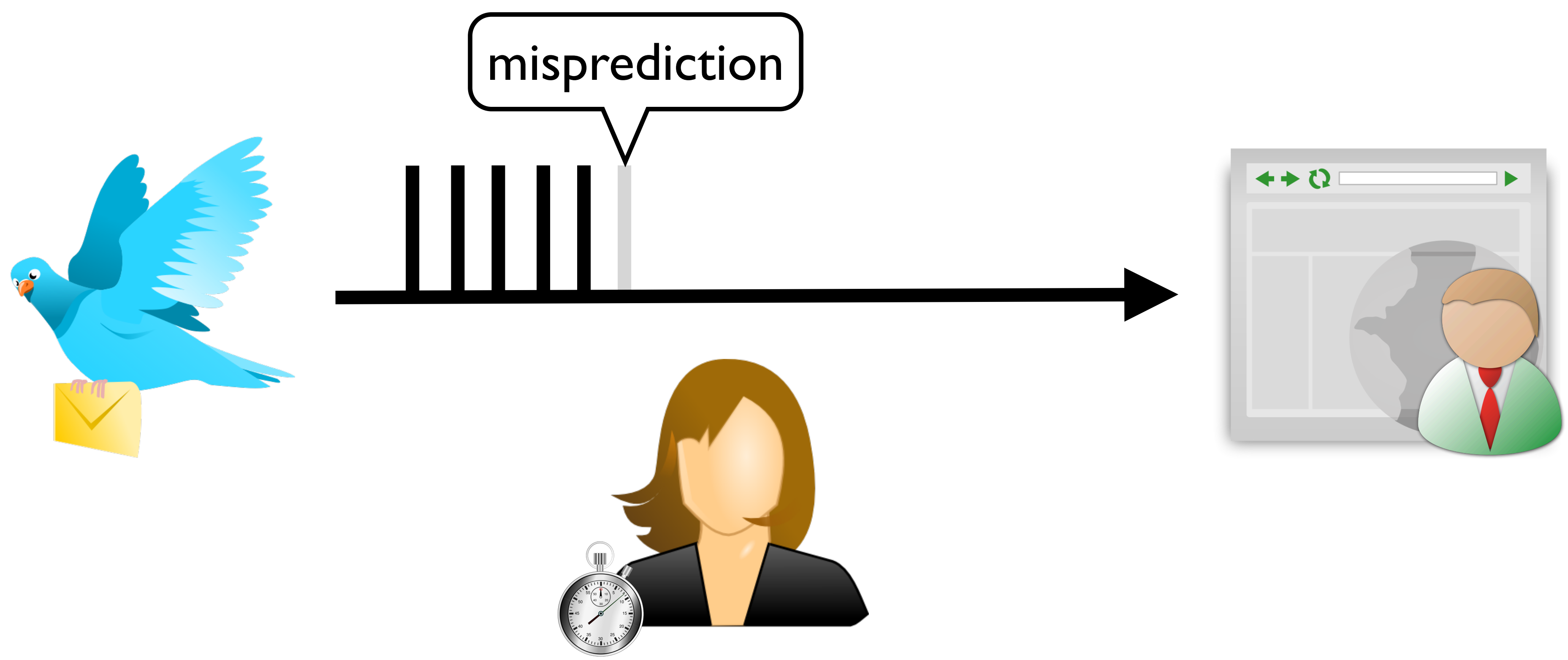
Mitigation

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



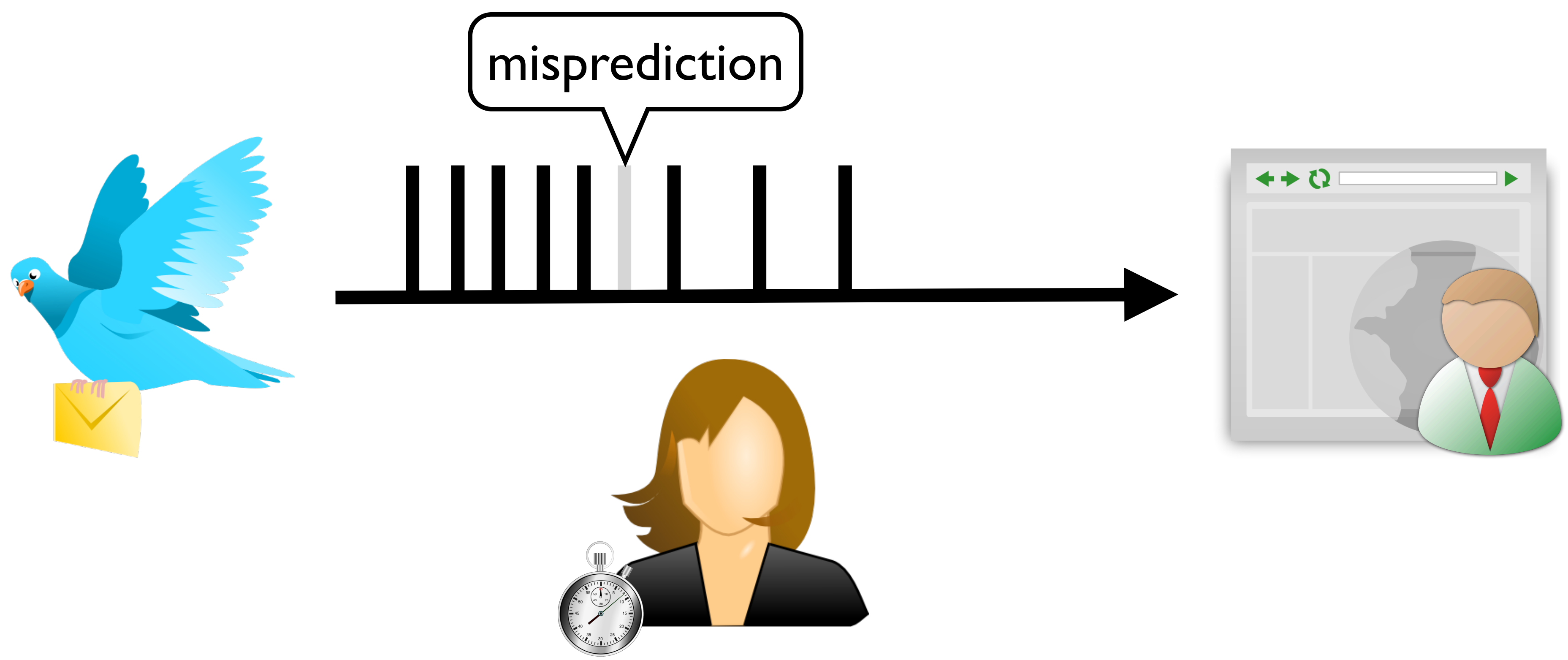
Mitigation

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



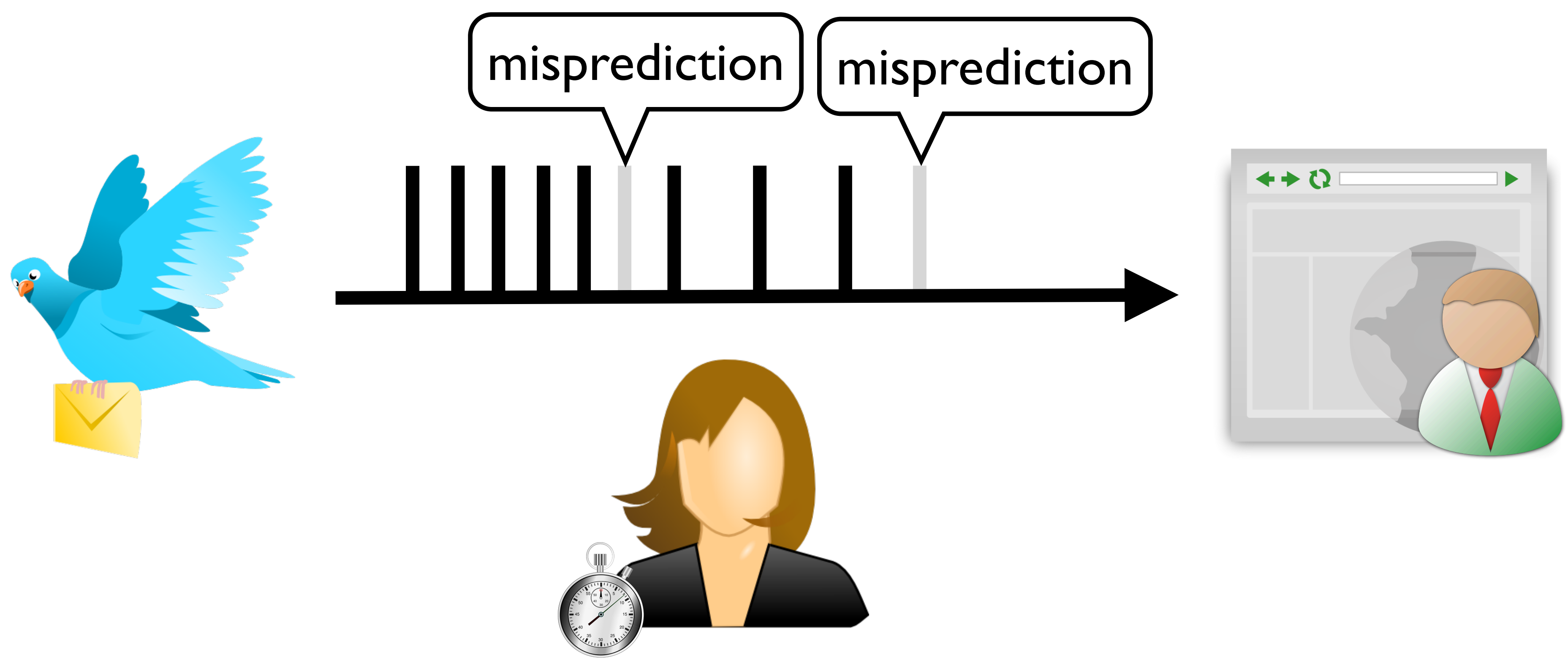
Mitigation

- 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 ➠ 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*