Addressing Covert Channels in a Concurrent Information Flow Control Language

Deian Stefan
Joint work with: Alejandro Russo, Pablo Buiras, Amit Levy, John Mitchell, and David Mazières
Information Flow Control

• Well-established approach to enforcing security
  ➤ Guarantees: preservation of confidentiality

• Suitable for executing untrusted code
  ➤ Policies specify where data can flow
  E.g., “Alice’s contacts may flow to her friends.”
  ➤ Do not need to analyze code
  E.g., “No execution of system call Y if executed X.”
Information Flow Control

Limitations & Motivation

• Limitations of IFC enforcement techniques:
  ➤ Static: inflexible when considering inherently dynamic systems (e.g., web apps and OSes)
  ➤ Dynamic: violations leaks through monitor

• Adoption setbacks:
  ➤ Lack of advanced features, including concurrency
  ➤ Covert channels
E.g., timing and cache attacks are practical!
Goal: develop an IFC-secure language

- Support concurrency constructs
- Address termination & timing covert channels
- Retain flexible programming interface

Approach: Extend existing Haskell IFC library!

- Associates a label with every piece of data: means for carrying policies associated with data
- Dynamically tracks and controls propagation of information within custom monad (LIO)
Overview of LIO

- **LIO monad**: associates a label with computation, called the “current label”

- **How is the current label used?**
  - Protects otherwise unlabeled data within scope
  - Used when enforcing flow of information to/from current context

---

*Read: Can Secret data flow to a Public computation?*  

*No, but current label is raised to allow flow!*
Overview of LIO

- **LIO monad**: associates a label with computation, called the “current label”
- How is the current label used?
  - Protects otherwise unlabeled data within scope
  - Used when enforcing flow of information to/from current context

*Write*: Can Secret computation write to Public file?

*No! Secret information may be leaked.*
Overview of LIO

Fine-grained policies

• Haskell has a strong type system \textcolor{red}{\Rightarrow} can associate label with data types (e.g., \texttt{Int})

• Example: \texttt{label Secret} \textcolor{red}{\Rightarrow} Protects document with label \texttt{Secret}
  \textcolor{red}{\Rightarrow} The current label must be no more restricting than \texttt{Secret} (e.g., fails if current label is \texttt{Top-Secret})
Overview of LIO
Fine-grained policies

• How do we use/inspect Labeled values?

• Example: unlabel
  ➤ Current label raised (if necessary) to level Secret
  ➤ Note: current label is upper bound on observed data
    preserves confidentiality of now-unlabeled data
Overview of LIO

Avoiding the label creep

• Reading sensitive data \(\Rightarrow\) raises current label
  ➢ Cannot lower current label (would violate IFC)
  ➢ Computation cannot write to “useful” channels

• toLabeled is used to prevent such label creeps!

• Example: toLabeled Secret 
  ➢ Execute:
    ➢ Return:

  Note: computation current label is unchanged
Overview of LIO

Avoiding the label creep

toLabeled Secret

Public

Secret

Unlabel & raise current label when value is needed!
LIO Security Guarantees

• Termination-insensitive non-interference
  ➢ Informally: If a program terminates, then confidentiality of data is preserved
  ➢ Standard & provided by Jif, FlowCaml, etc.

• Why only termination-insensitive?
  ➢ toLabeled susceptible to termination attacks!
Termination Attack

- Leaking a Secret-Labeled bit \( sb \):

\[
\text{bitLeak } sb = \text{ do } \\
\quad \text{outputPublic "bit is..." } \\
\quad \text{toLabeled Secret } $ \text{ do bit } \leftarrow \text{ unlabel sb } \\
\quad \quad \text{when } (bit == 1) \perp \\
\quad \text{outputPublic "0"}
\]
Adding Fire

• Recall first goal: *Support concurrency constructs*

• Suppose we add simple *fork* primitive...
Termination Attack v. 2

- Leaking a Secret-Labeled, public-bounded Int si:
  \[
  \text{intLeak } \text{si } \text{nrbits} = \text{forM\_} [0..\text{nrbits-1}] \ \lambda n \rightarrow \text{fork } \rightarrow \text{do }
  \text{outputPublic (“\text{nbit “ ++ show n ++ “ is...”) }
  \text{toLabeled Secret } \rightarrow \text{do int } \leftarrow \text{unlabel si }
  \quad \text{let bit } = \text{extractBit int n }
  \quad \text{when (bit == 1) } \perp
  \text{outputPublic “0”}
  \]
Internal Timing Attack

- Leaking a Secret-Labeled bit sb:

```haskell
bitLeak sb = do
  fork $ do toLabeled Secret $ do bit ← unlabel sb
            when (bit == 1) $ sleep 5
    outputPublic "T"
  fork $ do sleep 1
    outputPublic "F"
```
Fundamental Problem

- The possibly sensitive execution of a `toLabeled` computation affects when/if next non-sensitive step executes.

```
1. publicOutputOrMeasurement
2. toLabeled Secret computeOnSecretData
3. publicOutputOrMeasurement

Approach: Must have “pre-determined” execution time!
Solution: Decouple execution.
```

$\Delta t$
Solution: Threads
Fighting fire with fire

• Decoupling toLabeled computations
  ➢ Spawn new thread to execute sub-computation
  ➢ Immediately return a labeled “handle” to thread

• Concurrent LIO:
  ➢ lFork: used to spawn new labeled thread.
  ➢ lWait: forces evaluation of thread, but first raises current label to be at least as restrictive as the thread label.
Overview of IFork & IWait

IFork Secret

Public Secret

IWait

Has been raised!
Termination Attack

First Attempt

• Leaking a Secret-Labeled bit sb:

\[
\text{bitLeak } sb = \text{ do }
\]

\[
\text{outputPublic } \text{“bit is...” }
\]

\[
\text{IFork Secret }$ \text{ do bit } \leftarrow \text{ unlabel sb }
\]

\[
\text{when (bit } == 1) \perp
\]

\[
\text{outputPublic } \text{“0”}
\]
Termination Attack

Second Attempt

- Leaking a Secret-Labeled bit sb:

  \[
  \text{bitLeak } sb = \text{do} \quad \text{outputPublic } \text{“bit is...”} \\
  h \leftarrow \text{IFork } \text{Secret } $ \text{do} \text{ bit } \leftarrow \text{unlabel } sb \\
  \quad \text{when } (\text{bit } == 1) \perp \\
  \text{outputPublic } \text{“0”} \\
  \text{IWait } h
  \]
Internal Timing Attack
Attempt

- Leaking a Secret-Labeled bit sb:

```plaintext
bitLeak sb = do
  IFork Pub $ do IFork Secret $ do bit ← unlabel sb
  when (bit == 1) $ sleep 5
  outputPublic "T"
  IFork Pub $ do sleep 1
  outputPublic "F"
```

Diagram:
- bitLeak 0 → Public
- bitLeak 1 → Public
Scalability

- Performance impact of forking new threads
  - Minimal: Haskell’s threads are light-weight!

![Graph showing Overhead of Fork & Wait vs toLabeled](image)
Guarantees & Limitations

- Formalized concurrent LIO as call-by-name λ-calculus
  - Added support for communication primitives
  - Proved termination-sensitive non-interference

- Do not address covert channels outside API
  - Cache timing attacks
  - Leaks through memory exhaustion
Thank you!

$ cabal install lio