Multi-Ejecución Segura en Haskell

Arroyo Joaquin

Universidad Nacional de Rosario
Licenciatura en Ciencias de la Computación
Seguridad Informática

25 de febrero de 2025

Índice

Introducción

Se presenta un mecanismo de *seguridad del flujo de información basada en el lenguaje* para garantizar la **No Interferencia** en sistemas de cómputo.

Introducción

Se presenta un mecanismo de *seguridad del flujo de información basada en el lenguaje* para garantizar la **No Interferencia** en sistemas de cómputo.

Estos mecanismos tradicionalmente se realizan de forma:

- Estática (Sistemas de Tipos)
- Dinámica (Monitores en *runtime*)
- Combinación de ambas

Introducción

Se presenta un mecanismo de *seguridad del flujo de información basada en el lenguaje* para garantizar la **No Interferencia** en sistemas de cómputo.

Estos mecanismos tradicionalmente se realizan de forma:

- Estática (Sistemas de Tipos)
- Dinámica (Monitores en *runtime*)
- Combinación de ambas

Se presenta como novedad el uso del modelo de **Multi-Ejecución Segura** como mecanismo.

Introducción

Se presenta un mecanismo de *seguridad del flujo de información basada en el lenguaje* para garantizar la **No Interferencia** en sistemas de cómputo.

Estos mecanismos tradicionalmente se realizan de forma:

- Estática (Sistemas de Tipos)
- Dinámica (Monitores en *runtime*)
- Combinación de ambas

Se presenta como novedad el uso del modelo de **Multi-Ejecución Segura** como mecanismo.

Este es implementado en una biblioteca en Haskell.

Índice

No Interferencia

Para explicar coloquialmente el concepto de **No Interferencia** necesitamos:

No Interferencia

Para explicar coloquialmente el concepto de **No Interferencia** necesitamos:

- Un grupo de Usuarios U
- Un par de niveles de seguridad L y H
- Una categorización de U en dos grupos G_L y G_H.

No Interferencia

Para explicar coloquialmente el concepto de **No Interferencia** necesitamos:

- Un grupo de Usuarios U
- Un par de niveles de seguridad L y H
- Una categorización de U en dos grupos G_L y G_H.

Entonces, G_H **no interfiere** con G_L si toda acción realizada por usuarios de G_H no repercute sobre lo que pueden ver los usuarios de G_L.

Dicho de otra forma para G_L, G_H es **invisible**.

Índice

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Utiliza un **Retículo de Seguridad** L, donde los niveles están ordenados por una relación de orden parcial R.

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Utiliza un **Retículo de Seguridad** L, donde los niveles están ordenados por una relación de orden parcial R.

Se basa principalmente en los siguienes puntos:

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Utiliza un **Retículo de Seguridad** L, donde los niveles están ordenados por una relación de orden parcial R.

Se basa principalmente en los siguienes puntos:

- El programa se ejecuta una vez por cada nivel de seguridad.

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Utiliza un **Retículo de Seguridad** L, donde los niveles están ordenados por una relación de orden parcial R.

Se basa principalmente en los siguienes puntos:

- El programa se ejecuta una vez por cada nivel de seguridad.
- Se controlan las operaciones de **E/S**.

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Utiliza un **Retículo de Seguridad** L, donde los niveles están ordenados por una relación de orden parcial R.

Se basa principalmente en los siguienes puntos:

- El programa se ejecuta una vez por cada nivel de seguridad.
- Se controlan las operaciones de **E/S**.
- Se garantiza **Solidez**.

Multi-Ejecución Segura

Este modelo es presentado a partir del enfoque de Devriese y Piessens. Dicho enfoque garantiza la **No Interferencia**.

Utiliza un **Retículo de Seguridad** L, donde los niveles están ordenados por una relación de orden parcial R.

Se basa principalmente en los siguienes puntos:

- El programa se ejecuta una vez por cada nivel de seguridad.
- Se controlan las operaciones de **E/S**.
- Se garantiza **Solidez**.
- Se garantiza **Robustez**.

Índice

SME en Haskell

Lo primero a resaltar es la forma en que se representan las computaciones que realizan **Efectos Secundarios** en los lenguajes funcionales puros:

SME en Haskell

Lo primero a resaltar es la forma en que se representan las computaciones que realizan **Efectos Secundarios** en los lenguajes funcionales puros: **Tipos**.

SME en Haskell

Lo primero a resaltar es la forma en que se representan las computaciones que realizan **Efectos Secundarios** en los lenguajes funcionales puros: **Tipos**.

Esta forma de identificar dichas computaciones se adapta a **SME** ya que permite dar interpretaciones distintas según nivel de seguridad.

SME en Haskell

Lo primero a resaltar es la forma en que se representan las computaciones que realizan **Efectos Secundarios** en los lenguajes funcionales puros: **Tipos**.

Esta forma de identificar dichas computaciones se adapta a **SME** ya que permite dar interpretaciones distintas según nivel de seguridad.

```
f :: Int -> IO Int
f x = print x >> return x
```

SME en Haskell

Lo primero a definir en la implementación fue el **Retículo de Seguridad**.

SME en Haskell

Lo primero a definir en la implementación fue el **Retículo de Seguridad**.

Para simplificar se consideró un retículo finíto S con solo dos elementos: L y H

SME en Haskell

Lo primero a definir en la implementación fue el **Retículo de Seguridad**.

Para simplificar se consideró un retículo finíto S con solo dos elementos: L y H

```
data Level = L | H
```

Y se definió una relación de orden $\prec$ sobre estos elementos, donde $L \prec H$

SME en Haskell

Lo primero a definir en la implementación fue el **Retículo de Seguridad**.

Para simplificar se consideró un retículo finíto S con solo dos elementos: L y H

```
data Level = L | H
```

Y se definió una relación de orden $\prec$ sobre estos elementos, donde $L \prec H$

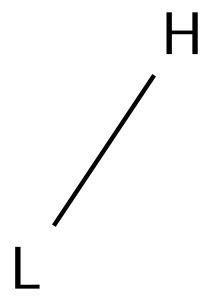

Figura: Diagrama de Hasse de S

SME en Haskell

Luego, se buscó reemplazar las acciones de **E/S** convencionales, por una descripción **pura** de ellas.

SME en Haskell

Luego, se buscó reemplazar las acciones de **E/S** convencionales, por una descripción **pura** de ellas.

Para esto se definió el tipo de datos *ME* el cuál fue instanciado como **mónada**:

SME en Haskell

Luego, se buscó reemplazar las acciones de **E/S** convencionales, por una descripción **pura** de ellas.

Para esto se definió el tipo de datos *ME* el cuál fue instanciado como **mónada**:

```
data ME a = Return a
| Write FilePath String (ME a)
| Read FilePath (String -> ME a)
```

SME en Haskell

Luego, se buscó reemplazar las acciones de **E/S** convencionales, por una descripción **pura** de ellas.

Para esto se definió el tipo de datos *ME* el cuál fue instanciado como **mónada**:

```
data ME a = Return a
| Write FilePath String (ME a)
| Read FilePath (String -> ME a)
```

Notar que solo se definieron como acciones de **E/S** a la lectura y escritura sobre archivos.

SME en Haskell

Luego, se buscó reemplazar las acciones de **E/S** convencionales, por una descripción **pura** de ellas.

Para esto se definió el tipo de datos *ME* el cuál fue instanciado como **mónada**:

```
data ME a = Return a
| Write FilePath String (ME a)
| Read FilePath (String -> ME a)
```

Notar que solo se definieron como acciones de **E/S** a la lectura y escritura sobre archivos.

La idea es que los usuarios contruyan programas seguros a partir de la interfaz que ofrece la **mónada** (`return` y >>=).

SME en Haskell

Luego, se buscó reemplazar las acciones de **E/S** convencionales, por una descripción **pura** de ellas.

Para esto se definió el tipo de datos *ME* el cuál fue instanciado como **mónada**:

```
data ME a = Return a
| Write FilePath String (ME a)
| Read FilePath (String -> ME a)
```

Notar que solo se definieron como acciones de **E/S** a la lectura y escritura sobre archivos.

La idea es que los usuarios contruyan programas seguros a partir de la interfaz que ofrece la **mónada** (`return` y >>=).

Y además que utilicen dos funciones: `readFile` y `writeFile`.

Índice

Intérprete para *ME*

En este punto tenemos definido:

Intérprete para *ME*

En este punto tenemos definido:

- Retículo de Seguridad *Level*
- Tipo *ME*, el cuál nos permite escribir programas

Intérprete para *ME*

En este punto tenemos definido:

- Retículo de Seguridad *Level*
- Tipo *ME*, el cuál nos permite escribir programas

Ahora, se necesita un intérprete de dichos programas, que actúe acorde a la especificación de **SME**.

Intérprete para *ME*

En este punto tenemos definido:

- Retículo de Seguridad *Level*
- Tipo *ME*, el cuál nos permite escribir programas

Ahora, se necesita un intérprete de dichos programas, que actúe acorde a la especificación de **SME**.

Este fue implementado a partir de la función `run`.

Intérprete para *ME*

En este punto tenemos definido:

- Retículo de Seguridad *Level*
- Tipo *ME*, el cuál nos permite escribir programas

Ahora, se necesita un intérprete de dichos programas, que actúe acorde a la especificación de **SME**.

Este fue implementado a partir de la función `run`.

```
run :: Level → ChanMatrix → ME a → IO a
run l _ (Return a) = return a
run l c (Write file o t)
  | level file ≡ l   = do IO.writeFile file o
                          run l c t
  | otherwise        = run l c t
run l c (Read file f)
  | level file ≡ l   = do x ← IO.readFile file
                          broadcast c l file x
                          run l c (f x)
  | level file ⊏ l   = do x ← reuseInput c l file
                          run l c (f x)
  | otherwise        = run l c (f (defvalue file))
```

Figura: Intérprete para la mónada *ME*

Índice

Orquestador

Por último, necesitamos ejecutar los programas con el intérprete bajo la **Multi-ejecución Segura**.

Orquestador

Por último, necesitamos ejecutar los programas con el intérprete bajo la **Multi-ejecución Segura**.

De esto se encarga la función `sme`:

Orquestador

Por último, necesitamos ejecutar los programas con el intérprete bajo la **Multi-ejecución Segura**.

De esto se encarga la función `sme`:

```
sme :: ME a → IO ()
sme t = do
  c ← newChanMatrix
  l ← newEmptyMVar
  h ← newEmptyMVar
  forkIO (do run L c t; putMVar l ())
  forkIO (do run H c t; putMVar h ())
  takeMVar l; takeMVar h
```

Figura: Multi-Ejecución Segura

Orquestador

Por último, necesitamos ejecutar los programas con el intérprete bajo la **Multi-ejecución Segura**.

De esto se encarga la función `sme`:

```
sme :: ME a → IO ()
sme t = do
  c ← newChanMatrix
  l ← newEmptyMVar
  h ← newEmptyMVar
  forkIO (do run L c t; putMVar l ())
  forkIO (do run H c t; putMVar h ())
  takeMVar l; takeMVar h
```

Figura: Multi-Ejecución Segura

El hilo principal queda bloqueado hasta que ambos ejecuciones de `t` escriban en sus variables de sincronización.

Índice

Conclusiones

- Se implementó SME en una biblioteca de aproximadamente 130 lineas.

Conclusiones

- Se implementó SME en una biblioteca de aproximadamente 130 lineas.
- Se introdujo como novedad la utilización de canales de comunicación indexados por niveles de seguridad.

Conclusiones

- Se implementó SME en una biblioteca de aproximadamente 130 lineas.
- Se introdujo como novedad la utilización de canales de comunicación indexados por niveles de seguridad.
- Si bien se implementó SME para dos niveles de seguridad y dos operaciones de **E/S**, sus respectivas extensiones son posibles.

¿Dudas?