Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Licenciatura en Ciencias de la Computación

Papers

Seguridad Informática

Joaquin Arroyo

2025

1. Secure Multi-Execution in Haskell

1.1. Resumen

En este paper se presenta un mecanismo de seguridad del flujo de información basada en el lenguaje para garantizar la propiedad de No Interferencia en sistemas en linea. Este tipo de mecanismos tradicionalmente se realizan de manera estática (sistemas de tipos), dinámica (monitores en tiempo real) o una combinación de ambas. En este paper se propone la utilización del concepto de Multi Ejecución Segura (SME) como mecanismo. Este concepto básicamente propone un programa varias veces, una por cada nivel de seguridad, utilizando reglas especiales para las operaciones de **E/S**. En comparación con los otros métodos, en este caso no debemos inspeccionar el código completo de la aplicación, solo sus operaciones de **E/S**. En lugar de diseñar nuevos lenguajes desde cero, la seguridad también puede proporcionarse mediante bibliotecas, y este enfoque ha sido demostrado en diversos lenguajes de programación y políticas de seguridad. Es por esto que método de seguridad es implementado a partir de una biblioteca en Haskell, de la cuál se presenta el código y un ejemplo de funcionamiento.

1.2. Secure Multi-execution

El enfoque de **ejecución múltiple segura** de Devriese y Piessens garantiza la **no interferencia** en sistemas con distintos niveles de seguridad. Se basa en un **retículo de seguridad** L, donde los niveles están ordenados por una relación $\preceq$. Solo se permiten filtraciones de datos si $\ell_1 \preceq \ell_2$, es decir, si el nivel de destino tiene al menos el mismo nivel de seguridad que el origen.

- **Ejecución separada por niveles:** El programa se ejecuta **una vez por cada nivel de seguridad**.
- **Control de E/S:**
  - **Salidas:** Cada nivel de ejecución solo produce salidas en su propio canal de seguridad.
  - **Entradas:** Si una entrada proviene de un nivel superior, se reemplaza por un valor predeterminado. Si proviene de un nivel igual o inferior, se reutiliza normalmente.
- **Solidez:** Una ejecución solo usa información de su nivel o inferiores, evitando filtraciones hacia niveles más bajos.
- **Precisión:** Si un programa cumple la no interferencia en una ejecución normal, su comportamiento será idéntico bajo ejecución múltiple segura.

En resumen, este método **blinda la información sensible** al asegurar que cada nivel de seguridad solo acceda a los datos que le corresponden.

1.3. Secure Multi-execution in Haskell

En la mayoría de los lenguajes de programación funcional pura, las computaciones con efectos secundarios, como las operaciones de entrada y salida, pueden distinguirse por su tipo. Por ejemplo, en Haskell, toda computación que realiza efectos secundarios debe codificarse como un valor de la mónada `IO`. Específicamente, un valor de tipo `IO a` es una una computación que puede tener efectos secundarios que, cuando se ejecuta, devuelve un valor de tipo `a`.

Esta manera en la que las mónadas identifican computaciones con efectos secundarios se ajusta particularmente bien a la idea de la ejecución múltiple segura, ya que permite dar diferentes interpretaciones a las operaciones de E/S según el nivel de ejecución especificado.

Para simplificar, consideramos un retículo de seguridad de dos elementos con L y H, donde $L \preceq H$ y $H \not\preceq L$. Los niveles L y H representan los niveles de confidencialidad pública y secreta, respectivamente. No obstante, la implementación presentada aquí funciona para un

retículo de seguridad finito arbitrario. En la Fig. 1 mostramos la implementación del retículo como elementos del tipo de datos `Level` y definimos la relación de orden $\preceq$ y la relación no reflexiva $\prec$.

$$\textbf{data } \mathit{Level} = L \mid H \textbf{ deriving } (\mathit{Eq}, \mathit{Enum})$$
$$\cdot \sqsubseteq \cdot, \cdot \sqsubset \cdot :: \mathit{Level} \to \mathit{Level} \to \mathit{Bool}$$
$$H \sqsubseteq L = \mathit{False}$$
$$\_ \sqsubseteq \_ = \mathit{True}$$
$$p \sqsubset q \;\; = p \sqsubseteq q \wedge p \not\equiv q$$

Fig. 1. Security lattice

Proponen una biblioteca que funciona reemplazando las acciones de **E/S** (es decir, los valores de la mónada `IO`) por una descripción pura de las mismas. En la **ejecución múltiple segura**, las acciones de **E/S** realizadas por dicho programa deben interpretarse de manera diferente dependiendo del nivel de seguridad vinculado a una ejecución dada. Por lo tanto, los programas que se ejecutan bajo ejecución múltiple segura no devuelven acciones de **E/S**, sino más bien una descripción pura de ellas. Con esto en mente, los programas seguros tienen el tipo $a \to \mathrm{ME}\, b$, donde la mónada `ME` describe los efectos secundarios producidos durante la computación. Cuando el programa se ejecuta, esas descripciones de **E/S** se interpretan según la especificación de la ejecución múltiple segura.

Para los niveles de seguridad L y H, el programa se ejecuta dos veces, donde las acciones de **E/S** se interpretan de manera diferente en la ejecución vinculada al nivel L, y en la vinculada al nivel H. La Figura 2 resume las ideas detrás de nuestra biblioteca. La función `run` ejecuta y vincula el programa al nivel de seguridad dado como argumento. Observe que la función `run` también es responsable de la interpretación de las acciones de E/S descritas en la mónada `ME`.

a) $a \longrightarrow IO\ b$

b) $a \longrightarrow ME\ b \xrightarrow[\text{run } H]{\text{run } L} IO\ b$

c) **data** $ME\ a = Return\ a$
$\quad \mid Write\ FilePath\ String\ (ME\ a)$
$\quad \mid Read\ FilePath\ (String \to ME\ a)$

Fig. 2. Type for a typical program with side-effects (a) and a secure multi-execution program (b), and definition of *ME* (c).

Para simplificar, solo se consideró la lectura y escritura de archivos como las posibles acciones de **E/S**. La función `level :: FilePath → Level` asigna niveles de seguridad a los archivos, indicando la confidencialidad de su contenido.

La mónada `ME` describe las acciones de **E/S** realizadas por los programas y está definida en la Figura 2. Los constructores `Return`, `Write` y `Read` modelan programas que realizan diferentes acciones.

1. El programa `Return x` simplemente devuelve el valor x sin realizar ninguna operación de **E/S**.
2. El programa `Write file x p` modela un programa que escribe la cadena x en el archivo `file` y luego se comporta como el programa p.
3. El programa `Read file g` modela un programa que lee el contenido x del archivo `file` y luego se comporta como el programa $g(x)$.

Técnicamente, `ME` es una mónada intermedia que proporciona un modelo puro de la lectura y escritura de archivos en la mónada `IO`.

Los usuarios de la biblioteca no escriben programas utilizando los constructores de `ME` directamente. En su lugar, utilizan la interfaz proporcionada por la mónada: `return :: a → ME a` y `(≫=) :: ME a → (a → ME b) → ME b`. La función `return` eleva un valor puro a la mónada `ME`. El operador ≫=, llamado *bind*, se utiliza para secuenciar las computaciones. Estas son las únicas operaciones primitivas para la mónada `ME`, y en consecuencia, los programadores deben secuenciar computaciones individuales explícitamente utilizando el operador `bind`.

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad\ ME} \textbf{ where} \\
\quad \mathit{return}\ x \qquad\qquad\quad = \mathit{Return}\ x \\
\quad (\mathit{Return}\ x) \quad \gg\!= f \quad = f\ x \\
\quad (\mathit{Write\ file}\ s\ p) \gg\!= f = \mathit{Write\ file}\ s\ (p \gg\!= f) \\
\quad (\mathit{Read\ file}\ g) \gg\!= f \quad = \mathit{Read\ file}\ (\lambda i \rightarrow g\ i \gg\!= f)
\end{array}
$$

Fig. 3. Definitions for *return* and ≫=

Los valores en `ME` se introducen con `Return`, por lo que este es el único caso donde se aplica f. En los otros dos casos, las operaciones de `Write/Read` se conservan y el `bind` con f se aplica de manera recursiva.

Además de `return` y `(≫=)`, la mónada `ME` tiene operaciones para denotar las acciones de E/S sobre archivos. Estas operaciones modelan las operaciones equivalentes en la mónada `IO` y se dan por las siguientes funciones:

```
writeFile :: FilePath → String → ME ()
writeFile file s = Write file s (return ())

readFile :: FilePath → ME String
readFile file = Read file return
```

1.4. An interpreter for the monad *ME*

$$
\begin{array}{l}
\mathit{run} :: \mathit{Level} \rightarrow \mathit{ChanMatrix} \rightarrow \mathit{ME}\ a \rightarrow \mathit{IO}\ a \\
\mathit{run}\ l\ \_\ (\mathit{Return}\ a) = \mathit{return}\ a \\
\mathit{run}\ l\ c\ (\mathit{Write\ file}\ o\ t) \\
\quad \mid \mathit{level\ file} \equiv l \quad = \textbf{do}\ \mathit{IO.writeFile\ file}\ o \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{run}\ l\ c\ t \\
\quad \mid \mathit{otherwise} \qquad = \mathit{run}\ l\ c\ t \\
\mathit{run}\ l\ c\ (\mathit{Read\ file}\ f) \\
\quad \mid \mathit{level\ file} \equiv l \quad = \textbf{do}\ x \leftarrow \mathit{IO.readFile\ file} \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{broadcast}\ c\ l\ \mathit{file}\ x \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{run}\ l\ c\ (f\ x) \\
\quad \mid \mathit{level\ file} \sqsubset l \quad = \textbf{do}\ x \leftarrow \mathit{reuseInput}\ c\ l\ \mathit{file} \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{run}\ l\ c\ (f\ x) \\
\quad \mid \mathit{otherwise} \qquad = \mathit{run}\ l\ c\ (f\ (\mathit{defvalue\ file}))
\end{array}
$$

Fig. 4. Interpreter for monad *ME*

La **Figura 4** muestra el intérprete para programas de tipo `ME a`. Intuitivamente, `run l c p` ejecuta `p` y vincula la ejecución al nivel de seguridad l

El argumento c se utiliza cuando los inputs de ejecuciones vinculadas a niveles más bajos necesitan ser reutilizados (explicado más abajo).

La implementación es notablemente cercana a la descripción informal de la ejecución múltiple segura.

Las **escrituras** solo se realizan cuando el nivel de confidencialidad del archivo de salida es el mismo que el nivel de seguridad vinculado a la ejecución.

Las **lecturas** se realizan cuando el nivel de confidencialidad de los archivos es el mismo que el nivel de seguridad vinculado a la ejecución. Además, los datos de esos inputs se difunden a ejecuciones vinculadas a niveles de seguridad más altos para ser reutilizados adecuadamente cuando sea necesario. Si el nivel de ejecución actual es más alto que el nivel de confidencialidad del archivo, el contenido del archivo se obtiene de la ejecución vinculada al mismo nivel de seguridad que el archivo. De lo contrario, los datos de entrada se reemplazan por un valor por defecto, esto se hace a partir de la función `defvalue :: FilePath → String` establece valores por defecto para diferentes archivos.

Para evitar introducir errores en tiempo de ejecución, adoptamos un valor por defecto para cada archivo (es decir, punto de entrada) en el programa. Observe que los inputs pueden ser utilizados de manera diferente dentro de los programas. Por ejemplo, los contenidos de algunos archivos podrían ser analizados como números, mientras que otros como cadenas de texto simples. Por lo tanto, elegir un valor constante por defecto, como una cadena vacía, podría generar errores en tiempo de ejecución al intentar analizar un número a partir de él.

Una ejecución vinculada al nivel de seguridad ℓ reutiliza los inputs obtenidos en ejecuciones vinculadas a niveles más bajos. Por lo tanto, se implementaron canales de comunicación entre ejecuciones, desde un nivel de seguridad ℓ' a un nivel de seguridad ℓ, si $\ell' \prec \ell$. En el intérprete, el argumento de tipo `ChanMatrix` consiste en una **matriz de canales de comunicación indexada por niveles de seguridad**. De esta manera, una ejecución vinculada al nivel ℓ' puede enviar sus inputs a la ejecución vinculada al nivel ℓ, donde $\ell' \prec \ell$. Los mensajes transmitidos en estos canales tienen el tipo `(FilePath, String)`, es decir, pares de un nombre de archivo y su contenido. La función `broadcast c l file x` difunde el par $(file, x)$ en los canales vinculados a ejecuciones en niveles de seguridad más altos, es decir, canales $c_{l,\ell}$ tales que $l \prec \ell$. La función `reuseInput c l file` coincide con el nombre del archivo $file$ como el primer componente de los pares en el canal $c_{levelfile,l}$ y devuelve el segundo componente, es decir, el contenido del archivo.

```
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
```

Fig. 5. Secure multi-execution

La ejecución segura multi-hilo es orquestada por la función `sme`. Esta función es responsable de **crear canales de comunicación** para implementar la reutilización de inputs, **crear variables de sincronización** para esperar que los diferentes hilos terminen y, **para cada nivel de seguridad**, **crear un nuevo hilo** que ejecute el intérprete en ese nivel.

Cuando se lanzan los procesos, uno por nivel de seguridad, que ejecutan la función `run`, el hilo principal queda bloqueado hasta que ambos procesos escriban dentro de sus variables de sincronización correspondientes.

El diseño de `sme` se centra en evitar filtraciones de información por datos explícitos, pero no se preocupa por fugas de tiempo. En consecuencia, si un atacante analiza tiempos de ejecución, podría inferir información sobre procesos de seguridad.

```
data CreditTerms = CT { discount :: Rational, ddays :: Rational, net :: Rational }
calculator :: ME ()
calculator = do loanStr  ← readFile "Client"
                termsStr ← readFile "Client-Terms"
                let loan     = read loanStr
                    terms    = read termsStr
                    interest = loan − loan ∗ (1 − discount terms / 100)
                    disct    = discount terms / (100 − discount terms)
                    ccost    = disct ∗ 360 / (net terms − ddays terms)
                writeFile "Client-Interest" (show interest)
                writeFile "Client-Statistics" (show ccost)
                  -- writeFile "Client-Statistics" (show ccost ++ loanStr)
```

Fig. 6. Financial calculator

1.5. A motivating example

En este escenario, consideramos que el monto de cada préstamo es información confidencial (secreta), mientras que el costo del crédito es público y, por lo tanto, está disponible para estadísticas. Al escribir nuestro programa utilizando la mónada ME, podemos estar seguros de que la información confidencial nunca se da para estadísticas. En otras palabras, la empresa consultora externa no aprenderá nada sobre el monto de los préstamos proporcionados por la empresa financiera. La Figura 6 muestra una posible implementación del programa para calcular los intereses y el costo del crédito. Los archivos ‘‘`Client`’’ y ‘‘`Client-Interest`’’ se consideran secretos (nivel H), mientras que ‘‘`Client-Terms`’’ y ‘‘`Client-Statistics`’’ se consideran públicos (nivel L). El código es autoexplicativo.

Si un programador escribe, por error o malicia, `show ccost ++loanStr` como la información a escribir en el archivo público (ver línea comentada), entonces la ejecución segura múltiple evita filtrar la información sensible en `loanStr` al darle la cadena vacía a la ejecución vinculada al nivel de seguridad L.

1.6. Concluding remarks

Se propuso una mónada y un intérprete para la ejecución segura múltiple. Se implementaron las ideas principales en una pequeña biblioteca de Haskell de unas 130 líneas de código y presentamos un ejemplo en ejecución. La implementación es compacta y clara, lo que facilita la comprensión de cómo funciona concretamente la ejecución segura múltiple. La transmisión de valores de entrada a ejecuciones en niveles más altos es una novedad de nuestra implementación. Esta decisión de diseño no está ligada a la implementación en Haskell, y la idea se puede utilizar para implementar la reutilización de entradas en cualquier enfoque de ejecución segura múltiple para cualquier lenguaje dado. La biblioteca está disponible públicamente.

Trabajo futuro Nuestro objetivo a largo plazo es proporcionar una biblioteca completa para la ejecución segura múltiple en Haskell. La mónada IO puede realizar una amplia gama de operaciones de entrada y salida. Es entonces interesante diseñar un mecanismo capaz de elevar, de manera tan automática como sea posible, las operaciones de IO a la mónada ME. Otra dirección para el trabajo futuro está relacionada con la desclasificación, o liberación deliberada de información confidencial. La desclasificación en la ejecución segura múltiple sigue siendo un desafío abierto. Debido a la estructura de los programas monádicos, creemos que es posible identificar y restringir los puntos de sincronización donde podría ocurrir la desclasificación. Luego, la desclasi ficación no podrá ocurrir arbitrariamente dentro de los programas, sino solo en aquellos lugares donde podamos dar algunas garantías sobre la seguridad de los programas. Para evaluar las capacidades de nuestra biblioteca, planeamos usarla para implementar una aplicación web de tamaño mediano. Las aplicaciones web son buenos candidatos para estudios de caso debido a su demanda de confidencialidad, así como a sus frecuentes ope raciones de entrada y salida (es decir, solicitudes y respuestas del servidor). También es nuestra intención realizar

pruebas de rendimiento para determinar la sobrecarga introdu cida por nuestra biblioteca. La biblioteca parece multiplicar el tiempo de ejecución por el número de niveles, pero dado que las operaciones de archivos solo se realizan una vez, la realidad podría ser mejor si el mecanismo de transmisión no es costoso.

2. RSK: Bitcoin Merge Mining is Here to Stay

Link al Paper: blog.rootstock.io/noticia/rsk-bitcoin-merge-mining-is-here-to-stay

2.1. Introduction to Bitcoin Mining

La **minería de Bitcoin** es el proceso que permite a Bitcoin resistir ataques Sybil (tipo de ataque en redes distribuidas donde un solo atacante crea múltiples identidades falsas para influir en el sistema.) y constituye la base del **Consenso Nakamoto**. Consiste en resolver problemas computacionales complejos mientras se selecciona un conjunto válido de transacciones para agregar al *ledger*, lugar donde se anotan todas las transacciones de Bitcoin.

Hoy en día, la mayoría de la minería se realiza a través de **pools de minería**, que ayudan a reducir la variabilidad en los pagos y los costos de mantenimiento. Estos pools siguen una arquitectura **cliente-servidor**, donde los mineros (clientes) se conectan a un **servidor de pool** que ejecuta software especializado como `Ckpool`, `Btcpool` o `Eloipool`.

El **servidor de pool** se comunica con un nodo Bitcoin a través de un canal RPC sin cifrar, obteniendo información sobre la mejor cadena de bloques actual y el hash del bloque padre. Luego, selecciona transacciones para incluirlas en el nuevo bloque.

2.2. Merge-Mining

Merge mining es una técnica que permite utilizar la misma potencia de minería que asegura una blockchain primaria para asegurar una blockchain secundaria.

El proceso consiste en incrustar el **identificador del bloque de la blockchain secundaria** (un hash criptográfico del nuevo bloque recientemente construido) en algún lugar del **bloque de la blockchain primaria que se está minando**. Este hash secundario, precedido por un texto descriptivo corto o **magic bytes**, se conoce como la **etiqueta de merge-mining**. El prefijo permite que la blockchain secundaria localice la etiqueta. Sin embargo, no debe haber ambigüedad en su ubicación: un bloque de la blockchain primaria debe asociarse con, como máximo, un único bloque de la blockchain secundaria.

Aunque el uso de hashing criptográfico para la vinculación evita trampas, los requisitos de seguridad para esta vinculación son mucho menores que los de la criptografía tradicional. De manera informal, el único requisito de seguridad para el **merge-mining** es que debe ser más difícil crear un bloque de la blockchain primaria que pueda asociarse con dos bloques de la misma blockchain secundaria que minar dos bloques diferentes de la blockchain primaria, uno para cada asociación, con la dificultad de la blockchain secundaria.

En el caso de **RSK**, la blockchain primaria es Bitcoin y la secundaria es RSK. La dificultad de los bloques de RSK actualmente equivale a una seguridad de 70 bits, mientras que la dificultad de Bitcoin equivale a 74 bits.

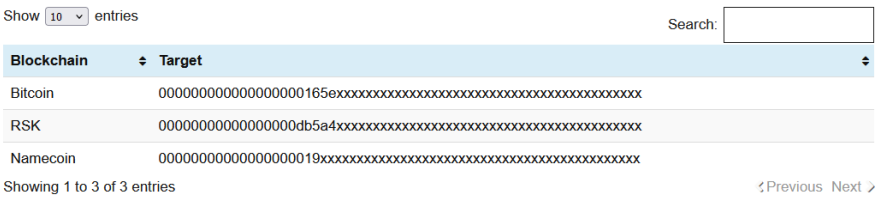
2.3. Proof of Work Proxy

En **merge-mining**, un **header de Bitcoin** sirve únicamente como un proxy de *proof-of-work* (PoW). La blockchain de **RSK** debe interpretar el PoW de un **header de bloque de Bitcoin** y buscar en el bloque de Bitcoin la etiqueta que establezca de manera única la relación con un **header de bloque de RSK**. Por lo tanto, se traduce transitivamente el PoW del bloque de Bitcoin al PoW del bloque de RSK.

Como se mencionó antes, la dificultad de la blockchain de RSK es menor que la dificultad de la blockchain de Bitcoin, por lo que muchos **headers de bloque de Bitcoin** que no resuelven el acertijo de PoW de Bitcoin serán soluciones válidas para el acertijo de PoW de RSK. Veamos esta distinción con más detalle.

Cada blockchain calcula una **dificultad esperada** para cada bloque. Esta dificultad es definida por todos los bloques previos con el fin de mantener aproximadamente constante el **tiempo**

promedio entre bloques. Internamente, la dificultad se traduce en un **target**, que es *inversamente proporcional* a la dificultad. El **target** es un número entero sin signo de 256 bits.


Show 10 entries Search:

Blockchain	Target
Bitcoin	0000000000000000000165exxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
RSK	00000000000000000db5a4xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Namecoin	0000000000000000000019xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Showing 1 to 3 of 3 entries Previous Next

Figura 1: Objetivos aproximados para una blockchain con merge-mining en el mismo día. RSK tiene un objetivo más alto porque los bloques son 20 veces más frecuentes.

Debido a la **imprevisibilidad** del *hash digest* criptográfico, un **id de header de bloque**, que es un *hash digest* criptográfico del contenido del header del bloque, se asume como una variable aleatoria distribuida uniformemente. Aunque esto puede no ser teóricamente cierto, funciona en la práctica, ya que no existen métodos prácticos conocidos para revertir las funciones de hash utilizadas en **Bitcoin** (`double SHA256`) o en **RSK** (`Keccak`).

Este *hash digest*, cuando se interpreta como un número sin signo, debe ser menor que el **target** para que el bloque represente una solución al acertijo de PoW. Por lo tanto, cuanto menor sea el **target**, más difícil será resolver el acertijo de PoW.

2.4. SPV Proofs

La **blockchain secundaria** no requiere el bloque completo de Bitcoin para validar el *proof-of-work* (PoW) del **header de Bitcoin** y asociarlo con el **header de RSK**. Esto se realiza de la siguiente forma:

1. No descarga todo el bloque de Bitcoin, sino solo el header.
2. Usa una prueba **SPV**, que es básicamente una prueba de que el tag de RSK existe dentro del árbol de Merkle del bloque de Bitcoin.
3. Si el tag está en el árbol de Merkle, significa que ese bloque de Bitcoin es válido para RSK, y puede aceptar el bloque en su propia cadena.

2.5. Targets Hierarchy

El **objetivo de RSK** generalmente será más alto que el de Bitcoin, porque los bloques de RSK son más frecuentes que los de Bitcoin, por lo que el acertijo de RSK es menos difícil de resolver. Por lo tanto, un **header de bloque de Bitcoin** que resuelva el acertijo de PoW de RSK puede no ser aceptado por la red de Bitcoin.

Estos **bloques intermedios** se llaman **"shares"**, y son requeridos por el servidor para la contabilidad.

Las **shares**, por lo tanto, proporcionan una mayor granularidad para contabilizar las contribuciones de los mineros. Las **shares** se transmiten regularmente al servidor para que pueda dividir de manera justa las ganancias futuras entre todos los clientes involucrados, ponderando sus contribuciones de hashing. Pero las **shares** también se transmiten porque una de ellas puede ser (por azar) una solución al acertijo actual de PoW de Bitcoin.

Así que los mineros no necesitan recibir la verdadera dificultad de PoW de Bitcoin (o el **target**) del servidor, y normalmente no saben si han resuelto un bloque de Bitcoin hasta que el pool se lo comunique. El servidor revisa cada share recibida, reconstruye el **header del bloque**, y si el *hash digest* de doble SHA256 del header es numéricamente inferior al **target** asociado con la dificultad actual de Bitcoin, lo envía al servidor, que lo difunde por la red.

Dado que cada blockchain secundaria puede tener una dificultad diferente, un **poolserver** capaz de hacer **merge-mining** debe hacer esta verificación para cada blockchain secundaria que maneja. Si el **header de Bitcoin** representa una solución válida para el acertijo de PoW de la blockchain de RSK, envía el **header del bloque de Bitcoin** al RSK, que añadirá el bloque asociado y lo enviará como válido a su red.

2.6. RSK tag Embedding

La **etiqueta de RSK** consiste en el identificador ASCII `RSKBLOCK:` concatenado con un fragmento de datos binarios que incluye el **hash digest** del **header del bloque de RSK** que está siendo minado. Como se mencionó anteriormente, la etiqueta debe ser identificada sin ambigüedad (no debe haber una forma de crear un bloque de Bitcoin que pueda ser asociado con dos bloques diferentes de RSK).

La etiqueta de RSK puede ubicarse en el campo **coinbase** de la transacción de generación, o en cualquiera de los **outputs** de la transacción de generación.

2.7. Merge-Mining improvements in the Upcoming Network Upgrade

La nueva actualización de red en RSK introduce un formato mejorado para la etiqueta de merge-mining. Esta mejora agrega información adicional para que los usuarios y sistemas automáticos puedan tomar decisiones informadas sobre la salud de la red y responder a situaciones anormales, como el doble gasto. Los nodos de RSK ahora podrán detectar y reaccionar autónomamente a estas situaciones.

El nuevo formato de la etiqueta tiene la siguiente estructura de 32 bytes:

- **20 bytes: Prefijo del hash para merge-mining (PREFIX)**: Identificador del bloque de Bitcoin relacionado con RSK.
- **7 bytes: Commit-to-Parents-Vector (CPV)**: Información sobre los bloques padres para entender las cadenas paralelas de RSK.
- **1 byte: Número de uncles en los últimos 32 bloques (NU)**: Limita a 255 el número de bloques uncles, ayudando a detectar la actividad minera paralela.
- **4 bytes: Número de bloque (BN)**: El número del bloque en la blockchain de RSK.

Estos cuatro campos deben ser verificados por consenso dentro de la red. La nueva etiqueta permite a los nodos de RSK monitorear la construcción de cadenas paralelas y competidoras de bloques RSK, incluso si no se transmiten a la red. Además, el hash del encabezado de RSK se recorta a 20 bytes para mantener la compatibilidad con el software de poolserver.

2.8. RSK Merge-Mining Security

La teoría del consenso Nakamoto de Bitcoin, que utiliza el *proof-of-work* (PoW), se basa en la seguridad termodinámica y teórica de juegos, no en la seguridad criptográfica. RSK, a través de su proceso de merge-mining, es seguro frente a atacantes irracionales que no pueden calcular 2^{80} operaciones de `hash` en menos de 30 segundos.

Para un atacante racional, el **merge-mining de RSK** es más rentable, ya que solo requiere aproximadamente 2^{69} operaciones, y está subsidiado por Bitcoin (los mineros aún deben minar, pero debido a que pueden usar el mismo esfuerzo de minería para ambas cadenas (Bitcoin y RSK), el costo de minería en RSK es mucho más bajo, lo que se describe como "subsidiado".).

Los atacantes irracionales necesitarían una cantidad mucho mayor de hardware, con un costo cercano a los cinco billones de dólares, y aunque pudieran resolver el PoW de RSK, no obtendrían beneficios económicos. Esto por como funciona la repartija de reompensas en RSK.

RSK utiliza una técnica no estándar para comprimir la transacción de generación, donde se transmite solo la cola de la transacción en lugar de toda la transacción, junto con un `midstate` de 64 bytes de SHA256. Este truco depende de una propiedad de SHA256 llamada **resistencia a colisiones de *freestart***, la cual es suficientemente segura. Destacar la diferencia entre esta resistencia a colisiones y la resistencia a colisiones normal.

En resumen, la seguridad de la etiqueta en RSK está asegurada para los próximos 20 años, aunque una futura actualización podría aumentar el tamaño del `hash` si las tendencias computacionales cambian radicalmente.

2.9. Summary

Entre el 40 % y el 51 % de los mineros de Bitcoin están actualmente realizando merge-mining de RSK, lo que hace que RSK sea la plataforma de contratos inteligentes más segura del planeta, en términos de **seguridad termodinámica**. El merge-mining de RSK es fácil, pero se debe tener cuidado de no alterar el funcionamiento normal del software del poolserver, incluso bajo las condiciones más adversas, como los ataques directos a la red de RSK. RSK Labs desarrolló varios plugins que fueron probados exhaustivamente contra una serie de condiciones de falla, para verificar la continuidad de la operación minera. Además, se realizaron medidas de eficiencia en los softwares de pools mineros para asegurar que la eficiencia del software de minería no se vea alterada.

2.10. Relaciones

El **hash** en las blockchains funciona como una **firma digital**. Aunque técnicamente no es lo mismo que una firma digital en el sentido clásico (que utiliza criptografía de clave pública y privada), tiene algunas similitudes importantes.

En una blockchain, el **hash** sirve para:

1. **Garantizar la integridad de los datos**: Cada bloque en una blockchain tiene un hash que se calcula a partir de la información contenida en el bloque. Cualquier cambio en los datos del bloque (por ejemplo, en las transacciones) cambiaría el hash, lo que haría evidente que los datos han sido alterados. Esta propiedad es similar a cómo una firma digital garantiza la integridad del mensaje firmado.

2. **Crear una "huella digital" única**: Al igual que una firma digital, el hash proporciona una **huella única** de un bloque o transacción. Este hash representa el contenido de ese bloque de manera compacta y única, funcionando como un identificador para el bloque. A diferencia de una firma digital que está vinculada a una clave privada, el hash es un identificador único para la información en el bloque.

3. **Verificar la autenticidad y no repudio**: Al estar vinculado al bloque, el hash también ayuda a **verificar la autenticidad** del bloque. Si un atacante intentara alterar el contenido de un bloque, el hash cambiaría, y cualquier nodo que verificara ese hash sabría que el bloque ha sido alterado. Esto se puede comparar con la función de una firma digital en términos de autenticación e integridad, ya que cualquiera puede verificar si el contenido es auténtico o ha sido modificado.

En resumen, aunque un **hash** no es una firma digital per se, cumple una función similar dentro de la blockchain al ofrecer **integridad, autenticidad y no repudio** para las transacciones y bloques. Sin embargo, las firmas digitales en blockchains (como las utilizadas en la firma de transacciones) se basan en criptografía de clave pública y privada, lo cual es un concepto diferente al hash.