UNIVERSIDAD DEL VALLE DE GUATEMALA

ALGORITMOS Y ESTRUCTURAS DE DATOS

SECCIÓN 20

Proyecto 1: Fase 1

Intérprete de Bitcoin Script

Cristopher Javier Chávez Toc - 25199

Mauricio Adrian Corado Castañeda - 25218

Joseph Alfredo Gongora Giron -25051

13 de enero de 2026

Introduccion

Bitcoin es una criptomoneda, pero también es un sistema distribuido que combina criptografía, redes peer-to-peer y estructuras de datos para permitir la transferencia de valor sin la necesidad de intermediarios. Desde su lanzamiento en 2008 por Satoshi Nakamoto, bitcoin fue diseñado con un enfoque en la seguridad, la verificabilidad y la descentralización sobre la expresividad computacional (Nakamoto, 2008).

En el centro de este sistema se encuentra Bitcoin Script, que es un lenguaje de programación simple y basado en pila que define las reglas por las cuales las salidas de las transacciones (UTXO) pueden ser gastadas. A diferencia de otros lenguajes de propósitos generales, Bitcoin Script fue diseñado como un mecanismo de validación determinista, que es capaz de ejecutarse de forma eficiente y segura en miles de nodos alrededor del mundo (Nakamoto, 2008).

El estudio del Bitcoin Script es fundamental, ya que es de suma importancia para poder comprender cómo es que la red garantiza la propiedad y el control de los fondos sin recurrir a autoridades centrales. Igualmente, su análisis permite observar como decisiones de diseño en estructuras de datos, límites computacionales y modelos de ejecución influyen directamente en la seguridad y escalabilidad de un sistema monetario descentralizado.

Fundamentos de Bitcoin Script

Bitcoin Script es el lenguaje de programación que hace funcionar cada transacción dentro de su red. A diferencia de otros lenguajes como Java o Python, que sirven para crear todo tipo de software, Script es intencionalmente limitado (Lightspark Team, 2025). Es un sistema basado en pila (stack) que procesa la información en un orden secuencial específico (Narayanan et al, 2016). Su función principal no es realizar cálculos complejos, sino actuar como un "candado y llave" digital. Define las condiciones que se deben cumplir para que los fondos conocidos como UTXO puedan moverse de una dirección a otra (Lightspark Team, 2025).

La característica más importante y debatida de este sistema es que no es Turing-completo. Esto significa que el lenguaje no permite crear bucles ni funciones que se llamen a sí mismas infinitamente (Maldonado, 2023). Esta restricción impide que el sistema resuelva cualquier tipo de problema matemático arbitrario, pero esto no es un fallo técnico (Lightspark Team, 2025).

La razón principal para esta limitación es evitar que la red se detenga o sufra ataques. En una red como Bitcoin, miles de computadoras (nodos) deben validar cada transacción de forma independiente. Si el lenguaje permitiera bucles infinitos, un atacante podría enviar una transacción con un código diseñado para nunca terminar. Esto atraparía a los procesadores de todos los nodos en un ciclo sin fin, congelando la red y provocando un colapso del sistema. Al restringir las instrucciones disponibles (Opcodes), también se minimiza el riesgo de que un error en el código permita el robo de fondos o la creación de monedas falsas. (Maldonado, 2023).

Al eliminar la posibilidad de bucles, Bitcoin garantiza tres cosas esenciales:
Es finito: Se sabe con certeza que el programa terminará en poco tiempo.
Es determinista: El resultado siempre es el mismo.
Es predecible: Se puede calcular cuánto esfuerzo y memoria tomará validar una transacción antes de siquiera ejecutarla (Maldonado, 2023).

Esta simplicidad elimina la necesidad de resolver problemas informáticos complejos sobre cuándo se detendrá un programa. La filosofía detrás de esto es que Bitcoin es "dinero programable", no una "computadora mundial". La mayoría de las transacciones financieras no necesitan lógica complicada; solo requieren verificar firmas, establecer tiempos de espera o requerir la aprobación de múltiples personas (multifirmas) (Lightspark Team, 2025).

La historia del desarrollo de Bitcoin muestra que esta seguridad ha sido una prioridad constante. En sus inicios, alrededor de 2009, el sistema incluía muchas más funciones, como operaciones matemáticas avanzadas y unión de textos. Sin embargo, en 2010, se descubrió que varias de estas funciones podrían usarse para atacar la red, por ejemplo, creando datos que crecen descontroladamente hasta agotar la memoria de los validadores. Como respuesta, se desactivaron muchas de estas funciones, consolidando la idea de poner la seguridad por encima de la funcionalidad (Murphy, 2025).

Flujo de validación

El motor de validación de Bitcoin es un proceso secuencial y determinista que une dos fragmentos de código: el candado y la llave. Para validar una transacción, la red no evalúa la transacción de forma aislada, sino que ejecuta un script compuesto (Fractal, 2023). Este proceso comienza con el ScriptSig (el script de desbloqueo que provee el usuario que gasta) y continúa inmediatamente con el ScriptPubKey (el script de bloqueo que definió el dueño anterior de los fondos) (Walker, 2025).

El entorno donde ocurre esta validación es la Pila (Stack). Esta estructura de datos opera bajo el principio LIFO (Last In, First Out), donde los elementos se empujan (push) hacia la cima y los opcode extraen (pop) los elementos superiores para procesarlos. La ejecución es lineal, de izquierda a derecha (Walker, 2025). Si en algún momento la pila viola una regla (como intentar duplicar un elemento que no existe) o si un operador de verificación falla, la transacción se marca inmediatamente como inválida (Fractal, 2023).

Pay-to-Public-Key-Hash (P2PKH)

Para entender la mecánica exacta del motor de validacion, analizamos el estándar P2PKH (ver figura 1). Su ejecución paso a paso ilustra el diseño del sistema:

1. Carga de Credenciales (ScriptSig): Inicialmente, la pila está vacía. El nodo ejecuta el script del usuario, que coloca dos datos en la cima: primero la Firma Digital (<sig>) y luego la Llave Pública (<pubkey>) (Fractal, 2023).
2. Duplicación y Hashing (ScriptPubKey - Inicio): El control pasa al script de bloqueo. El opcode OP_DUP duplica el elemento superior (la llave pública) para no perderlo, y acto seguido, OP_HASH160 transforma esa copia en un hash criptográfico (Fractal, 2023).

3. Verificación de Destino: El script empuja el hash de la dirección destino original (<pubKeyHash>). El opcode OP_EQUALVERIFY compara este hash con el que acabamos de calcular. Si coinciden, significa que la llave pública presentada corresponde a la dirección dueña de los fondos; ambos hashes se eliminan de la pila (Torres, 2024).
4. Verificación de Identidad: Finalmente, quedan en la pila la firma y la llave pública original. El opcode OP_CHECKSIG verifica matemáticamente que la firma haya sido generada por esa llave. Si la matemática es correcta, este operador consume ambos elementos y empuja un 1 (TRUE) a la pila (Walker, 2025).

La condición final de éxito es estricta. Para que la red acepte la transacción, una vez terminado todo el script, la pila no puede estar vacía ni contener un cero (OP_0). Debe quedar exactamente un elemento distinto de cero (habitualmente OP_1 o TRUE), lo que confirma que el dueño legítimo ha autorizado el gasto (Walker, 2025).

En P2PKH, los fondos quedan bloqueados a un hash de una clave pública. Para poder gastarlos, es de suma importancia presentar una firma válida y la clave pública correspondiente. Lo que ofrece este modelo es un equilibrio entre simplicidad, seguridad y eficiencia, que es la razón por la cual dominó la mayoría de las transacciones durante los primeros años de Bitcoin (Nakamoto, 2008).

Sin embargo, P2PKH presenta ciertas limitaciones cuando se requiere condiciones de gasto un poco más complejas, como las multifirmas. En esos casos, el script completo tiene la necesidad de ser revelado y almacenado en la blockchain, lo que genera un incremento en el tamaño de las transacciones y los costos relacionados a ello.

La pila (Stack) como Estructura LIFO

Bitcoin Script utiliza una pila de datos que siguen el principio Last In, First Out (LIFO), que es que el último elemento de la pila en ser insertado es el primero en ser retirado. Todos los opcodes operan solamente sobre los elementos ubicados en la cima de la pila, ya sea para duplicarlos, eliminarlos, compararlos o transformarlos (Antonopoulos, 2017).

Desde una vista algorítmica, Bitcoin Script opera como una máquina de estados de Forth simplificada. Las operaciones fundamentales de la pila, push y pop, presentan una complejidad temporal constante O(1). Esto es crítico en el ecosistema de Bitcoin, ya que en cada nodo de la red debe ejecutar y validar los scripts de cada transacción de manera independiente para alcanzar el consenso (Antonopoulos, 2017).

El modelo de ejecución basado en la pila LIFO minimiza las posibilidades de un ataque al evitar accesos aleatorios a la memoria, punteros complejos o estructuras anidadas que podrían derivar en desbordamientos de memoria. La ejecución es lineal y carece de bucles, lo que garantiza que el estado de la pila sea determinista. Es decir, ante un mismo script y datos de entrada, todos los nodos de la red alcanzaran el mismo resultado booleano (Nakamoto, 2008).

Límites de la Red y Control de Complejidad

Para salvaguardar la resiliencia de la red frente a ataques de Denegación de Servicio (DoS), Bitcoin impone límites estrictos a la computación de los scripts. Las principales restricciones son:
Un script individual no puede exceder los 10,000 bytes.
Se permite un máximo de 201 opcodes no triviales por script para evitar un uso excesivo de CPU durante la validación.
La pila no puede contener más de 1,000 elementos de forma simultánea.

Caso de Prueba y Traza de Ejecución: P2PKH (Entorno Simulado)

Para la validación práctica en el prototipo desarrollado, se implementará el flujo estándar P2PKH. Dado que el entorno es didáctico, las funciones criptográficas (OP_CHECKSIG, firmas y hashes) se comportarán como "Mocks" o simulaciones lógicas, operando sobre cadenas de texto en lugar de curvas elípticas reales.

El script de validación se compone concatenando el scriptSig (desbloqueo) con el scriptPubKey (bloqueo), resultando en la siguiente secuencia de ejecución lineal:
Secuencia de Instrucciones: <firma> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG (Flores, 2024).

Traza de la Pila (Stack Trace) paso a paso

A continuación, se detalla el estado de la pila (Deque) tras cada operación, asumiendo que la transacción es válida (Para más detalle, ver figura 2):

1. Estado Inicial:
    - Pila: [] (Vacía).
2. Carga de datos (scriptSig):
    - Se empujan a la pila la firma simulada y la llave pública.
    - Instrucción: PUSH <firma>, PUSH <pubKey>
    - Pila: [<firma>, <pubKey>] (Cima a la derecha).
3. Duplicación (OP_DUP):
    - Duplica el elemento en la cima (la llave pública) para poder verificar su hash sin perder el dato original necesario para la firma.
    - Pila: [<firma>, <pubKey>, <pubKey>]
4. Hashing (OP_HASH160):
    - Toma la llave pública de la cima y aplica el hash simulado.
    - Pila: [<firma>, <pubKey>, <HashedPubKey>]
5. Carga del Hash Esperado:
    - El script empuja el hash destino contenido en el scriptPubKey.
    - Pila: [<firma>, <pubKey>, <HashedPubKey>, <ExpectedHash>]
6. Verificación de Igualdad (OP_EQUALVERIFY):
    - Compara los dos elementos superiores (HashedPubKey y ExpectedHash). Si son iguales, los elimina y continúa; si no, la ejecución falla inmediatamente.
    - Pila: [<firma>, <pubKey>]

7. Verificación de Firma (OP_CHECKSIG):
    - Toma la <firma> y la <pubKey> restantes. En nuestro entorno simulado, el CryptoMock validará si la firma corresponde lógicamente a la llave pública.
    - Pila: [TRUE] (o 1).

Criterio de Aceptación: La transacción se considera válida únicamente si, al finalizar la ejecución, la pila no está vacía y el elemento en la cima es TRUE o un valor distinto a 0 (Flores, 2024).

Selección y Justificación de Java Collections Framework (JCF)

Para el intérprete de Bitcoin Script, seleccionamos tres estructuras de datos fundamentales del Java Collections Framework, priorizando eficiencia en tiempo de ejecución y de memoria.

1. La Pila Principal: Deque (Implementación: ArrayDeque)

Esta es la estructura central del intérprete (BitcoinStack en el UML), encargada de almacenar los datos y resultados de las operaciones. Funciona bajo el principio Last-In, First-Out.

Justificación: Aunque Java posee una clase Stack heredada, se ha seleccionado la interfaz Deque con su implementación ArrayDeque. Es más rápida que Stack al no ser sincronizada y más eficiente en memoria que LinkedList al no requerir la creación de objetos nodo adicionales para cada elemento (Morin, 2013).

Análisis de Complejidad:

- Tiempo: Las operaciones fundamentales push (addFirst) y pop (removeFirst) tienen una complejidad de O(1) amortizado. Esto garantiza que cada paso de ejecución del script tenga un costo constante, independientemente del tamaño de la pila (Morin, 2013).
- Espacio: O(n), donde n es el número de elementos en la pila. ArrayDeque redimensiona su arreglo interno dinámicamente sólo cuando es necesario (Morin, 2013).

2. Secuencia de Instrucciones: List (Implementación: ArrayList)

Utilizada en la clase ScriptEngine para almacenar la lista ordenada de instrucciones (List<OpCode>) que se obtienen tras parsear el scriptSig y scriptPubKey.

Justificación: El intérprete debe ejecutar el programa de izquierda a derecha. ArrayList es la opción óptima aquí porque ofrece un acceso rápido e iteración muy eficiente gracias a la localidad de referencia en memoria (los elementos están contiguos). Dado que la lista de instrucciones se construye una sola vez al inicio y luego solo se lee secuencialmente, no hay pérdida de eficiencia de inserción/borrado en posiciones intermedias (que sería la ventaja de LinkedList) (Downey, 2017).

Análisis de Complejidad:

- Tiempo: Agregar instrucciones al final durante el parseo (add) es O(1) amortizado. El acceso a una instrucción específica (get) es O(1) (Downey, 2017).
- Espacio: O(n), donde n es la cantidad de instrucciones en el script (Downey, 2017).

3. Fábrica de Operaciones: Map (Implementación: HashMap)

Esencial para la clase OpCodeFactory. Permite asociar los "tokens" de texto (ej. "OP_DUP", "OP_HASH160") con sus respectivas instancias concretas de la clase OpCode.

Justificación: El uso de un HashMap elimina la necesidad de estructuras condicionales extensas y complejas (como cadenas de if-else o switch). Esto hace que el código sea más limpio y extensible. Al recibir un token del script, el intérprete puede recuperar la operación correspondiente de manera instantánea (Adolfsson, 2023).

Análisis de Complejidad:

- Tiempo: La operación de búsqueda (get) y la de inserción (put) tienen una complejidad de O(1) promedio, gracias al cálculo de dispersión (hashing) de las claves tipo String. Esto asegura que decodificar el script no añada latencia al proceso de validación (Adolfsson, 2023)
- Espacio: O(k), donde k es el número de tipos de Opcodes soportados por el sistema (Adolfsson, 2023).

Conclusión

Nuestro análisis de Bitcoin Script demuestra que, en el contexto de un sistema monetario global descentralizado, la simplicidad constituye una característica de seguridad indispensable. La decisión original de diseñar un lenguaje que no es Turing-completo no surgió de una limitación tecnológica, sino que fue una estrategia deliberada para proteger la red. Al eliminar la posibilidad de bucles infinitos, así como restringir la complejidad de los comandos, Bitcoin garantiza que la validación de las transacciones sea siempre predecible, finita, además de resistente a ataques de denegación de servicio.

A través del estudio del flujo de validación, se evidencia que el uso de una estructura de pila bajo el principio LIFO resulta tanto eficiente como robusto. Este mecanismo permite que miles de nodos independientes verifiquen matemáticamente la propiedad de los fondos sin necesidad de confiar en un tercero, de esta forma, se asegura que el resultado sea idéntico en toda la red. Asimismo, la evolución de los scripts estándar, transitando del modelo básico P2PKH hacia soluciones avanzadas tales como SegWit o Taproot, confirma que el protocolo es capaz de adaptarse para ofrecer mayor privacidad junto con escalabilidad, sin sacrificar la seguridad de su capa base.

Anexos:

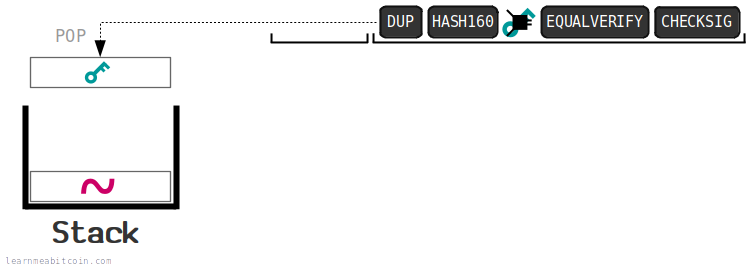

Figura 1. Diagrama técnico de la ejecución de un script P2PKH (Walker, 2025).

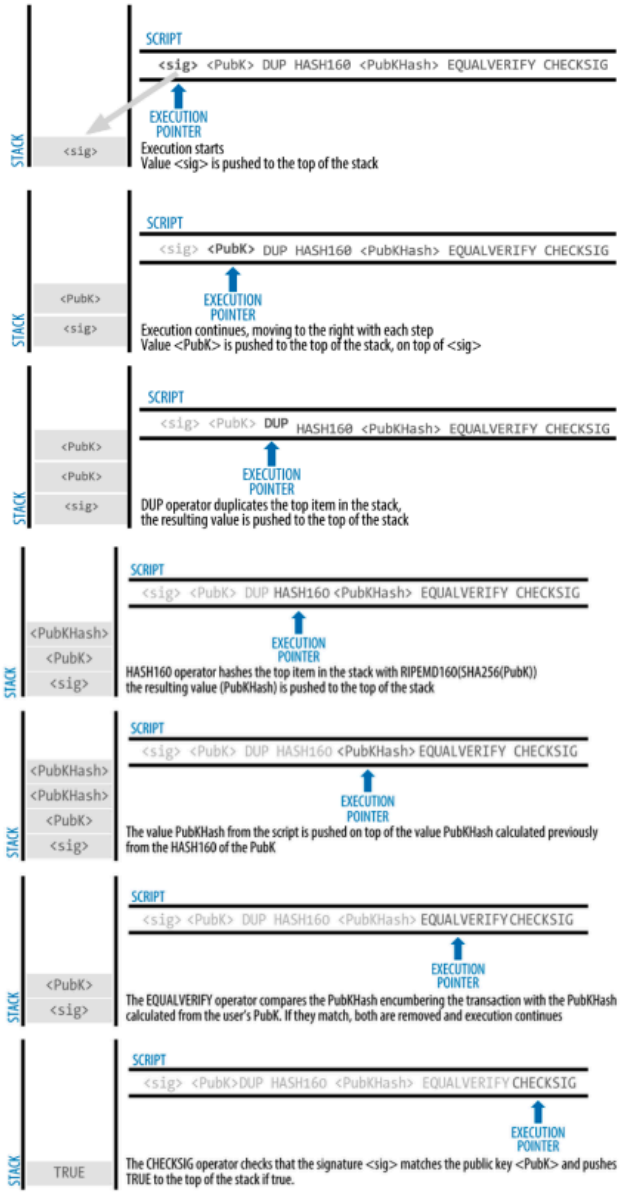

Figura 2: Diagrama de la traza de pila P2PKH (Flores, 2024).

References

Adolfsson, L. (2023). *Java Collections and What Influences Execution Time When Manipulating Collections of Strings*. https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9136657&fileOId=9136658

Antonopoulos, A. M. (2017). *Mastering Bitcoin: Programming the Open Blockchain (2a ed.)*. https://github.com/bitcoinbook/bitcoinbook?tab=readme-ov-file

Downey, A. (2017). *Think Data Structures: Algorithms and Information Retrieval in Java*. https://books.google.es/books?hl=es&lr=&id=oIQrDwAAQBAJ&oi=fnd&pg=PR2&dq=java+arraylist+time+complexity&ots=xgZmNMQYqz&sig=Vc8BQxeG06wyEdY5IxFtuC8CuiY#v=onepage&q&f=false

Flores, G. M. (2024). *Fundamentos matemáticos del Bitcoin*. https://upcommons.upc.edu/server/api/core/bitstreams/ea170f48-44f0-472c-ae51-49f670b60778/content

Fractal Bitcoin. (2025). *Bitcoin Script: A comprehensive primer for developers — Fractal Bitcoin.* https://fractalbitcoin.io/learn/bitcoin-script-primer-for-developers

Lightspark Team. (2025). *Bitcoin Script: The Programming Language of Money*. https://www.lightspark.com/glossary/bitcoin-script

Maldonado, J. (2023). *¿Qué es Turing Completo?* https://academy.bit2me.com/que-es-turing-completo/

Morin, P. (2013). *Open Data Structures: An Introduction, Volumen 9*. https://books.google.es/books?hl=es&lr=&id=ZZCJvrDe5bIC&oi=fnd&pg=PP11&dq=ArrayDeque+time+complexity&ots=4_ZqMMB3X8&sig=wrN5UlzqyrJ19xAQKiZcXjq4J7Q#v=onepage&q=ArrayDeque&f=false

Murphy, R. (2025). *Bitcoin Opcodes for Dummies*. https://medium.com/@DrRoyMurphy/bitcoin-opcodes-for-dummies-4cd6f10d744b

Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system*. https://bitcoin.org/bitcoin.pdf

Narayanan, A., Bonneau, J., Felten, E., Miller, A., & Goldfeder, S. (2016). *Bitcoin and Cryptocurrency Technologies*. https://d1wqtxts1xzle7.cloudfront.net/61427212/princeton_bitcoin_book20191204-61607-etkyte-libre.pdf?1575537871=&response-content-disposition=inline%3B+filename%3DBitcoin_and_Cryptocurrency_Technologies.pdf&Expires=1771010411&Signature=dcfWMCgx4JtRVvabp0NC

Torres, S. (2024). *ScriptPubKey: El código detrás de las transacciones Bitcoin.* https://keepcoding.io/blog/scriptpubkey/

Walker, G. (2025). *Bitcoin Script | A mini programming language.* https://learnmeabitcoin-com.translate.goog/technical/script/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc