Universidad Nacional Autonóma de México

Posgrado en Ciencias e Ingeniería De la Computación

FUNDAMENTOS DE CONTRATOS INTELIGENTES EN CADENAS DE BLOQUES

TESIS

QUE PARA OPTAR POR EL GRADO DE:

Maestro en Ciencia e Ingeniería en Computación

PRESENTA:

L. en C.C. Miguel Angel Piña Avelino

DIRECTOR DE TESIS:

Dr. Armando Castañeda Rojano

Posgrado en Ciencias e Ingeniería de la Computación

Ciudad Universitaria, CD. MX., 16 de octubre de 2019

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Posgrado en Ciencia e Ingeniería de la Computación, Universidad Nacional Autónoma de México

Tesis para obtener el grado de Maestro en Ciencias e Ingeniería en Computación
Primera Edición, 16 de octubre de 2019

Agradecimientos

A la UNAM

Por darme la oportunidad de ser parte de ella, por los conocimientos y valores que he adquirido y me han ayudado en mi formación personal y profesional.

Al Dr. Armando Castañeda Rojano

Director de esta tesis, por el apoyo, la asesoría y la paciencia brindada durante el desarrollo de la misma.

A mi Familia

Por estar conmigo en todo momento y por el apoyo brindado mientras realizaba mis estudios de maestría.

A mis Amigos

Gracias por el tiempo compartido e impulsarme a ser mejor cada día.

A CONACYT

Por la beca otorgada durante mis estudios de maestría, la cual fue parte primordial para que pudiera concluir mis estudios de manera satisfactoria.

Resumen

En este trabajo abordamos el estudio de las cadenas de bloques y los contratos inteligentes. Las cadenas de bloques han tenido auge en los últimos años debido a su uso como una alternativa a los sistemas de transferencia de activos tradicionales, tales como las transferencias bancarias o los sistemas de pago como *Paypal*. Además al agregar la operación de contratos inteligentes, se añade la posibilidad de agregar lógica arbitraria para la ejecución de transacciones. Dividimos el contenido de este trabajo en los siguientes capítulos. En el capítulo uno damos una introducción general del trabajo realizado. En el segundo capítulo, presentamos una introducción a los conceptos de criptografía de llave pública y privada, así como una descripción general del protocolo utilizado en cadenas de bloques como *Bitcoin* y *Ethereum*. En el tercer capítulo damos una introducción a los contratos inteligentes y sus casos de uso, tomando el caso de Ethereum. También introducimos el modelo de la máquina virtual de Ethereum que permite la ejecución de los contratos inteligentes y una breve introducción a uno de los lenguajes de alto nivel para implementarlos (*Solidity*). En el capítulo cuatro presentamos varias de las problemáticas que existen al usar contratos inteligentes y algunos patrones de diseño utilizados para mitigar varias de estas problemáticas. En el capítulo cinco, mostramos un par de soluciones a una problemática conocida como *Dependencia en el Orden de Transacciones*, la cual puede ser vista como una condición de carrera en la ejecución de los contratos. La primera solución es un patrón motivado por soluciones para problemáticas en computación concurrente, donde la dependencia en el orden de las transacciones es similar al problema de actualizar una variable compartida. La segunda solución es una propuesta a modificar el lenguaje Solidity y su máquina virtual, de modo que permita le ejecución de *funciones anónimas* para realizar transacciones. Por último, finalizamos con el capítulo seis en el que presentamos conclusiones y trabajo a futuro.

Índice general

Índice de figuras

Índice de códigos

No podemos esperar que gobiernos, u otras grandes corporaciones sin rostro nos otorguen privacidad fuera de su beneficencia. Es una ventaja para ellos hablar de nosotros, y nosotros debemos esperar que ellos hablen. Tratar de evitar su discusión es luchar contra las realidades de la información. La información no sólo quiere ser libre, anhela ser libre.

Manifiesto Cypherpunk

CAPÍTULO 1

Introducción

1.1 Contexto

Con el avance de la tecnología, en la actualidad, el comercio en Internet se basa principalmente en instituciones bancarias que sirven como terceros de confianza para procesar pagos electrónicos. Sin embargo, hay propuestas para realizar pagos electrónicos y otras transacciones sin el uso de un sistema bancario, ya sea por cuestiones de privacidad, confianza o seguridad. En los años ochenta, David Chaum introduce las primeras ideas de dinero digital como una alternativa al dinero físico [13]. Durante los años noventa, se desarrollaron propuestas de dinero digital, algunos de estas fueron: Digicash [12], E-gold [23], Bitgold [40], entre otros. Todas estas propuestas fueron precursoras de Bitcoin [33], que es una de las principales monedas digitales que actualmente operan. Bitcoin es un sistema distribuido para realizar transacciones, basado en pruebas criptográficas en lugar de confianza, haciendo que a esta moneda digital también se le nombre criptomoneda. Una de las particularidades que provee Bitcoin, es la de evitar el *doble gasto*.

El doble gasto es el riesgo de que una moneda digital pueda ser gastada más de una vez. Este es un problema potencial asociado a las monedas digitales, debido a que la información digital puede ser reproducida con una relativa facilidad y pueda ser utilizada para pagar dos o más veces con una sola moneda. Bitcoin resuelve este problema utilizando una red entre pares (*P2P — peer-to-peer*), en la que cada

participante ayuda a agregar y verificar y almacenar transacciones realizadas en el sistema. Estas transacciones son almacenadas en una estructura de datos conocida como *cadena de bloques* (también llamada *blockchain*).

Con el éxito de Bitcoin, surgieron varias criptomonedas y sistemas monetarios digitales basados en cadenas de bloques. Una de estos sistemas es Ethereum, que fue propuesto por Vitalik Buterin [7]. Ethereum al igual que Bitcoin, utiliza una red *P2P* y usa la estructura de cadena de bloques para almacenar las transacciones que se realizan, además del uso de criptografía de llave pública para realizar ejecutar y verificar dichas transacciones. La principal diferencia que hay entre Bitcoin y Ethereum, es que en este último, además de realizar transacciones, se puede agregar lógica adicional a las transacciones, utilizando programas que se ejecutan en la infraestructura de la cadena de bloques. A estos programa se les conoce como *Contratos Inteligentes*.

Los contratos inteligentes son una analogía a los contratos tradicionales, ya que permiten establecer acuerdos entre dos o más personas. Al ser programas computacionales, tienen ventajas sobre los contratos tradicionales; podemos listar por ejemplo, la comprobación automática de reglas y acuerdos, ejecución en minutos, menor costo, autonomía, precisión y seguridad criptográfica. Algunos ejemplos de casos de uso para contratos inteligentes son:

- Contratos para el comercio.
- Registros médicos.
- Registros de propiedad.
- Hipotecas.
- Seguros.
- Votaciones.
- Transacciones punto a puto.
- Cadenas de suministros.
- etc[1].

Con las aplicaciones anteriores, los contratos inteligentes se vuelven una propuesta interesante contra los modelos clásicos de contratos y ejecución de transacciones.

[1]Se pueden navegar los contratos inteligentes desplegados en la red de Ethereum en la siguiente dirección `https://etherscan.io/contractsVerified`

1.2 Problemática

La principal función de las cadenas de bloques es el intercambio de activos digitales a través de transacciones. Ethereum es una plataforma que permite extender las reglas operativas para el uso de transacciones, utilizando un lenguaje Turing-completo para implementar programas que se ejecutan en la infraestructura de la cadena de bloques. Con este nuevo modelo de cómputo, hay características que se tienen que estudiar. Un ejemplo, es la forma en que se ejecutan los contratos inteligentes dentro de la infraestructura de la cadena de bloques de Ethereum. En la práctica, la ejecución es determinista, pero hay partes de la infraestructura donde la ejecución tiene comportamientos no esperados.

Debido a que la función principal de las cadenas de bloques es la de permitir la ejecución de transacciones de activos y registrarlas, se vuelven un blanco atractivo para los atacantes que busquen explotar vulnerabilidades existentes en las mismas, así como en los contratos inteligentes. En [3] se propone una clasificación de vulnerabilidades en las cadenas de bloques. Esta clasificación se divide en tres tipos:

- Vulnerabilidades a nivel de lenguaje.
- Vulnerabilidades a nivel de máquina virtual.
- Vulnerabilidades a nivel de cadena de bloques.

En cada uno de estos niveles hay vulnerabilidades que pueden ser compuestas para tener ataques más complejos, haciendo que sean más difíciles de detectar y enfrentar. La clasificación en tres niveles permite establecer una taxonomía de las vulnerabilidades conocidas hasta el momento. En el capítulo 4 describiremos varias de las vulnerabilidades que existen y cómo es que se relacionan con los diversos niveles de la clasificación anterior. En este trabajo nos enfocaremos a la vulnerabilidad de **dependencia en el orden de las transacciones**.

Una característica de las cadenas de bloques que permiten la ejecución de contratos inteligentes, es que dichas ejecuciones sean siempre deterministas. Existen unos nodos llamados *mineros*, encargados de agregar nuevos bloques a la cadena de bloques. Los mineros ejecutan las transacciones y junto con sus resultados lo agregan a un bloque. Lo anterior permite que las transacciones sean ejecutadas y verificadas nuevamente por cada uno de los participantes del protocolo de la cadena de bloques.

La problemática de la dependencia en el orden de transacciones se da cuando un contrato es implementado utilizando supuestos sobre el estado de la cadena de bloques con el fin de modificarlo. El minero decide arbitrariamente el orden de ejecución de los contratos y puede suceder que ejecute un contrato que modifica el estado de la cadena de bloques, de modo que el contrato original, al ejecutarse quede en un estado inesperado o inconsistente, dando como resultado un aparente no-determinismo en la ejecución.

Para ilustrar esta problemática, pensemos que tenemos un contrato inteligente A, que en su implementación supone que hay valores en la cadena de bloques que se encuentran en cierto estado y desea modificarlos, adicionalmente existe otro contrato inteligente B que también quiere modificar el estado de esos valores, entonces, por azares del destino, ambas transacciones son tomadas por un mismo minero. Este minero decide libremente el orden de ejecución de dichas transacciones. Supongamos que B es ejecutado primero, lo cual hace que el resultado de la ejecución de A sea ó incorrecta ó que falle. Esto hace que la ejecución de los contratos inteligentes lleguen a un estado impredecible. En el capítulo 4 discutiremos más sobre esta problemática, la cual será el eje central de este trabajo.

1.3 Objetivos y Contribución

El objetivo principal de la tesis es explicar el modelo computacional las cadenas de bloques, así como proveer algunas técnicas para realizar la implementación de contratos inteligentes usando técnicas de cómputo concurrente, distribuido y de ingeniería de software, con el fin de mitigar algunas vulnerabilidades.

En este trabajo proponemos dos soluciones para el problema del ordenamiento de transacciones. La primera solución es un patrón de diseño, basándonos en la similitud que existe entre la problemática del ordenamiento de transacciones con el problema que soluciona la primitiva `compareAndSet` de cómputo concurrente. A partir de la idea de la primitiva, se construye el patrón de diseño para implementar contratos inteligentes y mitigar el problema atacado en esta tesis. La segunda solución es proponer una modificación a la máquina virtual de *Ethereum* y al lenguaje *Solidity*, de tal modo que se permita la invocación de transacciones en bloque, de modo similar a como se haría con una función anónima. Esta solución es útil para trabajar con contratos que ya hayan sido desplegados y necesite mitigarse el problema del

ordenamiento de transacciones.

1.4 Estructura de la Tesis

La organización de la tesis es la siguiente:

- El capítulo dos muestra la teoría detrás de las cadenas bloques, así como proveer una introducción de alto nivel a las mismas.
- El capítulo tres describe que son los contratos inteligentes, cómo es que funcionan y el lenguaje con el que pueden ser implementados, así como sus limitaciones.
- El capítulo cuatro discute que son los patrones de diseño y las vulnerabilidades existentes en la implementación de los contratos inteligentes. Además ayuda a describir la problemática que se plantea atacar a detalle.
- El capítulo cinco describe un nuevo patrón de diseño aplicado a los contratos inteligentes, para mitigar las problemáticas planteadas en el capítulo 4.
- El capítulo seis muestra por último las conclusiones y el trabajo a futuro.

Comprendo en principio la importancia de la criptografía de llave pública, pero todo se mueve mucho más rápido de lo que esperaba. No esperaba que se volviera un pilar en la avanzada tecnología de las comunicaciones.

Whitfield Diffie

CAPÍTULO 2

Cadenas de Bloques

En este capítulo describiremos la teoría que fundamenta la operación de la mayoría de las cadenas de bloques existentes, en particular discutiremos cómo opera Bitcoin y Ethereum. Las criptomonedas y las cadenas de bloques, como ya se mencionó, hacen uso de la criptografía para ejecutar la mayoría de las operaciones dentro de sus protocolos, de modo que provean herramientas para satisfacer anonimidad, seguridad y ayuden a prevenir el doble gasto utilizando consenso criptográfico, que describiremos con mayor detalle al final de este capítulo.

Para entender el resto del trabajo, daremos los conceptos necesarios para entender la operación de las cadenas de bloques, además de agregar referencias donde se puedan consultar exposiciones detalladas de cada uno de los temas. Los tópicos que discutiremos serán inicialmente las técnicas de criptografía utilizadas en las cadenas de bloques, seguido de una explicación sobre el modelo de operación de Bitcoin y Ethereum, y finalizamos el capítulo con una breve descripción sobre los algoritmos de consenso basado en prueba de trabajo y prueba de participación.

2.1 Criptografía

La criptografía se ha dedicado al estudio de las técnicas de cifrado en las comunicaciones, cambiando el contenido de un mensaje de modo que sea ininteligible

para receptores no autorizados y sólo pueda ser leído por el receptor esperado. Estas técnicas se usan en el arte, la guerra, la ciencia y la vida diaria. Inicialmente, la criptografía estaba basada en técnicas de sustitución y rotación de símbolos sobre un alfabeto. Durante y después de la segunda guerra mundial, la criptografía tomó un enfoque basado en el álgebra. Una definición actual de criptografía es la siguiente [27]:

> **Definición** *Criptografía* es el estudio de técnicas matemáticas relacionadas con aspectos de seguridad tales como privacidad, confidencialidad, integridad de datos, autenticación de entidades y autenticación de origen de datos.

El uso de la criptografía usualmente es para el envío de información que sea confidencial, segura e íntegra. Los esquemas de envío de información en criptografía, generalmente inician con información (un mensaje) que deseamos comunicar; a esta información se le conoce como *texto plano*. El *texto plano* es *cifrado* con la ayuda de una *función de cifrado* en combinación de una *llave de cifrado*. Este mensaje cifrado permite que sólo el destinatario pueda consultar la información que contiene, impidiendo que cualquier otro individuo que no sea el destinatario pueda conocer su contenido. El destinatario puede acceder a la información *descifrando* el mensaje, usando otra *función de descifrado* (que puede ser la misma función de cifrado), en combinación de una *llave de descifrado* (que puede ser la misma llave de cifrado). Este proceso se puede expresar de forma algebraica de la siguiente forma:

$$c = E_k(m) \tag{2.1}$$
$$m = D_k(c) \tag{2.2}$$

Donde c es el texto cifrado, m es el texto plano, E_k es la función de cifrado y D_k es la función de descifrado y k es la llave. La criptografía se puede categorizar en el número de llaves que son usadas para el cifrado y el descifrado, así como por su aplicación y uso. Las técnicas criptográficas más comunes son las siguientes:

> **Criptografía de llave secreta:** Usa una llave para cifrar y descifrar información, también conocida como *cifrado simétrico*.

> **Criptografía de llave pública:** Usa una llave para cifrar y otra distinta para descifrar información, también conocida como *cifrado asimétrico*. Modificando la operación del protocolo de cifrado, también se pueden proveer *firmas digitales*.

> **Funciones hash criptográficas:** Usa funciones matemáticas para "*cifrar*" de forma irreversible la información, con lo que provee una huella digital única.

Explicaremos a continuación como es que operan a alto nivel cada una de estas técnicas.

2.1.1. Criptografía de Llave Secreta

La *criptografía de llave secreta* emplea una sola llave tanto para cifrado como para descifrado. En general, un agente A tiene una llave secreta k, que comparte con otro agente B, entonces, el emisor A usa la llave k para cifrar el texto plano, transformándolo en un texto cifrado usando una función de cifrado y se la envía al receptor B. B descifra el texto cifrado, usando la misma llave k, que es dada como un parámetro a la función de descifrado. Tanto la función de cifrado y descifrado usan la misma llave, por lo que a la criptografía de llave secreta en ocasiones se le llama *cifrado simétrico*.

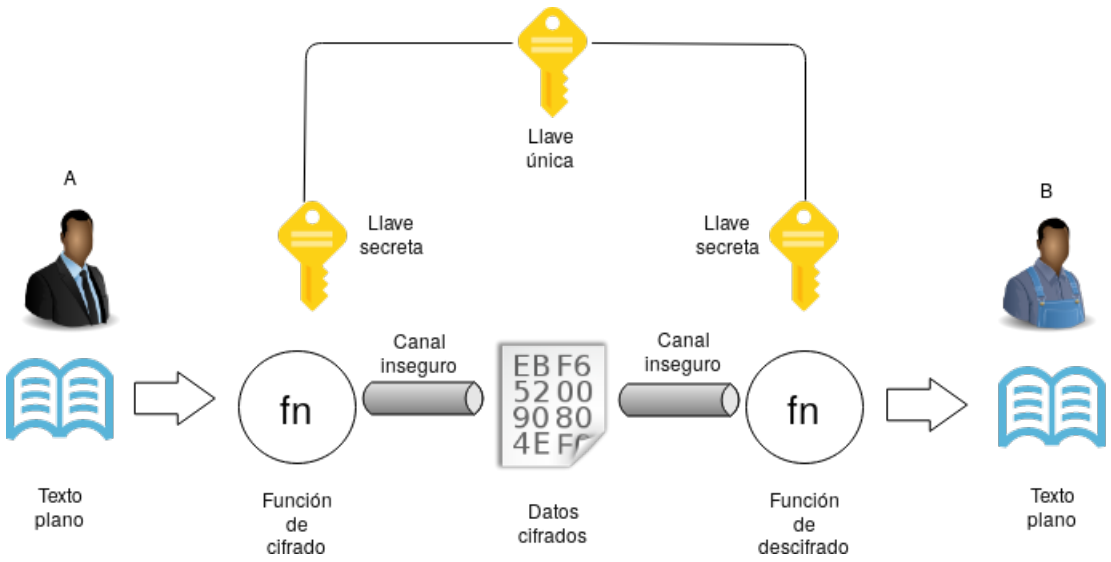

Figura 2.1: Criptografía de llave secreta

En esta categoría de criptografía, la llave debe ser conocida tanto por el emisor como por el receptor, lo que hace que de ahí obtenga el nombre de criptografía de llave secreta, por que la llave es un secreto compartido entre ambos. La mayor problemática que existe con este enfoque, es la distribución de la llave. Los esquemas de cifrado simétrico generalmente se dividen en dos categorías, *cifrado por bloques (block cypher)* y *cifrado de flujo (stream cypher)*, siendo los de bloque los más utilizados. El algoritmo AES (estándar actual) corresponde al esquema de cifrado por

bloques [17, 16]. En la figura 2.1 podemos observar de forma conceptual como es que se realiza el cifrado simétrico en alto nivel.

2.1.2. Criptografía de Llave Pública

En el cifrado asimétrico (criptografía de llave pública), un agente B tiene una llave pública e y su correspondiente llave privada d. La seguridad de estos sistemas radica en que dada la llave e, encontrar d es computacionalmente inviable. La llave pública e permite definir una función de cifrado E_e y la llave privada d permite definir una función de descifrado D_d. Cualquier agente A que desee enviar un mensaje m a B, obtiene una copia auténtica de la llave pública e de B y usa la transformación de cifrado para obtener un texto cifrado $c = E_e(m)$ y transmitir c a B. Para descifrar c, B aplica la transformación de descifrado $m = D_d(c)$ para obtener el mensaje original. Un ejemplo de la operación del cifrado asimétrico se puede observar en la figura 2.2.

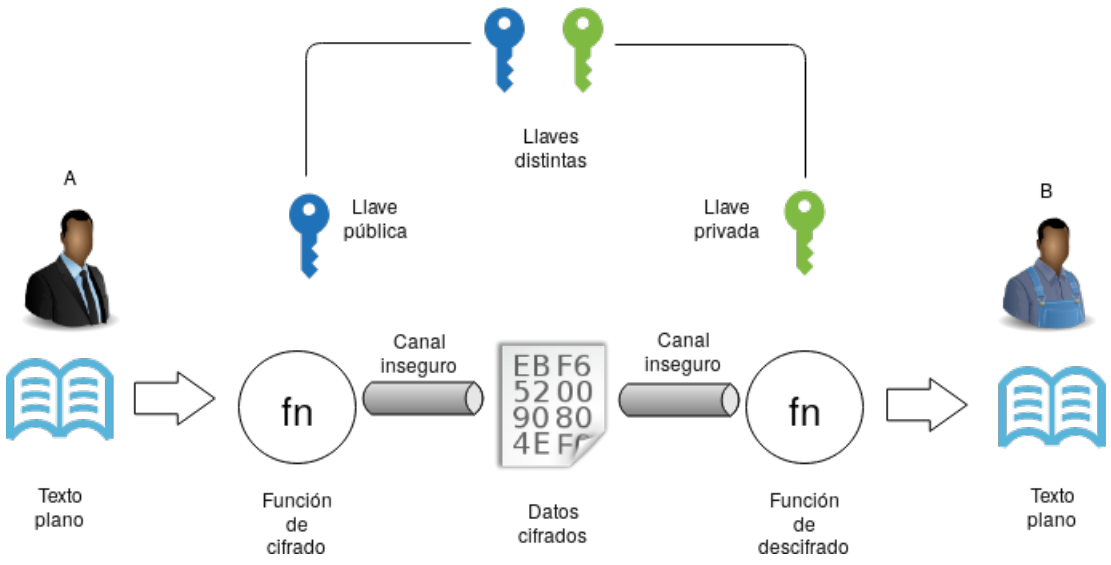

Figura 2.2: Criptografía de llave pública

Actualmente existen dos implementaciones del cifrado asimétrico, uno es el cifrado basado en curvas elípticas [29, 32] y otro es basado en la intratabilidad de la factorización de números primos [35]. Describamos algunas características acerca de estas implementaciones.

Cifrado basado en curvas elípticas: En este tipo de cifrado, se utilizan curvas elípticas del estilo $y^2 = x^3 + ax + b$ sobre un campo finito, generalmente $\mathbb{Z}_p$. El conjunto de soluciones $G = \{(x, y)|(x, y) \in \mathbb{Z}_p \text{ x } \mathbb{Z}_p\}$ de la ecuación, constituyen un grupo finito. A partir de este grupo, se construye la llave pública y la llave privada. Sea $P, Q \in G$ y $d \in \mathbb{Z}$, la siguiente ecuación describe la relación entre llave pública y privada:

$$Q = dP \tag{2.3}$$

Donde, Q es la llave pública, d es la llave privada y P es un punto fijo en G. El grupo G debe satisfacer qué, dados P y Q, calcular el valor de d es computacionalmente difícil. A esta dificultad se le conoce como el problema del logaritmo discreto [32]. El cifrado de datos se puede describir de la siguiente forma [1]:

- A desea mandar datos cifrados a B, para esto, debe conseguir la llave pública (Q) de B.
- A selecciona un entero k aleatoriamente.
- A calcula kP y kQ.
- M es el mensaje que A quiere mandar a B, entonces, A calcula $kQ \oplus M$.
- A manda a B los datos kP y $kQ \oplus M$[a].

El descifrado se realiza de la siguiente forma:

- B recibe los datos kP y $kQ \oplus M$.
- B calcula $d(kP) = kQ$.
- B descifra el mensaje haciendo $(kQ \oplus M) \oplus d(kP) = M$.

[a] $\oplus$ es la suma binaria

Cifrado RSA Este tipo de cifrado aprovecha la dificultad para realizar la factorización de números primos grandes. Para construir la llave pública y privada, se generan dos números primos distintos p y q, ambos del mismo tamaño tomando su número de bits para compararlos. Calculamos $n = pq$ y $\phi = (p - 1)(q - 1)$, posteriormente seleccionamos un entero aleatorio e que satisfaga la relación $1 < e < \phi$, de modo que el máximo común divisor entre e y ϕ sea 1, es decir $\gcd(e, \phi) = 1$. Usando el algoritmo extendido de Euclides, calculamos un número d, de modo que satisfaga $1 < d < \phi$ tal que $ed \equiv 1 \mod \phi$. A partir de

esto, definimos la llave pública como (n, e) y la llave privada es d. El cifrado de datos se puede describir de la siguiente forma [27]:

- A quiere mandar un mensaje a B, para esto debe conseguir la llave pública $((n, e))$ de B.
- Representar el mensaje como un entero m en el intervalo $[0, n-1]$.
- Calcular $c = m^e \mod n$.
- Enviar c a B.

El descifrado se realiza de la siguiente forma:

- B recibe c.
- B, usando la llave privada d recupera el mensaje $m = c^d \mod n$.

La llave pública no necesita mantenerse secreta, y, de hecho, debería estar públicamente disponible, siendo la única restricción la verificación de la autenticidad de la llave e, para garantizar que B es la única parte que va a conocer el mensaje enviado. Una ventaja primaria del cifrado de llave pública es que es más fácil proveer autenticación de llaves públicas que distribuir llaves secretas con seguridad, siendo esto último un requerimiento en los sistemas de cifrado simétrico.

El objetivo principal del cifrado de llave pública es proveer *privacidad* o *confidencialidad*. Debido a que la llave pública e de B es de conocimiento público, el cifrado de llave pública no provee por sí sola *autenticación del origen de datos* o *integridad de la información*. Por tal motivo, para proveer esas funcionalidades, se añaden otras técnicas como *códigos de autenticación de mensajes* y *firmas digitales*.

Los esquemas de cifrado de llave pública son típicamente más lentos que los algoritmos de cifrado de llave simétrica. Por esta razón, comúnmente es usada en la práctica para el transporte de llaves de cifrado simétrico, cifrando las llaves que servirán para establecer un canal con cifrado simétrico y para cifrar elementos de datos pequeños como números de tarjetas bancarias y PINs.

2.1.3. Funciones Hash

Las *funciones hash, de resumen o funciones digestoras*, toman un mensaje como entrada y producen una salida referida como *código de resumen*, *resultado de resu-*

men, *valor hash* o simplemente *hash*. De forma más precisa, una función hash h, mapea cadenas de bits de tamaño arbitrario a cadenas de bits de tamaño n, con n un número fijo. Para un dominio D y un rango R con h: $D \to R$ y $|D| > |R|$, la función es *muchos-a-uno*, implicando la existencia de *colisiones*, que es cuando dos entradas distintas tiene una misma salida para la función hash. Estas colisiones son inevitables. Incluso si restringimos h a un dominio de entradas de t bits con $(t \geq n)$, si h fuera "aleatoria", en el sentido de que todas las salidas fueran equiprobables, entonces cerca de 2^{t-n} entradas podrían caer en alguna salida ya conocida y dos entradas elegidas aleatoriamente, podrían tener probabilidad de $\frac{1}{2^n}$ de que tuvieran la misma salida.

Algunas propiedades de las funciones hash son:

- Son funciones deterministas, es decir, el mensaje digerido siempre arroja el mismo hash.
- Es rápido calcular el hash de cualquier valor.
- Es difícil computacionalmente el generar el mensaje a partir de su hash, siendo la única forma el intentar todos los mensajes posibles.
- Un pequeño cambio en el contenido del mensaje, genera un nuevo hash que no tiene relación con el hash original. Una buena función hash convierte un valor arbitrario en valores aparentemente aleatorios.
- Es difícil computacionalmente encontrar dos mensajes con el mismo hash.

Una característica importante del hash de un mensaje, es que se puede usar como una imagen compacta y representativa del mensaje de entrada, de modo que pueda ser usada como si fuera únicamente identificable con esa cadena. Las funciones hash son utilizadas para verificar la integridad de la información, ya que la más mínima variación, genera una salida distinta. Podemos observar este proceso en la figura 2.3. Actualmente se han especificado varios algoritmos para esta funciones de resumen, siendo las más conocidas las funciones de la familia SHA (SHA1, SHA256, SHA3) [18, 19].

2.1.4. Firmas Digitales

La *firma digital* de un mensaje es una cadena de datos adicional, que permite asociar al mensaje con alguna entidad donde se origina. La firma digital permite que la identidad del firmante y la integridad de la información puedan ser verificadas.

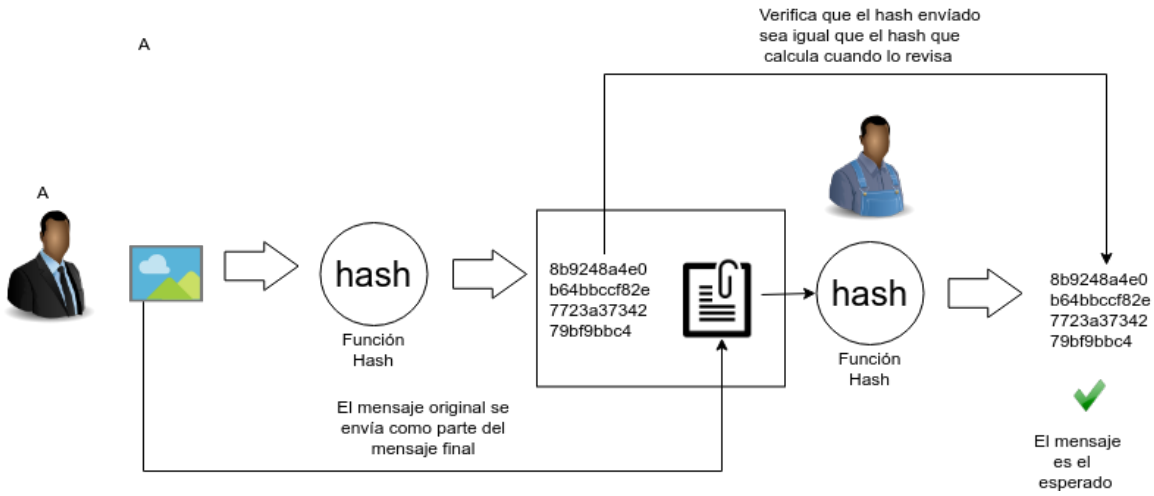

Figura 2.3: Función de resumen

Las firmas digitales tienen varias aplicaciones, como autenticación, integridad de la información y la no repudiación. Consideremos las siguientes definiciones:

- Una *firma digital* es una cadena de datos que asocia un mensaje con alguna entidad que lo genera.
- Un *algoritmo de generación de firmas digitales o algoritmo de generación de firma* es un método para producir una firma digital.
- Un *algoritmo de verificación de firma digital o algoritmo de verificación* es un método para verificar que una firma digital es autentica, es decir, fue creada en efecto por la entidad especificada.
- Una *esquema de firma digital o mecanismo* consiste de una algoritmo de generación de firma y un algoritmo de verificación asociado.
- Un *proceso de firmado digital* consiste de un algoritmo de generación de firma digital, junto con un método para transformar datos en mensajes que pueden ser firmados.
- Un *proceso de verificación de firma digital* consiste de un algoritmo de verificación, junto con un método para recuperar datos desde el mensaje.

En [27] clasifican a las firmas digitales en dos clases generales:

1. Las *firmas digitales con apéndice* requieren el mensaje original como entrada al algoritmo de verificación.

2. Las *firmas digitales con recuperación de mensaje* no requieren del mensaje original como entrada al algoritmo de verificación. En este caso, el mensaje es recuperado desde la firma digital.

En la sección 2.1 mencionamos brevemente que si modificábamos la operación del algoritmo de cifrado de llave pública, obteníamos firmas digitales. Describamos estos esquemas con un poco mayor de detalle y podremos ver la relación que tienen con los algoritmos de cifrado de llave pública.

Esquemas de firmas digitales con apéndice

Los esquemas de firma digital con apéndice son de los más utilizados en la práctica. Operan con ayuda de funciones hash y son menos propensos a ataques de falsificación existencial. Este esquema funciona de la siguiente forma:

Generación de firma y verificación (esquemas de firmas digitales con apéndice) Una entidad A genera una firma s para un mensaje m que posteriormente puede ser verificado por una entidad B.

1. *Generación de firma.* La entidad A debe hacer lo siguiente:
   a) Obtener la llave privada sk_A de A.
   b) Calcular $\tilde{m} = h(m)$ y $s* = S(\tilde{m}, sk_A)$, con $h(\ldots)$ una función hash y S la función de generación de firma digital, la cuál es equivalente al cifrado de llave pública utilizando la llave privada sk_A.
   c) La firma de A para m es $s*$. Tanto m, como $s*$ deben hacerse disponibles a las entidades que van a ser verificadoras.
2. *Verificación.* La entidad B debe hacer lo siguiente:
   a) Obtener la llave pública pk_A de A.
   b) Calcular $\tilde{m} = h(m)$ y $u = V(\tilde{m}, s*)$, con V función de verificación definida como

$$V(\tilde{m}, s*) = \begin{cases} true. & \tilde{m} = D'(s*, pk_A) \text{ con } D' \text{ la función de} \\ & \text{descifrado con } pk_A \text{ como llave.} \\ false, & \text{en cualquier otro caso.} \end{cases}$$

   c) Aceptar la firma digital si y sólo si $u = true$.

Un ejemplo del proceso de firma digital en alto nivel, lo podemos observar en la figura 2.4.

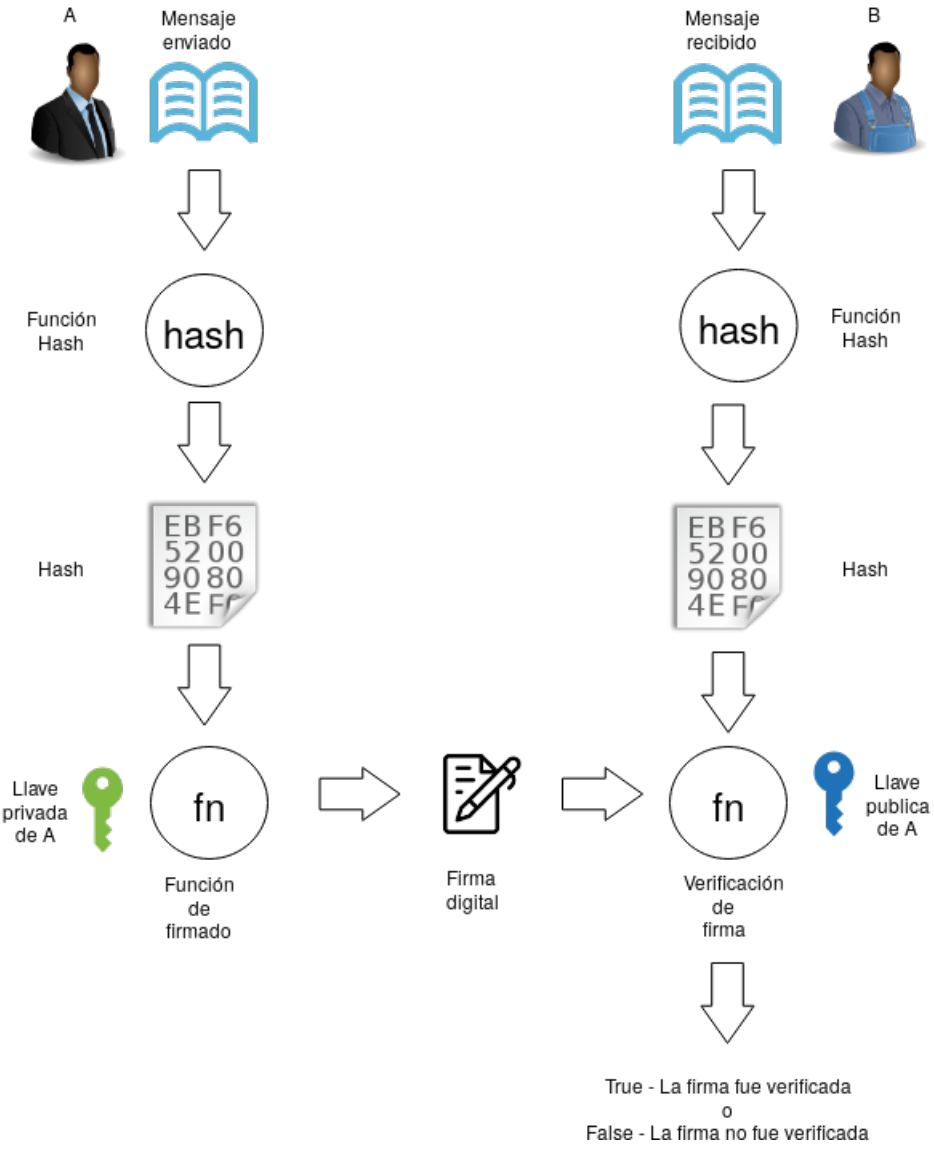

Figura 2.4: Esquema de firma digital con apéndice

Esquemas de firmas digitales con recuperación de mensajes

Los esquemas de firmas digitales con recuperación de mensajes tienen la característica de que los mensajes firmados pueden ser recuperados desde la misma firma. En la práctica, este tipo de firmado es utilizado para mensajes cortos. Además de los algoritmos de firmado y verificación digital, para su proceso necesitan de una función de redundancia. Esta función de redundancia R es una función que produce un mensaje redundante con la información del mensaje que se desea enviar y además debe existir una función inversa R^{-1} que permita obtener el mensaje original desde el mensaje redundante. Denotemos como $\mathcal{M}_{\mathcal{R}}$ a la imagen de R. La función R debe ser elegida con cuidado para evitar problemas discutidos en el capítulo 11 de [27]. Describamos la operación de estos esquemas de la siguiente forma:

> **Generación de firma y verificación (Esquemas de recuperación de mensajes)** Una entidad A genera una firma s para un mensaje m que posteriormente puede ser verificado por una entidad B. El mensaje m es recuperado desde la firma s.
>
> 1. *Generación de firma.* La entidad A debe hacer lo siguiente:
>   *a*) Obtener la llave privada sk_A de A.
>   *b*) Calcular $\tilde{m} = R(m)$ y $s* = S(\tilde{m}, sk_A)$, con S la función de generación de firma digital, la cuál es equivalente al cifrado de llave pública usando la llave privada sk_A y R una función de redundancia.
>   *c*) La firma de A para m es $s*$. Tanto m como $s*$ deben hacerse disponibles a las entidad que van a ser verificadoras.
> 2. *Verificación.* La entidad B debe hacer lo siguiente:
>   *a*) Obtener la llave pública pk_A de A.
>   *b*) Calcular $\tilde{m} = D(s*, pk_A)$.
>   *c*) Verificar que $\tilde{m} \in \mathcal{M}_{\mathcal{R}}$. Si $\tilde{m} \notin \mathcal{M}_{\mathcal{R}}$ rechazar la firma.
>   *d*) Recuperar m desde $\tilde{m}$ al calcular $m = R^{-1}(m)$.

Un ejemplo del proceso de firma digital con recuperación de mensaje, lo podemos observar en la figura 2.5.

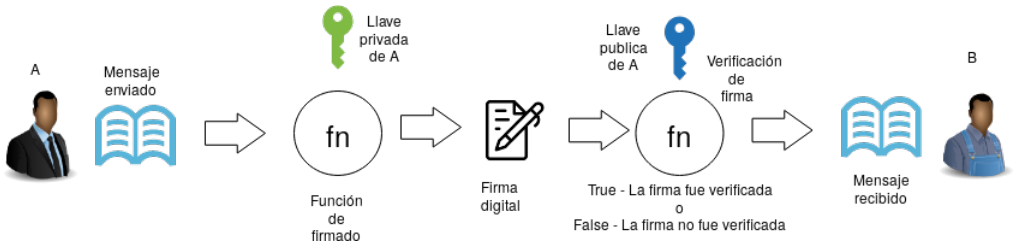

Figura 2.5: Esquema de firma digital con recuperación de mensaje

2.2 Bitcoin

Bitcoin es un protocolo basado en redes P2P que se utiliza como moneda digital, sistema de pago y mercancía. Está basado en la tecnología de cadena de bloques. El uso de las cadenas de bloques en las criptomonedas se puede concebir como un libro notarial público, distribuido y descentralizado que se utiliza para registrar transacciones a través de varias computadoras. Las transacciones son transferencias de activos en Bitcoin, que se realizan entre pares, las cuales se difunden a la red y se colectan en bloques.

La cadena de bloques es una lista creciente de registros de transacciones agrupadas en bloques. Cada bloque contiene un conjunto de transacciones válidas, donde cada uno de los bloques está enlazado de forma criptográfica. Para realizar este enlace entre los bloques, cada bloque contiene el hash criptográfico del bloque previo, una *marca de tiempo* y datos de transacciones, como se ilustra en la figura 2.6. Esto permite el intercambio de valores sin la necesidad de tener una autoridad central o un agente regulador. La cadena formada confirma la integridad de todos los bloques, desde el último añadido hasta el bloque génesis o bloque inicial.

Una transacción corresponde al envío de bitcoins dentro de la infraestructura de Bitcoin. Cada transacción hace referencia a salidas de transacciones previas como entradas para una nueva transacción, y todos los valores de entrada se convierten en nuevas salidas. Las transacciones no se encuentran cifradas para que puedan ser públicamente consultadas, así que cualquier persona que tenga acceso a la cadena de bloques puede conocer el estado de las cuentas del sistema, ya que sólo es necesario que las transacciones se ejecuten una en una y en orden. Además, una vez que las transacciones se integran con la cadenas de bloques, estas se vuelven irreversibles.

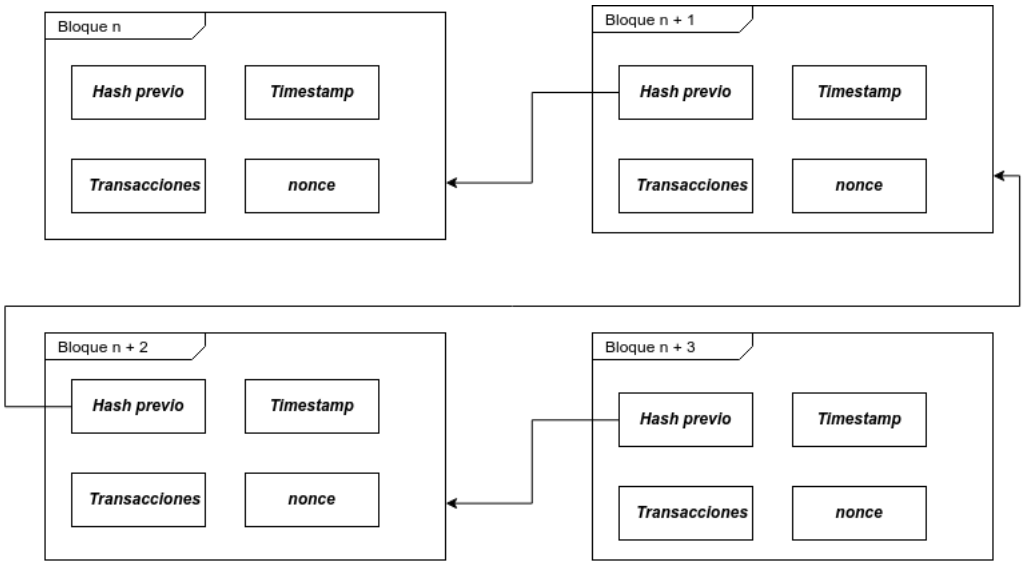

Figura 2.6: Secuencia de bloques de bitcoin, en la que cada bloque hace referencia al bloque anterior a través de su hash.

Al ser un protocolo P2P, los equipos que participan en la red de Bitcoin son iguales entre si y a estos equipos se les conoce como *nodos*. Los nodos se encargan de ejecutar el protocolo y proveer servicios para su operación. Todos los nodos proveen enrutamiento, descubrimiento de nuevos nodos y retransmisión de transacciones. Cada uno de los nodos participantes almacena una copia de la cadena de bloques. Aunque los nodos dentro de la red son iguales, cada uno puede asumir distintos roles dependiendo de la funcionalidad que estén apoyando. Los roles pueden ser: emisión de transacciones, retransmisión o propagación de transacciones o una combinación de las anteriores incluyendo minado (*mineros*). El minado es la actividad que realizan algunos nodos para intentar agregar nuevos bloques a la cadena de bloques. Se le llama minado por la semejanza a la actividad de estar removiendo tierra con maquinaria pesada para obtener oro en cantidad suficiente como para obtener ganancia, sólo que en el caso de Bitcoin, la maquinaria son las computadoras que realizan muchos cálculos computacionales para obtener incentivos, que son nuevas monedas y las comisiones de las transacciones.

2.2.1. Ciclo de Vida de una Transacción

Las transacciones son uno de las principales componentes que existen en Bitcoin. Se busca que ellas se puedan ejecutar de una forma segura, secreta y anónima. Las transacciones son creadas, propagadas en la red y validadas. A continuación describimos el ciclo de vida de una transacción:

1. Un nodo crea la transacción para realizar la transferencia de activos entre pares.
2. El emisor firma la transacción con una o más firmas digitales, las que permiten la autorización del gasto de los valores.
3. La transacción se propaga en la red y llega a los mineros, ellos la depositan en un conjunto de transacciones, de donde las toman para añadirlas a nuevos bloques.
4. La transacción es validada en el momento de que un minero mina un nuevo bloque de la cadena de bloques.
5. Si el minero satisface la prueba criptográfica, el bloque con la transacción se agrega a la cadena de bloques y el minero notifica a los demás nodos sobre el nuevo bloque. A este protocolo se le conoce como consenso criptográfico, y la versión que usa Bitcoin es llamada prueba de trabajo (*proof-of-work*) que se explicará con mayor detalle en la sección 2.3.1.
6. El bloque es añadido a las copias de la cadena de bloques de cada uno de los participantes.

Las transacciones son confirmadas cuando ya se agregaron suficientes bloques subsecuentes. Esto es debido a que puede suceder una bifurcación (fork) de la cadena de bloques, donde puede pasar que dos mineros distintos encuentren una solución al acertijo criptográfico. Entonces en ese momento existen una cadena de bloques con dos ramificaciones, cada una de ellas con bloques distintos. Ambas ramificaciones pueden crecer, pero sólo una de ellas se volverá la oficial. Bitcoin resuelve este problema tomando la cadena más larga. Después de unos cuantos bloques y se haya elegido la cadena oficial, la transacción se vuelve una parte permanente de la cadena de bloques y Bitcoin. Este ciclo se repite constantemente, mientras haya transacciones sin agregar y validar en la red. Podemos visualizar a grandes rasgos este ciclo de vida en la figura 2.7.

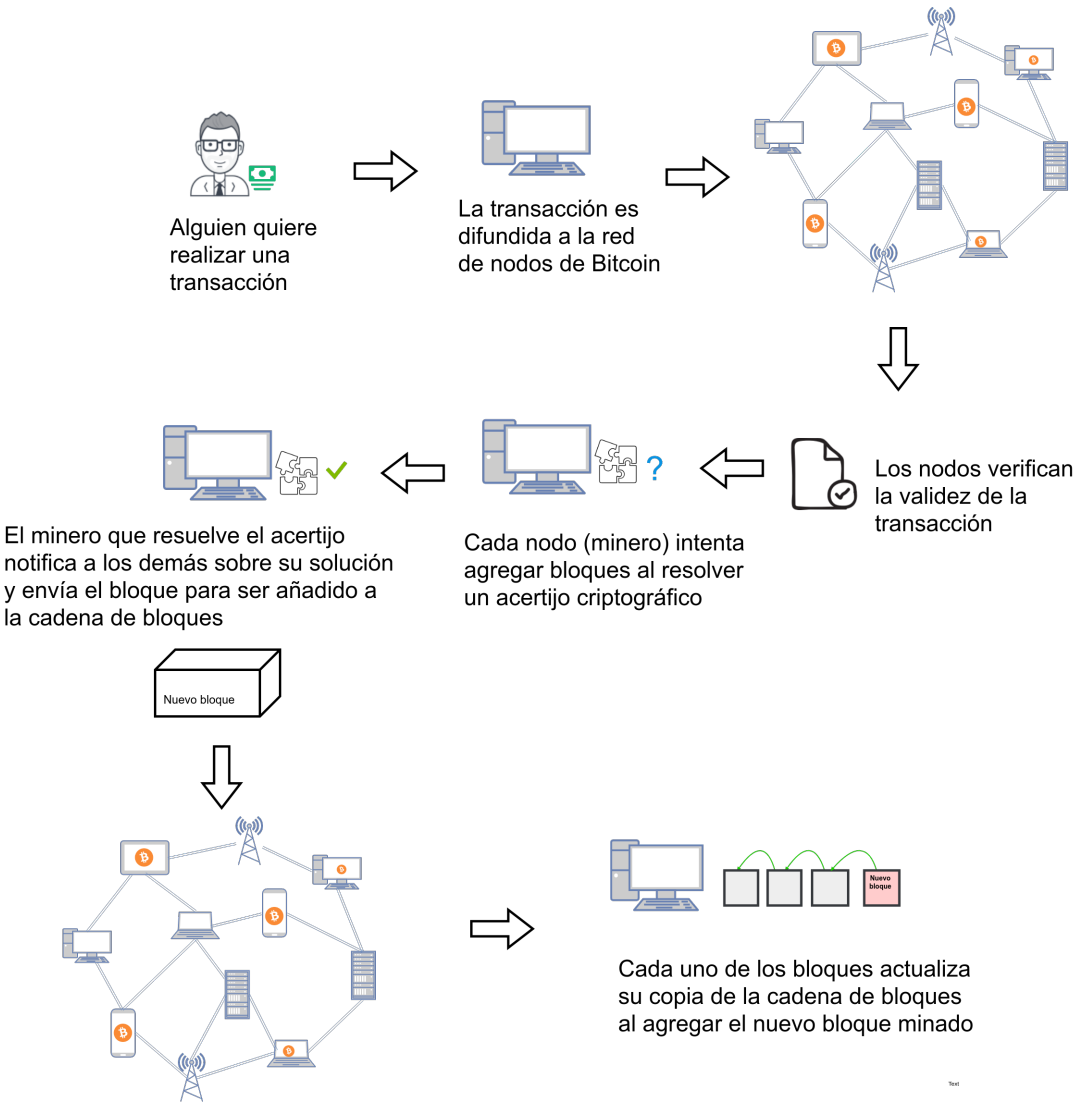

Figura 2.7: Ciclo de vida de una transacción

2.3 Algoritmos de Consenso

Las cadenas de bloques son un sistema distribuido y descentralizado, lo cuál significa que los participantes en el protocolo de la cadena de bloques son una parte importante de su ejecución. Esto garantiza que no hay un agente centralizador que

lleve el control de la ejecución del protocolo. Además, como la cadena de bloques maneja transacciones financieras, es importante que todos los participantes estén de acuerdo con el protocolo y estén sincronizados.

Durante el ciclo de vida de una transacción, hay un momento en que un minero genera un nuevo bloque para añadir a la cadena de bloques. Sin embargo, este proceso es realizado por todos los mineros participantes en el protocolo de la cadena de bloques. Entonces, ¿cómo llegan a consenso sobre el bloque a añadir en la cadena de bloques?

El *problema del consenso* en cómputo distribuido se refiere al hecho de que dado un sistema con múltiples procesadores p_i, los procesadores inician con valores individuales desde un conjunto particular V. Todos los procesadores son requeridos para producir salidas desde el mismo conjunto de valores V, sujeto a un acuerdo y condiciones de validez (Por validez, asumimos que si todos los procesadores inician con el mismo valor v, el único valor de decisión permitido es v) [31]. Existen varios algoritmos para resolver el consenso, en el que las soluciones propuestas tienen diferentes hipótesis del sistema en el que operan. En el caso de las cadenas de bloques, el valor de entrada es un nuevo bloque que satisface que fue generado usando prueba de trabajo o prueba de participación y la decisión es el bloque que satisfizo la prueba correspondiente.

Actualmente, hay dos algoritmos de consenso que se discuten y se usan mayoritariamente en las cadenas de bloques. El primero es el algoritmo de consenso denominado prueba de trabajo (*proof-of-work*) y el segundo es el de la prueba de participación (*proof-of-stake*). El primero es utilizado tanto por Bitcoin como por Ethereum, mientras que el segundo, actualmente forma parte de una estrategia por parte del equipo de desarrollo de Ethereum para remplazar el primero, además de formar parte de otros sistemas basados en cadenas de bloques.

2.3.1. Prueba de Trabajo

Tanto en Bitcoin como en Ethereum, el ciclo de vida de las transacciones son similares, cuando se mina un nuevo bloque para la cadena de bloques, el modo en que se elige ese bloque es a través de un algoritmo conocido como prueba de trabajo (*proof-of-work*). La idea básica de este algoritmo es que los mineros resuelvan un acertijo criptográfico, que es propuesto por el mismo sistema. Este acertijo consiste en que los mineros deben de calcular el hash del bloque que están proponiendo a la

cadena de bloques y ver que cumpla ciertas restricciones, por ejemplo, que la cadena del hash tenga una cantidad determinada de ceros al principio o que el valor del hash este entre ciertos rangos. Si el hash que calculan no cumple con el requisito, modifican un valor del bloque y vuelven a calcular el hash hasta que lo satisfagan. De este modo, la prueba de trabajo radica en que el minero pruebe que invirtió una cantidad significativa de trabajo para crearlo, de modo que se asegure que los nodos honestos y deshonestos tengan que invertir la misma cantidad de trabajo para añadir un bloque. Esto hace que para un nodo deshonesto, el agregar bloques fraudulentos no es un buen negocio, ya que el costo computacional de agregarlo es alto comparado con la ganancia de agregarlo a la cadena de bloques. Otra característica relacionada con la prueba de trabajo, es que el modificar la historia de la cadena de bloques, se requiere mucho costo computacional para modificar la cadena y tener un número mayor al 51 % de nodos deshonestos [42].

Esta prueba de trabajo es similar a la propuesta por Adam Back en Hashcash [4]. Aquí se aprovecha de la aparente naturaleza aleatoria de las funciones hash criptográficas. Una buena función hash criptográfica transforma datos arbitrarios en valores aparentemente aleatorios. Esto permite que si se modifica un sólo bit del conjunto de datos que se evalúa en la función hash, el valor de salida va a ser totalmente distinto y no va a tener correlación alguna con el valor original de los datos sin modificar. Esto permite que los valores de salida de la función de resumen sean impredecibles.

El encadenamiento de los bloques se realiza usando los valores de la función hash, haciendo imposible modificar transacciones en cualquier bloque sin modificar los bloques subsecuentes, lo que lleva como resultado, que el costo de modificar un bloque en particular, incremente con cada bloque añadido a la cadena de bloques, modificando el efecto de la prueba de trabajo.

Prueba de trabajo Cada minero toma un conjunto de transacciones y las añade a su bloque.

- El minero construye el bloque usando las transacciones elegidas, agrega una marca de tiempo y utiliza un número (*nonce*) que va a ser el que varía para generar el nuevo valor de resumen del bloque.
- Siguiendo el protocolo de la cadena de bloques, la red decide cuál va a ser el rango de valores que satisfacen su acertijo criptográfico (el hash a satisfacer). Esto lo hace calculando un número denominado *nbits* [41],

que es el umbral para el hash de los bloques que satisfacen el acertijo criptográfico. Este valor es calculado dividiendo el tiempo de la dificultad objetivo entre la dificultad actual para encontrar los bloques. Este valor permite fijar la cantidad de ceros en el prefijo que debe satisfacerse durante la prueba de trabajo.

- El minero calcula el valor hash del bloque más un *nonce* para incrementar el valor. Debido a que las funciones hash suelen devolver valores aparentemente aleatorios para distintos valores de entrada, no es posible encontrar el valor del hash que debe satisfacer el acertijo criptográfico más que intentar con todos los valores del *nonce*. Si el hash satisface que está en el rango, entonces el minero gana el consenso.
- El minero ganador notifica a todos los demás que ha resuelto el acertijo criptográfico y los demás nodos se encargan de probar que su nodo satisface el acertijo.
- Después de validar el bloque, los nodos se encargan de añadir el bloque a su copia de la cadena de bloques.

El proceso anterior se repite tantas veces como bloques se quieran generar para añadir a la cadena de bloques.

2.3.2. Prueba de Participación

Una de las problemáticas que tiene el algoritmo de consenso de prueba de trabajo referido en la sección 2.3.1, es que es altamente costoso. La página de *bitcoincharts.com*[1] refiere que se están calculando aproximadamente 6.35 trillones de hashes por segundo en la red de Bitcoin. Se han hecho estudios que muestran que el consumo energético es muy alto, por ejemplo, el proceso de minado de bloques consume 73.12 Terawatts, lo cuál es comparable al consumo de energía de Austria. En el caso de la huella de carbono, se producen 34.73 millones de toneladas de CO2, comparable a la huella de carbono de Dinamarca [15]. Aunque la prueba de trabajo es el primer algoritmo de consenso que se aplicó en las cadenas de bloques,se han desarrollado otros algoritmos más eficientes.

Uno de los algoritmos que discutiremos es el algoritmo de consenso conocido como prueba de participación. En este algoritmo, el minero (o validador como se le refiere en este tipo de algoritmo de consenso) se encarga de probar que tiene una cierta de cantidad de activos, lo que da representatividad dentro del conjunto

[1] `https://bitcoincharts.com/bitcoin/`

de interesados. La idea simple de este tipo de algoritmo de consenso, es elegir un grupo de validadores para proponer y votar el siguiente bloque. El peso de cada voto depende de la participación o interés propuesto por cada minero. La documentación de Ethereum [44] explica que el algoritmo para la prueba de participación se puede describir del siguiente modo:

- El sistema de la cadena de bloques realiza un seguimiento de un conjunto de validadores.
- Si hay nodos que tienen criptomonedas base, pueden realizar un depósito de sus criptomonedas como una forma de decir que están interesados en formar parte del conjunto de validadores en el protocolo.
- Los validadores comienzan a crear bloques a través de una de las variantes del algoritmo de consenso basado en prueba de participación. Los más comunes son los pruebas de participación basada en la cadena de bloques y prueba de participación estilo BFT (*Bizantine Fault Tolerant*).
- La elección del bloque se realiza y se agrega la cadena de bloque.
- Los validadores reciben una recompensa proporcional a su depósito.

En el algoritmo de consenso de prueba de participación basado en la cadena de bloques, se elige de forma aleatoria a un validador cada cierto tiempo y ese validador se encarga de crear un nuevo bloque que apunta a otro anterior (generalmente el último de la cadena). En el caso del consenso BFT, de forma aleatoria, a los validadores se les asigna el derecho de proponer nuevos bloque. Cada validador propone su bloque y para elegir el bloque a añadir, se realiza una serie de rondas de votos. Al final de las rondas se elige cuál es el bloque que se va a añadir a la cadena de bloques. Un ejemplo del uso de esta variante la podemos encontrar en la cadena de bloques Algorand [22], en la que hacen consenso con prueba de participación a través de esquemas de sorteos criptográficos, en la que asigna pesos a los usuarios, el cual es proporcional al valor monetario que ellos tienen. Otro protocolo que también usa un esquema bizantino es la nueva implementación de Ethereum para su algoritmo de consenso, al cual han denominado CASPER [9].

2.4 Ethereum

Además de Bitcoin, hay otras criptomonedas basadas en la tecnología de las cadenas de bloques. Una de las más importantes es Ethereum. Ethereum es una cadena

de bloques diseñada por Vitalik Buterin [7]. Esta plataforma difiere de Bitcoin en el hecho de que Bitcoin se especializa sólo en la ejecución de transacciones, mientras que Ethereum añade una capa adicional al permitir la ejecución de programas que extienden las capacidades de las transacciones. Estos programas son lo que denominan contratos inteligentes (*smart-contracts*). Esta capa permite la ejecución de los contratos inteligentes al añadir una máquina virtual que ejecuta los contratos inteligentes utilizando un lenguaje cuasi-Turing completo[2]. Al implementar estos contratos inteligentes, se pueden crear aplicaciones descentralizadas con sus propias reglas, formatos de transacción y funciones de transición de estado. En Ethereum, los contratos inteligentes son un atributo de una cuenta de contrato. Más adelante discutiremos acerca de los tipos de cuentas que existen.

Ethereum maneja el ciclo de vida y la ejecución de transacciones de manera similar a Bitcoin, agregando el hecho de que la ejecución del código de un contrato se realiza al construir un nuevo bloque. Cuando un minero construye un nuevo bloque, el código del contrato es incluido dentro de una transacción, dicho código es ejecutado por el minero, consultando su copia local de la cadena de bloques para determinar sus salidas. Las salidas son añadidas al bloque y si dicho bloque satisface la prueba de trabajo, entonces, de forma similar a Bitcoin, el minero agrega el bloque a la cadena de bloques y propaga el bloque a sus vecinos y cada uno de ellos se encarga de verificar las transacciones del bloque, así como realizar la ejecución del código del contrato y verificar que las salidas que genera sean iguales a las que contiene el bloque propagado.

Ethereum puede ser visto como una máquina de estados basada en transacciones, empezando con un estado inicial y de forma incremental, ir ejecutando transacciones para llegar a algún estado final. El estado puede incluir información sobre balance de cuentas, creaciones de contratos, ejecuciones, datos sobre el mundo, etc. Las transacciones representan un arco válido que representa la transición entre dos estados. Una transición válida es aquella que es producida por una transacción. Formalmente describimos:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_i, T) \tag{2.4}$$

donde Υ es la función de transición de Ethereum, σ_t corresponde con el estado en el tiempo t y σ_{t+1} el estado en el tiempo $t+1$. Las transacciones son recogidas en

[2]Decimos que es cuasi-Turing completo debido a que está limitado por la cantidad de gas disponible para ejecutar las instrucciones de los contratos

los bloques de la cadena de bloques. Desde el punto de vista de los estados, Ethereum puede ser una cadena de estados, en el que cada bloque dentro de la cadena de bloques determina el estado global en un tiempo t y al ejecutar las transacciones de un nuevo bloque que se agrega a la cadena, se llega a un nuevo estado global $t + 1$. Desde el punto de la implementación puede verse como una cadena de bloques de manera equivalente y desde el punto de vista de un libro notarial, se puede ver como una pila de transacciones realizadas a lo largo del tiempo.

El estado global es un mapeo entre direcciones y estados de cuenta. Una *cuenta* en Ethereum consiste de una dirección como identificador y un estado de cuenta, conceptualmente podemos ver cómo es una cuenta en la figura 2.8. Un estado de cuenta consiste de 4 campos:

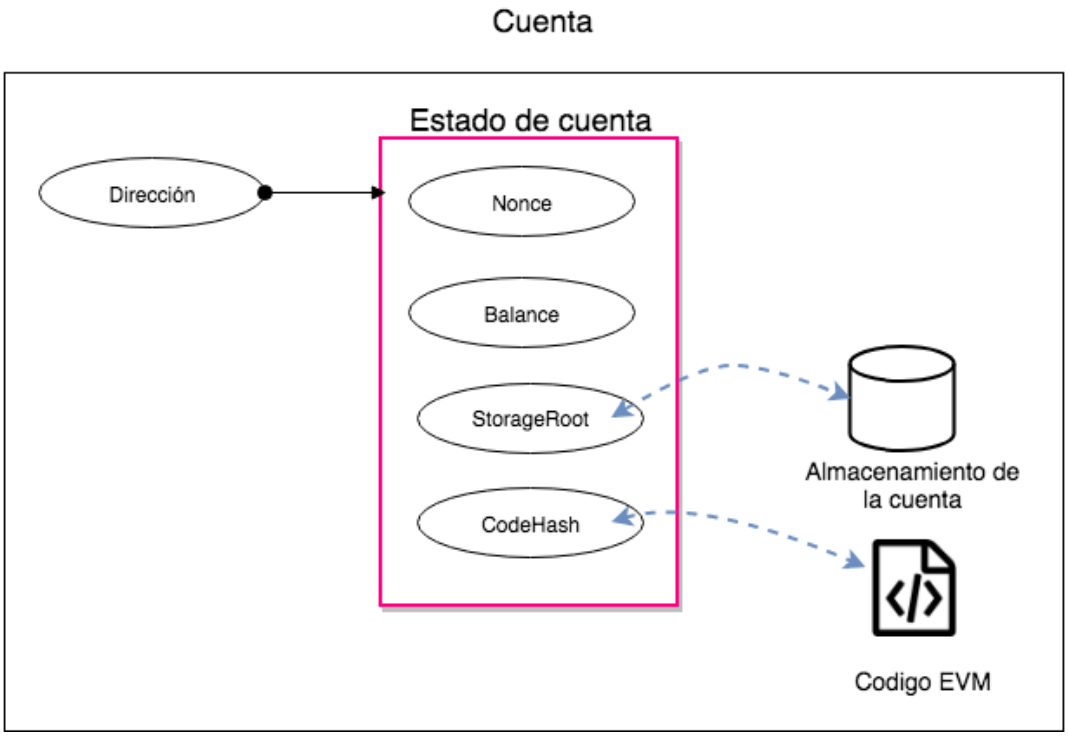

Figura 2.8: Una cuenta consiste de una dirección y un estado de cuenta

- ***Nonce*:** un valor escalar igual al número de transacciones enviada desde la dirección de la cuenta o el número de contratos creados por la cuenta.
- ***Balance*:** un valor escalar igual a la cantidad de Wei[3] de la que es dueña la cuenta.
- ***StorageRoot*:** Un hash de 256 bits que corresponde con el hash de la raíz de un árbol de Merkle-Patricia[4] que codifica el contenido almacenado por la cuenta.
- ***CodeHash*:** El hash del código de EVM de esta cuenta si es que hay. Este código es ejecutado si esta dirección recibe un mensaje. Además es inmutable y no puede ser modificado después de que se construye el objeto de la cuenta.

Existen dos tipos de cuentas:

1. Cuentas externas (*Externally owned acounts*), controladas por llaves privadas criptográficas. A través de la llave privada el usuario se autentica para poder utilizar su cuenta y realizar operaciones en la cadena de bloques (figura 2.9).
2. Cuentas de contrato (*Contract accounts*), que son cuentas que son controladas por el código de un contrato inteligente asociado (figura 2.10).

Una transacción T es una instrucción firmada criptográficamente construida por un agente externo a la cadena de bloques. Existen dos tipos de transacciones, la primera de ellas es la creación de contratos y la segunda un envío de mensaje, que puede ser a una cuenta externa o a una cuenta de contrato. Ambas se pueden ver

[3]En Ethereum, la denominación de una unidad de medida de las criptomonedas es el ether, pero hay otras denominaciones como el Wei que es la unidad más pequeña que hay. Una relación entre las medidas desde la más pequeña hasta la más grande son las siguientes:

- Wei
- Lovelace (1000 Wei),
- Babbage (1000 Lovelace)
- Shannon (1000 Babbage)
- Szabo (1000 Shannon)
- Finney (1000 Szabo)
- Ether (1000 Finney)

[4]De forma intuitiva, el árbol de Merkle-Patricia es estructura de datos arborescente, que mantiene sus valores en las hojas (las transacciones) y para todos aquellos nodos que no son hojas, el valor que almacena es el hash de la concatenación de los hashes de cada uno de sus hijos, de modo que este proceso se repita hasta llegar a la raíz del árbol.

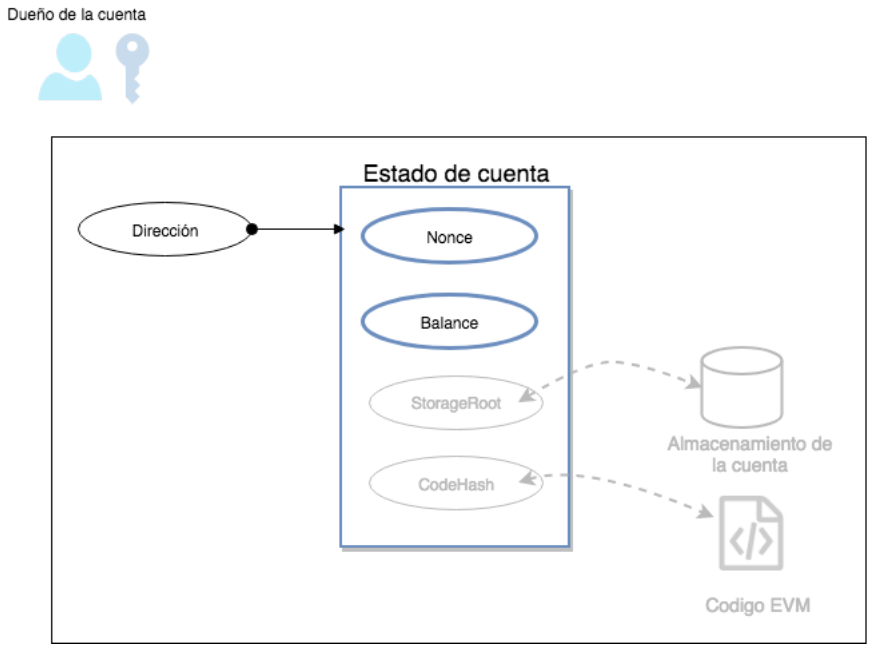

Figura 2.9: Las cuentas externas son controladas por llaves privadas, además de no contener código de la EVM.

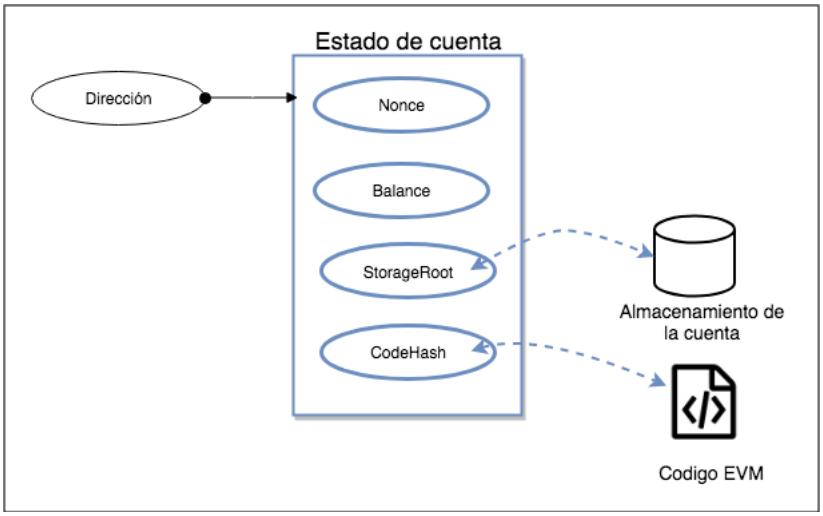

Figura 2.10: Las cuentas de contrato contienen código de la EVM y son controladas por ese mismo código.

como invocaciones que cambian el estado global de un estado σ_t a un nuevo estado σ_{t+1}. Durante la transacción de creación de contrato, se crea una nueva cuenta de contrato, la cual se añade a la cadena de bloques con su dirección y el nuevo estado de la cuenta, incluyendo el código del contrato asociado y su espacio de almacenamiento para los resultados de la ejecución del contrato. En el caso de la transacción de envío de mensajes, puede ser utilizado para realizar la ejecución de un contrato enviando datos y actualizar el estado. Una transacción contiene los siguientes campos:

- Nonce.
- Precio del gas.
- Límite de gas.
- Dirección a quien se envía la transacción. Es una dirección de 160 bits si corresponde con un mensaje o 0 si es la creación de un contrato.
- La cantidad de Wei transferidos.
- v, r, s, valores utilizados para la firma *ECDSA*[5] que es usada por Ethereum.
- Creación del contrato o datos del mensaje.

Los mensajes son realizados entre dos cuentas y consiste en un conjunto de datos (un conjunto de bytes), y un valor (especificado en Ether). Los mensajes pueden ser lanzados por una transacción o por la ejecución de código de la EVM. Los mensajes se pueden clasificar en cuatro tipos dependiendo si son lanzados por transacciones o por contratos.

1. Un mensaje puede ser lanzado por una cuenta externa y el receptor puede ser también una cuenta externa (envío de Ether).
2. Un mensaje puede ser lanzado por una cuenta externa y el receptor puede ser recibido por una cuenta de contrato (invocando funciones de contratos).
3. Un mensaje es enviado por una cuenta de contrato y el receptor puede ser una cuenta externa (recepción de Ether).
4. Un mensaje es enviado por una cuenta de contrato y el receptor puede ser una cuenta de contrato (comunicación entre contratos inteligentes).

El *ether* es la criptomoneda de Ethereum, la cual es utilizada para realizar el intercambio de activos, pagar cuotas por la ejecución de los contratos y ser el incentivo de los mineros. El pago de la ejecución de los transacciones y contratos se realiza con

[5] *Eliptic Curve Digital Signature Algorithm*

gas. El gas es una unidad que corresponde a la ejecución de una instrucción, a partir del cual, los mineros pueden darle un valor a los recursos consumidos al ejecutar una transacción. El precio de cada unidad de gas (*gas price*), está dado en términos de ethers, lo que permite que los mineros puedan monetizar su trabajo computacional al ejecutar las instrucciones que son dadas en las transacciones. La cantidad de gas a pagar por transacción (o contrato) está determinada por el valor de cada una de las operaciones que tiene, es decir, dependiendo de la dificultad o la rapidez con la que queremos que se ejecute la transacción (o contrato), la cantidad de cómputo requerido aumentará o disminuirá en proporción a dicha dificultad y la cantidad de gas a pagar aumentará o disminuirá de forma correspondiente[6].

Un punto importante en Ethereum respecto a los contratos inteligentes, es que estos no deben considerarse en su totalidad como una ley o un acuerdo legal, sino como agentes autónomos que ejecutan una parte de su código cuando se les envía un mensaje. En el capítulo 3 discutiremos a detalle más características acerca de los contratos inteligentes de Ethereum.

[6]En la literatura se le conoce como *transaction fee*.

Un contrato inteligente es un protocolo computarizado de transacciones que ejecuta los términos de un contrato. El objetivo general es satisfacer condiciones contractuales comunes (tales cómo términos de pago, gravamenes, confidencialidad e incluso cumplimientos), minimizando excepciones tanto maliciosos, como accidentales, y también minimizar la necesidad de tener intermediarios certificados.

Nick Szabo

CAPÍTULO 3

Contratos Inteligentes

Los contratos inteligentes son una forma de extender las capacidades de las transacciones en una cadena de bloques. La forma en que los contratos inteligentes extienden a las transacciones es a través de reglas que se pueden programar mediante un lenguaje cuasi-Turing completo. El ejemplo más común es el lenguaje de la EVM y Solidity, ambos lenguajes para Ethereum. En este capítulo discutiremos muchas de las propiedades que caracterizan a los contratos inteligentes, así como usos y lenguajes que actualmente se usan para implementarlos.

En la sección 3.1 introducimos conceptos sobre los contratos inteligentes. En la sección 3.2 discutiremos en profundidad cómo es que operan los contratos inteligentes, tomando como ejemplo el caso de Ethereum; en la sección 3.3 describiremos cuáles son los usos más comunes que se les dan a los contratos inteligentes. Finalizamos con las secciones 3.4 y 3.5 en donde discutiremos características acerca del lenguaje *solidity*, el cual es el lenguaje más utilizado para implementar contratos inteligentes y describir la estructura que deben tener para su ejecución.

3.1 ¿Qué son los Contratos Inteligentes?

Debido a la naturaleza inmutable de la cadena de bloques, así como su característica de ser un sistema distribuido en el que cada participante tiene una copia de la información de la cadena de bloques, la ejecución de contratos inteligentes se

puede realizar sin que sea corrompida o comprometida. Por ejemplo, si uno de los participantes pierde información, se corrompe o la información local es comprometida, los demás participantes se mantienen sin ser afectados. Pero, ¿qué es un contrato inteligente? Nick Szabo fue el primero en definir el concepto de contrato inteligente antes de la aparición de Ethereum [39]:

> Llamo a esos nuevos contratos "inteligentes", porqué son mucho más funcionales que sus ancestros inanimados basados en papel. Ningún uso de inteligencia artificial está implicado. Un contrato inteligentes es un conjunto de promesas, especificadas en forma digital, incluidos los protocolos dentro de los cuales las partes cumplen sus promesas.

Sin embargo, actualmente consideramos a los contratos inteligentes como un programa de computadora, cuya ejecución se realiza sobre la infraestructura de una cadena de bloques. Los contratos son código que se añade a la cadena de bloques a través de una cuenta de contrato y a través de mensajes (que son envíados dentro de transacciones), se invocan y se ejecutan cuando se incluyen en nuevos bloques durante la fase de minado. Uno de los principales objetivos de los contratos inteligentes es el automatizar las transferencias de activos entre pares, en el que se incluyen un conjunto de reglas específicas para garantizar que las transferencias se realicen de acuerdo con esas reglas.

Aunque podemos implementar contratos inteligentes a través del lenguaje de bajo nivel de la EVM, resulta difícil construirlos para la mayoría de las personas. Sin embargo, se han realizados esfuerzos para implementar lenguajes de alto nivel para su implementación. Actualmente *Solidity* [45], es el lenguaje de alto nivel por defecto para el desarrollo de contratos inteligentes. Existen otros lenguajes que se encuentran en estado experimental o en estado beta que se encuentran en desarrollo, por ejemplo, Viper [46] es un lenguaje con un enfoque *pythonico* en su desarrollo. Otro ejemplo es LLL [43], este es un lenguaje de bajo nivel para la EVM con una sintaxis basada en *S-expressions*, similar a lo provisto en lenguajes tipo Lisp, siendo uno de los primeros lenguajes desarrollados por el equipo de desarrrollo de Ethereum. Existen varios más, pero aún se encuentran en estado experimental.

De todos estos lenguajes, *Solidity* es el lenguaje que vamos a utilizar, debido a que es lenguaje que actualmente tiene un soporte activo por el equipo de Ethereum y en la literatura, la mayoría de las problemáticas que se describen para esta plataforma, son descritas en términos de este lenguaje.

3.2

¿Cómo Operan los Contratos Inteligentes?

En la sección 3.1 hablamos sobre los contratos inteligentes. Mencionamos que, a pesar de que la primera definición dada por Nick Szabo [39], en la que hacia una referencia a protocolos y promesas computacionales usados en la ejecución de contratos en un sentido legal y jurídico, actualmente se entienden como programas de computadora que se ejecutan sobre la infraestructura de una cadena de bloques de forma distribuida, accionados en el momento en que alguna transacción los invoque y se cumpla una o más condiciones definidas dentro del programa.

En Ethereum, los contratos inteligentes ayudan al intercambio de dinero, propiedades, activos o cualquier otro valor de una forma transparente, libre de conflictos y sin el uso de servicios de terceros para lidiar con las transacciones entre pares. Generalmente representan un acuerdo entre dos personas a través de código que se va a ejecutar sobre la infraestructura de la cadena de bloques. Esto permite que se almacenen en la cadena de bloques y que sean inmutables.

¿Cómo funcionan los contratos inteligentes? Para explicar su funcionamiento, presentemos el modelo de la máquina virtual de Ethereum, el cual nos ayudará a entender la forma en que se realiza la ejecución de los contratos inteligentes.

3.2.1. Máquina Virtual de Ethereum

La propuesta principal de Ethereum radica en la ejecución de contratos inteligentes. Para poder realizar esto, agrega una capa adicional que permite la ejecución de los contratos de Ethereum. Esto es posible gracias a lo que se conoce como *Máquina Virtual de Ethereum (EVM)* [45]. Con esta máquina virtual, en un estilo similar al de la máquina virtual de Java, permite la ejecución de bytecode, que es un lenguaje de bajo nivel, en el que secuencias de bytes representan una operación en la máquina virtual. A este lenguaje se le conoce como "*Ethereum Virtual Machine Code*" (código de la EVM). El modelo de ejecución de la EVM es cuasi-Turing completo debido a que las computaciones realizadas están acotadas por un parámetro, el *gas*, que limita la cantidad de cálculos realizados.

La EVM es un modelo que especifica, cómo el estado del sistema es alterado a partir de instrucciones de bytecode y una pequeña tupla de datos de entorno. Esta máquina virtual está basada en una arquitectura simple basada en pilas. El tamaño

de palabra utilizado por la EVM es de 256 bits. Este valor es escogido para facilitar el trabajo con esquemas de valores hash de 256 bits y con operaciones criptográficas basadas en curvas elípticas. El modelo de almacenamiento es similar al de memoria, pero en lugar de que sea un arreglo de bytes, es un arreglo de palabras direccionable, además de que el almacenamiento no es volátil y es mantenido como parte del estado del sistema [49]. Conceptualmente, este modelo lo podemos observar en la figura 3.1.

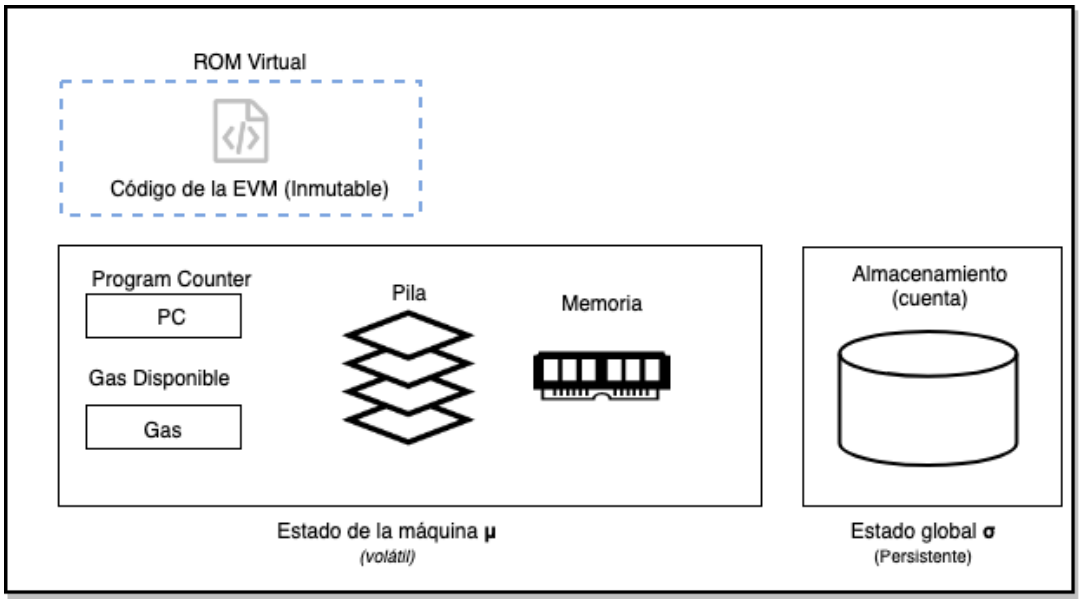

Figura 3.1: Arquitectura de la máquina virtual de Ethereum

Una parte importante del modelo de ejecución es el *fee* (cuota) especificado en gas, utilizado para realizar la ejecución de las transacciones. Este valor se utiliza en tres circunstancias distintas:

1. Pagar el costo de los cálculos realizados por un contrato.
2. Utilizado para pagar la invocación a un mensaje o una creación de contrato.
3. Pagar el incremento de uso de memoria durante la ejecución de un contrato.

Dependiendo de la cantidad de *fee*, se limitará la cantidad de cálculos que realizará la ejecución de un contrato inteligente. Cuando se ejecuta un contrato inteligente, se pasa de un estado global σ_t a otro estado global σ_{t+1} a partir de un mensaje generado por una transacción. La información del código de la EVM en el estado

σ_t es envíado a la máquina virtual y es ejecutado, para posteriormente ser actualizado en la sección de almacenamiento de la cuenta de contrato y llegar al estado σ_{t+1}.

El espacio de memoria con el que opera la máquina virtual se divide en tres tipos de espacio donde guardar información:

1. Pila (*stack*), es un contenedor tipo *LIFO*, donde los valores pueden ser insertados o expulsados. Tiene un tamaño de 1024 con entradas de 256 bits. Todas las operaciones son realizadas en este espacio de memoria y son accedidas a través de varias instrucciones tales como PUSH, POP, COPY, SWAP, etc[1].
2. Memoria (*memory*), que es un arreglo de bytes expandible lineal que puede ser direccionable a nivel de bytes. Se puede acceder a ella con instrucciones del estilo MSTORE, MSTORE8, MLOAD, etc. Todas las locaciones en memoria son bien definidas inicialmente en cero.
3. Almacén de cuenta (*storage account*), que es un diccionario donde se pueden guardar valores de estilo llave/valor. Difiere de los dos anteriors en que, en el momento de terminar algún cálculo, la pila y la memoria se reinician, mientras que el *almacén* persiste como información de la cuenta.

El modo de ejecución del código consiste en un ciclo infinito que evalua la operación actual, la cual está indexada por el contador de programa de la máquina virtual (*program counter, pc*), e incrementa el contador de programa en uno, hasta que el código se encuentre en una condición de error o alcance alguno de los código de detención (`STOP` o `RETURN`). Las operaciones son accedidas y procesadas en la pila y esta estructura se encarga de comunicarse con la memoria y el almacén de cuenta. Todo lo anterior es realizado mientras haya gas que permita realizar la ejecución. Podemos observar esto en la figura 3.2.

De acuerdo con la documentación de Ethereum, la ejecución del código de la EVM se puede definir de forma sencilla. Mientras la EVM esté ejecutándose, un estado computacional se puede definir a partir de la siguiente tupla: (`block_state, transaction, message, code, memory, stack, pc, gas`). La descripción de cada uno de los elementos se puede revisar en la tabla 3.1. En el inicio de cada ronda de ejecución, la instrucción actual es elegida por el `pc`-ésimo byte del código (`code`) (o el byte cero si `pc` $\leq$ `len (code)`), y cada instrucción tiene su propia definición en términos de cómo afecta a la tupla.

[1]Los códigos de operación se pueden consultar en `https://github.com/crytic/evm-opcodes`

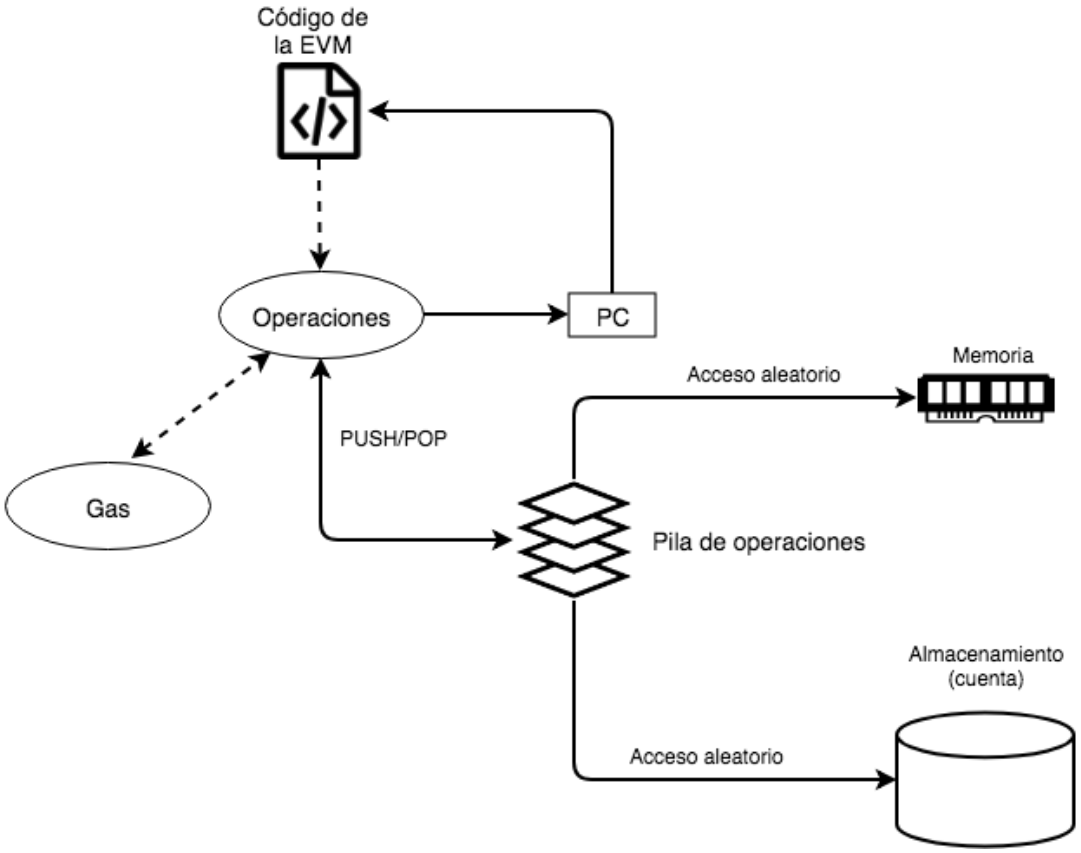

Figura 3.2: Modelo de ejecución de la máquina virtual de Ethereum.

3.3 Usos

Los contratos inteligentes son programas quasi-Turing completos, lo que permite que se puedan implementar reglas de negocio complejas e interesantes. Estas reglas de negocio permiten que se puedan crear acuerdos comerciales digitales fiables. Al estar construidas sobre la tecnología de la cadena de bloques, su despliegue, ejecución y validación siguen las mismas reglas que cualquier otra transacción en la cadena de bloques, permitiendo que sean fácilmente verificables y que sean publicamente consultadas.

Al usar contratos inteligentes, el objetivo de las transacciones realizadas en la

block_state	Es el estado global que contiene todas las cuentas, balances y almacenes.
transaction	Información relacionada con la transacción que se está evaluando.
message	Mensaje que envía como parte de la transacción.
code	El bytecode del contrato inteligente.
memory	Corresponde a la memoria de los espacios mencionados anteriormente.
stack	Corresponde a la pila de los espacios mencionados anteriormente.
pc	El contador de programa.
gas	La cantidad de gas disponible para realizar la transacción.

Tabla 3.1: Atributos de una tupla de ejecución en Ethereum

cadena de bloques se vuelve el no tener que depender de terceros, de tal modo que se permita el acceso a todas las partes de los acuerdos implementados. Esto garantiza que la lógica implementada en contratos inteligentes sea realizada en términos de ejecución automática, rápida, directa, barata y transparente.

Algunos de los casos de uso que tienen los contratos inteligentes se listan a continuación [11]:

- Identidad digital
- Intercambios financieros
- Cadenas de suministros (*supply-chains*)
- Hipotecas
- Escrituras
- Registros de impuestos
- Seguros
- Internet de las cosas
- Autoría y derechos de propiedad intelectual
- Ciencias de la salud
- Almacenamiento de datos financieros

3.3.1. Identidad digital

El uso de contratos inteligentes permite a los usuarios llevar el control de su identidad digital, la cual contiene datos, reputaciones y activos digitales. Esto permite que los usuarios decidan qué datos comparten a sus contrapartes. Las contrapartes no tendrán que tener datos confidenciales para verificar las transacciones. Esto reduce la responsabilidad de manejo de información sensible, al mismo tiempo que mejora los requisitos de interacción con los usuarios. Además de aumentar el cumplimiento, la flexibilidad e interoperabilidad del manejo de información personal.

Algunas características importantes de esto son las siguientes:

- **Identificación auto-soberana.** Esto permite que una persona sea capaz de actualizar la información de identificación, que se encuentre disponible en cualquier momento, además de que no sea accesada por cualquiera sin consentimiento. La mejor parte es que no hay terceros involucrados con la validación de la identidad.
- **Reducción del robo de identidades.** El uso de contratos inteligentes para la gestión de identidades permite reducir el robo de identidades. Además si se le agregan protocolos criptográficos es mucho mayor el beneficio.
- **Identificación digital para clientes bancarios.** El reconocimiento de clientes incluye el proceso de validación y verificación de documentos por parte de bancos. Este proceso suele ser largo y burocrático. El uso de cadenas de bloques permite a las instituciones financieras acceder a los datos de los clientes sin tener que estar iniciando un nuevo proceso cada vez que se busca reconocer a un cliente.
- **Gobiernos e identidad digital.** Los gobiernos pueden acceder a los datos de las personas, lo cuál abre la posibilidad de utilizar su identidad digital en elecciones, captura de impuestos y algunos otros servicios.

3.3.2. Intercambios financieros

Los contratos inteligentes pueden agilizar las transferencias internacionales de bienes a través de una rápido pago comercial, al tiempo que permiten una mayor liquidez de los activos financieros. También pueden mejorar la eficiencia de financiamiento para compradores, proveedores e instituciones.

La integración de los sistemas de intercambio financiero, los requisitos de tecnología y los ecosistemas fuera de la cadena son importantes para el éxito.

3.3.3. Cadenas de suministros

Los contratos inteligentes pueden proporcionar visibilidad en tiempo real para cada paso de los productos en una cadena de suministro. Los dispositivos pueden registrar cada paso a medida que un producto se traslada de una fábrica a los estantes de las tiendas, además de registrar el estado del producto en cada paso. La importancia de los contratos inteligentes en cadenas de suministro es facilitar el seguimiento del inventario a nivel granular, beneficiando el financiamiento de la cadena de suministro, los seguros y el riesgo. El seguimiento y verificación implementados reducen el riesgo de robo y fraude.

Además, las identidades de los actores en la cadena de suministro deben ser verificadas a lo largo del tiempo, incluyendo empresas, instituciones, individuos, sensores, instalaciones y productos.

3.3.4. Registros de impuestos

El uso de contratos inteligentes permite que se realicen pagos automáticos de impuestos, lo que ahorraría las multas. Como los datos se almacenan en la cadena de bloques, todos los que estén verificados pueden consultar los registros de impuestos, haciendo que la transparencia de dichos registros impida la comisión de fraudes.

3.3.5. Hipotecas

Los contratos inteligentes pueden automatizar los contratos hipotecarios al conectar automáticamente a las partes, lo que proporciona un proceso sin fricción y menos propenso a errores. El contrato inteligente puede procesar automáticamente el pago y liberar gravámenes de los registros de tierras cuando se paga el préstamo.

También permite mejorar la visibilidad de los registros para todas las partes, facilitando el seguimiento y la verificación de los pagos. Reducen los errores y los costes asociados a los procesos manuales. La identidad digital es un requisito clave.

3.4

Solidity

Solidity es un lenguaje orientado a objetos y de alto nivel para la implementación de contratos inteligentes. Su diseño está basado en el lenguaje JavaScript. Su diseño permite que su código sea compilado a bytecode, en particular en el lenguaje de la EVM. Soporta tipado estático, herencia, modularidad y tipos definidos por el usuario.

Una de las principales herramientas en Solidity son las funciones, las cuales son utilizadas de diferentes formas. Las funciones que producen valores se les conoce como *expresiones*. Mientras, que aquellas que indican una acción se le conocen como *enunciados*. Los programas escritos en Solidity pueden ser compuestos de enunciados y expresiones. En el momento de declarar una función, se define el alcance de la visibilidad que puede tener dicha función. Se definen dos tipos de invocaciones de programas (interno y externo) y cuatro tipos de visibilidad que se combinan con las invocaciones anteriores (pública, externa, privada e interna).

La invocación interna de funciones se refiere a una invocación de función dentro del mismo contrato, mientras que la invocación externa, hace referencia a la invocación de funciones desde otros contratos.

Los tipos de visibilidad de funciones se dividen en externa, pública, interna o privada, siendo la visibilidad pública la que se utiliza por defecto. Para variables de estado, la visibilidad externa no puede ser utilizada. Una descripción más detallada de las visibilidades la tenemos a continuación:

- **Externa:** Como parte de la interfaz del contrato, las funciones con la visibilidad *externa*, pueden ser invocadas por otros contratos y por transacciones. En el caso de las invocaciones internas, si **f** es una función con la visibilidad *externa*, esta no podrá ser invocada usando **f ()**, si no **this.f ()**.
- **Pública:** Las funciones que son públicas, que forman parte de la interfaz del contrato, pueden ser invocadas por mensajes y de forma interna. En el caso de las variables de estado públicas, van a tener un *getter* automático para acceder a ellas.
- **Interna:** Las funciones internas y variables de estado sólo están disponible dentro del contrato que las define y sus derivados.
- **Privada:** Las funciones privadas y variables de estado son sólo visibles para el

contrato que las define. No pueden ser accedidas por contratos derivados.

Al ser un lenguaje de tipado estático, todo valor dentro del lenguaje tiene un tipo definido, el cual es especificado en tiempo de compilación. Los tipos pueden interactuar con otras expresiones que contienen operadores. Entre los tipos que define Solidity tenemos:

- Booleans
- Enteros
- Números de punto fijo
- Direcciones
- Miembros de direcciones
  - Balance y transferencia
  - Send
  - Call, callcode y delegatecall
- Arreglos de bytes de tamaño fijo
- Literales de direcciones
- Cadenas
- Hexadecimales
- Enums

Además de los tipos, tenemos funciones globales especiales que son utilizadas principalmente para proveer información sobre la cadena de bloques, así como proveer algunas operaciones matemáticas y criptográficas. Por ejemplo, `keccak256 (...)` que calcula el hash de un elemento en SHA-3, `sha256 (...)` para generar hash de la función SHA-256, etc. Además de funciones que hacen referencia a los contratos en si mismos, tales como **selfdestruct (address)**[2] para autodestruir un contrato, **assert (condition)** la cual lanza una excepción si la condición no se cumple, **revert ()** para abortar y revertir una transacción y su estado.

[2]La autodestrucción o suicidio de un contrato se refiere a la opción de vaciar los fondos del contrato envíandolo a la dirección dada y limpiando el estado de un contrato haciéndolo "inutilizable" para futuras invocaciones.

3.5

Estructura de un Contrato Inteligente

Para mostrar la estructura de un contrato inteligente, mostremos el código de uno que fue escrito en Solidity. Para ilustrarlo, mostramos la implementación de un contrato sencillo para proveer una nueva moneda. Este contrato nos servirá para mostrar propiedades del lenguaje. Podemos ver la implementación del contrato en el código 3.1.

```
1  pragma solidity >=0.4.22 <0.7.0
2
3  contract Coin {
4      // The keyword "public" makes those variables
5      // readable from outside.
6      address public minter;
7      mapping (address => uint) public balances;
8
9      // Events allow light clients to react on
10     // changes efficiently.
11     event Sent(address from, address to, uint amount);
12
13     // This is the constructor whose code is
14     // run only when the contract is created.
15     function Coin() public {
16         minter = msg.sender;
17     }
18
19     function mint(address receiver, uint amount) public {
20         if (msg.sender != minter) return;
21         balances[receiver] += amount;
22     }
23
24     function send(address receiver, uint amount) public {
25         if (balances[msg.sender] < amount) return;
26         balances[msg.sender] -= amount;
27         balances[receiver] += amount;
28         emit Sent(msg.sender, receiver, amount);
29     }
30 }
```

Código 3.1: Contrato para votaciones

Este contrato implementa una forma sencilla de sub-moneda dentro de Ethereum. Es posible generar monedas de la nada (`línea 19`), pero sólo la persona que creo

el contrato puede hacerlo (se puede implementar otras formas de generar monedas). Además cualquiera puede enviar monedas y recibirlas sin la necesidad de asociar un nombre o algún sistema de identificación, sólo se necesita tener una cuenta en Ethereum (`línea 24`).

Además del significado semántico del contrato, la estructura de los contratos inteligentes está dada por los siguiente elementos:

- Directiva *pragma*.
- Directivas de importación.
- Declaración del contrato.
- Declaración de tipos.
- Declaración de variables de estado.
- Declaración de funciones.

3.5.1. Directiva pragma

La palabra reservada *pragma* es utilizada para establecer ciertas características del compilador respecto a una versión o rango de versiones. Las directivas pragma siempre son locales respecto al archivo de un contrato, así que se tiene que añadir a todos los archivos si se quiere aplicar a todo el proyecto. Los archivos de código fuente pueden ser anotados con una versión pragma para evitar rechazos al compilar con versiones futuras del compilador que podrían introducir cambios incompatibles. Por ejemplo, la directiva pragma de la `línea 1` del código 3.1 está definido de la siguiente forma:

```
1 pragma solidity >=0.4.22 <0.7.0;
```

Código 3.2: Directiva pragma

La directiva pragma mostrada en el código 3.2 muestra que la versión del compilador que se debe utilizar para compilar el contrato debe ser a partir de la versión 0.4.22 y menor a la 0.7.0. El especificar las versiones se debe a que el compilador se encuentra cambiando constantemente, ya sea corrigiendo bugs o agregando nuevas características.

3.5.2. Directiva de importación

Solidity soporta enunciados para importar código desde otros contratos, de modo que se pueda construir contratos modularizados. De manera global, uno puede importar código como se muestra en el código 3.3.

```
1 import "filename";
```

Código 3.3: Directiva de importación

El enunciado anterior importa todos los símbolos globales (variables de estado, funciones) que estén definidos en "filename" en el contexto actual del contrato que lo invoca. Sin embargo esta forma de importar código no es recomendada, ya que puede ensuciar el espacio de nombres del contrato. Por ejemplo, si se añaden nuevos símbolos globales a "filename", estos se encuentran disponibles en el contrato. La mejor opción es importar símbolos específicos de forma explícita. Por ejemplo, en el código 3.4, se muestran formas alternativas de importar código.

```
1 import * as symbolName from "filename"; // Los simbolos estaran
     disponibles con symbolName.symbol
2 import "filename" as symbolName; // Alternativa a la linea anterior
3 import {symbol1 as alias, symbol2} from "filename"; // Importa de forma
      especifica los simbolos symbol1 y symbol2
```

Código 3.4: Directivas de importación alternativas

3.5.3. Declaración de contrato

La declaración de un contrato permite asignarle un nombre al contrato. Dicho contrato puede contener declaraciones de variables de estado, funciones, modificadores, eventos, estructuras y tipos enumerados. Un ejemplo de la declaración de un contrato es la `línea 3` del código 3.1.

Otra característica de los contratos es que pueden implementar herencia de forma similar a como se realiza en la programación orientada a objetos. Se pueden heredar funciones y variables de estado. Un ejemplo sencillo es el mostrado en el código 3.5. Para indicar la derivación desde otro contrato, se usa la palabra *is*. Los contratos derivados pueden acceder a todos los miembros no privados, incluyendo funciones declaradas como internal y variables de estado.

```
contract owned {
    address payable owner;
    constructor() public { owner = msg.sender; }
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

Código 3.5: Herencia en contratos

3.5.4. Declaración de tipos

Solidity es un lenguaje tipado estáticamente, lo que significa que cada variable tiene un tipo que necesita ser declarado explícitamente y dicho tipo no va a cambiar. Además de los tipos que define el lenguaje, podemos definir nuestros propios tipos a partir de estructuras (*structs*), por ejemplo, en el contrato para votar (código 3.6), en las `lineas 1 y 9`, se definen dos tipos a través de *struct*, un votante (*Voter*) y una propuesta (*Proposal*) para un contrato utilizado para realizar votaciones.

```
struct Voter {
      uint weight; // weight is accumulated by delegation
      bool voted;  // if true, that person already voted
      address delegate; // person delegated to
      uint vote;   // index of the voted proposal
  }

  // This is a type for a single proposal.
  struct Proposal {
      bytes32 name;   // short name (up to 32 bytes)
      uint voteCount; // number of accumulated votes
  }
```

Código 3.6: Tipos definidos por usuario

3.5.5. Variables de estado y funciones

Las variables de estado, son aquellas variables cuyos valores son almacenados permanentemente en el contrato. Por ejemplo, en el caso del contrato de votaciones

(código 3.1), en la `línea 7`, se define una variable que va a existir mientras el contrato sea operable.

En el caso de las funciones, estas se pueden entender como unidades de código dentro de un contrato. Las invocaciones a funciones pueden suceder de forma interna, como invocaciones de otras funciones o externamente, donde una transacción puede invocar a una función.

Yo no soy un gran programador. Sólo soy un buen programador con grandes hábitos.

Martín Fowler

CAPÍTULO 4

Vulnerabilidades en Contratos Inteligentes y Patrones de Diseño

Iniciamos en la sección 4.1 describiendo una clasificación propuesta en [3] de vulnerabilidades que existen dentro de la cadena de bloques de Ethereum. En la sección 4.2 describiremos qué son los patrones de diseño y para que son utilizados; en la sección 4.3 se describirá la relación que se da cuando se implementan contratos inteligentes y el uso de patrones de diseño. Esto último se debe a que los patrones ayudan a resolver problemáticas y vulnerabilidades relacionadas con la ejecución y el diseño de programas, de modo que también se puedan aplicar a los contratos inteligentes. Algunas de las vulnerabilidades explicadas en la sección 4.1, pueden ser explotadas en otras cadenas de bloques distintas, por lo que terminaremos el capítulo con la sección 4.4, donde se describirán algunos patrones de diseño orientados a aspectos de comportamiento en contratos inteligentes, seguridad, actualización de contratos y de optimización en el uso de gas.

4.1 Vulnerabilidades en Contratos Inteligentes

Desde el despliegue de Ethereum y los contratos inteligentes, se han identificado varias vulnerabilidades que han resultado en pérdidas económicas dentro del ecosistema de Ethereum. Uno de los ejemplos más conocidos, es el ataque a la Organización Autónoma Descentralizada (DAO) [37], en el que se robaron cerca de 150 millones

de dolares en ese momento. Este evento causó que los desarrolladores de contratos tengan una mayor conciencia sobre vulnerabilidades existentes. En [3] propone una clasificación de vulnerabilidades de acuerdo al nivel donde se genera la vulnerabilidad. La clasificación se divide en tres niveles y en cada nivel hay varias vulnerabilidades ya identificadas. Estos niveles son:

- Vulnerabilidades a nivel de lenguaje.
- Vulnerabilidades a nivel de máquina virtual.
- Vulnerabilidades a nivel de cadena de bloques.

A continuación daremos una descripción de cada nivel y mostraremos algunas de la vulnerabilidades conocidas en cada nivel.

4.1.1. Vulnerabilidad a Nivel de Lenguaje

En este nivel, las vulnerabilidades están asociadas a ciertas debilidades en el diseño de primitivas del lenguaje o en el comportamiento que pueden tener en el lenguaje. El lenguaje que vamos a utilizar es Solidity. Podemos describir algunas problemáticas asociadas con el lenguaje. Listamos algunas de las problemáticas existentes en este nivel y posteriormente describiremos cuál es la problemática que atacan.

- *Call to the unknown*
- *Exception disorder*
- *Gasless send*
- *Reentrancy*
- *Keeping secrets*

Call to the unknown

Esta vulnerabilidad se aprovecha en la ejecución de primitivas que permiten transferir ethers y además, estas primitivas pueden invocar a una función *fallback* (*fallback function*[1]), que pueden tener efectos laterales al ejecutarse en el invocador/recipiente. Esta vulnerabilidad puede ser lanzada por una de las siguientes tres funciones: `call`, `send` o `delegatecall`.

[1]Las funciones *fallback* o funciones por defecto, son funciones especiales dentro de los contratos inteligentes que no tienen un nombre y que no reciben argumentos, las cuales pueden ser implementadas arbitrariamente. La función por defecto es ejecutada cuando se le envía una firma vacía, por ejemplo, cuando se le envía ether al contrato.

- `call` es una función que invoca a otra función (o a un contrato o a si misma), transfiriendo ethers a la función invocada. Por ejemplo, la invocación a la función `ping` de un contrato `c` (como se muestra en el código 4.1), la cuál es identificada por los primeros 4 bytes de su hash (obtenida al aplicar la función `sha3`), transfiere la cantidad `amount` (en wei) a `c`, con `n` el parámetro de la función `ping`. Una característica de esta función es que si la función invocada no existe, se invoca la función *fallback*.

```
1  c.call.value(amount)(bytes4(sha3(''ping(uint256)'')), n)
```

Código 4.1: Invocando la función Ping con `call`

- La función `send` es utilizada para transferir ether desde un contrato en ejecución a algún receptor r, utilizando la sintaxis `r.send (amount)`. Después de que el ether ha sido enviado, `send` ejecuta la función fallback del receptor.
- La función `delegatecall`, permite que las funciones invocadas sean ejecutadas en el entorno del contrato invocador. Por ejemplo, si se ejecuta el código 4.2, la función `ping`, internamente utiliza la variable `this` dentro del cuerpo de la función, permitiendo que la variable haga referencia a la dirección del invocador y no a la del contrato, y en el caso de que se realice la transferencia de ethers a algún receptor `d` (por ejemplo, vía `d.send (amount)`), el ether es tomado del balance del invocador.

```
1  c.delegatecall.value(amount)(bytes4(sha3(``ping(uint256)'')), n)
```

Código 4.2: Invocando la función Ping con delegatecall

Exception disorder

En el lenguaje de Solidity, hay varias situaciones que pueden causar una excepción. Algunos ejemplos en que se lanza una excepción son:

- La ejecución se queda sin gas.
- La pila de llamadas alcanza su límite.
- El comando `throw` es ejecutado.

Sin embargo, Solidity no lanza las excepciones de manera uniforme, hay dos diferentes comportamientos basados en la forma en que se llaman los contratos entre

sí. Si la invocación es través de una cadena de llamadas, cuando se lanza la excepción, se revierten todos los efectos laterales y el resultado de la invocación. Si se realiza una invocación a través del uso de `call`. Entonces, sólo se revierten los efectos laterales, pero el resultado de la invocación se mantiene.

Gasless send

Cuando se utiliza la función `send` para transferir ether a un contrato, es posible incurrir en una excepción *out-of-gas* (falta de gas). Para los programadores el envío de ether no está asociado a la ejecución de código, causando que la excepción sea inesperada para ellos. La razón de esto es sutil, ya que la función `send` al ejecutarse invoca a la función *fallback* del contrato, además de que el envío está acotado a 2300 unidades de gas. Entonces, cuando se envía ether con `send` y no se agrega el suficiente gas para ejecutar el fallback, se lanza la excepción.

Reentrancy

En computación, a una función o rutina es llamada reentrante si puede ser ejecutada por varios hilos o invocaciones en un mismo proceso de forma segura. En este contexto, "seguro" significa que la función alcanza su resultado esperado, independientemente del estado de la ejecución de cualquier otro proceso [28]. Las transacciones en Ethereum, son ejecutadas de forma atómica y secuencial. Esto puede inducir en los programadores la creencia de que cuando se invoca una función no recursiva, está no puede ser vuelta a invocar después de su terminación. Sin embargo, el mecanismo de *fallback* permite realizar invocaciones al término de una función y dependiendo de su implementación, se puede volver a invocar a la función que había finalizado su ejecución. Esto puede permitir que se ejecuten funciones vulnerables y se ejecuten ciclos hasta que se acabe el gas.

Un ejemplo importante de esta vulnerabilidad, es el Ataque a la "Organización Autónoma Descentralizada" (DAO[2], por sus siglas en inglés), el cual aprovecho esta vulnerabilidad en un contrato desplegado en la cadena de bloques de Ethereum. En su momento causó que un total de 3.6 millones de ethers fueran robados. Para ilustrar este ataque consideremos lo siguiente:

- Se despliega un contrato inteligente que permite a los usuarios gestionarse en una organización que puede recolectar fondos y ejecutar acciones decididas a través de los votos de los usuarios.

[2]Descentralized Autonomous Organization

- Las acciones que puede realizar la organización son contratos inteligentes propuestos por la comunidad.
- Hay un periodo inicial de recolección de fondos, en el que los usuarios agregan fondos a la organización y reciben a cambio *tokens* criptográficos que representan una acción dentro de la organización. Los fondos recaudados sirven para permitir la ejecución de los contratos que sean elegidos para ejecutarse por la organización.
- Cuando se termina el periodo de recolección de fondos, la Organización está lista para operar y llama a votación.
- La votación es realizada por los miembros de la Organización y deciden los contratos a ejecutar.

El contrato que llevaba el control del *DAO*, tenía una función que permitía a sus integrantes recuperar su dinero invertido. Consideremos el código discutido en [3], el cual es una versión simplificada del contrato del *DAO*, este código lo mostramos en el código 4.3. Este contrato tiene tres funciones:

1. La función `invest (to)`, la cuál permite a un usuario invertir sus activos en el DAO.
2. La función `queryCredit (to)`, que devuelve el valor invertido por un usuario.
3. La función `withdraw (amount)`, que permite devolver parte o la totalidad de la inversión de un usuario.

Podemos realizar un ataque que se robe todo el ether del contrato *SimpleDAO*, de la siguiente forma: despleguemos en la cadena de bloques el contrato *Mallory* descrito en el código 4.4.

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function invest(address to){
4      credit[to] += msg.value;
5    }
6    function queryCredit(address to) returns (uint) {
7      return credit[to];
8    }
9    function withdraw(uint amount) {
10     if (credit[msg.sender] >= amount) {
11       msg.sender.call.value(amount)();
```

```
12          credit[msg.sender] -= amout;
13        }
14      }
15    }
```

Código 4.3: DAO Simplificado

```
1   contract Mallory {
2     SimpleDAO public dao = SimpleDAO(0x123);
3     address owner;
4     function Mallory() {
5       owner = msg.sender;
6     }
7     function () {
8       dao.withdraw(dao.queryCredit(this));
9     }
10    function getJackpot() {
11      owner.send(this.balance);
12    }
13  }
```

Código 4.4: Contrato Mallory

```
1   m.call.value(1)(bytes4(sha3(``dao.invest(address)'')), m.owner)
```

Código 4.5: Donación de ether para lanzar el ataque a SimpleDAO

El contrato *Mallory* se compone de lo siguiente:

- Una instancia del contrato *SimpleDAO* para explotar la vulnerabilidad.
- Constructor del contrato. Asigna a la variable *owner* quien es el dueño del contrato.
- Función *fallback*, que invoca a la función $SimpleDAO.withdraw$, pasando como parámetro una invocación a $SimpleDAO.queryCredit(this)$, esta función devuelve el total enviado por el contrato a *SimpleDAO*.
- La función *getJackpot* que envía todo el balance del contrato al dueño.

Para que la vulnerabilidad pueda ser explotada, el atacante, a través de una instancia del contrato *Mallory*, debe realizar una transacción en la cadena de bloques donde está desplegado el contrato de *SimpleDAO*, enviando ether al contrato. Una forma de hacerlo es como se muestra en el código 4.5. Al finalizar su ejecución, esta transacción invoca a la función *fallback* de *Mallory*, lo cuál provoca que se invoque a la función *withdraw* de *SimpleDAO*. Esto provoca un ciclo que hace lo siguiente:

- La invocación a *withdraw* recibe como parámetro la cantidad de ether invertido.
- La condicional de la línea 10 del código 4.5 siempre se cumple, ya que el crédito del que dispone es igual a la cantidad que está pasando como parámetro a la función.
- Se ejecuta la instrucción de la línea 11 del código 4.5. Como es una función *call*, cuando termina de ejecutarse, esta invoca a la función *fallback* del invocador, que en este caso es el contrato *Mallory*.
- Regresa el control a la instancia m del contrato y entra en la función *fallback* y volviendo a ejecutar *withdraw*.

Los pasos anteriores se van a repetir tantas veces lo permita alguna de las siguientes condiciones:

1. Se acabe el gas.
2. La pila de llamadas se agote.
3. La cantidad de ethers del contrato DAO se vuelva cero.

Keeping secrets

Los campos en los contratos pueden ser públicos o privados, es decir, directamente legibles por cualquiera cuando son públicos o no directamente legibles por otros usuarios ó contratos cuando son privados. Sin embargo esto no garantiza su privacidad, ya que para establecer los valores en dichos campos, es necesario que se envíe una transacción a los mineros y estos, durante la ejecución, establezcan dichos valores. Pero debido a la naturaleza de las cadenas de bloques, cualquiera puede inspeccionar las transacciones realizadas e inferir el contenido de los campos dentro de un contrato. Esto afecta directamente a contratos que implementan juegos multijugador donde es necesario mantener secreto el estado de los jugadores respecto a los otros adversarios.

Veamos un ejemplo de como se podría explotar esta problemática. Consideremos un juego de 2 jugadores definido de la siguiente forma: cada jugador elige un número, si la suma es par, el jugador 1 gana, en otro caso, el jugador 2 gana. Veamos el código 4.6 que implementa este juego en Ethereum.

```
1  contract OddAndEvens {
2    struct Player {
3      address addr;
4      uint number;
5    }
```

```
6     Player[2] private players;
7     uint tot = 0;
8     address owner;
9     function OddAndEvens() {
10      owner = msg.sender;
11    }
12    function play(uint number) {
13      if (msg.value != 1 ether) throw;
14      players[tot] = Player(msg.sender, number);
15      tot++;
16      if (tot == 2) andTheWinnerIs();
17    }
18    function andTheWinnerIs() private {
19      uint n = players[0].number + players[1].number;
20      players[n\%2].addr.send(1800 finney);
21      delete players;
22      tot = 0;
23    }
24    function getProfit() {
25      owner.send(this.balance);
26    }
27  }
```

Código 4.6: Contrato OddsAndEvens

En el contrato mostrado en el código 4.6, se almacena la información de los dos jugadores en el arreglo `players`. Como este como es `private`, otros contratos no pueden leer directamente de el. Para unirse al juego, cada jugador debe enviar 1 ether cuando invocan la función *play*. Si la cantidad transferida es diferente, se lanza una excepción. Una vez que el segundo jugador se ha unido al juego, se ejecuta la función *andTheWinnerIs*, transfiriendo 1.8 ethers al ganador. El resto, se queda como ganancia para el dueño del contrato y estas ganancias son recolectadas a través de la función `getProfit`.

Entonces, ¿cómo se puede explotar este contrato? Un adversario puede lanzar un ataque en el que siempre puede ganar. Para hacer esto, el atacante debe ser el segundo jugador del juego y esperar a que el primer jugador invoque `play`. Cuando el primer jugador se una, el atacante podrá inferir la información del primer jugador. Aunque la información del primer jugador dentro del contrato es privada y puede ser observada por el atacante, si se puede inferir a partir de observar las transacciones realizadas en la cadena de bloques e inspeccionar la transacción donde el primer jugador invocó a `play`. Entonces el atacante puede invocar a `play` con un valor que le convenga. De este modo, explotamos la vulnerabilidad descrita aquí.

4.1.2. Vulnerabilidad a Nivel de Máquina Virtual

Las vulnerabilidades explotadas a nivel de máquina virtual, buscan aprovecharse de debilidades que se pueden dar en tiempo de ejecución. Varias de ellas tienen que ver con instrucciones erróneas que se encuentran en contratos que ya han sido desplegados en la cadena de bloques. Esto se debe a que los contratos una vez desplegados en la cadena de bloques, ya no pueden ser modificados, lo cuál implica que no hay un forma directa de arreglar o corregir un contrato en caso de un error o equívoco. A continuación listaremos un par de problemáticas que suelen suceder a este nivel de abstracción y posteriormente las describiremos con mayor detalle.

- *Immutable bugs.*
- *Ether lost in transfer.*

Immutable bugs

Al desplegarse un contrato en la cadena de bloques, este no puede ser modificado. Los usuarios pueden confiar en que las implementaciones de los contratos inteligentes funcionan como se espera, pero si el contrato contiene un error, no hay una forma directa de corregirlo. Los desarrolladores de contratos inteligentes, tienen que anticipar formas de terminar un contrato en caso de fallo, porque de no ser así, la inmutabilidad de los errores puede ser utilizada para robar ethers o para hacerlos irrecuperables. En todos los casos, la única forma de recuperarse es, invalidando, mediante "*hard-fork*", el sufijo de la cadena de bloques que comienza a partir del ataque. Un ejemplo de esto, es el problema que se generó con el ataque al *DAO*. La solución a este ataque fue realizar un hard-fork de Ethereum, donde no todos los participantes estuvieron de acuerdo con esto, dividiéndose en dos grupo, los que aceptaron el *hard-fork* (Ethereum) y los que no (Ethereum-clasic) [8].

Ether lost in transfer

Al momento de enviar ethers, hay que especificar la dirección del receptor, y estas direcciones son secuencias de 160 bits. Debido a la longitud de las cadenas que representan las direcciones, es común que hayan fallos al capturar una dirección y debido a eso, el ether transferido se pierda. Debido a que el ether perdido no puede ser recuperado, es necesario que los programadores se aseguren manualmente de que las direcciones son correctas.

Consensys, una empresa dedicada a implementar soluciones basadas en Ethereum, realizó un análisis de la cantidad de ether perdido a causa de errores tipográficos, descubriendo que al menos 12,622 ethers se perdieron, esto se traduce en una perdida de de 8.84 millones de dólares por el precio que tenía cada ether en ese momento (1 eth $\approx$ \$700) [34].

4.1.3. Vulnerabilidad a Nivel de Cadena de Bloques

En este nivel, las vulnerabilidades explotadas tienen que ver con el estado de la cadena de bloques. Algunas de estas vulnerabilidades se aprovechan del conocimiento (o desconocimiento) del estado en que se encuentra los contratos con sus balances, o la imposibilidad de generar aleatoriedad dentro de la cadena de bloques para realizar cómputo. A continuación listaremos algunas vulnerabilidades y posteriormente daremos una descripción de las mismas.

- *Unpredictable state.*
- *Generating randomness.*

Unpredictable state

Cuando hablamos del estado global δ_i en la cadena de bloques, nos referimos al estado en que se encuentran los contratos desplegados en la cadena de bloques, cada uno de ellos definidos en términos de sus campos y su balance. Cuando se ejecuta una transacción, esta modifica el estado global y genera un nuevo estado δ_{i+1}. Sin embargo, cuando se ejecuta una transacción T, no se puede estar seguro si la transacción va a ejecutarse en el mismo estado que cuando fue enviada para ser ejecutada, esto es debido a que puede haber ocurrido que se ejecutaran transacciones entre el momento en que se envío la transacción original T y el momento en que el minero la ejecuta. Incluso si la transacción es ejecutada entre dos estados globales δ_i y δ_j consecutivos, no hay garantía de que el estado sea el esperado. Una de las razones por la que pasa esto, es debido a que los mineros deciden el orden en que ejecutan las transacciones dentro de su bloque propuesto.

Consideremos una transacción T que supone que el estado de la cadena de bloques es δ_i. Esta transacción se ejecuta en un estado $\delta_j \neq \delta_i$, esto se debe a que al menos una transacción T_j dentro del mismo bloque fue ejecutada antes de T, provocando el cambio de estado δ_i a δ_j, causando que T tenga una salida distinta a la esperada. A esta problemática se le conoce como *Transaction Ordering Dependence* [14], discuti-

remos con mayor detalle esta problemática en el capítulo 5, donde propondremos un par de soluciones para mitigarla.

Generating randomness

La ejecución de los contratos inteligentes siempre es determinista, es decir, todos los mineros que ejecutan una transacción van a tener siempre los mismo resultados. Entonces, para simular elecciones no deterministas, varios contratos (como juegos o loterías), generan números pseudoaleatorios, donde la elección de la semilla es elegida de manera única por todos los mineros. Esto representa un problema, ya que se podrían desarrollar contratos con valores precalculados y engañar a los contratos que hacen uso de la pseudo-aleatoriedad. Algunas formas de evitar esto es tomar como semilla el hash del timestamp de algún bloque que va a aparecer en el futuro. Como todos los mineros tienen la misma vista de la cadena de bloques, este valor elegido va a ser el mismo para todos.

4.2 ¿Qué son los Patrones de Diseño?

El diseño de software es una tarea difícil, pero el crear componentes reusables es incluso más difícil [21]. Dependiendo del paradigma del lenguaje con el que se está implementando un producto de software, puede ser que se busque crear componentes que permitan resolver problemas recurrentes. Ya sea en el caso de los lenguajes orientados a objetos, donde se busca crear métodos, clases, interfaces y jerarquías de herencia con una correcta granularidad, o en el caso de los lenguajes funcionales donde se busca la existencia de funciones de primera clase, que sean modulares, que satisfagan la transparencia referencial, funciones que hagan uso de estructuras de datos inmutables, etc. Una característica importante del diseño de estos componentes es que deben satisfacer una alta cohesión y un bajo acoplamiento.

Una manera de implementar software que satisfaga las características anteriores, es haciendo uso de la experiencia de los desarrolladores. Esto es posible al reutilizar soluciones que funcionaron en el pasado para problemas similares a los que se están resolviendo actualmente. Si la solución fue buena, esta se reutiliza una y otra vez, lo que da origen a un patrón. Estos patrones describen un problema que ocurre múltiples veces en un entorno y también describe una solución al mismo problema, permitiendo que pueda ser utilizada más adelante sin tener que intentar resolver el problema nuevamente [2]. Los patrones de diseño son técnicas comúnmente usadas

para definir guías de diseño, buenas prácticas y para resolver problemas recurrentes.

Los patrones se describen en términos de un esquema dividido en tres partes [38]:

- **Contexto:** Una situación que describe cómo es que se genera el problema. Especificar el contexto correcto para el patrón es difícil, ya que no es posible determinar todas las situaciones en las que un patrón puede ser aplicado, ya sea de forma general o específica.
- **Problema:** El problema recurrente del contexto anterior. Inicia con una especificación del problema capturando la esencia del problema concreto que debemos resolver. Para construir una solución, hay aspectos que deben ser considerados tomando en cuenta la naturaleza del problema. Algunos ejemplos de esto son:
  - Requerimientos que la solución debe satisfacer.
  - Restricciones que deben ser consideradas.
  - Propiedades que la solución debe tener.
- **Solución:** Una solución probada del problema. La solución del patrón debe mostrar cómo se debe resolver el problema, especificando la estructura de la solución, la configuración, arquitectura de la solución implementada, etc. Además de la descripción del comportamiento que debe tener la solución, para poder considerar los aspectos dinámicos de la solución.

Una vez que se descubre que se necesita un patrón, se tiene que observar el cómo aplicarlo a cada una de las circunstancias. Una parte importante de los patrones, es que no se aplican ciegamente, ya que en muchas ocasiones se tienen que adaptar en términos de los proyectos en los que se aplican. Cada vez que se aplica un patrón, se modifica un poco en cada proyecto. Esto permite que observemos una misma solución aplicada varias veces sobre el tiempo, pero nunca exactamente la misma.

4.3 Patrones de Diseño en Contratos Inteligentes

Los contratos inteligentes son programas de software ejecutados sobre una cadena de bloques. Estos contratos tratan con activos y reglas de negocio, lo que hace importante diseñar buenos contratos que no tengan problemas en tiempo de ejecución. Algunos problemas comunes en los contratos son: ejecuciones incorrectas, problemas de tipos o incluso brechas de seguridad, ya que una falla en estos programas puede

desembocar en altas pérdidas económicas. Como se mencionó en la sección 4.2, el diseño de software es una tarea difícil. En el caso de los contratos inteligentes, se pueden implementar patrones de diseño en ellos, de modo que se mitiguen diversas problemáticas.

En el caso de Ethereum, el lenguaje de alto nivel para implementar contratos inteligentes es *Solidity*. Este es el lenguaje por defecto, debido a su popularidad y su similitud con el lenguaje *JavaScript*. En la documentación del lenguaje lo describen como un lenguaje orientado a contratos [45], pero debido que está fuertemente inspirado en *JavaScript*, encontramos características de programación orientada a objetos. Podemos encontrar mucha similitud al momento de definir clases, objetos, métodos, funciones, componentes y otros elementos del lenguaje.

Esto permite que podamos aplicar a los contratos inteligentes muchos de los patrones de diseño existentes para lenguajes orientados a objetos. Muchas soluciones a diversas problemáticas definidas en términos de lenguajes orientadas a objetos se han propuesto en [21], así como problemáticas de cómputo distribuido y concurrente en [6] y en términos de patrones arquitectónicos [38, 20].

El trabajo de esta tesis lo dirigimos a la construcción de patrones de diseño aplicados a contratos inteligentes, de modo que nos ayuden a mitigar la vulnerabilidad de *Transaction Ordering Dependence*, la cual describimos en alto nivel en la sección 4.1.

4.4 Clasificación

En la comunidad de Ethereum, se han desarrollado propuestas de patrones de diseño. En [48] se han recopilado varios patrones y se han dividido en la siguiente clasificación:

- Patrones de comportamiento.
- Patrones de seguridad.
- Patrones de actualización.
- Patrones económicos.

en patrones de comportamiento,

En esa compilación se ha incluido la explicación de como funciona cada uno de los patrones e implementaciones de ejemplo para los mismos. Cada patrón incluye antecedentes, implicaciones e información adicional. A continuación, haremos una descripción sobre las clasificaciones de los patrones.

4.4.1. Patrones de Comportamiento

Los patrones que están en esta categoría, son patrones que hacen referencia al comportamiento esperado de un contrato inteligente durante su ejecución. A continuación describiremos cada uno de los patrones mostrados en [48].

Guard check En muchas ocasiones los contratos tienen comportamiento no esperados o las entradas no operan como se espera. Se busca que durante la ejecución de los contratos, la lógica implementada sea ejecutada sólo cuando se cumplan las condiciones especificadas. En caso de que no sea así, se revierten todas las acciones realizadas por el contrato inteligente.

State Machine Un comportamiento esperado de un contrato inteligente es que este funcione de manera similar a una máquina de estados, donde el contrato inteligente tiene un estado inicial y a través de invocaciones realizadas por transacciones, el contrato sigue un conjunto de estados hasta llegar a un estado final. En cada uno de estos estados, el contrato puede ser accedido en diversas formas y proveer diferentes funcionalidades a los usuarios.

Oracle (oráculo) Todo cómputo realizado en la cadena de bloques tiene que ser validado por cada nodo de la red de la cadena de bloques. Esto implica que toda la información se encuentra dentro la misma cadena de bloques para ser consultada. Sin embargo, en ocasiones es necesario obtener información del mundo exterior, ya sea para consultar el resultado de un evento deportivo u obtener información del clima para la que un contrato inteligente pueda tomar una solución.

Randomness Este patrón ayuda con la problemática de generar aleatoriedad en Ethereum, ya que es difícil o incluso imposible generar un verdadero número aleatorio a través de software, se busca a través de ciertas técnicas proveer esta aleatoriedad. Algunas técnicas usadas son:

- Generación de números pseudoaleatorios usando el hash de un bloque.
- Generación de números aleatorios usando oráculos.
- Generación de números pseudoaleatorios a través de colaboración de nodos dentro de la cadena de bloques.

4.4.2. Patrones de Seguridad

Debido a la naturaleza económica asociada a la ejecución de contratos inteligentes, se busca desarrollar patrones que ayuden a mitigar problemas de seguridad, ya sea en el acceso a valores dentro de los contratos, reducción en el ataque de agentes maliciosos, envío de ethers de forma de segura o deshabilitar un contrato en caso de una emergencia crítica.

Access Restriction En las cadenas de bloques no es posible garantizar una completa privacidad para los contratos. No se puede prevenir que cualquiera lea el estado del contrato o se hagan inferencias a partir de las transacciones realizadas en la cadena de bloques. Lo que se puede hacer es realizar una restricción de acceso al contrato desde otros contratos. Esta restricción de acceso debe ser realizada bajo criterios adecuados. Ya sea declarando atributos de los contratos como privados o que las funciones del contrato sean accedidas bajo ciertas circunstancias, o también que esas mismas restricciones sean aplicadas a varias funciones, de modo que aumente la seguridad contra el acceso no autorizado.

Checks Effects Interactions Una opción para mitigar el problema de la reentrancia es el identificar las instrucciones que pueden permitir que un atacante se quede con el flujo del programa. Una vez realizada la identificación, se modifica el contrato para que las variables del contrato se actualicen antes de que las instrucciones del atacante se vuelvan a ejecutar.

Secure Ether Transfer El envío de Ether entre usuarios o entre contratos es una operación poco común en el ecosistema de Ethereum, pero es necesaria y es una característica importante. Una forma de hacerlo es verificando primero si el destinatario con el que queremos enviar transferencias y si lo es, continuamos con la operación de envío.

Emergency Stop Ayuda a deshabilitar un contrato en caso de alguna emergencia. En muchas ocasiones, incluso el software que ha sido auditado y probado puede contener bugs o código que no tiene el funcionamiento esperado. Muchas veces, los errores no son detectados hasta que están en ejecución o son aprovechados por un atacante. Cuando una falla ha sido detectada, es difícil arreglarla, más aún cuando el código es inmutable. Aunque hay varios patrones de actualización de código, la aplicación de esos patrones toma tiempo y los atacantes puede aprovecharse de ellos. Con este patrón, se puede añadir la posibilidad de pausar un contrato, prevenir el abuso de bugs no descubiertos o preparar el código para posibles fallas.

4.4.3. Patrones de Actualización

Los patrones de actualización nacen como una respuesta a la búsqueda de implementaciones de contratos para los que es necesario actualizarlos en un futuro, ya sea para agregar nuevas funcionalidades o modificar las existentes. Los patrones actualización permite introducir cierta "mutabilidad" a los contratos, por ejemplo, dividiéndolos en módulos que pueden ser virtualmente actualizados.

Proxy Delegate Esta actualización se denomina virtual por qué los contratos existentes no pueden ser modificados. Son reemplazados por otros nuevos y su dirección es actualizada en el almacenamiento de un contrato especial que funciona como un Proxy[3] que delega las llamadas a los nuevos contratos. Entonces todas las peticiones que se realizan, se realizan al proxy y los usuarios no necesitan conocer que nuevos contratos han sido liberados.

External storage Otro patrón que está intimamente relacionado con *proxy delegate* es el patrón de *external storage*, donde las variables y todos los datos que un contrato puede usar, son almacenados en un contrato externo y el contrato original puede ser deshabilitado o actualizado sin mayor problema.

4.4.4. Patrones Económicos

Los patrones económicos son utilizados para la optimización de gas y que el uso del gas por las instrucciones sea utilizado de formas eficientes. Por ejemplo, en solidity no existe una forma de comparar cadenas, cosa que en otros lenguajes existe por defecto. Una propuesta para esto es con el patrón *string equality comparison*, el cual básicamente lo que hace es realizar el hash de dos cadenas y compararlas, además de comparar que las dos cadenas tienen la misma longitud. Además de este patrón, se puede aprovechar la forma en que se almacena la información en el *Storage* de Ethereum. Los datos creados en una estructura pueden ordenarse según el espacio que utiliza cada tipo. Esto permite reducir los costos de interacción y además se optimiza para variables de estado con tamaño estático. A este patrón se le conoce cómo *thight variable packing*.

[3]El patrón Proxy, es un patron de diseño de software estructural que tiene como propósito proporcionar un objeto intermedio entre el cliente y el objeto a utilizar.

Se ha escrito más código "bueno" en lenguajes designados como "malos" que en lenguajes designados como "maravillosos" (mucho más).

Bjarne Stroustrup

CAPÍTULO 5

Dependencia en el Orden de Transacciones

En este capítulo proponemos dos nuevas soluciones que ayudan a mitigar el problema de la dependencia en el orden de transacciones. Comenzamos describiendo con detalle el problema en la sección 5.1. En la sección 5.2 describimos en alto nivel las propuestas de las soluciones que ayudan a mitigar el problema del ordenamiento de transacciones y en las secciones 5.3 y 5.4 muestran a detalle cada una de las soluciones propuestas.

5.1 Dependencia en el orden de las transacciones

Una de las ventajas de los contratos inteligentes es que permiten definir reglas de negocio para distintas problemáticas de carácter económico. Actualmente, un problema común es el comercio electrónico automatizado. Con la ayuda de contratos se pueden implementar funciones para la compra y venta de productos y/o servicios.

Consideremos el contrato de una tienda en linea, donde venden un producto y el dueño del producto puede actualizar el precio del producto. Una versión simplificada de este contrato lo observamos en el código 5.1.

```
1  contract Tienda {
2    uint private precio;
3    uint private prodsDisp;
```

```
4     address private dueno;
5
6     /*codigo*/
7     function actualizaPrecio(uint _precio) {
8       if (msg.sender == dueno) precio = _precio;
9     }
10
11    function compra(uint cantidad) returns (uint) {
12      if (cantidad > prodsDisp || msg.value < cantidad * precio)
13          throw;
14      prodsDisp -= cantidad;
15      /*mas codigo*/
16    }
17
18    function obtenPrecio() returns (uint) {
19      return precio;
20    }
21  }
```

Código 5.1: Venta de productos usando contratos inteligentes

El contrato mostrado en el código 5.1 define tres funciones:

- Función para actualizar el precio del producto vendido por la tienda (línea 7). Esta función sólo puede ser invocada por el dueño de la tienda. Un ejemplo de cómo enviar una transacción de actualización es mostrada en el código 5.2.
- Función para realizar la compra del producto por los clientes (línea 11). La función recibe como parámetro la cantidad de elementos a comprar. Un ejemplo de cómo enviar una transacción de compra es mostrada en el código 5.3.
- Función para realizar la consulta del precio del producto (línea 18).

En *Ethereum*, para ejecutar transacciones invocando a las funciones antes descritas, es necesario enviar los datos de la transacción en una cadena, la cual consiste en concatenar los primeros 4 bytes del hash de la función invocada, más el parámetro que recibe la función, además de indicar la dirección de la cuenta que invoca a la transacción y la dirección del contrato invocado. Para tener una notación que nos permita referirnos a las transacciones de una manera más sencilla, nombraremos a las transacciones de la siguiente manera:

T_c: Transacción de compra de producto, la cual es enviada por algún cliente.

T_aa: Transacción de actualización de precio, la cual es enviada por el dueño del contrato.

T_q: Transacción de consulta de precio, la cual puede ser enviada tanto por el cliente, como por el dueño del contrato.

```
var price =
"0000000000000000000000000000000000000000000000000000000000000000ff"
eth.sendTransaction({
  from: eth.accounts[0],
  to: contractAddress,
  data: bytes4(sha3("actualizaPrecio(uint)")) + price
})
```

Código 5.2: Transacción para actualizar el precio de un producto.

```
var newValue=
"0000000000000000000000000000000000000000000000000000000000000001";
eth.sendTransaction({
  from: eth.accounts[0],
  to: contractAddress,
  data: bytes4(sha3("compra(uint)")) + newValue
})
```

Código 5.3: Transacción para comprar un producto.

5.1.1. Ejecuciones Esperadas

Consideremos algunas transacciones enviadas por el cliente y por el dueño del contrato. Comencemos con la ejecución esperada por el cliente. Esta transacción puede interactuar con posibles transacciones enviadas por el dueño del contrato.

1. El cliente envía una transacción de consulta de precio T_q.
2. El cliente envía una transacción de compra T_c.
3. El dueño envía una transacción de actualización T_a.

Cada una de las transacciones enviadas pueden ser añadidas en bloques distintos y posiblemente consecutivos. Podemos observar un ejemplo de esta ejecución en la figura 5.1. En esta figura, representamos cada bloque como una línea punteada y dentro de cada línea, representamos como segmentos a las transacciones que se ejecutaron y fueron incluidas en el bloque. Remarcamos con una escala de grises a las transacciones en las que estamos interesados.

De igual forma, mostremos una ejecución esperada para el dueño del contrato.

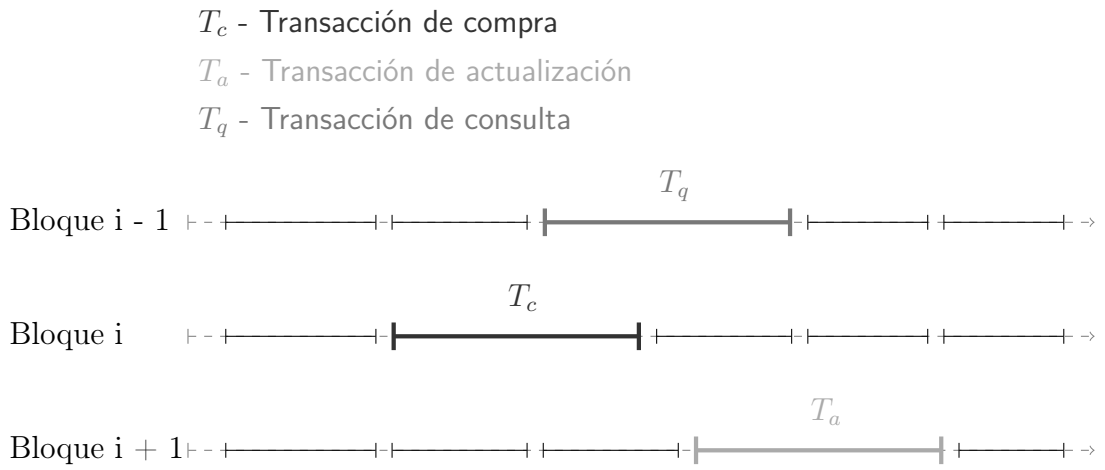

Figura 5.1: Ejecución esperada por el cliente, interactuando con una transacción lanzada por el dueño del contrato.

1. El dueño envía una transacción de actualización T_a.
2. El cliente envía una transacción de consulta de precio T_q.
3. El cliente envía una transacción de compra de producto T_c.

Al igual que con la ejecución del cliente, mostremos una figura para ilustrar la ejecución esperada por el dueño del contrato. Podemos observar lo anterior en la figura 5.2.

Las transacciones anteriores son incluidas en bloques distintos, pero a pesar de que pueden no estar en bloques consecutivos, las transacciones mantienen el orden de la ejecución esperada tanto por el dueño, como por el cliente. Sin embargo, puede que esto no ocurra de esta forma. En la sección 2.4, mencionamos que las transacciones deben de pagar una cantidad de gas para poder ser ejecutadas. En la práctica, los mineros eligen de su conjunto de transacciones de entrada a aquellas que tienen una mayor cantidad de gas para ejecutarse, causando que haya transacciones que sean ejecutadas primero en lugar de otras que fueron enviadas antes. Esto puede causar que las transacciones que fueron enviadas en distintos tiempos, sean incluidas en un mismo bloque o en bloques posteriores.

Otra situación a considerar, es que cuando el minero elige las transacciones que va a incluir en su bloque, el orden de ejecución es arbitraria, provocando ejecuciones

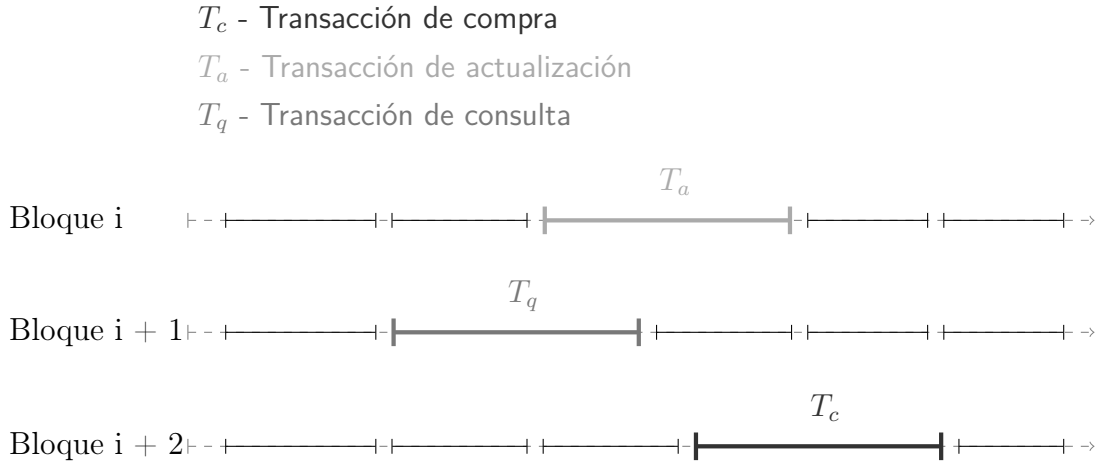

Figura 5.2: Ejecución esperada por el dueño del contrato, interactuando con transacciones lanzadas por un cliente.

no esperadas por los que envían las transacciones. Estas ejecuciones no esperadas las explicaremos más adelante.

5.1.2. Ejecuciones no esperadas

Volvamos a considerar la notación para las transacciones enviadas por el cliente y el dueño del programa:

T_c: Transacción de compra de producto enviada por algún cliente.

T_a: Transacción de actualización de precio enviada por el dueño del contrato.

T_q: Transacción de consulta de precio, la cual puede ser enviada tanto por el cliente, como por el dueño del contrato.

De manera similar a como describimos las ejecuciones esperadas, vamos a describir las no esperadas por el cliente y el dueño del contrato. Comencemos con las ejecuciones no esperadas por el cliente. Consideremos el caso cuando en un mismo bloque son agregadas las transacciones enviadas por el cliente y el dueño. Ambos envían transacciones en el siguiente orden:

1. El cliente envía una transacción de consulta de precio T_q.
2. El cliente envía una transacción de compra T_c.

3. El dueño envía una transacción de actualización T_a.

Sin embargo, en el bloque las ejecuciones de las transacciones son añadidas en el siguiente orden: [T_q, T_a, T_c]. Esto causa que el cliente compre el producto a un precio distinto (el cual puede ser mayor) al que deseaba. Observamos esto en la figura 5.3.

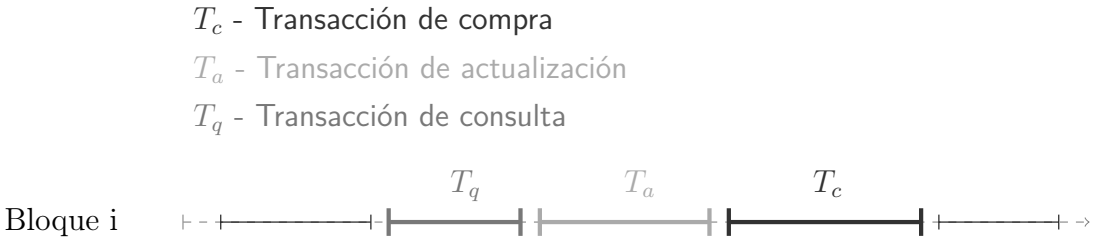

Figura 5.3: Ejecución no esperada por el cliente, la combinación de transacciones es [T_q, T_a, T_c].

Otra combinaciones de ejecuciones de transacciones añadidas al bloque son: [T_a, T_q, T_c] y [T_a, T_c, T_q]. Estas ejecuciones las podemos observar en las figuras 5.4 y 5.5 respectivamente. En ambas, el cliente sigue comprando a un precio distinto e incluso, en la última secuencia, la consulta de precio se ejecuta al final.

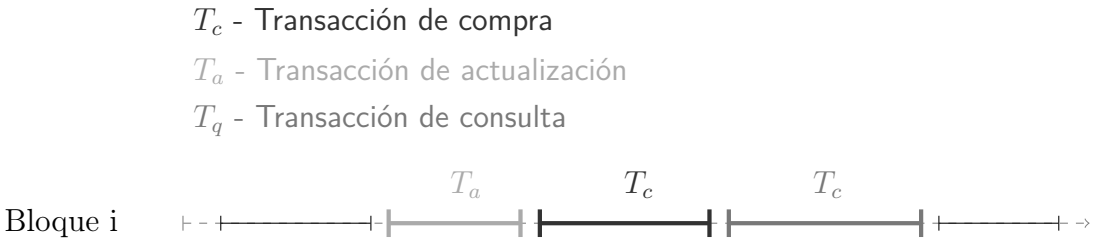

Figura 5.4: Ejecución no esperada por el cliente, la combinación de transacciones es [T_a, T_c, T_q].

En el caso del dueño del producto, tenemos una situación similar, en la que la transacción de actualización de precio se ejecute después de al menos una transacción de compra, lo que puede resultar en perdidas para él. Esto lo podemos observar en

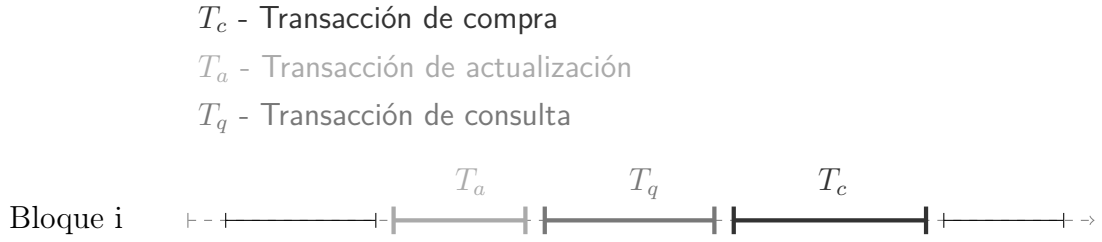

Figura 5.5: Ejecución no esperada por el cliente, la combinación de transacciones es $[T_a, T_q, T_c]$.

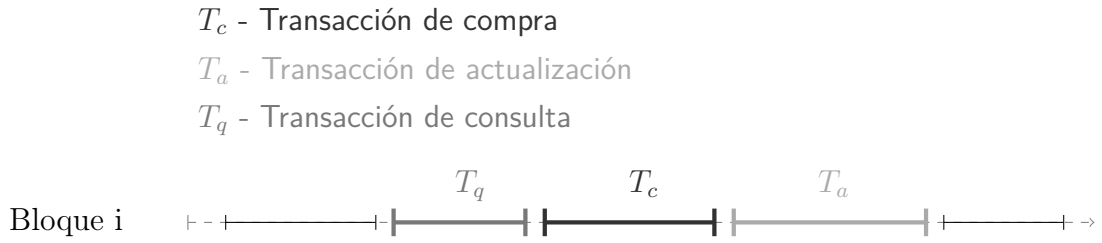

Figura 5.6: Ejecución no esperada por el dueño del contrato, interactuando con transacciones lanzada por el cliente.

la figuras 5.6 y 5.7.

A este tipo de ejecuciones, donde la salida está determinada por el orden de la evaluación arbitraria de las transacciones, se le conoce como condición de carrera [47]. Esto sucede por que las transacciones "compiten" por ser evaluadas primero y agregadas al bloque de transacciones. Esto puede provocar inconsistencias con los resultados esperados, así como comportamientos impredecibles y que parezcan no compatibles con un sistema determinista. A esto es lo que conocemos como el **problema del ordenamiento de transacciones**.

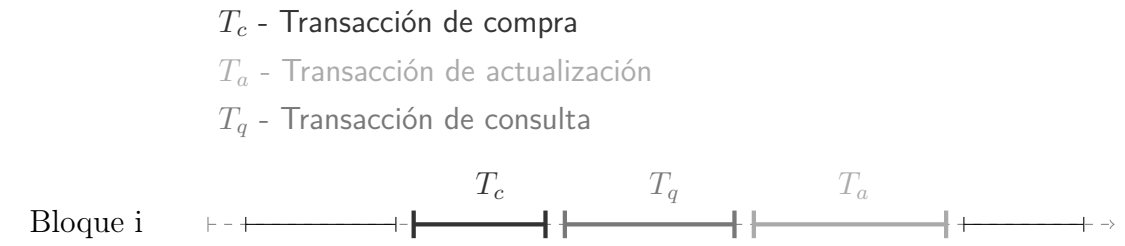

Figura 5.7: Ejecución no esperada por el dueño del contrato, interactuando con transacciones lanzadas por el cliente.

5.2 Soluciones al Problema del Ordenamiento de Transacciones

El problema del ordenamiento de transacciones puede ser transformado en un problema de cómputo concurrente. Para entender cuál es la relación que hay entre ambos, expliquemos en que consiste un sistema concurrente. Dentro de los sistemas operativos, los procesos generalmente son ejecutados por un procesador de principio a fin. Con la aparición de los sistemas operativos, se diseñaron formas de ejecutar múltiples procesos en un sólo procesador, dando la sensación de que se ejecutaban paralelamente. Esto es posible gracias a que se diseñaron estrategias para ejecutar procesos de forma parcial y por tiempos, de modo que pudieran compartir recursos entre sí. En los sistemas monoprocesador, cada proceso tenía su propia memoria y lo único que compartían era el procesador. En tiempos más recientes, se diseñaron arquitecturas multiprocesador, donde ahora los procesos podían ser ejecutados de forma arbitraria entre los distintos procesadores y para poder realizar esto, los procesadores comenzaron a compartir memoria. Esto implica un reto para mantener coordinación durante la ejecución de procesos, ya que al compartir un recurso como la memoria, es posible que entre ellos lleguen a estados inconsistentes o se bloquen entre ellos. Para una mayor información, podemos revisar los capítulos 2 al 6 del libro *Art of Multiprocessor Programming* de Maurice Herlihy [25].

Existe una similitud entre la ejecución de procesos en multiprocesadores y la ejecución de transacciones durante el proceso de minado. En el modelo de sistemas concurrentes con multiprocesadores, los datos que se encuentra en memoria compartida la podemos comparar con los datos que ya se encuentra en la cadena de bloques;

de manera análoga, el procesador es equivalente al minero y los procesos son similares a las transacciones [36].

En los sistemas concurrentes, una forma de actualizar el valor de una variable compartida de forma atómica es utilizando la primitiva `compareAndSet` [25]. Este método es también conocido en la literatura como `compareAndSwap`. El método `compareAndSet` toma dos argumentos, un valor esperado y un valor de actualización. Si el valor que se quiere actualizar es igual al valor esperado, entonces lo reemplaza por el valor de actualización, en otro caso, lo deja sin cambios. El método regresa un *boolean* que indica si el valor ha cambiado. Debido a que en Ethereum no existe primitivas de concurrencia, podemos tomar la idea de `compareAndSet` para implementar una función que nos permita realizar las operaciones de actualización o de compra de forma segura. La idea anterior es la base de la primera solución propuesta para mitigar el problema del ordenamiento de transacciones. Esta primera propuesta la describimos con mayor detalle en la sección 5.3.

La propuesta anterior se puede implementar en la fase de diseño e implementación del contrato inteligente. Pero, ¿qué pasa con los contratos que han sido desplegados en la cadena de bloques y no se pueden modificar? Una opción sería utilizar un patrón de actualización cómo los que mencionamos en la sección 4.4.3. Sin embargo, es posible que haya contratos ya fueron desplegados y no exista forma de actualizarlos usando los patrones de actualización. Un enfoque distinto es el de permitir que el minero obtenga secuencias de transacciones que pueda ejecutar de forma atómica.

La ejecución secuencial atómica se puede realizar al agrupar las transacciones en bloques, de modo que la ejecución se parezca a la evaluación de un método. Una propuesta para realizar esto es que los bloques sean ejecutados como si fueran *funciones lambda* con *closures* [30]. A grandes rasgos una función lambda es una función anónima que es utilizada en el contexto en que fue creada. Un *closure* permite asociar algunos datos con una función que opera con esos datos. En el caso de las ejecuciones de transacciones en la cadena de bloques, el closure estaría asociado con los valores y funciones desplegados para ser usadas dentro de la función anónima. Para que esto funcione, es necesario hacer modificaciones en la máquina virtual de Ethereum para soportar las funciones lambda. En la sección 5.4 discutiremos con mayor detalle esta propuesta.

Los propuestas anteriores tienen dos enfoques distintos, por lo que tienen varias diferencias. La primera propuesta va enfocada hacia la parte de diseño e implemen-

tación de contratos, en donde desde estas etapas se identifica que puede darse la problemática del ordenamiento de transacciones. Y la segunda propuesta va orientada a proveer herramientas para agregar características en la máquina virtual para poder agregar una pequeña cantidad de lógica adicional a las transacciones, de modo que puedan corregirse o mitigarse ciertos problemas que puedan darse durante el proceso de minado.

5.3 Primera Solución: Patrón de diseño

Una forma de atacar el problema del ordenamiento de transacciones, es la de implementar la primitiva `compareAndSet` para realizar la compra de productos en el caso de los clientes. Sin embargo, debido a la semántica que proveen los contratos inteligentes, se pueden simular la primitiva usando condicionales y/o aserciones. Describamos esta solución en términos de un patrón de diseño.

De acuerdo a la descripción que dimos en 4.2, describamos un patrón de diseño para mitigar el problema del ordenamiento de transacciones. Esta solución la vamos a basar en las propiedades de `compareAndSet`. Definamos el esquema del patrón describiendo el contexto, problema y solución.

- **Contexto:** Un contrato en el que se tiene una variable que puede ser modificada por más de una función. Un ejemplo es el contrato mostrado en el código 5.1.
- **Problema:** Dos (o más) transacciones que invocan a funciones distintas que modifican un mismo estado (variable), pueden ser ejecutadas por un minero dentro de un mismo bloque. Debido a que los mineros pueden elegir de forma arbitraria el orden de ejecución de las transacciones, la salida no es la esperada. La solución debe satisfacer las siguientes características:
  - La solución debe añadir un incremento pequeño en el costo total de gas utilizado por la función.
  - La solución debe estar basada en condicionales y aserciones.
- **Solución:** Se identifica el estado compartido (variable o variables) que pueden causar una condición de carrera. Una vez identificada la variable (o variables), se deben modificar las funciones que pueden causar la condición de carrera de modo que incluyan en sus parámetros el valor esperado del estado concurrente y el valor de la actualización. Dentro del cuerpo de la función, al inicio se debe

comparar si el valor dado como parámetro es igual al del estado concurrente. Si es igual, se actualiza el valor del estado concurrente, si no, se aborta la transacción. Esta idea es la misma que la función `compareAndSet` (código 5.4), sólo que adaptada a la función que tiene el problema. Notemos que la ejecución de la función es totalmente atómica, debido a que cada transacción se ejecuta de forma secuencial durante el minado del bloque.

```
1  public synchronized int compareAndSet(int expectedValue,
2  int newValue) {
3      int readValue = value;
4      if (readValue == expectedValue) {
5         value = newValue;
6      }
7      return readValue;
8  }
```

Código 5.4: Ejemplo de implementación de la función `compareAndSet` en Java.

Analicemos el código 5.1 para aplicar la solución propuesta descrita en el patrón de diseño anterior. Observamos que la función `compra` definida en las líneas 11–16, tiene un posible condición de carrera que se puede dar al tener transacciones que modifiquen el valor de la variable `precio`. Esta variable puede ser modificada la función `actualizaPrecio` y dependiendo del orden de minado de las transacciones encargadas de modificar el precio y comprar producto, pueden darse varias de las ejecuciones no esperadas descritas en la sección 5.2.

Podemos volver a implementar la función `compra` con condicionales (`if`) para que sea similar a la primitiva `compareAndSet`, esto lo mostramos en el código 5.5. Sin embargo, la API[1] de *Solidity* provee dos funciones que permiten reemplazar el uso de condicionales y además satisfacer la restricción del patrón de diseño de no añadir un mayor incremento, lo que lo hace una implementación idiomática.

```
1  contract Tienda {
2    uint private precio;
3    uint private prodsDisp;
4    address private dueno;
5
6    /*codigo*/
```

[1]Application Programming Interface (Interfaz de programación de aplicaciones), es un conjunto de funciones, métodos y subrutinas que ofrecen las bibliotecas de software.

```
7     function actualizaPrecio(uint _precio) {
8       assert(msg.sender == owner);
9       precio = _precio;
10    }
11
12    function compra(uint _precio, uint cantidad) returns (uint) {
13      require(_precio == precio, "El precio ha cambiado");
14      assert(cantidad > prodsDisp || msg.value < cantidad * precio);
15      prodsDisp -= cantidad;
16      /*mas codigo*/
17    }
18
19    function obtenPrecio() returns (uint) {
20      return precio;
21    }
22  }
```

Código 5.5: Código actualizado con las aserciones y condicionales para evitar la condición de carrera.

Observamos en el código 5.5, que en la líneas 8, 13 y 14, se reemplaza la condicional `if`, por dos funciones: `assert` y `require`. La forma idiomática de usar `assert` es para probar que se cumplan ciertas invariantes en la ejecución, validar el estado después de cambio, etc. `Require` es utilizada para validar entradas del usuario, validar respuestas o mensajes de otros contratos, etc. Además de esta diferencia, cada uno tiene un nivel de eficiencia diferente. `Assert` compila al código de operación `0xfe`, el cual es un código que lanza una condición de error y consume todo el gas restante de la transacción, mientras que `require` compila al código de operación `0xfd`, el cual devuelve el gas remanente al que invocó la transacción[2].

Regresando al diseño de la función, también observamos que es muy similar a las implementaciones clásicas de `compareAndSet`. Podemos observar que en la firma de la función tenemos un valor esperado, mientras que mantenemos el valor de actualización, que es la cantidad de elementos a comprar (línea 12). En la siguiente línea, tenemos la evaluación condicional que es equivalente a la condicional que verifica si el estado de la variable no ha cambiado. Si no ha cambiado, entonces continua con la ejecución. La principal diferencia a nivel de lógica de negocio, es que la función

[2] ¿Por qué elegir una función respecto a otro si pareciera que hacen lo mismo? Como ya se mencionó anteriormente, `require` es utilizado en situaciones donde hay que validar datos externos, donde es posible que falle las condiciones esperadas. Pero en el caso de `assert`, si el código está bien implementado, la ejecución nunca debería de lanzar la ejecución de `assert`, si esto sucede es porqué hay algún error en el código.

`compareAndSet` sólo actualiza un valor de forma atómica, mientras que la función que simula `compareAndSet`, ejecuta más instrucciones, pero también se realiza de forma atómica por la naturaleza misma de la ejecución de las transacciones durante el proceso de minado.

Esta solución es efectiva, porqué sabemos que la función `compareAndSet` tiene un *número de consenso* infinito[3], por lo que puede coordinar cualquier número de hilos [25]. Al modelar un patrón basado en esta primitiva, podemos garantizar que el patrón permite solucionar cualquier problema de coordinación, en particular para el problema del ordenamiento de transacciones.

5.4 Segunda Solución: Transacciones con closures

La solución basada en patrones de diseño funciona bien si aún no hemos desplegado nuestro contrato en la cadena de bloques. ¿Qué pasa si el contrato 5.1 ya se ha desplegado? Una solución sería implementar un contrato adicional que implemente un *patrón adaptador*, donde se haga la verificación y se realice la compra. Un ejemplo de contrato que implementa un *patrón adaptador* lo observamos en el código 5.6.

```
1  contract TiendaAdaptador {
2
3    Tienda tienda = Tienda(0x1234);
4
5    function compra(uint _precio, uint cantidad) returns (uint) {
6      require(_precio == tienda.obtenPrecio(), "El precio ha cambiado")
           ;
7      tienda.compra(cantidad);
8    }
9  }
```

Código 5.6: Contrato que implementa el patrón *Adaptador* para envolver la función de compra del contrato 5.1.

El contrato del código 5.6 podría ser una buena solución a nivel de patrones de diseño, pero, un problema que tiene, es que cada cliente tendría que implementar un contrato similar al previo o que el dueño provea dicha funcionalidad, lo que implicaría

[3]El número de consenso de una primitiva de sincronización, es el número de hilos que puede coordinar para resolver problemas de coordinación.

un mayor gasto en gas tan sólo para desplegarlo. Necesitamos una forma de invocar transacciones, de modo que podamos ejecutar múltiples transacciones en orden. Una alternativa es proponer un cambio en la manera de invocar transacciones a nivel de máquina virtual.

Una transacción invoca una función a la vez. Si un usuario quisiera saltarse la implementación del contrato 5.6 y realizar la compra, no existe una forma de realizar la validación del precio y comprar el producto de forma atómica. ¿Cómo podríamos hacer esto?

Una propuesta para resolver este problema, es que la máquina virtual de Ethereum permita la construcción *closures*. Un *closure* es una expresión anónima (*lambda*) asociada a un entorno, que permite enlazar variables libres dentro de la expresión hacia un valor del entorno [30]. Al definirlas de este modo, un minero podría ejecutar las funciones definidas dentro del closure de forma atómica, ya que uno de los valores que se da como parámetro al crear una transacción, es la dirección del contrato que contiene la función. Una propuesta de invocación de transacciones en términos del ejemplo del problema de este capítulo, lo podemos observar en el código 5.7, donde notamos que es muy similar a las transacciones mostradas en los códigos 5.2 y 5.3.

```
var newValue=
"0000000000000000000000000000000000000000000000000000000000000001";
var expectedPrice=
"0000000000000000000000000000000000000000000000000000000000000001";
var amount=
"0000000000000000000000000000000000000000000000000000000000000001";
  eth.sendTransaction({
    from: eth.accounts[0],
    to: contractAddress,
    data: bytes4(sha3("compra(uint)")) + newValue,
    l_function: (uint _precio, uint cantidad) -> {
      require(_precio == obtenPrecio(), "El precio ha cambiado");
      compra(cantidad);
    }
    l_params: (expectedPrice, amount),
  })
```

Código 5.7: Propuesta de transacción con soporte a funciones lambda con closures.

En el código de la transacción 5.7, se agregan dos campos que nombramos como `l_function` y `l_params`. En el campo `l_function`, definimos una función lambda asociada a la dirección del contrato con el que queremos interactuar, mien-

tras que en `l_params` definimos los parámetros que le pasamos a la función definida en `l_function`. Suponiendo que el compilador de Solidity y la máquina virtual de Ethereum operan como esperamos, esa transacción asocia la dirección (y contenido) del contrato a la función lambda definida en el campo `l_function`, entonces, cuando se ejecute la función anónima, está será un *closure*, que invoca a los parámetros definidos en `l_params`, y las variables libres estarán enlazadas a variables y funciones definidas en el contrato invocado.

Esta solución es interesante porque permite construir secuencias de transacciones para ser ejecutadas por el minero de forma atómica. A partir de las instrucciones que tenga definida se podría determinar la cantidad de gas que necesita. Sin embargo, para que esto funcione es necesario que se modifique el código (añadir opcodes que permitan la invocación secuencial de transacciones) de la máquina virtual y del compilador del lenguaje.

Uno de los beneficios que tiene esta solución, es la de permitir verificar el estado de la cadena de bloques al momento de querer ejecutar alguna función que tenga dependencias del estado. Además de poder implementar soluciones pequeñas basadas en el patrón adaptador sin tener que declarar un nuevo contrato y desplegarlo. Esto puede ayudar a mitigar no sólo el problema de la dependencia en el orden de las transacciones, si no también a algunas otras vulnerabilidades mientras se diseñan algunas otras soluciones para reemplazar a los contratos. Algunos casos de uso en los que esta solución puede ayudar, los listamos a continuación.

- *Gasless send* [4.1.1], al verificar que haya suficiente gas para invocar alguna transacción.
- *Perdida de ethers en transferencias* [4.1.2], al validar la dirección de un contrato o cuenta.
- *Estado impredecible* [4.1.3], al verificar estado antes de ejecutar alguna función.

Mientras que la mayoría de tecnologías tienden a automatizar a los trabajadores que están en la periferia realizando tareas menores, las Blockchain automatizan el centro. En lugar de dejar al taxista sin trabajo, blockchain deja a Uber sin trabajo y permite que los taxistas trabajen directamente con el cliente.

Vitalik Buterin

CAPÍTULO 6

Conclusiones y Trabajo a Futuro

6.1 Conclusiones

La cadena de bloque es una estructura fundamental para la ejecución de criptomonedas, contratos inteligentes, cadenas de suministros y otras aplicaciones. En este trabajo, en el capítulo 2 mostramos las bases teóricas necesarias para introducirnos en el estudio de las cadenas de bloques, donde vimos las bases criptográficas para entender su funcionamiento. También explicamos a alto nivel cómo funcionaba *Bitcoin* e introdujimos *Ethereum*. En el capítulo 3, explicamos con mayor detalle lo que son los contratos inteligentes, cómo es que funcionan en Ethereum, así como los casos de uso que tiene. En el capítulo 4 describimos algunas de las vulnerabilidades de los contratos inteligentes y su clasificación, además de mostrar que podemos mitigar varias de ellas con la ayuda de los patrones de diseño. Y en el capítulo 5 estudiamos con mayor detalle una vulnerabilidad en particular y propusimos dos soluciones.

Estudiando el funcionamiento de los contratos inteligentes en *Ethereum*, entendimos cómo opera la infraestructura que permite su ejecución en las cadenas de bloques. Con esto, comprendimos varias problemáticas que son aprovechadas por las vulnerabilidades clasificadas en [3]. La clasificación se realiza en tres niveles: *Nivel de cadena de bloques*, *nivel de máquina virtual* y *nivel de lenguaje*. Cada uno de estos niveles contiene varias vulnerabilidades y pueden ser combinadas para realizar ataques más complejos.

De entre todas las vulnerabilidades, elegimos el *problema del ordenamiento de transacciones* y al ver que existía una relación entre poder actualizar un valor de forma atómica en sistemas concurrentes y verificar el estado para que el flujo de la transacción se complete, propusimos una solución que toma las ideas de la primitiva `compareAndSet` de cómputo concurrente y proveer un patrón de diseño, lo que implica una solución desde las etapas de diseño e implementación.

De igual forma, nos preguntamos qué pasaría si el contrato ya se encuentra desplegado y proponemos una solución que tiene un impacto en la infraestructura de *Ethereum*. Esta segunda solución requiere un trabajo mucho mayor, ya que es necesario modificar la máquina virtual, de tal modo que se incluyan nuevos opcodes que permitan la ejecución de funciones anónimas con *closures* y se modifique también el lenguaje de *Solidity* para que exista una correspondencia con lo anterior. Esto no representa un trabajo trivial, ya que esta modificación debe ser propuesta para ser integrada en la especificación formal de *Ethereum* (*Yellow Paper*) [49]. Después de esto, cada implementación del cliente de *Ethereum* que ejecuta la máquina virtual debe ser actualizada.

Aunque la tecnología es nueva, aún hay muchas cosas por mejorar y dichas mejoras pueden provenir de diversas áreas de computación. En este trabajo, para atacar nuestro problema, utilizamos conceptos de *Ingeniería de Software*, *Cómputo Concurrente* y *Programación Funcional* que en principio parecería que no están relacionadas con la problemática, pero entre más crece la infraestructura y los problemas que se resuelven con ella, más áreas de conocimiento se necesitan para que se den nuevas soluciones, continúe operando y se añadan nuevas características.

A pesar de lo anterior, también es importante mencionar que existe una gran similitud entre las cadenas de bloques y mucha de la teoría del área de cómputo distribuido. Ya lo mencionamos en la sección 5.3, donde "*varios de los algoritmos y técnicas usadas en las cadenas de bloques son entendidas como variantes de algoritmos conocidos y técnicas clásicas de cómputo distribuido*", así como permitir que "*los lenguajes para contratos inteligentes deberían de tener un modelo explícito de concurrencia para hacer a los programadores conscientes de conocidas dificultades y peligros de la concurrencia*" [24].

6.2 Trabajo a Futuro

Actualmente, hay un crecimiento importante en nuevas propuestas para cadenas de bloques. *Bitcoin* y *Ethereum* son cadenas de bloques públicas, es decir, cualquiera se puede unir a su protocolo y participar. También existen cadenas de bloques privadas o permisionadas (como Hyperledger de IBM [10] o R3 Corda [5]), donde se eligen los nodos que pueden participar en el protocolo y cual es el rol que tendrán en la cadena de bloques. Con estas nuevas propuestas se van creando nuevas aplicaciones para su uso.

En la sección 3.3 mencionamos algunos de los casos de uso de los contratos inteligentes, mucho del trabajo a futuro que hay en esta área es la de crear nuevos protocolos de operación para los casos de uso existentes y los que se puedan crear, no sólo de contratos inteligentes, si no de las cadenas de bloque en general. En el trabajo a futuro, se encuentra la investigación en temas de protocolos criptográficos para proponer nuevos modelos de consenso como los propuestos para *Z-Cash* [26] que utilizan protocolos basados en *Zero Knowledge Proofs* o en *Algorand* [22], donde proponen nuevos protocolos de acuerdos bizantinos basados en técnicas de clasificación criptográfica. Además de que estas soluciones deberían permitir la ejecución de contratos inteligentes que se beneficien de la ejecución de estos protocolos y técnicas.

Bibliografía

[1] Ansi x9.62, public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ecdsa), September 1998. American National Standards Institute, X9-Financial Services.

[2] Alexander, C. *The timeless way of building*, vol. 1. New York: Oxford University Press, 1979.

[3] Atzei, N., Bartoletti, M., and Cimoli, T. A survey of attacks on ethereum smart contracts. In *Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[4] Back, A. Hash cash: A partial hash collision based postage scheme. `http://www.hashcash.org/papers/announce.txt`, 2001.

[5] Brown, R. G., Carlyle, J., Grigg, I., and Hearn, M. Corda: an introduction. *R3 CEV, August 1* (2016), 15.

[6] Buschmann, F., Henney, K., and Schmidt, D. C. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, UK, 2007.

[7] Buterin, V. A next-generation smart contract and decentralized application platform-ethereum whitepaper, 2014.

[8] Buterin, V. Hard fork completed. `https://blog.ethereum.org/2016/07/20/hard-fork-completed/`, July 6, 2016.

[9] Buterin, V., and Griffith, V. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).

[10] Cachin, C. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers* (2016), vol. 310, p. 4.

[11] ccn. Smart contracts: 12 use cases for business and beyond. https://www.ccn.com/smart-contracts-12-use-cases-for-business-and-beyond, 2019.

[12] Chaum, D. Blind signatures for untraceable payments. In *Advances in cryptology* (1983), Springer, pp. 199–203.

[13] Chaum, D., Fiat, A., and Naor, M. Untraceable electronic cash. In *Conference on the Theory and Application of Cryptography* (1988), Springer, pp. 319–327.

[14] Consensy. Ethereum smart contract best practices - known attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/, 2019.

[15] Digiconomist. Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption, 2019.

[16] FIPS, P. 140-2. *Security Requirements for Cryptographic Modules 25* (2001).

[17] FIPS, P. 197, advanced encryption standard (aes), national institute of standards and technology, us department of commerce, november 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2009.

[18] FIPS, P. 180-4. *Secure hash standard (SHS), March* (2012).

[19] Fips, P. Fips pub 202: Sha-3 standard: Permutation-based hash and extendable-output functions. *National Institute of Standards and Technology (NIST)* (2015).

[20] Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[21] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1995.

[22] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 51–68.

[23] Gold & Silver Reserve, I. Synopsis of e-gold transactions. https://web.archive.org/web/19980627133928/http://www.e-gold.com/unsecure/synopsis.htm#redeem, 1998.

[24] Herlihy, M. Blockchains from a distributed computing perspective. *Commun. ACM 62*, 2 (Jan. 2019), 78–85.

[25] Herlihy, M., and Shavit, N. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[26] Hopwood, D., Bowe, S., Hornby, T., and Wilcox, N. Zcash protocol specification. *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep.* (2016).

[27] Katz, J., Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. *Handbook of applied cryptography*. CRC press, 1996.

[28] Kerrisk, M. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[29] Koblitz, N. Elliptic curve cryptosystems. *Mathematics of computation 48*, 177 (1987), 203–209.

[30] Krishnamurthi, S. *Programming Languages: Application and Interpretation*. Brown University, 2012.

[31] Lynch, N. A. *Distributed algorithms*. Elsevier, 1996.

[32] Miller, V. S. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques* (1985), Springer, pp. 417–426.

[33] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Cypherpunks Mailing List* (2008).

[34] Pfeffer, C. J. Over 12,000 ether are lost forever due to typos. https://media.consensys.net/over-12-000-ether-are-lost-forever-due-to-typos-f6ccc35432f8, 2018.

[35] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[36] Sergey, I., and Hobor, A. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security* (2017), Springer, pp. 478–493.

[37] Siegel, D. Understanding the dao attack. `http://www.coindesk.com/understanding-dao-hack-journalists`, 2016.

[38] Stal, M., Buschmann, F., and Meunier, R. Pattern-oriented software architecture—a system of patterns, 1996.

[39] Szabo, N. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)* (1996).

[40] Szabo, N. Bit gold, 2008.

[41] Team, B. Bitcoin developer guide - nbits. `https://bitcoin.org/en/developer-reference#target-nbits`, 2019.

[42] Team, B. Bitcoin developer guide - proof of work. `https://bitcoin.org/en/developer-guide#proof-of-work`, 2019.

[43] Team, E. Lll, low-level-like-lisp. `https://solidity.readthedocs.io/en/v0.5.3/lll.html`, 2019.

[44] Team, E. Proof of stake faqs. `https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs`, 2019.

[45] Team, E. Solidity docs. `https://solidity.readthedocs.io/en/develop/`, 2019.

[46] Team, E. Vyper, pythonic smart contracts language for evm. `https://github.com/ethereum/vyper`, 2019.

[47] Unger, S. H. Hazards, critical races, and metastability. *IEEE Trans. Computers 44* (1995), 754–768.

[48] Volland, F. A compilation of patterns and best practices for the smart contract programming language solidity. `https://fravoll.github.io/solidity-patterns/`, 2018.

[49] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper 151* (2014), 1–32.