UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

MARCO DE REFERENCIA PARA LA TRAZABILIDAD Y ALMACENAMIENTO DE ACTIVOS DIGITALES

TESIS
QUE PARA OPTAR POR EL GRADO DE
MAESTRO EN CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:
JOSÉ ANTONIO JIMÉNEZ MIRAMONTES

TUTORES PRINCIPALES
DRA. ROCÍO ALDECO-PÉREZ, FI-UNAM
DR. SERGIO RAJSBAUM GORODEZKY, IM-UNAM

MIEMBROS DEL COMITÉ TUTOR
DR. ARMANDO CASTAÑEDA ROJANO, IM-UNAM
DR. JOSÉ DAVID FLORES PEÑALOZA, FC-UNAM
DR. JAIME CAMACHO ESCOTO, FI-UNAM

CIUDAD UNIVERSITARIA, CDMX, FEBRERO 2023

Universidad Nacional
Autónoma de México

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.

A la Universidad por todos los conocimientos proporcionados.
Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por su patrocinio que permitió cursar la maestría y al proyecto UNAM-PAPIIT TA 101021 por el apoyo otorgado para las publicaciones derivadas de este trabajo.
A mis papás y a mi hermana por su apoyo incondicional.
A la Dra. Rocío Aldeco-Pérez por su confianza, enseñanzas y su amistad.
A Mariana Garibay por su amor y por siempre creer en mi.
A mis gatos por acompañarme en noches de desvelo.
Es gracias a ustedes que es posible el presente trabajo.

Índice general

Índice de figuras V

Índice de tablas VII

1. Introducción 1

2. Marco teórico 3
- 2.1. Conceptos criptográficos de interés . . . 3
  - 2.1.1. Cifrado . . . 3
    - 2.1.1.1. Simétrico . . . 4
    - 2.1.1.2. Asimétrico . . . 4
  - 2.1.2. Función hash criptográfica . . . 5
    - 2.1.2.1. Árbol de Merkle . . . 6
  - 2.1.3. Firma digital . . . 8
- 2.2. Linaje electrónico . . . 8
- 2.3. Blockchain . . . 10
  - 2.3.1. Tipos de Blockchain . . . 12
  - 2.3.2. Protocolos de consenso . . . 14
    - 2.3.2.1. Proof of Work . . . 15
    - 2.3.2.2. Proof of Stake . . . 16
    - 2.3.2.3. Proof of History . . . 17
  - 2.3.3. Contratos inteligentes . . . 18
- 2.4. Blockchain Solana . . . 19
  - 2.4.1. Cuentas . . . 21
  - 2.4.2. Transacciones . . . 22
  - 2.4.3. Programas . . . 24
- 2.5. InterPlanetary File System . . . 24
- 2.6. Trabajos relacionados . . . 28

3. Descripción del marco de referencia 31
- 3.1. Análisis . . . 32
- 3.2. Subir imagen . . . 35
- 3.3. Modificar permisos . . . 38

3.4. Descargar imagen . . . 39
3.5. Editar imagen . . . 40
3.6. Atomicidad en las operaciones sobre imágenes . . . 42

4. Implementación **43**
4.1. Front-end . . . 44
4.1.1. Criptografía . . . 44
4.1.2. Imágenes . . . 45
4.1.3. Comunicación con IPFS . . . 46
4.1.4. Comunicación con Solana . . . 47
4.2. Contrato inteligente . . . 51
4.2.1. entrypoint.rs . . . 52
4.2.2. instruction.rs . . . 53
4.2.3. processor.rs . . . 54
4.2.4. state.rs . . . 55
4.3. Ejemplos de uso . . . 55
4.3.1. Subir imagen . . . 56
4.3.2. Modificar permisos . . . 59
4.3.3. Compartir llaves . . . 60
4.3.4. Descargar Imagen . . . 61
4.3.5. Editar Imagen . . . 62
4.4. Registro de mensajes durante transacciones . . . 62

5. Resultados **63**
5.1. Consulta de transacciones y visualización de imágenes . . . 63
5.2. Consulta del linaje electrónico . . . 66

6. Conclusiones **71**

A. Código **73**
A.1. Código Typescript . . . 73
A.1.1. utils.ts . . . 73
A.1.2. cryptography.ts . . . 75
A.1.3. solana-com.ts . . . 82
A.2. Código Rust . . . 97
A.2.1. entrypoint.ts . . . 97
A.2.2. instruction.rs . . . 98
A.2.3. processor.rs . . . 100
A.2.4. state.rs . . . 108
A.2.5. error.rs . . . 113

Bibliografía **115**

Índice de figuras

2.1. (a) Algoritmo de cifrado y (b) Algoritmo de descifrado (modificado de [9]). 3
2.2. Tipos de cifrado que existen dentro de la Criptografía [11]. 4
2.3. (a) Cifrado de bloques y (b) Cifrado de flujo [11]. 4
2.4. Proceso de cifrado y descifrado asimétrico. 5
2.5. Función hash (modifcado de [9]). 5
2.6. Árbol de Merkle (modificado de [14]). 7
2.7. Árbol de Merkle (modificado de [14]). 7
2.8. Proceso de firma digital. 9
2.9. Estructura de bloque 11
2.10. Árbol de Merkle con las transacciones de un bloque. 11
2.11. Funcionamiento Blockchain. 12
2.12. Relación entre las 3 propiedades principales de Blockchain [22]. 13
2.13. Secuencia de hashes que alimentan PoH. 18
2.14. Secuencia de hashes con el registro de transacciones. 18
2.15. Ejemplo de cuentas en Solana una cuenta de programa y una cuenta que almacena información [31]. 22
2.16. Anatomía de una transacción. 23
2.17. Direccionamiento basado en la ubicación del archivo y basado en el contenido. Basado en [35] 25
2.18. Anatomía del identificador de contenido. 25
2.19. Construcción del cid de la imagen a partir de bloques. 26
2.20. Construcción del cid de la imagen a partir de bloques [41]. 27
2.21. DAG de Merkle después de la modificación del directorio [41]. 28

3.1. Arquitectura del marco de referencia 32
3.2. Estructuras de datos para representar parte del linaje electrónico de la imagen. 34
3.3. Proceso de subir una imagen 35
3.4. Estructura de datos para compartir llaves para descifrar imágenes . . . 36
3.5. Proceso de compartir llave simétrica de una imagen privada 37
3.6. Proceso de modificar permisos para una imagen 39
3.7. Proceso de descargar una imagen 40

3.8. Diferencia entre imagen original y editada 41
3.9. Proceso de editar una imagen 41
3.10. Proceso de editar una imagen 42

4.1. Arquitectura de la implementación 43
4.2. Procesos del *front-end* en los que se involucran llaves criptográficas. . . 45
4.3. Comparación de la imagen original y la modificada. 46
4.4. Flujo de proceso del contratos inteligente 52
4.5. Representación de funcionamiento dentro de instruction.rs. 53
4.6. Representación del funcionamiento dentro de **processor.rs** 54
4.7. (a) Deserialización de la información (unpack) y (b) Serialización de la información (pack). 55
4.8. (a)Crear cuenta y (b) Subir imagen. 56
4.9. Datos de instrucción para la operación subir imagen. 57
4.10. Selección de la instrucción upload image. 57
4.11. Selección de la función process_upload_img_data 58
4.12. (a) Deserialización de la información en la cuenta y (b) Serialización de la información hacia la cuenta. 58
4.13. Modificación de permisos. 59
4.14. Datos de instrucción para la operación modificar permisos. 59
4.15. Modificación de los campos de interés para modificar permisos. 60
4.16. Mensajes generados por tipo de operación. 62

5.1. Visualización general de una transacción. 64
5.2. Detalles del contrato inteligente. 65
5.3. Visualización de la información referente a la imagen subida a IPFS . . 65
5.4. Visualización de la imagen subida a IPFS. 66
5.5. La imagen y su información almacenada en la cuenta. 66
5.6. Transacciones en las que está involucrada la imagen padre. 67
5.7. La imagen padre y su información almacenada en la cuenta. 68
5.8. Transacciones en las que está involucrada la imagen original. 68
5.9. Mensajes generados durante la transacción de modificación de permisos. 69
5.10. La imagen original y su información almacenada en la cuenta. 69
5.11. Linaje electrónico completo. 70

Índice de tablas

2.1. Contenido de una cuenta de Solana 21

4.1. Operaciones de imágenes y las cuentas involucradas 48
4.2. Operaciones de imágenes y datos de instrucción relacionados 50
4.3. Número de operación y operación relacionada 51

Capítulo 1

Introducción

Para 2021, el número aproximado de archivos digitales en el mundo era de 15 trillones [1], la mayoría de estos archivos residen en redes sociales o en alguna plataforma de nube. Estos archivos en su mayoría tienen un valor para cada uno de sus propietarios, de ahí que podamos considerarlos activos digitales. Al estar alojados en el Internet, el control que los propietarios tienen sobre ellos es casi inexistente. Estos activos pueden ser duplicados con facilidad o caer en manos de un tercero que no necesariamente es de confianza, surgiendo la necesidad de controlar y saber qué sucede con dichos activos digitales.

De esos 15 trillones de archivos digitales, 5 trillones son imágenes. Estos activos digitales pueden ser editados sin el consentimiento del dueño para volverse el elemento central de una campaña de desprestigio o de *fake news* [2, 3] o compartidos a terceros sin el consentimiento del dueño como sucedió en el caso que motivó la Ley Olimpia [4].

Dada la naturaleza abierta del internet, borrar un activo digital se vuelve sumamente retador rayando en lo imposible. Una solución alternativa es tener la historia de lo que ha sucedido con dicho activo como evidencia de las operaciones realizadas y de ahí tomar acciones que pueden ir desde lo personal hasta lo legal. Esta historia se denomina linaje electrónico. El "linaje electrónico" nos permite conocer el origen y los procesos por los que pasa un activo digital generando la propiedad de trazabilidad [5]. Este registro de procesos da la posibilidad al dueño del activo de saber qué es lo que ocurre con su propiedad. Para lograr esto, las aplicaciones deben generar documentación de los procesos que le están ocurriendo a los activos digitales y almacenarlos de forma segura para que no sean alterados y sean confiables.

La primeras propuestas para crear y almacenar el linaje electrónico involucran un sistema centralizado donde se encuentran todos los registros de estos procesos [5]. Esta situación obliga a tener un tercero de confianza sin garantía alguna de su comportamiento. Para resolver esta problemática, se propone el uso de esquemas distribuidos y descentralizados, una opción es Blockchain. Blockchain es una base de datos distribuida cuya tecnología engloba el uso de criptografía, algoritmos de consenso y modelos económicos. Blockchain combina redes punto a punto y algoritmos de consenso distribuidos para resolver problemas de sincronización que aparecen en bases de datos

distribuidas tradicionales [6].

El uso de Blockchain garantiza el no repudio e integridad de la información ya que genera un registro inalterable de transacciones mediante el uso de funcionas hash criptográficas y firma digitales. También ofrece transparencia y trazabilidad y que cualquiera con acceso a la Blockchain pueda realizar consultas de transacciones y conocer el origen de dichas transacciones.

Sin embargo, las Blockchain públicas más usadas son lentas y consumen altas cantidades de electricidad, además de no ser eficientes en el almacenamiento de archivos [6]. Esto es consecuencia del tipo de protocolo de consenso usado.

Por esta razón, la comunidad de Blockchain ha desarrollado nuevas propuestas que pretenden mejorar el consumo eléctrico y el tiempo de transacciones mediante el uso de protocolos de consenso más eficientes. Entre las propuestas más conocidas se tienen Blockchains como Cardano, Polkadot, o Algorand, las cuales usan el protocolo de consenso Proof of Stake donde la cantidad de tokens que uno posee es el factor que decide quien crea el siguiente bloque en lugar del poder de cómputo que uno posee, como sucede en Bitcoin.

En vista de esto, se propone el uso de la Blockchain llamada Solana para almacenar el linaje electrónico de imágenes, es decir, el historial de las operaciones que se realizarán sobre ellas. Esta Blockchain utiliza Proof of Stake en combinación con el protocolo denominado Proof of History (Prueba de Historia) descrito en el trabajo de Yakovenko [7]. Para el caso del almacenamiento de imágenes, se propone el uso de otro protocolo diseñado para este fin llamado IPFS (Inter Planetary File System) [8].

Como consecuencia, se obtendrá una solución distribuida, eficiente y confiable debido a las propiedades ofrecidas por estas tecnologías.

Esto se formalizará en un marco de referencia que permitirá verificar su correcto funcionamiento, además de su uso en otros activos digitales que no sean necesariamente imágenes. Si bien este marco de referencia puede aplicarse en cualquier Blockchain, se presenta un prototipo en la Blockchain de Solana, para verificar el correcto funcionamiento del marco de referencia. Finalmente, se analiza el linaje generado mostrando que es posible controlar y saber qué sucede con las imágenes que se procesan en este prototipo.

Este trabajo se divide en seis capítulos, siendo el primero esta introducción. El Capítulo 2 comprende el marco teórico que abarca todos los conceptos utilizados para la realización de este trabajo. En el Capítulo 3 se describe a detalle el marco de referencia propuesto para, posteriormente, en el Capítulo 4 detallar su implementación. En el Capítulo 5 se muestran y analizan los resultados obtenidos y finalmente, en el Capítulo 6 se presentan las conclusiones y trabajo futuro. Adicionalmente, se incluye un anexo con el código desarrollado para la implementación de este trabajo.

Capítulo 2

Marco teórico

2.1. Conceptos criptográficos de interés

En esta sección del capítulo se presentan un conjunto de conceptos criptográficos que son usados en los capítulos y secciones posteriores.

2.1.1. Cifrado

El cifrado busca hacer que la información sea incomprensible para asegurar la confidencialidad de la misma, esto se logra a partir de un algoritmo de cifrado y de un valor secreto llamado llave [9]. Un algoritmo de cifrado consiste en dos funciones, una de cifrado y una de descifrado. La función de cifrado E convierte un mensaje en texto claro m a un texto cifrado c haciendo uso de una llave e de tal que $c = E(e, m)$, mientras que la función de descifrado D convierte el texto cifrado c en un texto claro m utilizando una llave d tal que $m = D(d, c)$ [10]. Estas dos funciones pueden observarse en la figura 2.1. Además, en la figura 2.2 se muestra la clasificación de los algoritmos de cifrado, dividiéndose estos en dos grupos: los simétricos y los asimétricos.

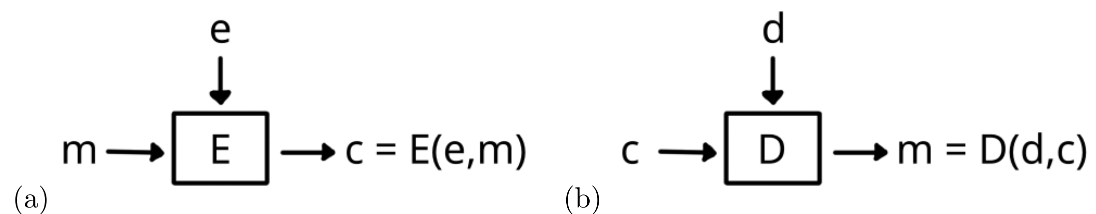

Figura 2.1: (a) Algoritmo de cifrado y (b) Algoritmo de descifrado (modificado de [9]).

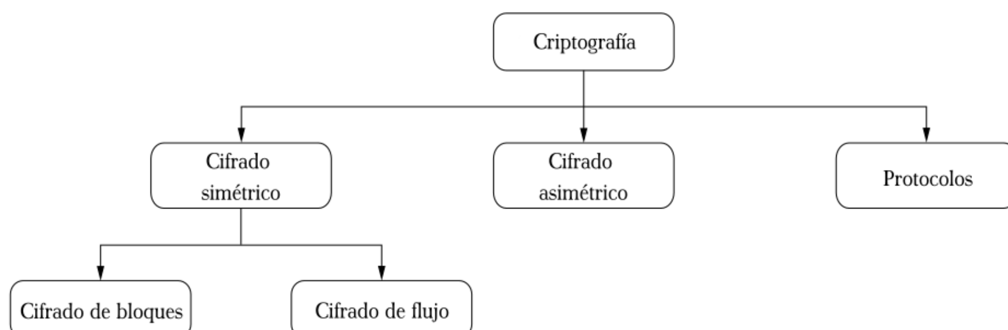

Figura 2.2: Tipos de cifrado que existen dentro de la Criptografía [11].

2.1.1.1. Simétrico

Un algoritmo de cifrado es simétrico si para las funciones E y D es computacionalmente sencillo determinar la llave d únicamente conociendo la llave e y determinar e a partir de d. En la mayoría de los algoritmos simétricos $e = d$ por lo que solo se utiliza una llave [10].

Este tipo de algoritmos, a su vez, se dividen en dos tipos: cifrado de bloques y cifrado de flujo. El cifrado de bloques (figura 2.3 a) es un tipo de cifrado que divide el mensaje en texto claro en cadenas (bloques) de tamaño fijo t para después cifrar los bloques con la misma llave, un bloque a la vez. El cifrado de flujo (figura 2.3 b) es un caso especial del cifrado de bloques donde el tamaño del bloque es igual a 1, en este caso se cifran los bits de forma individual al aplicar la operación XOR entre un bit del texto claro y un bit de un flujo de claves generado a partir de la llave.

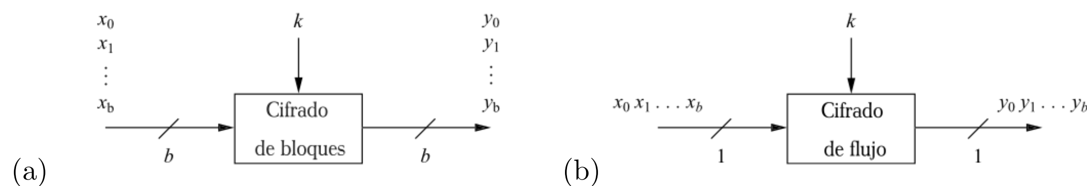

Figura 2.3: (a) Cifrado de bloques y (b) Cifrado de flujo [11].

2.1.1.2. Asimétrico

También conocido como cifrado de llave pública, en este caso para el par de funciones E y D se tiene la propiedad de que conociendo E es computacionalmente inviable encontrar el mensaje en texto claro m tal que $c = E(e, m)$ dado el texto cifrado c, lo que quiere decir que conociendo la llave de cifrado e es inviable determinar la llave de descifrado d [10].

E es una función trampa de un solo sentido, una función fácil de calcular pero que presenta dificultad en el calculo de su inversa D a menos que se conozca un valor (de ahí el nombre trampa) que permita un cálculo sencillo de la inversa. En este caso la llave d funciona como la trampa de la función, proporcionando la información necesaria para poder calcular de manera sencilla la función D y obtener el mensaje $m = D(d, c)$.

De este modo se ve la diferencia entre el cifrado simétrico, ya que aquí la entidad que utilice cifrado asimétrico contará con dos llaves: una para cifrar que es de dominio público (llave pública) y una llave para descifrar que mantiene en secreto (llave privada). La figura 2.4 muestra cómo sería este tipo de cifrado y descifrado, en donde Alice cifra un mensaje m con la llave pública e de Bob y él puede descifrarlo utilizando su llave privada d. Esto es posible porque la llave e es inversa de la llave d.

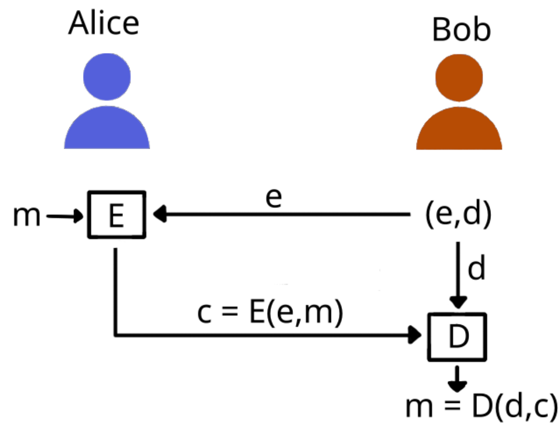

Figura 2.4: Proceso de cifrado y descifrado asimétrico.

2.1.2. Función hash criptográfica

Es una función computacionalmente eficiente que mapea cadenas binarias de longitud arbitraria a cadenas binarias de una longitud fija llamadas valores hash [10]. La figura 2.5 muestra el concepto de una función hash, para una entrada x la función hash h entrega una salida $y = h(x)$. Para una cadena binaria en particular, dicho valor hash "puede verse como la huella digital de la misma" [11].

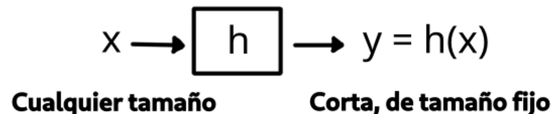

Figura 2.5: Función hash (modifcado de [9]).

Paar and Pelzl [11] mencionan los requerimientos deseados para las funciones hash criptográficas, dichos requerimientos son prácticos y de seguridad. A continuación se enumeran dichos requerimientos.

Requerimientos prácticos

- Computacionalmente eficiente. Aunque el tamaño del mensaje sea muy grande el valor hash debe ser rápido de calcular.
- Salida con tamaño fijo. Sin importar el tamaño de la entrada el valor hash siempre tendrá el mismo tamaño.
- Sensible a los bits de entrada. Cualquier cambio en la entrada arrojará un valor hash diferente.

Requerimientos de seguridad

- Unidireccionalidad. Significa que la función es de un solo sentido: dado un valor hash de salida y es computacionalmente inviable obtener la entrada x de tal forma que $y = h(x)$
- Resistente a colisiones. Para dos entradas distintas x_1 y x_2, para la función hash es improbable arrojar el mismo valor hash $h(x_1) = h(x_2)$, es decir, el generar dos entradas diferentes con valores hash iguales debe ser computacionalmente inviable.

La seguridad que proporcionan este tipo de funciones es que garantizan la integridad de la información al ofrecer evidencia de que dicha información no fue modificada. Si una función hash es segura (cumple con las propiedades mencionadas) entonces dos piezas de información diferentes tendrán valores hash diferentes con muy alta probabilidad, ya que cualquier cambio que sufra la información se verá reflejado en el valor hash [9].

2.1.2.1. Árbol de Merkle

Una aplicación de las funciones hash criptográficas es el Árbol de Merkle, el cual es una estructura de datos ideada por Merkle [12] para almacenar y autenticar firmas de manera eficiente para poder emplear el esquema de firma de Lamport-Diffie. Un árbol de Merkle puede ser utilizado para autenticar con una sola firma un conjunto de datos y ayudar a que la verificación de un solo dato se realice de manera independiente de los demás [13].

Para un conjunto de datos, el árbol de Merkle correspondiente se crea calculando el hash de cada uno de los datos, dichos hashes serán las hojas del árbol. Posteriormente cada nodo padre corresponderá al valor hash de la concatenación de los valores hash de ambos nodos hijos (en el caso de un árbol binario, derecho e izquierdo). Este proceso se repetirá hasta formar el nodo raíz del árbol, el cual es el valor hash que depende de todos los valores calculados anteriormente.

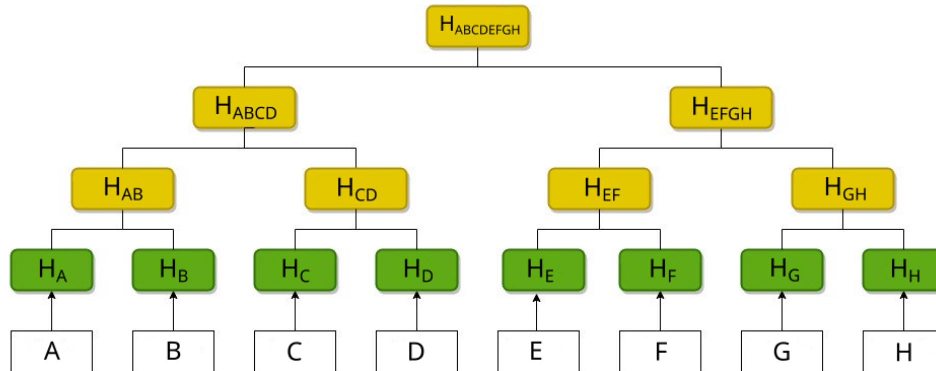

Figura 2.6: Árbol de Merkle (modificado de [14]).

La figura 2.6 muestra un árbol de Merkle. Se tiene el conjunto de datos A, B, C, D, E, F, G, cada hoja corresponderá al valor hash de cada dato dando como resultado las hojas H_A, H_B, etc. Para el primer par de hojas su nodo padre será el valor hash $H_{AB} = H_A \mid H_B$ donde $\mid$ representa la operación de concatenación. Esto será igual para todos los nodos hasta llegar a la raíz que corresponde al valor hash $H_{ABCDEFGH}$.

Se puede detectar una alteración de la información a través del árbol de Merkle de un conjunto de datos ya que cualquier modificación en un dato modificará la raíz del árbol de Merkle. Además, para verificar si cierta información forma parte del conjunto de datos representado por el árbol basta con tener la rama que va desde la raíz hasta la hoja con todos los hashes generados en el camino (figura 2.7) e ir calculando y verificando cada uno de los valores hash. En este caso, si se quiere verificar que el valor E es parte del árbol basta con tener los hashes H_E, H_F, H_{EF}, H_{GH}, H_{ABCD}, H_{EFGH} y la raíz del árbol $H_{ABCDEFGH}$.

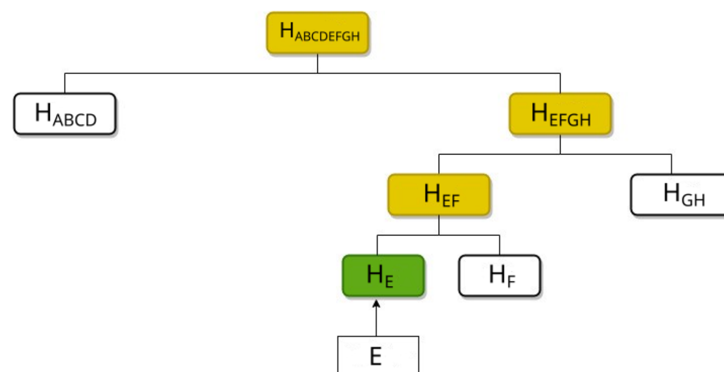

Figura 2.7: Árbol de Merkle (modificado de [14]).

Un árbol de Merkle reduce la cantidad de hashes calculados necesarios para asegurar que la información no fue manipulada, sirviendo como una huella digital del conjunto

de datos almacenado [14].

2.1.3. Firma digital

La firma digital es una herramienta fundamental para ofrecer autenticación, autorización y no-repudio de una pieza de información, brindando un medio para ligar la identidad de una entidad a dicha pieza [10]. Esto ocurre para casos en los que una entidad trate de negar una acción o alterar información que concierne a otras dos personas.

El proceso de firma consiste en pasar de un mensaje m a una firma s mediante una transformación S realizada por la entidad que firma y que debe permanecer secreta. Dicha firma puede ser verificada mediante una transformación V que es públicamente conocida. Generalmente estas transformaciones son algoritmos públicos, cada algoritmo está determinado por una llave por lo que el algoritmo de firma tiene una llave k que es secreta y el algoritmo de verificación tiene una llave l que es pública [10].

Esto último puede realizarse mediante la criptografía de llave pública donde todo usuario cuenta con un par de llaves, una pública y una privada. En el caso de firma digital, la entidad que firma utiliza su llave privada mientras que la entidad que recibe la firma utiliza la llave pública de la entidad que firma para verificar dicha firma [11].

Los pasos para firmar digitalmente y verificar dicha firma se mencionan a continuación:

Firma

1. Calcular la firma s para el mensaje m
2. Transmitir el par (m, s)
3. Transmitir la llave privada

Verificación

1. Calcular la verificación u para la firma s
2. Aceptar la firma si $u = \text{true}$ o rechazarla si $u = \text{false}$

En la figura 2.8 pueden observarse los pasos. El calculo de la firma se realiza obteniendo como entradas tanto el mensaje como la llave privada de Bob, la última es necesaria ya que la firma depende de esta llave. También se aclara que es necesario adjuntar la firma al mensaje y enviarlos juntos ya que "Una firma digital sin el mensaje es equivalente a una firma en un papel sin el contrato" [11] con lo cual se pierde el contexto de la firma. Por último para verificar la firma de Bob, Alice ingresa el mensaje, la firma y la llave pública de Bob.

2.2. Linaje electrónico

El linaje o *provenance* puede ser definido como la derivación desde un origen particular hasta un estado específico de un elemento [5]. Un ejemplo común en el que se

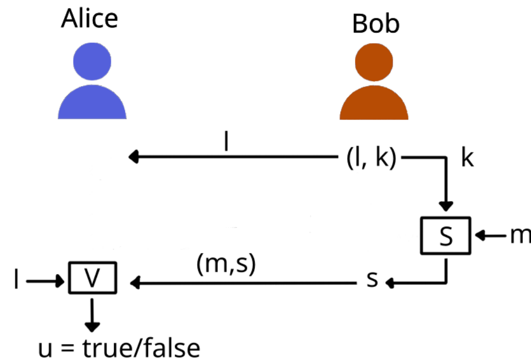

Figura 2.8: Proceso de firma digital.

observa el linaje es en el arte donde se tiene un registro de todos los dueños que ha tenido una determinada obra de arte o incluso podrían registrarse las restauraciones o procesos por los que esta obra ha pasado. Esta idea puede aplicarse al mundo digital dando origen al **linaje electrónico** que no es más que el proceso que conduce a un dato [15] y el registro de ese proceso. El linaje electrónico apoya la integridad de la información y de los procesos documentando las entidades, sistemas y procesos que operan y contribuyen a los datos de interés, sirviendo como un registro histórico inalterable de la duración de los datos y sus orígenes [16]. Sus usos son diversos incluyendo el evaluar la calidad de la información, atribuir el origen de un resultado computacional, reproducir ejecuciones previas de aplicaciones o como evidencia en auditorías electrónicas como lo mencionan Aldeco-Pérez y Chávez [15].

Si se desea tener el linaje electrónico de algún proceso computacional es necesario registrar cada proceso que describa lo que ocurrió en tiempo de ejecución (lo que se denomina documentación) y cualquier tipo de aplicación debe generar dicha documentación para, posteriormente, almacenarla en un sistema dedicado únicamente al linaje que proporciona un almacenamiento seguro y persistente a largo plazo. Una vez almacenada la información, ésta puede ser extraída mediante consultas al sistema para ser analizada y, con el paso del tiempo, el almacenamiento del linaje tal vez requiera de mantenimiento [5].

Este registro está compuesto por un conjunto de aseveraciones creadas por los servicios involucrados en el proceso. En este contexto, el construir el linaje electrónico de imágenes permitirá que los dueños de las mismas conozcan los procesos por lo que dichas imágenes pasan teniendo la seguridad de saber qué ocurre con su propiedad y poder tomar acciones en caso de que algo no les parezca. Generalmente, las soluciones para preservar y asegurar el linaje requiere de entidades centralizadas por lo que hay que confiar en que dicha información estará segura bajo el cuidado de dichas entidades.

2.3. Blockchain

Blockchain es una base de datos distribuida y descentralizada. Esta tecnología integra técnicas como criptografía, algoritmos, modelos económicos y matemáticas dónde se combinan redes punto a punto y algoritmos de consenso distribuido para resolver problemas de sincronización que presentan las bases de datos distribuidas tradicionales [17].

Esta base de datos distribuida puede verse como una lista creciente de registros llamados bloques que se unen entre sí mediante técnicas criptográficas, específicamente las funciones hash. A través de dichas técnicas y de la descentralización, se puede crear un registro histórico inalterable y transparente para activos digitales [18].

Blockchain se basa en tres pilares fundamentales:

- Descentralización
- Transparencia
- Inmutabilidad

La descentralización significa que nadie tiene el poder completo sobre la información almacenada en la Blockchain. Cada miembro de la red almacena por su cuenta esta información y cualquier modificación debe ser aceptada después de llegar a un acuerdo entre todos. En este caso no hay una autoridad central y sus funciones se dispersan entre todas las entidades que forman parte de la red. Respecto a la transparencia, cualquiera que inspeccione una Blockchain pública o forme parte de una Blockchain privada puede consultar las transacciones y todos los detalles de ésta. La inmutabilidad busca asegurar que una vez que un bloque se añade a la Blockchain, sea casi imposible alterar las transacciones dentro de éste debido a las funciones hash criptográficas que se utilizan. Esto resulta benéfico cuando se decida auditar información ya que el emisor y receptor pueden confiar en que la información no ha sido alterada [18].

A pesar de que esta idea surgió con la intención de ser utilizada como libro contable para la criptomoneda Bitcoin, Blockchain puede ser aplicada en diferentes campos [19]. Mattila [20] menciona algunos campos en los que está tecnología puede ser utilizada como, por ejemplo, en registros de propiedad, micropagos, manejo de identidad y reputación en línea, registros para cadenas de suministros e información centrada en el producto, internet de las cosas (IoT), sistemas de votación y organizaciones autónomas descentralizadas.

Los bloques con los que se construye una Blockchain contienen todas las transacciones de la red que ocurren en un determinado espacio de tiempo. Una transacción de Blockchain puede considerarse como un registro de datos estáticos públicos que muestra la ejecución de alguna instrucción de programación. En el caso de Bitcoin, una transacción es la transferencia de activos entre direcciones [21]. Siendo más específicos, una transferencia entre una cuenta emisora y una o más cuentas receptoras, aunque podría verse a una transacción como cualquier tipo de proceso que se quiera almacenar

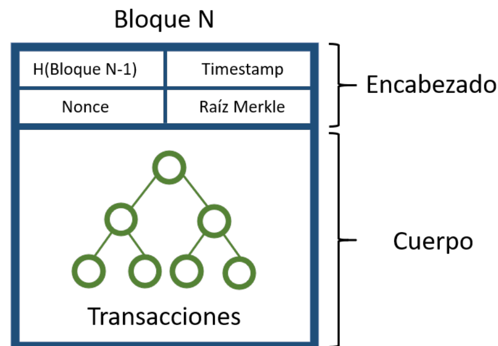

Figura 2.9: Estructura de bloque

en la Blockchain. Explorando más en los bloques, un bloque se divide en dos partes: el encabezado del bloque y el cuerpo del mismo, como se muestra en la Figura 2.9. En el caso de Bitcoin se tiene que dentro del cuerpo del bloque se encuentra la lista de transacciones e información de estas transacciones como el tiempo en el que ocurren, el valor o motivo de la transacción y la información sobre quienes participan en dicha transacción en forma de llaves públicas. Cada transacción es firmada digitalmente por parte del emisor [18]. Las transacciones se almacenan en el bloque haciendo uso de un árbol de Merkle (como se explicó en la Sección 2.1.2.1). Eso se muestra en la Figura 2.10.

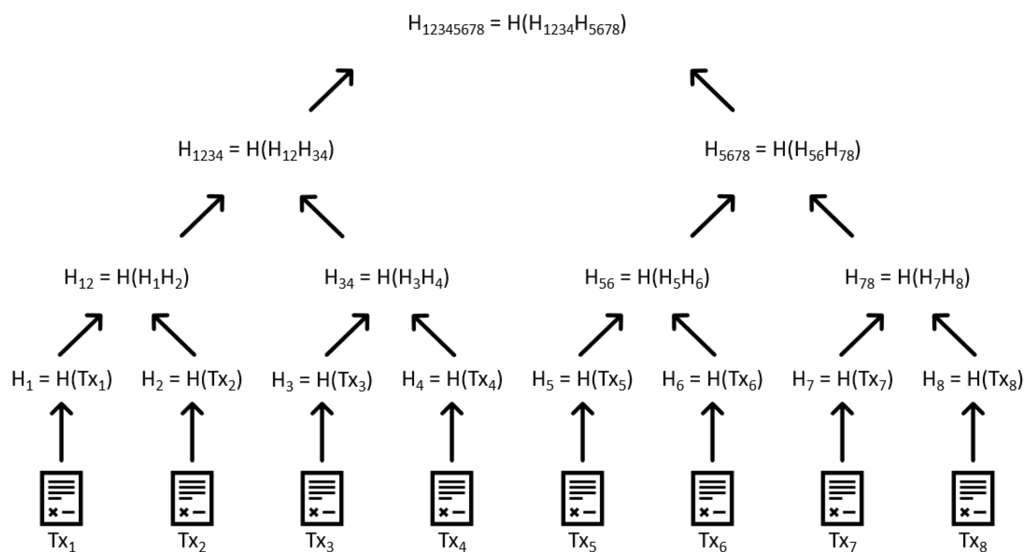

Figura 2.10: Árbol de Merkle con las transacciones de un bloque.

En el encabezado del bloque, por otro lado, se tiene la estampa de tiempo en la que se generó el bloque, un *nonce*[1], la raíz del árbol de Merkle y una referencia al bloque anterior, específicamente el hash del bloque anterior. Si bien la descripción anterior del contenido de un bloque fue tomada de Bitcoin, el resto Blockchains muestra un comportamiento similar.

Para que una Blockchain pública funcione, los nodos presentes en la red crearán bloques o validarán las transacciones. De manera general, para que un bloque sea generado y añadido en la Blockchain (figura 2.11) primero debe ocurrir una transacción entre dos entidades (los usuarios en color rojo) y después esa transacción debe ser validada para ser añadida al bloque [18]. Cuando ocurre una transacción, el nodo emisor registra la información y la firma digitalmente antes de propagarla a través de la red, posteriormente otros nodos (en color azul) recibirán la información y verificarán la autenticidad de ésta mediante la firma digital para después almacenarla en el bloque. Una vez generado el bloque se ejecuta un algoritmo de consenso para que se valide el bloque y este algoritmo de consenso utilizado por la red almacenará la información haciendo válido el bloque al añadirlo a la Blockchain y todos los nodos en la red admitirán este bloque en su registro de la Blockchain y extenderán la cadena [6]. Lo anterior es una explicación general pero cada Blockchain implementa un protocolo de consenso específico que depende del objetivo de cada red.

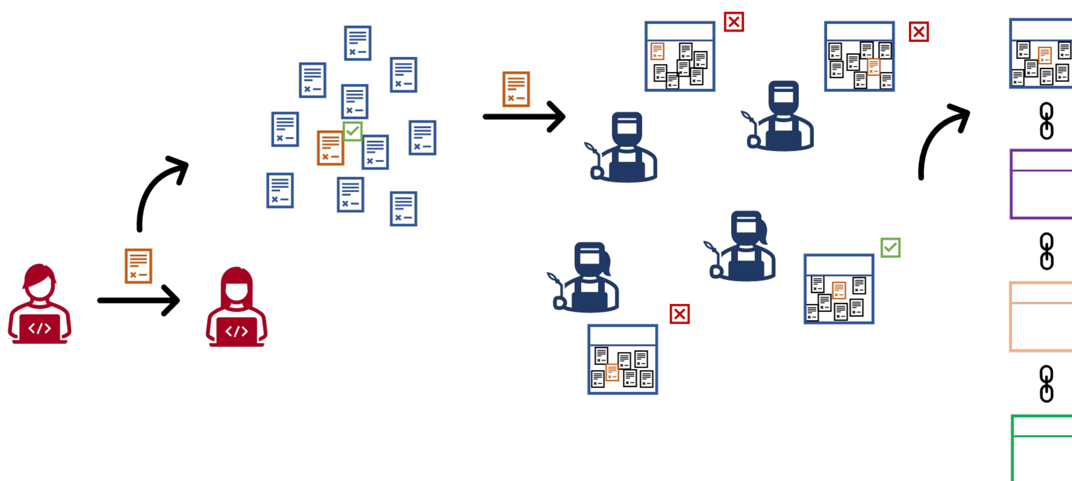

Figura 2.11: Funcionamiento Blockchain.

2.3.1. Tipos de Blockchain

Existen diferentes tipos de Blockchain que, dependiendo del control que se tenga para la participación en la red, pueden ser públicas, privadas o de tipo consorcio.

[1] Acrónimo de “number used only once”, un valor aleatorio que se utiliza una única vez.

Las Blockchains públicas son una red abierta para todos, donde está permitido que cualquier computadora se pueda unir a la red, validar transacciones y formar parte del proceso para llegar a un consenso ya sea viendo, leyendo o escribiendo información en un bloque y añadiéndolo a la Blockchain [18]. Cabe señalar que el poder de voto de algún usuario es generalmente proporcional a su posesión de recursos de la red como lo son: poder computacional, riqueza de tokens, espacio de almacenamiento, entre otros [22]. Este tipo de Blockchains son descentralizadas y libres de una autoridad central por lo que el control pertenece a la red y con lo cual se requiere la máxima seguridad para impedir que alguna organización o individuo tome el control de la misma, entre mayor distribución de la red exista, mayor es su dificultad para realizar un ataque [23].

Las Blockchains privadas son cerradas a unos cuantos y no permiten que cualquier computadora pueda realizar las verificaciones de la red. En este caso regresa la figura del intermediario donde, comúnmente, una empresa u organización es la autoridad central de la red y se encarga de verificar y validar las transacciones hechas por los nodos de la red. El hecho de tener una autoridad central brinda un nivel de velocidad y eficiencia más alto a la red sacrificando la seguridad descentralizada de una Blockchain pública ya que la seguridad corre por parte de la empresa que brinda el servicio [6].

Por último, las Blockchains de tipo consorcio están en un punto medio entre las públicas y las privadas. Aquí, se tiene un conjunto de nodos que pueden verificar y validar transacciones o bloques después de un proceso de registro y verificación de identidad, esto en lugar de permitir que cualquier individuo con una conexión a internet o una sola entidad tengan el control. Este tipo de Blockchains brindan las ventajas de velocidad y eficiencia asociadas a las Blockchains privadas sin concentrar todo en una sola organización [6, 18].

Dependiendo del tipo de Blockchain que se utilice se cumplen diferentes atributos deseados en una Blockchain.

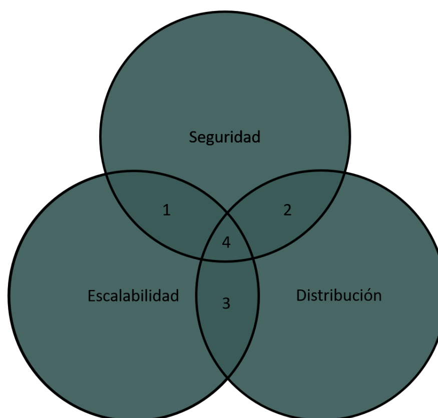

Figura 2.12: Relación entre las 3 propiedades principales de Blockchain [22].

Blockchain, al ser una base de datos distribuida hace que toda la información esté repartida entre todos los nodos de la red y, por lo tanto, exista **distribución** de los da-

tos. Por otro lado, el uso de técnicas criptográficas como las funciones hash y las firmas digitales proporcionan **seguridad** y ayudan a que ningún grupo o entidad modifique la información para su beneficio. Finalmente, la **escalabilidad** está relacionada con la habilidad de la Blockchain para mantener las operaciones sin presentar aumento en el tiempo de procesamiento incluso si la red aumenta su tamaño [18].

En la figura 2.12 se pueden observar las diferentes configuraciones que se pueden tener con los 3 atributos antes mencionados. El número uno se refiere a una Blockchain privada o de consorcio donde la configuración está ideada de forma que se aumente la velocidad de transacciones para mayor escalabilidad y mantener la seguridad. Esto se realiza sacrificando la distribución ya que se centraliza el control de la red. La opción dos distribuye las tareas aumentando la descentralización y mantiene la seguridad sacrificando la velocidad de transacciones lo cual dificultaría la escalabilidad, siendo esto común en las Blockchain públicas. La tercera opción es el caso no deseado debido a que se tiene descentralización y velocidad de transacciones pero sin seguridad, siendo la última una prioridad. Por último el escenario ideal es el número cuatro en el que se tienen los tres atributos, distribución, seguridad y escalabilidad [23].

2.3.2. Protocolos de consenso

Cuando se tiene un sistema centralizado, una autoridad central tiene todo el control y es la única entidad que toma decisiones y ejecuta acciones. Esto no es así cuando se tiene un sistema descentralizado porque dicha figura de autoridad desaparece y todos los miembros del sistema tienen el mismo peso. Debido a esto, las entidades tendrán que llegar a acuerdos entre sí que sean equitativos y benéficos para todos los participantes del sistema, todo esto nuevamente en un entorno en el que no existe confianza.

Esta problemática se puede mostrar con el ejemplo del problema de los generales bizantinos, el cual surge debido a un conjunto de generales que comandaban un porcentaje del ejército bizantino que rodeaba una ciudad. Algunos estaban a favor de la opción de atacar la ciudad mientras que otros generales preferían la opción de retirarse. Sin embargo, el ataque no tendría éxito si sólo una porción de los generales atacaban la ciudad por lo que un desafío mayor era conseguir un consenso para atacar o retirarse en un ambiente diseminado [6]. Resolver este problema es lo que abrió el camino para Blockchain y dio lugar a un conjunto de algoritmos para conseguir que todos los miembros de la red se pongan de acuerdo entre sí y lleguen a un consenso, en este caso del siguiente bloque a ser añadido a la blockchain.

La finalidad de un protocolo de consenso para una Blockchain es asegurar que todos los nodos participantes estén de acuerdo en una historia de transacciones común de la red y debe cumplir con los requerimientos de terminación, acuerdo, validez e integridad. **Terminación** se refiere a que, en cada nodo honesto (entendiendo por honesto que no tiene un comportamiento malintencionado y no propaga información incorrecta hacia la red), una transacción nueva debe ser rechazada o aceptada en la Blockchain dentro del contenido de un bloque. Existe **acuerdo** cuando cada nueva transacción y el bloque que la contiene deben ser aceptados o rechazados por todos los nodos honestos y para

cada bloque aceptado se debe asignar el mismo número de serie por cada nodo honesto. **Validez** significa que, si cada nodo recibe una misma transacción o bloque, debe ser aceptado en la Blockchain. Por último, hay **integridad** si, en cada nodo honesto, todas las transacciones aceptadas son consistentes entre sí y todos los los bloques son generados correctamente además de ser encadenados por hashes en orden cronológico [22].

Dependiendo también del tipo de Blockchain varía el tipo de protocolo de consenso a utilizar. En las Blockchains públicas se utilizan los protocolos de consenso basados en pruebas mientras que en Blockchains privadas se utilizan protocolos basados en votos. En un protocolo basado en prueba, los nodos que se unan a la red deben probar un cierto requerimiento definido por la red para tener el derecho de agregar un bloque a la Blockchain y así obtener una recompensa si su bloque es el que se añade. Existen diferentes protocolos dependiendo de la prueba solicitada. Algunos ejemplo son:

Proof of Work (PoW). Probar poder de ćomputo resolviendo un problema criptográfico díficil.

Proof of Stake (PoS). Probar que se tienen más recursos monetarios en la blockchain.

Proof of Elapsed Time (PoET). Probar que cierto tiempo ha pasado.

Proof of Space (PoSp). Probar que se tiene capacidad de almacenamiento.

Por otro lado, un protocolo de consenso basado en voto requiere que se elija un líder a través del voto de los participantes. Este líder será el encargado de agregar bloques y de intercambiar el resultado con la red antes de añadir el bloque a la Blockchain, si un nodo quiere añadir un bloque se debe hacer una revisión donde al menos x nodos accedan al bloque, donde x es un umbral definido previamente. Estos protocolos se clasifican en:

Tolerantes a fallos bizantinos. Si ciertos nodos se coluden, el sistema completo seguirá funcionando sin alterar la veracidad de la información almacenada.

Tolerantes a fallos. Si ciertos nodos fallan, el sistema completo seguirá funcionando de manera normal.

Debido a que este trabajo busca utilizar una Blockchain pública solo se profundizará en los protocolos de consenso basados en pruebas más usados que son PoW y PoS.

2.3.2.1. Proof of Work

Como se mencionó anteriormente, el protocolo de Proof of Work se basa en probar el poder de cómputo con el que un nodo cuenta para poder generar un bloque. Lo que busca este protocolo es desalentar los ataques de negación de servicio y spams que

pueden afectar a la red al hacer que se cumpla con un trabajo demandante. En PoW los nodos que se encargan de crear bloques se les conoce como mineros y el protocolo requiere que estos nodos prueben que el trabajo que han realizado los califique para recibir el derecho de agregar nuevos bloques a la Blockchain [18].

Aquí los mineros compiten entre sí utilizando su poder de cómputo calculando la solución de un desafío criptográfico. Este desafío involucra encontrar un *nonce*, un número que se encuentra en el encabezado del bloque y da como resultado un valor hash del bloque menor al valor definido por la red. Cada minero prueba diferentes *nonces* y calcula los valores hash del bloque, resolviendo el desafío por medio de la fuerza bruta. Una vez que un minero genera un hash que cumpla con las especificaciones, este transmite su bloque a los demás nodos de la red para que comprueben la validez del valor hash y, de ser válido, lo transmiten de manera que este bloque se propague a través de la red y la Blockchain esté actualizada para todos [24].

PoW es el protocolo más antiguo para Blockchain ya que fue utilizado para la red de Bitcoin, la primer criptomoneda. Esto hace que sea el protocolo con mayor tiempo de actividad y que haya superado ya varios desafíos a su seguridad y estabilidad, lo cual es una ventaja considerable. Otra ventaja que presenta este protocolo es que es muy sencillo verificar la solución del desafío criptográfico por las propiedades de la función hash a pesar de que calcular dicha solución sea muy complicado. Por último, y como se mencionó anteriormente, este protocolo desalienta a quienes quieran atacar mediante spams o negación de servicio debido a los costos computacionales. Por otro lado, también existen desventajas como el gasto de energía relacionado al consumo eléctrico por parte de todos los mineros que intentan resolver el desafío criptográfico, esto también va relacionado con una posible centralización ya que mineros que tengan acceso a un mayor poder de cómputo o tarifas eléctricas más baratas pueden tener una ventaja para generar bloques válidos en comparación de otros y monopolizar el control de la red [18].

2.3.2.2. Proof of Stake

Con motivo del gran gasto de energía eléctrica que representa el protocolo Proof of Work se tiene una alternativa más eficiente en la que, en lugar de que el nodo que resuelva primero el desafío criptográfico es el que puede añadir su bloque a la Blockchain, el nodo que tenga más monedas de la red apostadas en comparación de otros nodos es el que puede crear el próximo bloque. Aquí los nodos que generan los bloques se llaman forjadores y los que quieran forjar un bloque deben congelar sus fondos mediante una transacción especial a la red que funciona como depósito, si se apuesta una mayor cantidad de fondos entonces se tendrá una mayor posibilidad de ser elegido para crear el siguiente bloque.

Este protocolo se basa en que si las entidades tienen más fondos en la red, menor será su intención de atacar a la misma. El problema es que este tipo de selección es injusto porque quien tenga más fondos dominará a la red por lo que se tienen dos procesos de selección alternativos: uno que se basa en el valor hash creado y la cantidad de fondos

apostados donde se elige la mayor apuesta y el valor hash menor; y otro donde se toman en cuenta el tiempo que se tienen apostados los fondos y la cantidad apostada, una vez que se creó el bloque el tiempo de apuesta se vuelve cero y el forjador debe esperar un cierto tiempo antes de volver a generar un bloque [18].

Como se mencionó anteriormente, PoS tiene un consumo de electricidad reducido en comparación con PoW por lo que tampoco hay que preocuparse de estar en una zona con bajos costos de electricidad. Otro punto es que hace sencillo participar a más personas porque lo importante es tener fondos y no requieren de un hardware costoso para minar, lo cual también va relacionado con mayor descentralización en el sentido de que no permite concentrar el poder de la red para alguien con poder de cómputo o bajos costos de electricidad. En las desventajas es que la red puede verse influenciada por entidades que tengan una gran cantidad de recursos si no hay algo para evitarlo, también está el problema de que es un protocolo muy joven y todavía no se prueba su sostenibilidad a largo plazo [18].

2.3.2.3. Proof of History

Proof of History (PoH) no es un protocolo de consenso como tal pero ayuda a acelerar la sincronización entre los nodos de la red y permite que haya más velocidad en las transacciones de la Blockchain. PoH fue ideado por Yakovenko [7] para la Blockchain Solana, la cual busca cumplir con los tres atributos mencionados anteriormente: distribución, escalabilidad y seguridad. Solana combina su idea de PoH con el protocolo PoS para llegar a un acuerdo sobre los bloques.

Lo que PoH busca es tener un registro del tiempo en el que ocurren todas las transacciones y que todos los nodos en la red tengan este registro para que dichas transacciones puedan ser validadas de forma rápida, esto debido a que en los protocolos antes mencionados se debe esperar un cierto tiempo llamado “tiempo de bloque”para que las transacciones sean validadas. PoH proporciona una forma de verificar de forma criptográfica que el tiempo ha pasado entre dos eventos.

Para lograr esto, se utiliza una función hash. Dicha función, como se mencionó anteriormente en la sección 2.1.2, es una función de un solo sentido que tiene la propiedad que el resultado no puede ser predicho a partir de su entrada con lo cual es necesario ejecutarla para poder obtener dicho resultado. La clave es correr esta función una y otra vez iniciando con un valor aleatorio y donde el resultado será la entrada de la siguiente ejecución, esto puede verse más a detalle en la figura 2.13. La información puede ser concatenada junto con la entrada de la función hash y el número de veces que fue ejecutada la función para proporcionar una estampa de tiempo que pruebe que esa información fue generada antes que la función hash siguiente.

Debido a que la función hash debe ejecutarse forzosamente para obtener una salida, no hay forma de obtener un cierto valor hash en un cierto índice sin tener que ejecutar la función ese número de veces. Por ejemplo, observando nuevamete la figura 2.14 se nota que para obtener el hash 300 se tuvo que ejecutar anteriormente la función hash 300 veces, Con esto se asegura que cierto tiempo tuvo que pasar entre un hash y otro.

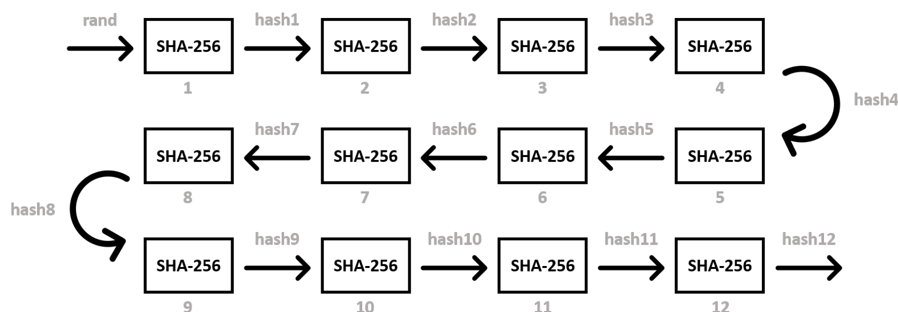

Figura 2.13: Secuencia de hashes que alimentan PoH.

Al incluir transacciones de la red también se puede asegurar que cierto tiempo paso entre una transacción y otra, en la figura 2.14 se puede ver que la transacción 1 tuvo que ocurrir antes de la transacción 2 ya que el hash 600 depende de los hashes anteriores incluido el hash 336 que depende de la transacción 1. La ventaja de esto es que cualquier nodo puede verificar de forma rápida toda la secuencia de hashes dando como resultado un proceso rápido de validación de la secuencia.

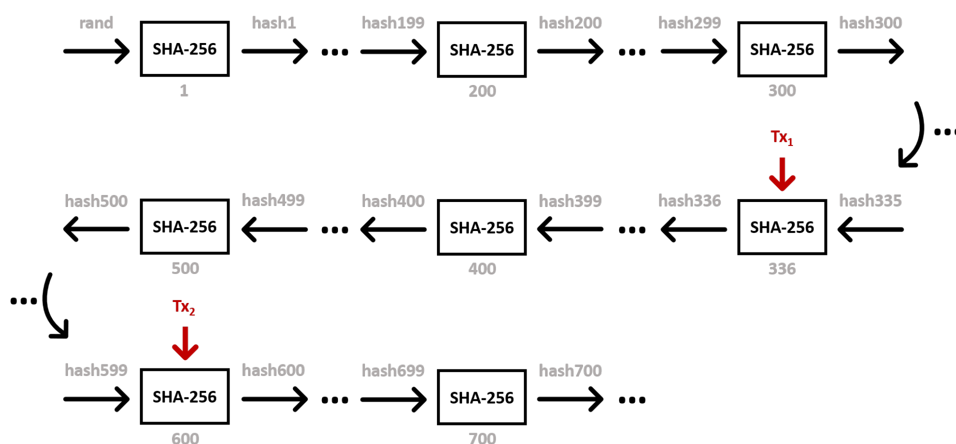

Figura 2.14: Secuencia de hashes con el registro de transacciones.

2.3.3. Contratos inteligentes

Un contrato inteligente es un programa de computadora que consiste en un conjunto de reglas que se ejecutan en la Bockchain, contiene todas las sentencias if/else y los acuerdos establecidos entre las partes involucradas en una negociación. Su integración con Blockchain brinda mucha flexibilidad para desarrollar diseñar e implementar problemas del mundo real al ejecutar código sin la intervención de terceros [25].

Debido a las características y propiedades de Blockchain, estos contratos son descentralizados y seguros. Sería imposible para cualquiera que deseará alterar el contrato modificar cualquier contenido del mismo ya que éste se encuentra asegurado criptográficamente y es inmutable [26].

Los contratos inteligentes pueden aplicarse en diversos escenarios como:

- Cadena se suministros.
- Internet de las cosas.
- Servicios médicos.
- Industria de derechos digitales.
- Aseguradoras.
- Sistema financiero.

2.4. Blockchain Solana

Solana es una Blockchain pública que pretende realizar un gran número de transacciones de manera eficiente y rápida. Para lograr eso utiliza una combinación de los protocolos antes mencionados, Proof of Stake y Proof of History. La utilización de Proof of History, como se mencionó anteriormente, permite la sincronización de la red de manera rápida.

Quién se encarga de mantener el registro inmutable de los eventos ocurridos en la red es el cluster, un conjunto de computadoras que trabajan juntas para verificar la salida de programas, rastrear la posesión de activos, entre otras [27]. En Solana existen diferentes tipos de clusters, cada con un propósito diferente. Estos se presentan en la siguiente lista.

- **devnet**. Este cluster sirve para que usuarios prueben sus aplicaciones, programas y equipos para ser validadores. Los tokens en este clusters no son reales, esto es, no tiene valor monetario.
- **testnet**. Este cluster tiene como fin ser el espacio para contribuyentes hagan pruebas de estrés a la red enfocándose en rendimiento, estabilidad y comportamiento de validadores.
- **mainnet**. Cluster en el que interactúan los usuarios realizando transacciones reales de Solana, sus token tienen valor monetario.

El token nativo de Solana recibe el nombre de SOL, el cual sirve para pagar por acciones que realicen en el cluster como pueden ser almacenar información, que un nodo ejecute un programa o que valide el resultado del mismo. A la fracción de un SOL se

le conoce como lamport[1] los cuales se utilizan en ciertas transacciones que no tengan un costo alto. Un lamport equivale a 0.000000001 SOL [27].

Para interactuar con el token SOL se requiere una billetera (o *wallet* por sus siglas en inglés), la cual almacena un par de llaves creadas usando el algoritmo asimétrico de curva elíptica ed25519, compuesto por una llave pública y una llave privada. La llave pública, como se mencionó anteriormente, puede compartirse con cualquier usuario de la Blockchain y funge como la dirección receptora de tokens. Cuando se quiera enviar una cierta cantidad de SOL a un usuario, o conocer cierta información de su wallet es necesario tener su llave pública. La llave privada se usa para firmar las transacciones o realizar algún cambio en la *wallet*. Esta llave debe permanecer privada y almacenada de manera segura, ya que quien la tenga, tendrá acceso a los tokens dentro de la correspondiente wallet [28].

La forma en la que se integra Proof of History con el protocolo de consenso Proof of Stake en Solana es la siguiente:

1. Se ejecuta un algoritmo de PoS donde se crea de forma pseudoaleatoria un horario de líderes que tendrá una cierta duración de tiempo, llamada época (*epoch*), donde los nodos del cluster con mayor cantidad de fondos apostada podrán participar más veces durante esta época.

2. De acuerdo al orden que dicte el horario de líderes cada líder tendrá un intervalo de tiempo, el cual se le llama *slot*, en el que este líder se dedicará a generar la secuencia de hashes y todas las transacciones que ocurran en ese tiempo serán añadidas a dicha secuencia, ejecutando así PoH.

3. Una vez que se cumpla cierto tiempo dentro del *slot*, el líder compartirá la secuencia de hashes firmada digitalmente por él mismo a los demás nodos que tendrán la tarea de verificar la firma digital del líder y validar las transacciones. Si la verificación es correcta, los validadores firmarán digitalmente la secuencia y esto contará como un voto. Después de obtener dos terceras partes de los votos la secuencia será válida y podrá integrarse a la Blockchain. Una vez que se cumpla el tiempo del *slot*, el siguiente nodo en la lista tomará el papel de líder y realizará el mismo proceso.

La forma en la que se propaga la información a través de la Blockchain de Solana es a traveś del protocolo *Turbine*. En lugar de que el líder en turno deba compartirle a cada nodo de la red el bloque correspondiente, este lo divide en pedazos y transmite dichos pedazos entre los nodos.

El cluster se divide en vecindarios que no son más que conjuntos de nodos, cada nodo se encarga de compartir cualquier información que reciba con los nodos de su vecindario y con un pequeño conjunto de nodos que están en otros vecindarios. Durante un *slot*, el líder distribuye los pedazo entre los nodos validadores que se encuentran en el primer vecindario. Cada validador comparte su información con su vecindario y

[1] Nombrado así en honor al investigador Leslie Lamport.

también la retransmite con algún nodo presente en otro vecindario, todo esto se repite entre todos los vecindarios hasta que cada nodo tenga el bloque completo. De esta forma se transmite la información de manera rápida y sin redundancia entre nodos [29].

A continuación se explican más a detalles 3 componentes principales de la Solana: cuentas, transacciones y programas.

2.4.1. Cuentas

Las cuentas son componentes esenciales dentro de la blockchain de Solana. Las cuentas son estructuras que se utilizan para almacenar un programa ejecutable o información que pueda ser utilizada entre transacciones. Toda cuenta en Solana tiene una dirección que corresponde a una llave púbica de 256 bits y un dueño que se identifica con la dirección de una cuenta de programa [30, 31]. La tabla 2.1 muestra la información completa que se encuentra dentro de una cuenta.

Tabla 2.1: Contenido de una cuenta de Solana.

Campo	Descripción
Lamports	El número de lamports que pertenecen a la cuenta.
Owner	El programa dueño de la cuenta.
Executable	Indica si la cuenta es ejecutable y, por lo tanto, se trata de un programa. En caso de no ser ejecutable la cuenta únicamente almacena información.
Data	La información almacenada por la cuenta en forma de un arreglo de bytes. Si la cuenta es ejecutable aquí es donde se almacena el programa.
Rent_epoch	La siguiente época en la que la cuenta deberá pagar renta.

Una cuenta ejecutable tiene un valor true en el campo *Executable* y representa un programa que puede ser ejecutado por la Blockchain. Las cuentas pueden ser editables o no, si una cuenta es editable esto quiere decir que esta cuenta sirve para almacenar información dentro del campo *Data*. Si por el contrario una cuenta no es editable, esta será de solo lectura y únicamente permitirá leer la información contenida en ella. Si se intenta modificar dicha información, el resultado será un error en ejecución.

Además de indicarse si la cuenta puede editarse, también se puede aclarar si la cuenta está firmando una transacción dentro de los metadatos de la misma transación. En caso de ser así, significa que el dueño de la llave privada asociada a la dirección (llave pública) de la cuenta autoriza la ejecución de dicha transacción.

Como se mencionó anteriormente, toda cuenta tiene un dueño, el cual es un programa. Únicamente el programa dueño de la cuenta puede modificar la información

que ésta almacena y retirar sus lamports. Si un programa no es dueño de la cuenta, únicamente puede añadir lamports y leer su información.

Almacenar información en cuentas tiene un costo, esto significa que hay que pagar una renta en lamports para mantener esta información en la Blockchain para futuras transacciones. De manera periódica se revisan las cuentas y se recolecta la renta. En caso de que una cuenta no tenga la cantidad necesaria de lamports o esta cantidad sea igual a cero, la cuenta será desasignada y la información se perderá.

En cada época se paga renta y el campo rent_epoch indica dicha época. Las cuentas pueden estar exentas de pagar renta si la cantidad de lamports que guardan es equivalente a 2 años de renta, esto permite que la cuenta persista.

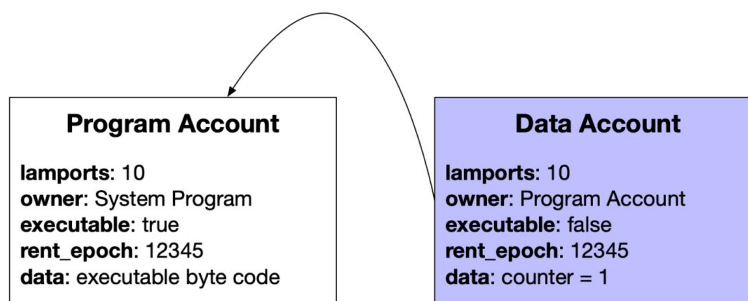

Figura 2.15: Ejemplo de cuentas en Solana una cuenta de programa y una cuenta que almacena información [31].

La figura 2.15 muestra un ejemplo de dos cuentas, una almacena el código de un programa contador y la otra es una cuenta que almacena al contador. Al momento de la ejecución del programa, este accederá a la variable contador almacenada en la otra cuenta y aumentará en uno al contador.

2.4.2. Transacciones

Los usuarios ejecutan programas al enviar una transacción a uno de los clusters de la Blockchain. La figura 2.16 muestra la anatomía de una transacción. Una transacción está compuesta por un arreglo de firmas, cada una es la firma digital del mensaje que sucede a dicho arreglo. Por otro lado, el mensaje contiene un encabezado seguido de un arreglo que contiene direcciones de cuentas, el hash de bloque más reciente y un arreglo de instrucciones.

El encabezado del mensaje se conforma de tres valores de 8 bits. El primero es el número de firmas requeridas en la transacción, el segundo el número de cuentas de solo lectura que requieren firmas y el tercer valor es el número de cuentas de solo lectura que no requieren firmas.

El arreglo de las direcciones de cuentas está compuesto primero por las cuentas que requieren firmas. De estas cuentas, las que requieren acceso de lectura-escritura van

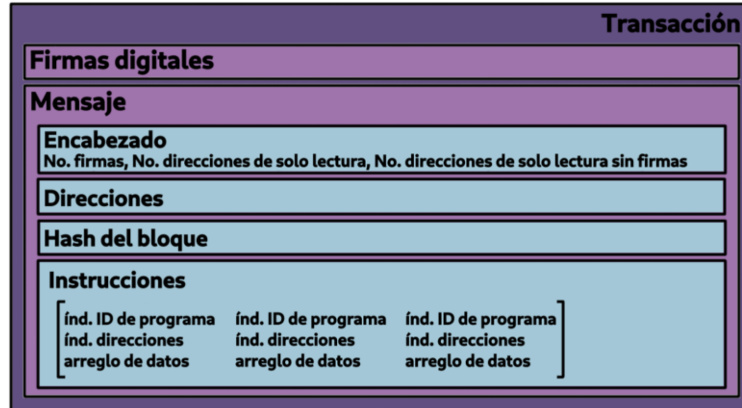

Figura 2.16: Anatomía de una transacción.

primero seguidas de las cuentas de solo lectura y al final del arreglo están las cuentas que no requieren firmas, del mismo modo las cuentas de lectura-escritura van antes que las cuentas de solo lectura. Cabe aclarar que las cuentas que requieren firmas se identifican como la llave pública de un par de llaves ed25519.

El hash del bloque es un valor hash de 32 bytes producto del algoritmo sha-256 que indica el momento donde el cliente observó por última vez la Blockchain. El uso de este hash permite darle un tiempo de vida a las transacciones además de evitar una posible duplicación de las mismas. Si el hash de bloque es demasiado viejo o si una transacción es idéntica a otra, entonces dicha transacción es rechazada.

Una transacción puede estar conformada por múltiples instrucciones por lo que el último arreglo mencionado puede tener uno o más elementos. Una instrucción es la unidad operacional básica en Solana y cada instrucción contiene el índice del identificador del programa que es llamado para ejecutar dicha instrucción, un arreglo de índices de cuentas y un arreglo de datos. Las instrucciones son ejecutadas de forma atómica y en orden, si falla una instrucción toda la transacción falla.

El índice del identificador de programa apunta a una dirección de cuenta en el arreglo de cuentas presente en el mensaje, dicha cuenta es la que almacena el programa a ejecutar. El programa puede ser uno nativo o uno en cadena (*on-chain* en inglés), los cuales se describirán en la siguiente subsección.

Los índices de cuentas también apuntan al mismo arreglo de cuentas solo que ahora las direcciones son de las cuentas involucradas en la instrucción, dando como resultado un subconjunto de las cuentas especificadas en la transacción que se pasarán al programa. Las cuentas utilizadas sirven para almacenar el estado del programa y también cumplen el papel de ser entradas o salidas del mismo.

Por último, los datos de instrucción consisten en un arreglo de bytes que se pasa al programa junto con las cuentas, dichos datos normalmente se encuentran serializados y pueden indicar el tipo de operaciones que se ejecutarán, al igual que contener información que esas operaciones requieran para su ejecución.

Toda esta información acerca de las transacciones fue tomada de [32] y [33].

2.4.3. Programas

Los programas en Solana no son más que código ejecutable que interpreta las instrucciones que se encuentran dentro de cada transacción en la Blockchain. Los programas no pueden guardar estado por lo que requieren interactuar con la información almacenada en cuentas. Como se mencionó anteriormente, los programas también se almacenan en cuentas que están marcadas como ejecutables [34].

Solana contiene ciertos programas nativos que se requieren para ejecutar un nodo validador, el par de programas que son de interés para este trabajo son el System Program y BPF Loader. El primero se encarga de crear nuevas cuentas, asignar datos de cuenta, asignar cuentas a programas, transferir lamports entre cuentas y pagar tarifas de transacción, mientras que el segundo es el responsable de implementar, actualizar y ejecutar programas en la Blockchain.

También existen los programas *on-chain* que son creados por usuarios para que cualquiera pueda ejecutar e interactuar con ellos en la Blockchain, es decir, se encuentran dentro de la Blockchain. Los programas en Solana son desarrollados comúnmente en el lenguaje de programación Rust y son conocidos como contratos inteligentes en otras Blockchains. Cabe aclarar que, como se busca que la explicación del marco de referencia presentado en esta tesis sea lo más general posible e independiente de la Blockchain utilizada, a partir de este punto se manejará únicamente el término contrato inteligente en lugar de programa.

2.5. InterPlanetary File System

El InterPlanetary File System (IPFS) es un sistema distribuido para almacenar y acceder a archivos, sitios web, aplicaciones e información. Siendo una red de almacenamiento punto a punto (*peer-to-peer*), el contenido que se deseé consultar puede ser accedido a través de pares ubicados en cualquier parte del mundo que pueden transmitir información, almacenarla o ambas.

En los sistemas centralizados, la comunicación ocurre entre un servidor que tiene cierta información guardada y el usuario que desea consultar esa información. Estas consultas se realizan mediante URLs que indican la localización de dicho contenido. En IPFS, la información es accedida mediante el contenido y no mediante la localización por lo que la conexión se da entre el usuario y cualquier equipo en el mundo que tenga almacenada la información deseada, a esto se le conoce como direccionamiento de contenido [35].

La figura 2.17 muestra la diferencia entre las URLs en sistemas centralizados y las que son utilizadas en IPFS. Puede observarse que en un URL normal se especifica el dominio en donde se encuentra el contenido al que se está accediendo mientras que en

Basado en ubicación → https://en.wikipedia.org/wiki/Aardvark

Basado en contenido → ipfs://bafybeiaysi4s6lnjev27ln5icwm6tueaw2vdykrtjkwiphwekaywqhcjze/wiki/Aardvark

Figura 2.17: Direccionamiento basado en la ubicación del archivo y basado en el contenido. Basado en [35]

IPFS se tiene una cadena de caracteres única poco comprensible, esta cadena recibe el nombre de identificador de contenido (`cid`).

Todo contenido en IPFS tiene un `cid` el cual permite que pueda ser encontrado y accedido dentro de IPFS. Dicho identificador es el valor hash de dicho contenido, lo cual permite que todo contenido tenga un hash único. Además, si el contenido es modificado entonces el valor hash o `cid` de este cambiará.

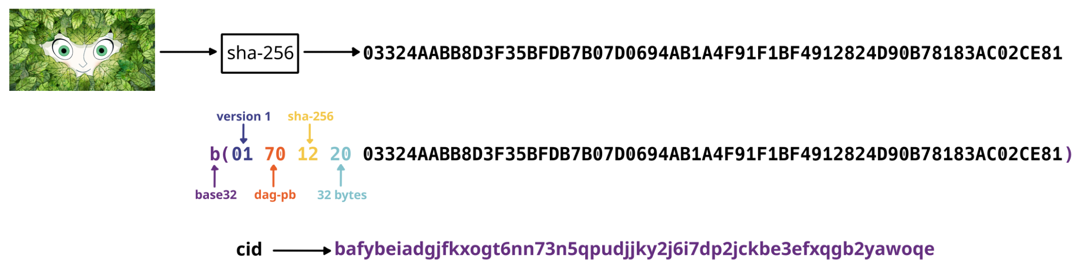

Figura 2.18: Anatomía del identificador de contenido.

La anatomía del `cid` se explica con detenimiento en [36] y se presenta en la figura 2.18. El componente principal del `cid` es el valor hash de la imagen de interés que viene acompañado de un conjunto de prefijos que aportan detalles sobre dicho valor hash y la información del contenido, dichos prefijos se explican a continuación.

- **Codificación del `cid`**. Es una letra que indica cuál fue la codificación utilizada para el cid, el valor por defecto es una **b**, corresponde a la codificación en *base32*.
- **Versión de `cid`**. Valor que muestra la versión del `cid` que se está utilizando, en este caso la versión utilizada es la 1 que corresponde al valor hexadecimal **0x0**.
- **Codificación empleada en los datos**. Valor que denota cómo está codificado el contenido, por defecto el codificador utilizado es dag-pb [1] que corresponde al valor hexadecimal **0x70**.
- **Función hash empleada**. Valor que indica el algoritmo utilizado para generar el hash del archivo, el valor por defecto es sha-256 cuyo valor en hexadecimal es **0x12**.

[1] Codificador que es el medio principal para codificar la información de un archivo haciendo uso de Protocol Buffers [37]

- **Largo del valor hash**. Valor que señala el largo (en bytes) del valor hash. Como la función hash por default es sha-256 entonces el largo de la cadena es de 256 bits que corresponde 32 bytes presentando así el valor hexadecimal **0x20**.

En [38] se muestran los diferentes tipos de codificación para el `cid` que existen, mientras que [39] contiene todos los valores posibles de codificadores para el contenido y de las opciones de funciones hash. Cabe aclarar que existe una versión antigua de `cid` (versión 0) donde solo se considera el valor hash junto con los prefijos que indican el largo del valor hash y la función hash utilizada. Dicho cid es codificado en base *base58btc* utilizado por bitcoin.

La limitante que presenta la versión 0 es que en caso de utilizarse otro codificador para el contenido no es posible distinguirlo. Además, como se tienen dos versiones de cid, es mejor utilizar la versión 1 que es más completa para poder distinguir entre ambas versiones. Utilizando los valores por defecto para los prefijos, siempre se tendrá a la cadena '`bafybei`' al inicio de los cid generados ya que siempre se tendrán los prefijos **b**, **0x01**, **0x70**, **0x12** y **0x20**.

IPFS utiliza un grafo acíclico dirigido de Merkle (DAG Direct Acyclic Graph por sus siglas en inglés) para representar los directorios y archivos. Esta DAG es similar a un árbol de Merkle con la diferencia de que el último es un árbol balanceado en donde cada nodo del árbol no puede tener más de un padre y debe existir una raíz [40].

La DAG de Merkle se construye mediante identificadores de contenido, donde el identificador de contenido funciona como arista del grafo mientras que los nodos están compuestos por la información codificada. Todo archivo es dividido (dependiendo de su tamaño) en bloques y cada bloque tendrá un cid por lo que el cid del archivo completo está compuesto por todos los cid de los bloques. El hecho de dividir el archivo en bloques permite que en caso de que haya archivos o directorios semejantes, estos compartan los bloques que son iguales, creando un manejo más eficiente de la información.

Figura 2.19: Construcción del cid de la imagen a partir de bloques.

En la figura 2.19 puede observarse la DAG de Merkle que corresponde a una imagen. Dicha imagen se divide en 17 bloques y cada bloque tiene un cid que, en conjunto, generan el cid de toda la imagen. A su vez, el cid del directorio se calcula tomando en cuenta todos los cid de los archivos que se encuentran dentro del directorio.

La DAG de Merkle para un directorio se puede observar en la figura 2.20, como se menciona en [41] se tiene un directorio "fotos" que contiene a su vez dos directorios "peces" y "gatos" y cada uno cuenta con imágenes. Las hojas corresponden a los archivos codificados y los `cid` de estos son las aristas de la gráfica, cada padre corresponde a una lista de `cid` de sus hijos hasta componer el nodo raíz del directorio.

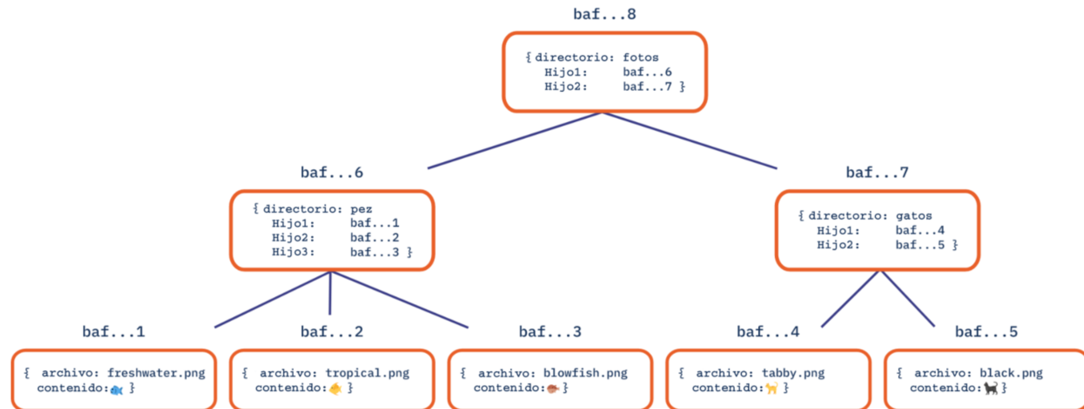

Figura 2.20: Construcción del cid de la imagen a partir de bloques [41].

Utilizar DAGs de Merkle permite que exista distribución de los archivos ya que, cualquiera que tenga la gráfica almacenada puede compartirla. También permite obtener nodos de la gráfica (o recursos) en paralelo ya que, al contener identificadores de contenido, se puede asegurar que el contenido es el correcto.

En caso de que se haga una modificación en algún directorio, reflejar dichos cambios es relativamente eficiente. Por ejemplo, si se desea eliminar el directorio de "peces" del DAG de la Figura 2.20 y añadir un directorio "perros", se creará una nueva DAG que puede observarse en color azul en la figura 2.21. El directorio "gatos" (en color verde) se mantendrá con el mismo cid y no habrá una modificación del mismo. Además puede observarse que ambas DAG comparten el directorio gatos por lo que no hay necesidad de duplicarlo. Esto ejemplifica otra ventaja de utilizar DAGs de Merkle, la deduplicación de datos.

Para acceder al contenido, IPFS utiliza una tabla hash distribuida (DHT Distributed Hash Table, por sus siglas en inglés), la cual de manera distribuida mapea llaves a valores. En este caso los identificadores de contenido sirven como llaves mientras que los valores son los pares que almacenan el contenido al que se desea acceder. A su vez, se cuenta con un segundo mapeo de identificadores únicos de pares a la dirección con la cual se puede conectar con dicho par.

Para encontrar a los pares que almacenan el contenido, se busca en la DHT utilizando el cid del contenido deseado. Posteriormente, cuando se tiene al par que tiene dicho contenido, se vuelve a realizar una búsqueda en la DHT para encontrar la ubicación del par. Con esto se puede observar que obtener el contenido deseado requiere de dos consultas a la DHT.

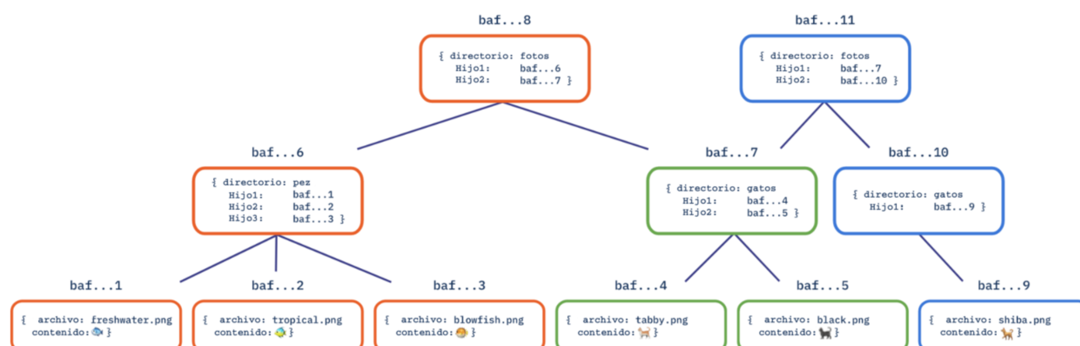

Figura 2.21: DAG de Merkle después de la modificación del directorio [41].

Una vez que se tiene la ubicación del par que almacena el contenido solicitado, se utiliza el módulo Bitswap que envía una lista con los bloques que conforman el contenido de interés y el par envía esos bloques de regreso. La ventaja de utilizar los identificadores de contenido es que se puede obtener el cid de cada bloque y comparar con los cid solicitados, verificando si se recibió la información correcta.

2.6. Trabajos relacionados

Todos los conceptos mencionados en las secciones anteriores entran en juego para generar una propuesta que ayude a los usuarios a tener control sobre su información y a poder conocer todo lo que llegue a ocurrir con la misma. Existen propuestas de almacenar el linaje electrónico usando Blockchain, Azaria et al. [42] proponen un esquema descentralizado para el manejo de registros médicos con el uso de la Blockchain Ethereum. Si bien esto ofrece la integridad del linaje al utilizar Blockchain, el uso de Ethereum representa un número bajo de transacciones procesadas y un costo de procesamiento alto, generando limitantes para la escalabilidad. Además de problemas de escalabilidad, el almacenamiento de los registros médicos presenta problemas ya que éste sería a través de los mismos proveedores de servicios médicos por lo que no se puede asegurar que estén libres de un uso indebido.

Tomando en cuenta lo anterior, Mani et al. [43] presentan propuestas en las que se utiliza la Blockchain permisionada Hyperledger para gestionar el linaje electrónico de los registros médicos y almacenar los archivos en IPFS. La ventaja de utilizar una Blockchain privada es que se pueden realizar un gran número de transacciones a bajo costo y evitar los problemas de escalabilidad, el uso de IPFS brinda el almacenamiento de manera descentralizada por lo que no hay que depender de un tercero que tenga control sobre los archivos.

Por otro lado, Sifah et al. [44], presentan el uso de Blockchain para el linaje electrónico de archivos en la nube. Este trabajo contempla también el uso de Ethereum con las problemáticas antes mencionadas. Una propuesta similar, que también hace uso de

Ethereum, es la de Abhishek et al. [45] donde los metadatos de las operaciones se almacenan en IPFS y el hash de los metadatos se procesa en la Blockchain. Como ambas propuestas tratan archivos almacenados en la nube, se tiene la desventaja de que el almacenamiento es centralizado y es necesario el confiar en esa entidad encargada.

Khatal et al. [46] proponen un marco de referencia para el linaje electrónico de archivos de texto con el uso de Blockchain y de IPFS. Esta propuesta brinda un almacenamiento descentralizado mediante IPFS con la ventajas que esto conlleva y el uso de Ethereum, con las mismas problemáticas tratadas anteriormente en el caso de los registros médicos.

Por último, enfocándose en imágenes, Igarashi et al. [47] y Sharma et al. [48] proponen un sistema de trazabilidad para fotografías y un marco de referencia para la publicación de imágenes con su linaje, respectivamente. Ambas propuestas se basan en la Blockchain Ethereum, presentando la misma problemática antes mencionada. Además, la primer propuesta se enfoca únicamente en fotografías tomadas con cámaras excluyendo otro tipo de imágenes, mientras que la segunda propuesta, a pesar de utilizar IPFS, solo se concentra en subir y compartir imágenes.

Capítulo 3

Descripción del marco de referencia

Como se mencionó anteriormente, el linaje electrónico debe ser almacenado en un sistema dedicado únicamente para este propósito, que proporcione seguridad y persistencia ya que contiene información sensible. El almacenamiento del linaje electrónico suele realizarse mediante soluciones centralizadas donde se confía en que la entidad que proporcione dicho almacenamiento asegure el cumplimiento de las características antes mencionadas.

El marco de referencia propuesto plantea que el linaje electrónico de las imágenes y las mismas imágenes sean almacenados en una Blockchain y en un sistema distribuido, respectivamente. Ésto proporcionará integridad al linaje, manteniendo segura dicha información y evitando que pueda ser modificada, además de que sea persistente a largo plazo. Por otro lado, el almacenar las imágenes de manera distribuida evitará depender de un tercero para que dichas imágenes estén seguras.

Se busca que este marco pueda aplicarse en cualquier tipo de Blockchain, sin importar el algoritmo de consenso utilizado o si se trata de una Blockchain pública o una privada. También podría utilizarse cualquier sistema distribuido para almacenar las imágenes. Sin embargo, existe una inclinación por utilizar una Blockchain pública buscando descentralizar el almacenamiento del linaje electrónico brindando más confianza a la información contenida en el mismo, además de que la misma Blockchain cuente con un protocolo de consenso eficiente y rápido pensando en términos de bajo consumo eléctrico y en caso de que exista un número elevado de transacciones y es por eso que se eligió utilizar la Blockchain de Solana.

El uso de IPFS para el almacenamiento de imágenes tiene la ventaja de ser descentalizado, seguro y evitar duplicación del contenido además de ser un sistema gratuito a comparación de otros sistemas como Filecoin o Arweave.

La arquitectura del sistema propuesto puede observarse en la figura 3.1. Esta figura muestra la manera en que un usuario se comunica con una interfaz o *front-end*[1] dentro de un navegador. Desde ahí, el usuario podrá solicitar el realizar funciones sobre las imágenes que desea compartir o descargar. A través de contratos inteligentes (mencionados en la subsección 2.3.3 y representados en la figura 3.1 como pequeños cuadros)

[1]Es la parte de un programa o dispositivo a la que un usuario puede acceder directamente. [49]

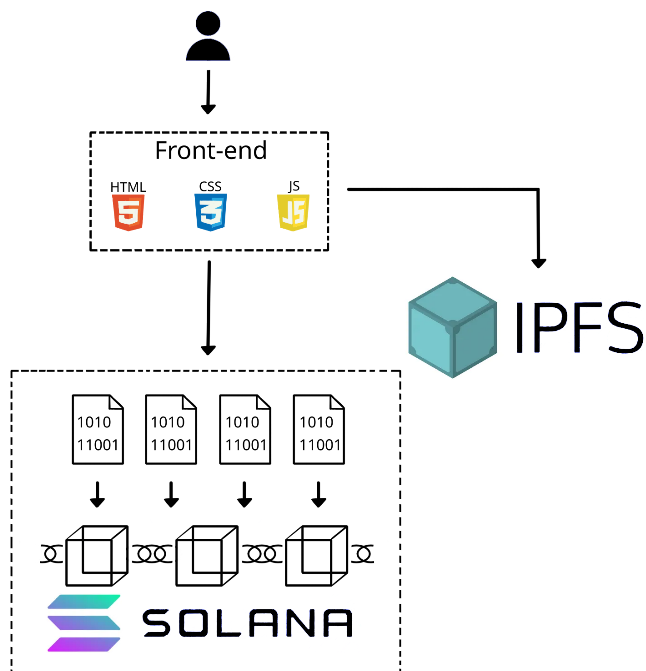

Figura 3.1: Arquitectura del marco de referencia

que son ejecutados en la Blockchain se realizan las operaciones sobre las imágenes y a su vez se registran estas operaciones como transacciones en la misma Blockchain. Por último, la o las imágenes que sean parte de las operaciones realizadas son almacenadas en IPFS, desde donde podrán ser accedidas en cualquier momento. Puede darse el caso en el que IPFS no sea llamado y en otras ocasiones se necesitará almacenar o retirar una imagen del mismo.

3.1. Análisis

En el párrafo anterior se habla sobre operaciones que involucran imágenes, estas operaciones están definidas de acuerdo a los diferentes casos que puedan presentarse durante el ciclo de vida de una imagen. Se puede tomar el ejemplo de un usuario que quiere compartir una imagen en internet, este sube su imagen a una plataforma; otros usuarios que también tienen acceso podrían buscarla, descargarla o incluso editarla para mejorarla o cambiar detalles (en caso de tener permisos de edición). Posteriormente, podría darse el caso de que el usuario quiera cambiar los permisos que tenía para la imagen debido a que ya no desea compartirla o no quiera que la editen.

A partir de este análisis se definieron las operaciones que el esquema debe soportar. Con esto, se determinó qué información será almacenada dentro de la Blockchain así

como el diseño de los correspondientes contratos inteligentes. Así, se definieron las siguientes operaciones a realizar sobre las imágenes en el esquema.

- Subir imagen
- Modificar permisos
- Descargar imagen
- Editar imagen

Antes de describir a detalle cómo se ejecutan las operaciones, es importante hablar acerca de la información que se genera a partir de la ejecución de dichas operaciones. Lo que se busca es tener un registro de los procesos que ocurren con las imágenes pero es igual de importante conocer de donde vienen estos. Así, para conocer el origen de una operación realizada, se usa la llave pública de cada usuario de la Blockchain. De esta manera, es posible tener un registro que contenga un elemento digital del origen de determinada operación.

Debido a las limitaciones que se tienen dentro de la Blockchain en cuanto al tamaño de información que puede ser almacenada y procesada, en lugar de trabajar con imágenes se utilizará un identificador que apuntará a las imágenes que se encuentren dentro de IPFS. Dicho identificador será el mismo que utiliza IPFS (`cid`), esto para referirnos a cada imagen de manera única. Adicionalmente, debido a que este identificador de contenido es generado con el hash del archivo (como se mencionó en el capítulo anterior), también brinda la confianza de que la información almacenada en la Blockchain es la correcta y no se trata de la información de otra imagen.

Además del identificador de contenido, es importante diferenciar si la imagen es una edición, una original o mostrar si tiene permiso para ser editada o vista. Eso puede lograrse con indicadores booleanos cuyos valores darán información extra adicional la imagen.

Si se da una edición, se busca conocer qué cambios se realizaron (lo cual puede no ser evidente en ciertos casos) para dar lugar a dicha edición. Esto es posible mediante una imagen de diferencia que compara a la imagen original y a la imagen editada, por lo cual también hay que diferenciar si se trata de una imagen diferencia mediante el uso de otro indicador booleano.

La información que alimentará parte del linaje electrónico de una imagen dada se almacenará dentro de la Blockchain mediante una serie de estructuras de datos (figura 3.2) que varían dependiendo del tipo de operación realizada, estas estructuras se almacenarán en la Blockchain mediante el uso de cuentas para almacenamiento de información (mencionadas en la subsección 2.4.1) y que forman parte del estado del contrato inteligente, dando la posibilidad de ser modificadas o no dependiendo de la operación que se esté realizando.

La primera estructura llamada **ImgData** contiene la información (metadatos) de la imagen que está involucrada dentro de las operaciones *subir*, *editar* y *modificar permisos*. Dicha estructura está compuesta por:

```
struct ImgData {
 owner: String,
 CID: String,
 parent: String,
 child: bool,
 diff: bool,
 public: bool,
 editable: bool
}
```

```
struct DwnldLog {
 downloader: Pubkey,
 CID: String
}
```

Figura 3.2: Estructuras de datos para representar parte del linaje electrónico de la imagen.

- La llave pública del usuario que sube la imagen, lo que permite conocer quién es su propietario.
- El identificador de contenido (`cid`) de la imagen, el cual ayudará a encontrarla y diferenciarla dentro de IPFS.
- El `cid` del archivo padre que, en caso de tenerlo, indica de qué imagen derivó la imagen actual.
- Un indicador booleano para saber si la imagen es hija de otra imagen (verdadero) o no (falso).
- Un indicador booleano que, de ser verdadero, muestra si la imagen es una diferencia entre una imagen padre y una imagen derivada.
- Un indicador para saber si la imagen es pública (verdadero) o no (falso), este indicador está relacionado al cifrado de la imagen.
- Un indicador booleano para establecer si la imagen es editable con un valor verdadero o de solo lectura con un valor falso.

La razón por la que se tienen indicadores para mostrar si una imagen es una edición o una diferencia es ser capaces de diferenciar de manera más sencilla entre las dos imágenes, ya que ambas comparten la imagen padre y solo tener el `cid` del padre no es suficiente para conocer de qué tipo de imagen se trata (edición o diferencia). Por otro lado, la estructura **DwnldLog** de la figura 3.2 es utilizada en las operaciones *editar* y *descargar*, dicha estructura sirve para documentar qué se realizó una descarga al registrar:

- La llave pública del usuario que descargue la imagen.
- El `cid` correspondiente a la imagen.

El resto del linaje electrónico se compone de los registros de las operaciones realizadas, dichos registros se almacenan directamente en la Blockchain ya que cada vez que se lleve a cabo una operación sobre una imagen, esta operación realizará una transacción y todas las transacciones se registran automáticamente en la Blockchain.

3.2. Subir imagen

Al ejecutar la operación de subir una imagen, se ejecuta un contrato inteligente que sube una imagen a IPFS. Este contrato guarda la estructura de datos **ImgData** en la Blockchain. En este caso al ser una imagen origen, su linaje comienza con dicha imagen por lo que no tiene padres o hijos. De esta manera, el `cid` del padre es una cadena vacía y los indicadores `child` y `diff` tendrán un valor de falso. Por otro lado, los indicadores `public` y `editable` dependen de lo que el usuario decida hacer con su imagen.

La figura 3.3 muestra lo que ocurre al subir una imagen. El primer paso consiste en ejecutar el contrato inteligente para almacenar la información de la imagen dentro de la Blockchain (paso 1). Si la transacción fue válida, la Blockchain regresará una respuesta exitosa (paso 2) y dicha imagen podrá subirse a IPFS donde permanecerá disponible, completando así el paso 3.

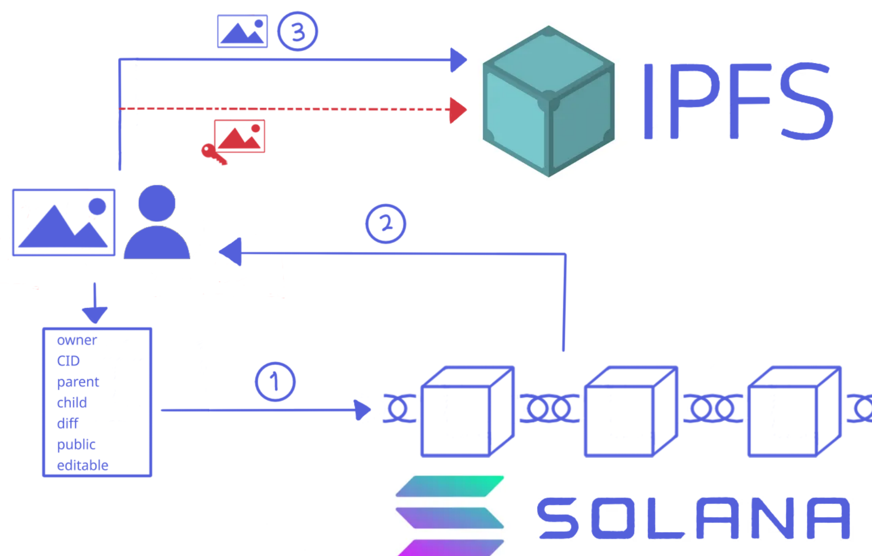

Figura 3.3: Proceso de subir una imagen

Se requiere que el guardar la información en la Blockchain y subir la imagen a IPFS sucedan de manera atómica, es decir, es necesario que ambas tareas se ejecuten de manera exitosa para poder completar la operación de subir la imagen. Esto es para evitar, por ejemplo, que se complete la subida de información a la Blockchain pero falle la comunicación con IPFS y la imagen no se encuentre presente, lo cual requeriría que se ejecutara de nuevo el contrato inteligente y representaría costos adicionales en la Blockchain para el usuario, además de añadir registros de transacciones incompletas al linaje electrónico.

Hay que aclarar que todo contenido en IPFS es público, lo que significa que cual-

quiera que tenga en su posesión el `cid` de la imagen podrá acceder a ella. Si el usuario quisiera que su imagen fuera confidencial esta tendría que ser cifrada. Si este fuera el caso, el indicador `public` tendrá un valor falso y dicha imagen se cifrará antes de ser subida a IPFS, almacenando en la Blockchain el `cid` de la imagen ya cifrada. En la misma figura 3.3 puede verse este proceso opcional dentro del paso 3, en color rojo.

Tomando en cuenta esta necesidad de confidencialidad, se ideó un método para mantener oculto el contenido de cualquier imagen de toda entidad que no tenga permiso del dueño de dicha imagen.

En este caso, si el dueño de la imagen desea que ésta sea privada, él creará una llave de cifrado simétrico con la que cifrará dicha imagen para que solo las personas permitidas tengan acceso a ésta. Esto crea la necesidad de administrar las llaves simétricas para cifrado y también crear una forma para compartir dichas llaves para otorgar acceso a los correspondientes usuarios.

Se propone entonces utilizar la Blockchain para compartir dichas llaves. Para esto, se presupone que los usuarios ya cuentan con un par de llaves para cifrado asimétrico[1] con la finalidad de que se cifre una llave simétrica con una llave pública que permita únicamente al dueño de la llave privada asociada descifrar dicha llave simétrica para poder acceder a la imagen. Para realizar el intercambio de llaves se utilizará la estructura de datos **SecKeyRequest** que se muestra en la figura 3.4. Dicha estructura, al igual que las anteriores, se almacenará en el estado del contrato inteligente y está compuesta por:

- La llave pública que utiliza en la Blockchain el usuario interesado en acceder a la imagen.
- La llave pública para cifrado del mismo usuario para cifrar la llave simétrica.
- El `cid` de la imagen a la que se quiere tener acceso.
- Un arreglo vacío donde irá la llave simétrica cifrada para acceder a la imagen.

```
struct SecKeyRequest {
 petitioner: String,
 petitionerPublicKey: String,
 CID: String
 secretKey: Array,
}
```

Figura 3.4: Estructura de datos para compartir llaves para descifrar imágenes

[1]Este par de llaves para cifrado es diferente al par de llaves utilizado para la Blockchain, el cual se usa únicamente para firmar transacciones y manejar fondos dentro de la misma Blockchain, no para cifrar.

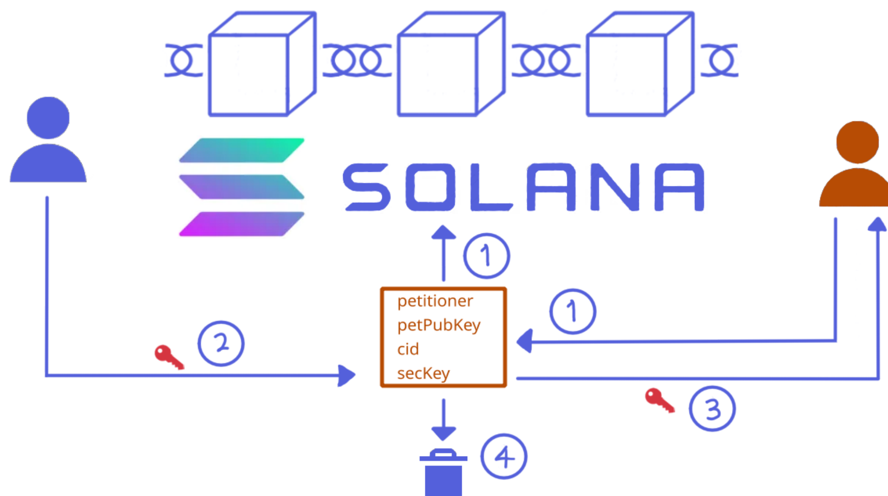

Figura 3.5: Proceso de compartir llave simétrica de una imagen privada

El proceso de compartir una llave puede observarse en la figura 3.5. En el primer paso, el usuario interesado en acceder a la imagen en cuestión realiza una solicitud subiendo a la Blockchain la estructura **SecKeyRequest** mencionada anteriormente. Posteriormente, el dueño de la imagen recibe una notificación[1] y, de aceptar compartir la llave, responde a la solicitud subiendo al campo `secretKey` de la estructura **SecKey-Request** la llave cifrada con la llave pública de cifrado proporcionada por el solicitante (paso 2). Una vez que la llave cifrada se encuentre en la Blockchain, el solicitante obtiene la llave en el paso 3 y por último se elimina la estructura **SecKeyRequest** del estado del contrato inteligente en el paso 4. Este método de compartir llaves permite que un usuario interesado en acceder a una imagen cifrada pueda obtener la llave para descifrarla de forma segura si tiene el permiso por parte del dueño de dicha imagen.

La razón por la cual se utiliza la estructura `SecKeyRequest` para esta compartición de llaves es que ésta permite registrar cada uno de los pasos descritos anteriormente. Cada que vez que se genere, modifique o elimine una estructura **SecKeyRequest** se generará una transacción en la Blockchain que se ligará con cada uno de esos procesos. De esta forma, se puede tener documentación de cuándo ocurrió una solicitud o una compartición de llaves y qué llaves públicas de la Blockchain estuvieron involucradas.

En caso de que un usuario malintencionado acceda a la información almacenada en la estructura, no podrá obtener la llave simétrica ya que ésta se encuentra cifrada con la llave pública del usuario que solicitó la imagen y sin la llave privada asociada será computacionalmente difícil obtener dicha llave simétrica.

Por otro lado, si un usuario quiere hacerse pasar por otro (impersonificación) para

[1] Hasta el momento no hay una implementación del método para notificar al dueño de la imagen, podría ser mediante una notificación en el *front-end* o una notificación directa por parte del solicitante.

engañar al dueño de la imagen y que éste le comparta su llave privada, mientras el dueño de la imagen se asegure que la llave pública Ed25519 dentro de la estructura corresponde a la llave pública de la persona a la que realmente solicita la imagen, no habrá ningún problema. Por otro lado, únicamente el contrato inteligente puede modificar la información dentro de la estructura por lo que no será posible para alguien externo suplantar la llave pública Ed25519 por la propia.

IPFS previene el caso en que un atacante cifre una imagen pública que no es suya para evitar que el dueño pueda acceder a la misma (denegación de servicio). Esto es posible ya que la modificación del contenido de la imagen (producto del cifrado) generará un nuevo archivo con un `cid` diferente por lo que la imagen original se mantendrá accesible a través de su `cid` correspondiente. Además, en caso de que un nodo que contiene a la imagen se encuentre fuera de línea, la imagen estará disponible entre los diferentes nodos que accedan a almacenar dicha imagen, evitando así que la imagen no esté disponible.

3.3. Modificar permisos

Para modificar permisos es necesario definir cuáles de estos se manejarán dentro el marco de referencia. Estos permisos están relacionados con los campos `public` y `editable` de la estructura **ImgData** que, según su valor, significan lo siguiente:

- **Solo lectura.** Permite que un usuario vea la imagen pero no pueda descargarla ni editarla, en este caso el indicador `editable` tiene un valor de 0.
- **Editable.** El usuario puede descargar y editar la imagen debido a que el indicador `editable` tiene el valor 1.
- **Público.** Cualquier usuario puede ver la imagen y el indicador `public` es 1.
- **Privado.** Nadie puede acceder a la imagen si no tiene permiso del dueño ya que la imagen se encuentra cifrada y la llave de descifrado no es compartida, el indicador `public` es 0.

El proceso de modificar permisos consiste en realizar una modificación de la información de la imagen que ya se encuentra almacenada en la estructura **ImgData** dentro del estado del contrato inteligente, los campos que pueden ser modificados únicamente son `public` y `editable`, como se mencionó anteriormente. La imagen 3.6 muestra como sería el proceso, el usuario actualizará los campos de interés en el paso 1, los cuales pueden observarse en color verde. Una vez que la transacción haya sido aceptada, se tendrá una respuesta positiva de la Blockchain completando el paso 2. Si se desea que la imagen sea pública o deje de serlo, se tendría que cambiar el archivo almacenado en IPFS por el archivo cifrado o descifrado (paso opcional en color rojo), eliminando el archivo anterior de IPFS porque no tendría sentido mantener dentro de IPFS la imagen a la que se le cambiaron los permisos. Debido a que el cifrado/descifrado de la imagen

genera un `cid` diferente, se tiene que reemplazar el mismo `cid` (también en color verde) con el de la nueva imagen para evitar que el `cid` registrado apunte a un archivo inexistente. En el caso donde se modifica el campo `public`, el proceso debe de ser atómico de la misma manera que en la sección 3.2 para asegurar que la información modificada se encuentre en la Blockchain y que la imagen cifrada o descifrada esté presente en IPFS.

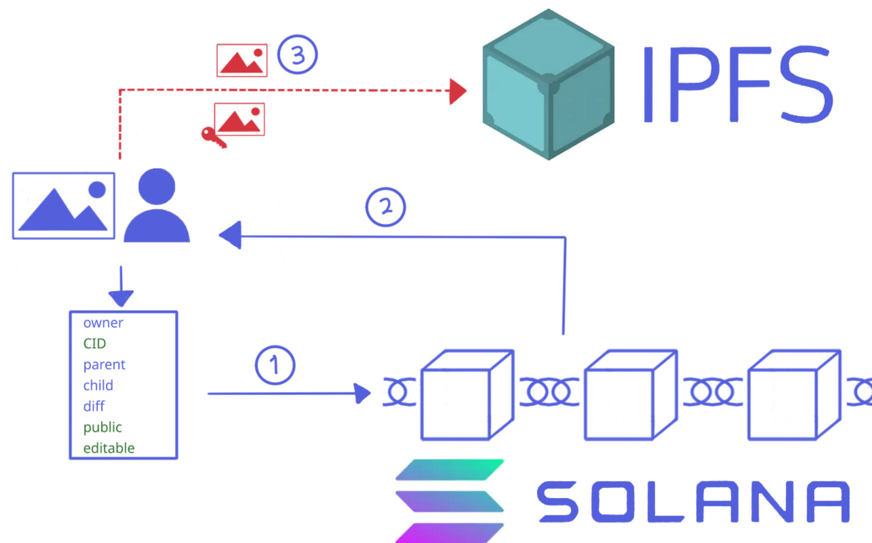

Figura 3.6: Proceso de modificar permisos para una imagen

3.4. Descargar imagen

En caso de que un usuario quiera descargar una imagen, se requiere tener los permisos de descarga (valor del campo `editable` igual a 1) para la imagen y después ejecutar el correspondiente contrato inteligente. La ejecución del contrato inteligente permite que el linaje electrónico de la imagen registre la descarga y así poder tener conocimiento de todos los usuarios que descarguen dicha imagen. En la figura 3.7 se observa el proceso de descarga, el usuario guarda la estructura de datos **DwnldLog** en la Blockchain durante el paso 1 para registrar la llave pública del usuario que realiza la descarga e identificar la imagen descargada mediante su `cid`. Si se obtiene una respuesta exitosa por parte de ésta (paso 2), se procede a obtener la imagen desde IPFS en el paso 3. Este proceso de descarga también debe ocurrir de manera atómica para evitar que no haya un registro fallido de la descarga o que, a pesar de haberse registrado la descarga, no se obtenga dicha imagen.

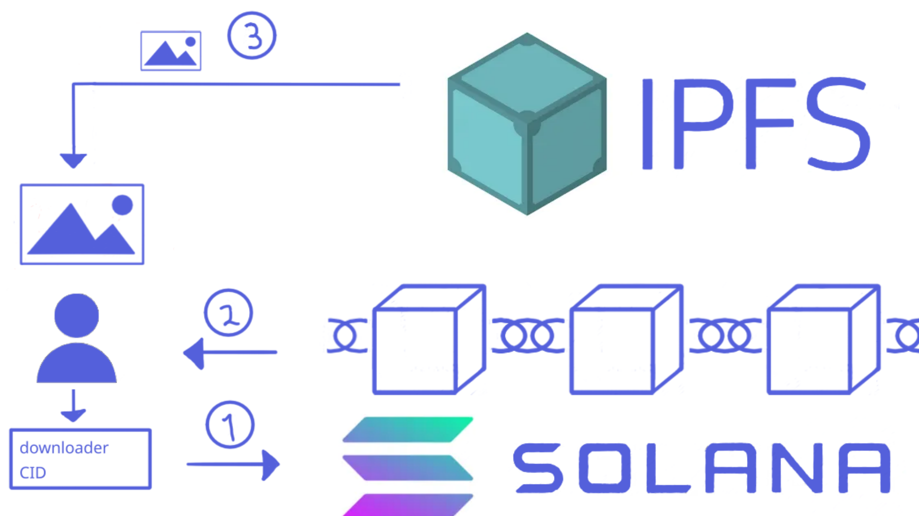

Figura 3.7: Proceso de descargar una imagen

3.5. Editar imagen

Antes de describir los procesos que se realizan dentro de la edición de una imagen, es importante comentar que, al realizarse la edición de una imagen, se crea una nueva imagen que contiene los cambios realizados. Esta nueva imagen es hija de la imagen original, por lo que es importante conservar ambas imágenes al igual que una tercera que muestre las diferencias entre ambas. Esto se muestra en la figura 3.8. Al conservar estas tres imágenes se contribuye a que el linaje electrónico registre el proceso completo de edición, es decir, qué fue lo que se le hizo a la imagen y cuál fue el resultado después de la modificación en lugar de reportar únicamente que se realizó una edición.

Al añadir 2 nuevas imágenes al sistema, la imagen editada y la diferencia, se tendrá, a su vez, que almacenar las estructuras de datos **ImgData** correspondientes. El campo padre contendrá el `cid` de la imagen original y los campos `child` y `diff` tendrán un valor verdadero dependiendo de si se trata de la imagen editada o de la diferencia, siendo el campo `child` correspondiente a la primera y el campo `diff` a la segunda.

Como se puede ver, la operación de editar está compuesta por las operaciones de descargar y subir imagen. Esto se muestra en la figura 3.9, donde se observa que para editar una imagen primero se obtiene ésta mediante una descarga y, una vez editada, se sube la edición y la diferencia anteriormente descritas. En los pasos del 1 al 3 se inicia el proceso de descarga descrito con anterioridad y, una vez descargada la imagen, el usuario en color naranja tiene en su posesión la imagen que desea editar. Ya que se edite dicha imagen, existe una nueva imagen derivada de la original (también en color naranja) que debe ser subida por el usuario que realizó la edición. Para lograr esto, se ejecuta el contrato inteligente correspondiente en el paso 4 donde se almacenará la

Figura 3.8: Diferencia entre imagen original y editada

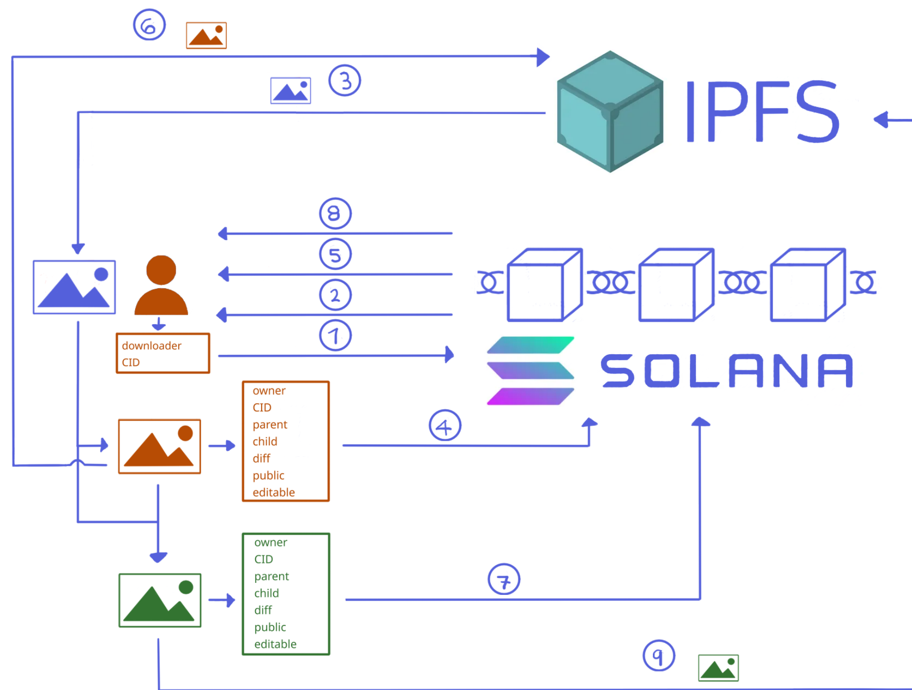

Figura 3.9: Proceso de editar una imagen

estructura **ImgData** en la Blockchain que corresponde a la información de la imagen editada. Cuando se ejecute con éxito el contrato inteligente, la Blockchain regresará una respuesta positiva al usuario en el paso 5 y la imagen se subirá a IPFS en el paso 6. Como también se realizó una comparación entre la imagen original y su edición, se tiene una imagen diferencia (resaltada con color verde) y cuya información se almacenará en

la Blockchain dentro de su propia estructura **ImgData** durante el paso 7. De tener una respuesta exitosa en el paso 8, el paso 9 consiste en almacenar la imagen en IPFS. Es necesario que el proceso de subir la edición y la imagen diferencia sea atómico, ya que se requiere que ambas estén registradas en la Blockchain, dentro de IPFS y se garantice que no se haya subido una sin la otra. Por otro lado, como se ejecuta la operación de descargar una imagen descrita anteriormente se asegura la atomicidad de esta operación completa.

De este modo se tiene un conjunto de operaciones sobre imágenes que, al ejecutarse en la Blockchain, alimentarán el linaje electrónico de cualquier imagen. Ya que cómo serían las operaciones definidas para imágenes y cómo sería su funcionamiento, en el siguiente capítulo se hablará acerca de la implementación realizada para llevar a la práctica este diseño.

3.6. Atomicidad en las operaciones sobre imágenes

Como se mencionó anteriormente, debe existir atomicidad al realizar las operaciones descritas. Por esto se entiende que, tanto el registro de la información en la Blockchain como la operación dentro de IPFS deben completarse para que toda la operación sea válida y en caso de que uno de los dos procesos falle, toda la operación también lo hará.

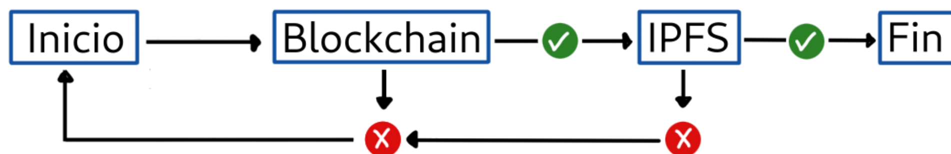

Figura 3.10: Proceso de editar una imagen

La figura 3.10 muestra como se ejecuta la idea anteriormente mencionada para las operaciones que involucran imágenes. En caso de que la ejecución del contrato inteligente dentro de Blockchain sea exitosa, se procede a ejecutar la tarea deseada en IPFS y, si esta es de igual forma exitosa, se finaliza la operación sobre la imagen. En caso contrario, si falla la ejecución del contrato inteligente, inmediatamente falla la operación. Por último, si llegara a pasar que la ejecución del contrato inteligente es exitosa pero ocurre un error con IPFS, también falla la operación y no se completa.

Una posible solución para este último caso, la cual no se desarrolla en el presente trabajo, sería el ejecutar un contrato inteligente que revierta el registro de la información dentro de la Blockchain. Como toda transacción queda registrada dentro de Blockchain, en este caso existirá un registro de la ejecución exitosa del contrato inteligente y un registro de la transacción realizada para anular los cambios debido al fallo de IPFS.

Capítulo 4

Implementación

Retomando la arquitectura mostrada en el capítulo anterior, en la figura 4.1 ahora describimos más a fondo como interactúan los componentes de esta. En cualquier operación, el flujo de trabajo partirá desde el usuario que quiera ejecutar alguna operación sobre una imagen y para lograr esto tendrá que comunicarse con el *front-end*. El *front-end* permitirá al usuario realizar una petición al System Program de Solana para crear una cuenta en la que se almacenará alguna de las estructuras de datos descritas con anterioridad. Una vez que se tenga la cuenta, ésta pasará al contrato inteligente donde se ejecutará la operación seleccionada y, en caso de finalizar sin errores, el *front-end* se comunicará con IPFS para enviar o recibir las imágenes.

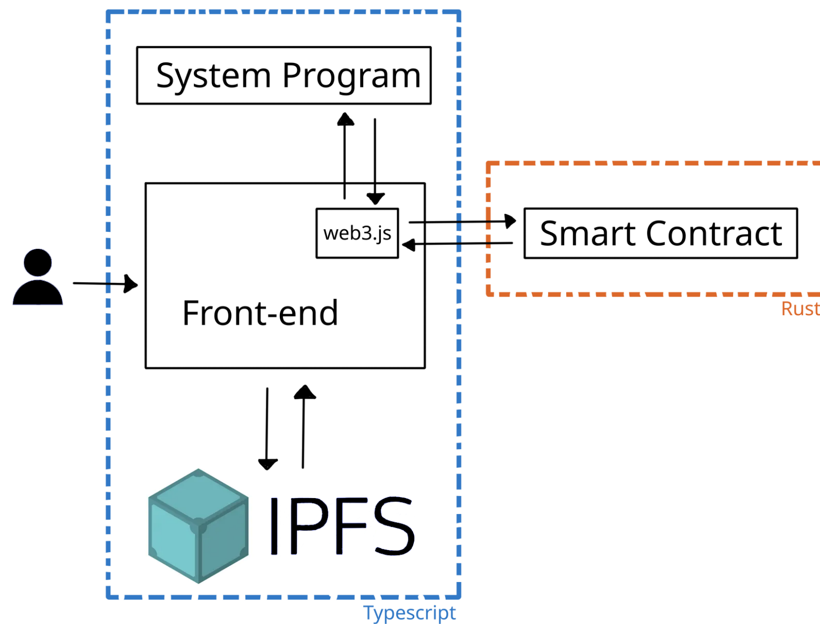

Figura 4.1: Arquitectura de la implementación

La implementación del marco de referencia propuesto se realizó haciendo uso de los lenguajes de programación Rust y Typescript, siendo el primero el utilizado para los contratos inteligentes en la Blockchain y el segundo para la interacción con el usuario y con IPFS en el *front-end*. El código completo puede observarse en el anexo A.

4.1. Front-end

El *front-end* desarrollado en Typescript se encarga de que el usuario pueda interactuar con la Blockchain y con IPFS pero también permite el manejo de las imágenes y ejecuta las funciones criptográficas necesarias para la implementación del marco de referencia por lo que el *front-end* puede dividirse en los siguientes módulos:

- **Criptografía.** Este módulo se encarga de generar llaves simétricas para cifrado de imágenes, generar el par de llaves de RSA, cifrar y descifrar imágenes.
- **Imágenes.** Módulo responsable de realizar la comparación entre las imágenes original y editada.
- **Comunicación con IPFS.** Módulo donde se suben, descargan y eliminan imágenes.
- **Comunicación con Solana.** Este módulo es el que se encarga de establecer conexiones con la Blockchain, llamar al System Program para crear cuentas, llamar al contrato inteligente para ejecutar las operaciones sobre imágenes y también consultar transacciones.

A continuación se explica a detalle cada uno de estos módulos.

4.1.1. Criptografía

Como se describió en el capítulo anterior, es necesaria la implementación de diversos algoritmos criptográficos para lograr la confidencialidad de las imágenes en caso de que un usuario lo desee. Para esta implementación se utilizó la biblioteca `crypto.js` [50], la cual es una implementación de OpenSSL para utilizarse en Javascript o Typescript y permite realizar cifrado, descifrado, cálculo de hashes, entre otras cosas. El cifrado que decidió utilizarse es el AES de 256 bits debido a su seguridad y rendimiento, con lo que se requiere que para cada imagen que quiera ser cifrada se tenga una llave simétrica asociada.

La figura 4.2 muestra los procesos descritos a continuación en los que se involucran llaves simétricas. El cifrado de imágenes se realizó mediante una función a la que se llamó `encryptImage`, dicha función recibe la imagen a cifrarse y una llave simétrica para el cifrado. Dentro de la función se especifica la utilización del algoritmo AES y se genera un vector de inicialización con la función `randomFill` para después aplicar el cifrado a la imagen.

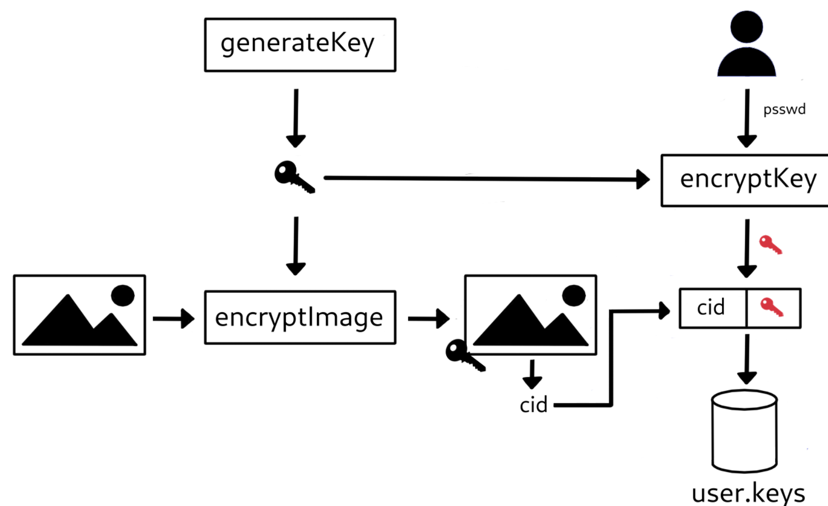

Figura 4.2: Procesos del *front-end* en los que se involucran llaves criptográficas.

La generación de llaves simétricas se realizó a partir de la función `generateKey` en la que se necesita especificar el largo de la llave (256 bits) y el algoritmo para el que se va utilizar dicha llave (AES). Como es de suma importancia mantener las llaves simétricas seguras, las mismas serán cifradas (también utilizando el cifrado AES de 256 bits) con una llave única generada a partir de la función de derivación de llaves `scrypt`, la cual utiliza una contraseña proporcionada por el usuario para derivar la llave. Esto garantiza que las llaves estén seguras y que únicamente las personas con permiso del dueño de la imagen puedan acceder a la misma ya que una llave cifrada no podrá descifrar la imagen que igualmente se encuentra cifrada.

La función `encryptKey` cifra las llaves simétricas, solo se requiere ingresar la llave creada y dentro de esta función se solicitará la contraseña al usuario para generar (mediante `scrypt`) la llave simétrica que permitirá cifrar la llave de la imagen. Una vez que la llave simétrica esté cifrada, ésta se almacenará junto con el `cid` de la imagen cifrada para conocer a qué imagen está asociada dicha llave. Para esta implementación las llaves se almacenan en un archivo llamado `user.keys` dentro de la computadora del usuario y, del mismo modo, el hash de la contraseña con la cual son cifradas se almacenan en el archivo `user.psswd`.

4.1.2. Imágenes

La comparación de imágenes que se realiza al existir una edición de imagen se implementó usando la biblioteca `jimp` [51]. Esta tiene un método llamado `diff` para realizar dicha comparación. Se tiene como entrada las imágenes que se quieren comparar, en este caso las imágenes original y editada. Con dichas imágenes se inicializan dos objetos, cada objeto contiene la información y datos de una imagen como lo son su resolución,

formato, entre otros.

Al realizar la diferencia entre imágenes, se resaltan los cambios de valores en píxeles entre una y la otra. En este caso, el resultado de la diferencia realizada es una imagen cuyos píxeles en color rojo representan los píxeles que fueron modificados. También se tendrá a la imagen original con un cierto porcentaje de transparencia como puede observarse en la figura 4.3.

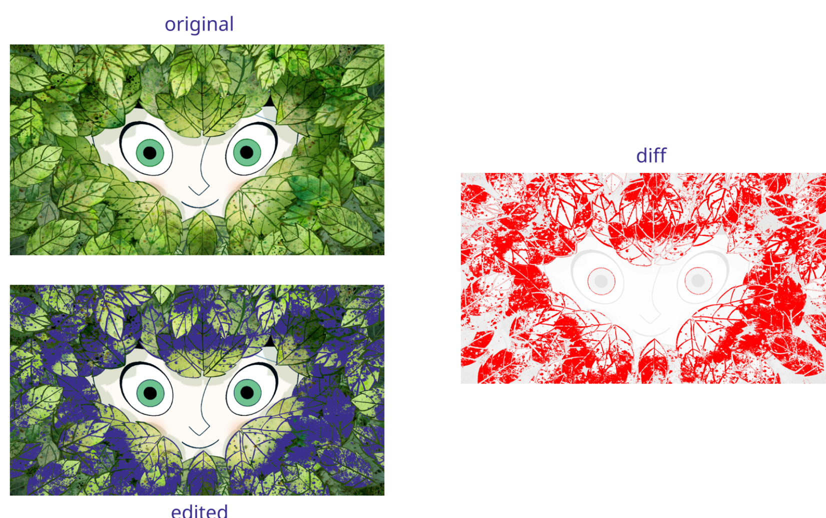

Figura 4.3: Comparación de la imagen original y la modificada.

4.1.3. Comunicación con IPFS

Para la comunicación del *front-end* con IPFS, se utilizó la biblioteca `jsipfs` [52] que permitirá obtener el *cid* de una imagen y almacenarla, descargarla o eliminarla dependiendo de la operación que se esté realizando. Para lograr esto, se generaron funciones que utilizan los métodos proporcionados por `jsipfs`. Cada función inicia una instancia de IPFS para posteriormente utilizar el método requerido. A continuación se describen las funciones y los métodos de `jsipfs` que estas utilizan.

- `getCid` utiliza el método `add` el cual tiene el parámetro **onlyHash** que calcula el cid del archivo sin agregarlo a IPFS. Esto genera el `cid` de la imagen sin subirla a IPFS, lo cual es útil ya que este se requiere para generar la estructura **ImgData** previo a subir una imagen.

- `uploadToIpfs` utiliza el método `add` con el parámetro **pin** para permitir que el archivo persista en el nodo de IPFS al cual se sube, de este modo se puede subir la imagen a IPFS para completar las operaciones *subir*, *editar* y *modificar* permisos (en caso de que deba subirse la imagen cifrada al cambiar el campo `public`).

- `getFromIpfs` utiliza el método `cat` y regresa un *buffer* que representa a la imagen. De esta forma es posible obtener una imagen de IPFS, ya sea para descargarla o para cifrarla en caso de que se modifiquen los permisos de la misma.
- `removeFromIpfs` utiliza el método `pin.rm` para eliminar la permanencia de la imagen en el nodo y posteriormente llama al colector de basura para eliminar completamente la imagen. Esto permite que la imagen sea eliminada de IPFS en caso de que se requiera, por ejemplo, en el caso de que se cambie el permiso de la imagen y se deba reemplazar la imagen cifrada o en claro para dar paso a su contraparte.

4.1.4. Comunicación con Solana

La comunicación con Solana se hace mediante la biblioteca `web3` de Solana o `solana/web3.js` [53] como se mencionó al inicio del capítulo. Dicha comunicación consiste en generar cuentas para almacenar información, obtener información de la Blockchain, conectar con contratos inteligentes mediante el envío de transacciones, entre otras cosas.

El primer paso es establecer una conexión con la Blockchain mediante la función `establishConnection`, donde se utiliza la función `getRpcUrl` para mostrar el cluster elegido a partir de la configuración que se tiene de Solana y elegir la conexión a la mainnet, devnet, testnet o incluso utilizar una máquina propia con una conexión a localhost.

Para este trabajo se decidió utilizar la devnet ya que esta opción es el espacio de pruebas de la Blockchain. Al establecer la conexión, es posible enviar transacciones y que estás realicen las operaciones descritas en el Capítulo 3.

Para almacenar información en la Blockchain es necesario crear cuentas. Como se mencionó en el capítulo anterior, el tipo de operación define qué información se guardará dentro de una cuenta y cuánto espacio ocupará. Debido a esto se tiene una serie de funciones que crearán cuentas con el tamaño adecuado para realizar la operación necesaria, dichas funciones son: `createImgDataAccount`, `createDownloadLogAcount` y `createSecKeyRequestAccount`. La tabla 4.1 muestra como se relacionan el tipo de cuenta y el tamaño de la misma con el tipo de operación que se desea ejecutar. La primer columna corresponde a las operaciones descritas en el capítulo anterior, la segunda columna muestra los tipos de cuentas asociados y la tercera columna indica el tamaño de cada tipo de cuenta (en bytes). Es necesario aclarar que la operación "Modificar permisos" no requiere crear una cuenta pero está relacionada con el tipo de cuenta `imgDataAccount`. Por otro lado, "Editar imagen" se relaciona tanto con `imgDataAccount` como con `dwnldLogAccount` para poder completar el proceso de edición que se mencionó en el capítulo anterior.

Cada una de las funciones mostradas en la tabla 4.1 utilizan un par de funciones proporcionadas por `web3.js` de nombre `createWithSeed` y `createAccountWithSeed` para crear una cuenta derivada que guarda la información relacionada con la operación y que solo será controlada por el contrato inteligente. Lo que `createWithSeed` realiza

Tabla 4.1: Operaciones de imágenes y las cuentas involucradas - Cada operación del marco de referencia involucra tipos de cuentas distintos.

Operación	Tipo de cuenta involucrada	Tamaño de la cuenta(bytes)
Subir imagen	`imgDataAccount`	179
Modificar permisos	`imgDataAccount`	179
Descargar imagen	`dwnldLogAccount`	112
Editar imagen	`imgDataAccount`	179
	`dwnldLogAccount`	112
Compartir llaves	`secKeyRequestAccount`	1428

es obtener una llave pública que corresponderá a la cuenta que almacenará información basándose en la llave publica del usuario y una semilla, dando como resultado un valor de 256 bits. Por otro lado, `createAccountWithSeed` genera la transacción que crea dicha cuenta, para realizar esto es necesario ingresar los siguientes datos:

- La llave pública del usuario que está creando la cuenta.
- La llave pública de la cuenta que pagará la transacción (en este caso es la misma llave del usuario que creará la cuenta).
- La cantidad de *lamports* (fracciones de SOL) que almacenará la cuenta. Esta debe ser la cantidad exacta para evitar pagar renta.
- La llave pública generada con `createWithSeed`.
- El identificador de contrato inteligente que equivale a la llave pública del mismo.
- La semilla con la que se generó la llave pública dentro de `createWithSeed`.
- El tamaño (en bytes) de la información que se almacenará en la cuenta.

Para determinar el tamaño de la información que se almacena en cada cuenta es necesario implementar las estructuras de datos mostradas anteriormente (Capítulo 3) en Typescript con la finalidad de representar dicha información. Para esto, se crea una clase por cada una de las estructuras donde los atributos de las clases corresponden a los campos de dichas estructuras. Posteriormente se genera un esquema que relaciona a la clase de Typescript con la estructura que se manejará en Rust mediante un mapa. Este mapa permitirá realizar la serialización de acuerdo al formato Borsh [1] que es

[1] Borsh significa *Binary Object Representation Serializer for Hashing.* Es un formato que prioriza la consistencia, la seguridad y la velocidad. [54]

utilizado por Solana para serializar la información dentro de las cuentas. El resultado de la serialización es un arreglo de bytes. Al obtener el tamaño de este arreglo, se tendrá el tamaño de la cuenta en bytes.

Una vez que se tiene la cuenta deseada, se llama a alguna de las funciones `uploadImage`, `permissions`, `downloadImage`, `editImage`, `requestKey`, `shareKey` u `obtainKey`. Cada una de las funciones tiene parámetros de entrada diferentes como puede verse a continuación.

- `uploadImage`. Sus parámetros de entrada son la llave pública del usuario que sube la imagen, la llave pública de la cuenta creada, la imagen, el indicador de si la imagen es pública, el indicador de si la imagen es editable, el `cid` de la imagen padre (edición), el indicador de si la imagen es una edición (edición), el indicador de si la imagen es una diferencia (edición) y la llave pública de la imagen original (edición).

- `permissions`. Sus parámetros de entrada son la llave pública del usuario que va a cambiar permisos, la llave pública de la cuenta con la información de la imagen, el indicador de si la imagen es pública y el indicador de si la imagen es editable.

- `downloadImage`. Sus parámetros de entrada son la llave pública del usuario que descarga la imagen, la llave pública de la cuenta creada y el `cid` de la imagen.

- `editImage`. Sus parámetros de entrada son la llave pública del usuario que efectuará le edición, la llave pública de la cuenta con información de la imagen a editar, la llave pública de la cuenta que tendrá la información de la edición, llave pública de la cuenta que tendrá la información de la imagen de diferencia, la imagen original, la edición, indicador de si la edición es pública, indicador de si la edición es editable, indicador de si la imagen diferencia es pública e indicador de si la imagen diferencia es editable.

- `requestKey`. Sus parámetros de entrada son la llave pública del usuario solicitante, el cid de la imagen y la llave pública de la cuenta creada.

- `shareKey`. Sus parámetros de entrada son la llave pública del usuario que comparte la llave simétrica, la llave pública del usuario solicitante, la llave pública de la cuenta creada.

- `obtainKey`. Sus parámetros de entrada son la Llave pública del usuario solicitante y la llave pública de la cuenta creada.

Los últimos 4 parámetros que se manejan en la función `uploadImage` son opcionales debido a que son necesarios solamente cuando ocurra una edición (ya que se llama a esta misma función dentro de `editImage`). En caso de sólo subir una imagen estos se descartan, considerando únicamente los parámetros hasta el indicador de edición.

Cada una de las funciones se encarga de llamar al contrato inteligente para ejecutar la operación correspondiente y, finalmente, comunicarse con IPFS (si es el caso) al

momento de completarse la ejecución del contrato inteligente de manera exitosa. Para llamar al contrato inteligente se crea una instrucción al generar una nueva instancia de la clase `TransactionInstruction` mediante el método constructor que recibe los tres siguientes parámetros:

- Un vector con las llaves públicas de las cuentas involucradas.
- La llave pública que sirve como identificador del contrato inteligente a ejecutar.
- Un vector con los datos de instrucción que se utilizarán en la ejecución del contrato inteligente (opcional).

En este caso, las cuentas involucradas siempre van a ser la cuenta del usuario que desea realizar la operación y la cuenta creada (o ya existente) que almacena información. Es necesario especificar si las cuentas son editables o si son cuentas que firman la transacción a ejecutar. Por otro lado, los datos de instrucción son una colección de información que permitirá que se ejecuten las tareas que realiza el contrato inteligente, cada operación tiene diferentes datos de instrucción como se ve resumido en la tabla 4.2.

Tabla 4.2: Operaciones de imágenes y datos de instrucción relacionados - Cada operación del marco de referencia cuenta con diferentes datos de instrucción que pasarán al contrato inteligente.

Operación	Datos de instrucción
Subir imagen	operation number, cid, parent cid, child, diff, public, editable
Modificar permisos	operation number, cid, public, editable
Descargar imagen	operation number, cid
Editar imagen	-
Compartir llave	operation number, rsa public key, cid (`requestKey`) operation number, encrypted rsa public key (`shareKey`) operation number, accept value (`obtainKey`)

La operación editar imagen no cuenta con datos de instrucción porque, como se ha mencionado anteriormente, esta operación está conformada por las operaciones subir y descargar imagen, por lo que cada una de éstas manejará sus datos de instrucción cuando sean ejecutadas para completar esta operación. Por otro lado, la operación compartir llave está compuesta por 3 procesos que son: solicitar la llave, compartir la llave y obtener dicha llave. Estos procesos se ejecutan mediante las funciones `requestKey`, `shareKey` y `obtainKey` respectivamente y cada uno tiene sus datos de instrucción.

Si bien todos los datos de instrucción varían dependiendo de la operación, lo que todos comparten es el número de operación. Este es un byte cuyo valor determina la operación que se ejecutará dentro del contratos inteligente y procesará el resto de los datos de instrucción de acuerdo a la operación. Los valores que puede tomar el número de operación van del 0 al 5 y en la tabla 4.3 puede observarse el número de operación y la operación que representan junto con la función relacionada.

Tabla 4.3: Número de operación y la función relacionada - cada número de operación tiene una función asociada, la cual realiza la operación.

Número de operación	Operación	Función
0	Subir imagen	`uploadImage`
1	Modificar permisos	`permissions`
2	Descargar imagen	`downloadImage`
3	Compartir llave	`requestKey`
4	Compartir llave	`shareKey`
5	Compartir llave	`obtainKey`

Una vez que está construida la instrucción, la función `sendAndConfirmTransaction` es llamada para ejecutar el contrato inteligente en la Blockchain para que la operación deseada sea registrada. Si la transacción es exitosa, y de requerirlo la operación, se llama a alguna de las funciones mencionadas anteriormente para interactuar con IPFS.

4.2. Contrato inteligente

Como se mencionó anteriormente, el contrato inteligente que contiene las funciones anteriormente descritas está implementado en Rust. Este es un único contrato inteligente en el que se elige qué operación se ejecutará de las previamente descritas (Capítulo 3). El contrato inteligente está compuesto por 5 diferentes archivos que se describen a continuación.

- **entrypoint.rs** Este archivo es la entrada al contrato inteligente, da formato a la información para su correcto ingreso al contrato inteligente.
- **instruction.rs** En este archivo se realiza la selección de la instrucción a ejecutar.
- **processor.rs** En este archivo se ejecuta la instrucción seleccionada.
- **state.rs** En este archivo se definen las reglas de serialización y deserialización para poder almacenar información en la Blockchain.

- **error.rs** En este archivo se realiza todo lo relacionado con la administración de errores.

El flujo de ejecución del contrato inteligente puede observarse en la figura 4.4. La función `sendAndConfirmTransaction` es la primera en ejecutarse desde el *front-end*. Esta ingresa al archivo **entrypoint.rs** la información serializada necesaria (cuentas involucradas, identificador del contrato inteligente y datos de instrucción) ahí deserializa para pasar a **processor.rs** (1). En processor.rs los datos de instrucción se envían a instruction.rs (2) en donde se define la instrucción a ejecutar y se da formato a dichos datos. Los datos, ya dentro de una estructura, regresan a processor.rs (3). Ahí se decide a qué función se llama de las 6 posibles para ejecutar la instrucción seleccionada (como puede verse en la tabla 4.3). Al ejecutar la función correspondiente se crea comunicación con state.rs (4 y 5) para obtener o mandar información a las cuentas que almacenan la información deseada. Finalmente se tiene la ejecución exitosa de este contrato inteligente (6). En caso de existir un error como, por ejemplo, ingresar un número de operación diferente a los mostrados en la tabla 4.3 o modificar una cuenta sin tener permisos, la transacción será rechazada.

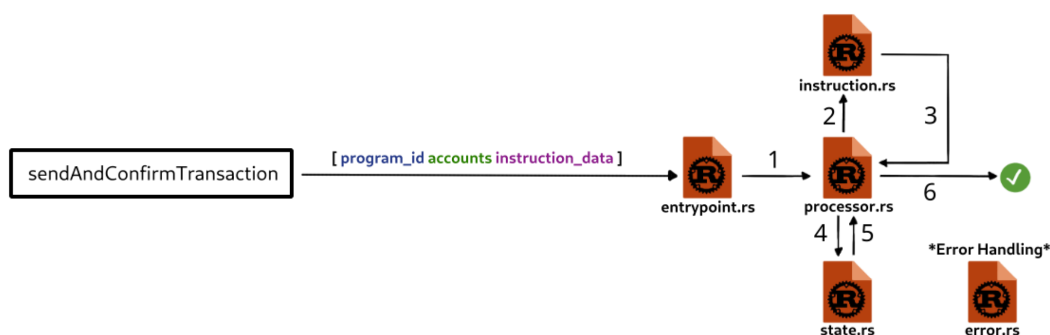

Figura 4.4: Flujo de proceso del contratos inteligente

A continuación describimos más a detalle cada uno de estos archivos.

4.2.1. entrypoint.rs

Como su nombre lo dice, este archivo es la entrada al programa. Este recibe como entrada un arreglo de bytes que contiene los parámetros serializados necesarios para la ejecución del contrato inteligente (el identificador del contrato inteligente, las cuentas involucradas y los datos de instrucción) y los deserializa. Una vez deserializados, estos se introducen a una función de procesamiento de instrucciones. Dentro de dicha función, los parámetros pasan a **processor.rs** para ser procesados, continuando con la ejecución del contrato inteligente. En resumen, **entrypoint.rs** se encarga de preparar la información para que tengan el formato adecuado para la correcta ejecución del contrato inteligente, esto se realiza mediante la deserialización y eventual serialización de

la información.

4.2.2. instruction.rs

Este archivo determina qué instrucción se ejecutará. Para lograr esto se hace uso de una declaración tipo `switch`, la cual evaluará el primer elemento del arreglo de bytes que representa a los datos de instrucción como se presento en la tabla 4.3. Dependiendo de dicho valor, se le dará formato al resto del arreglo de bytes de acuerdo a la operación que se ejecutará.

Este proceso está representado en la figura 4.5. En esta, se tiene como entrada el arreglo de bytes que corresponden a los datos de instrucción provenientes de **processor.rs**. Tomando el primer byte del arreglo, existen 6 posibles opciones (valores del 0 al 5) para dar el formato a los datos de instrucción, incluyendo una séptima opción que contempla todos los valores que no corresponden a una de las 6 opciones y arroja un error en caso de ser así. Dependiendo del valor, el arreglo de bytes restante se convertirá en una estructura con los campos necesarios para ejecutar la función correcta.

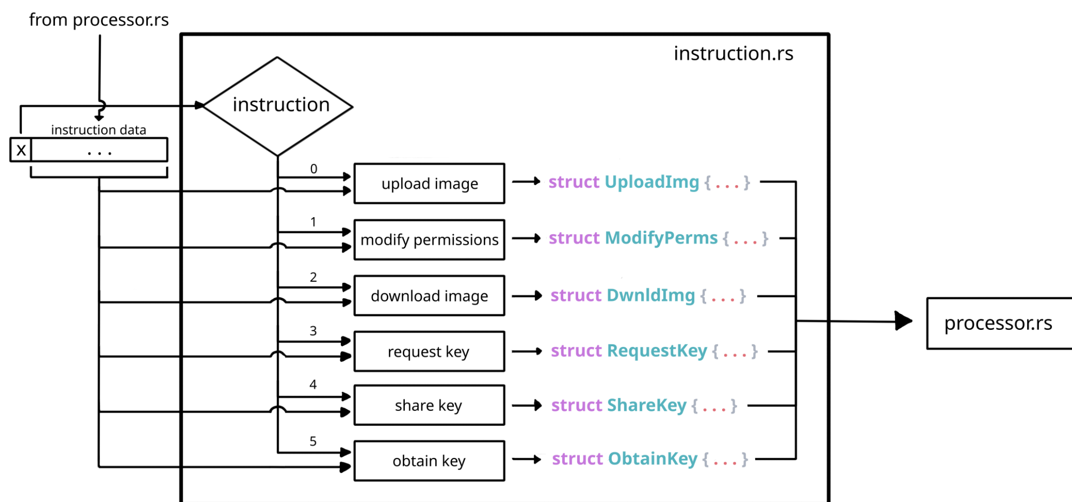

Figura 4.5: Representación de funcionamiento dentro de instruction.rs.

Como puede observarse, las 6 instrucciones son: *upload image*, *modify permissions*, *download image*, *request key*, *share key* y *obtain key*. Las instrucciones generan las estructuras **UploadImg**, **ModifyPerms**, **DwnldImg**, **RequestKey**, **ShareKey** y **ObtainKey** respectivamente. La estructura seleccionada será devuelta a **processor.rs** y permitirá que se continúe con la ejecución de la operación deseada. A grandes rasgos, **instruction.rs** se encarga de crear las estructuras de datos necesarias que definen la instrucción que será ejecutada en el archivo **processor.rs**.

4.2.3. processor.rs

Este archivo se encarga de ejecutar las funciones clave del marco de referencia y que se muestran en la tabla 4.3. Dicho proceso esta representado en la figura 4.6 donde se muestra la recepción de los datos de instrucción, las cuentas involucradas y el identificador del propio contrato inteligente provenientes de **entrypoint.rs**. Estos se envían a **instruction.rs** para definir qué instrucción va a ejecutarse. Una vez definida la instrucción y devuelta la estructura de datos correspondiente, se utiliza nuevamente una declaración tipo `switch` para evaluar el tipo de estructura devuelta y así llamar a la función correspondiente. Debido a que la estructura devuelta por **instruction.rs** es dependiente de la operación seleccionada, la función elegida será la que ejecute dicha operación sin posibilidad de que otra función que no corresponde a dicha operación sea ejecutada.

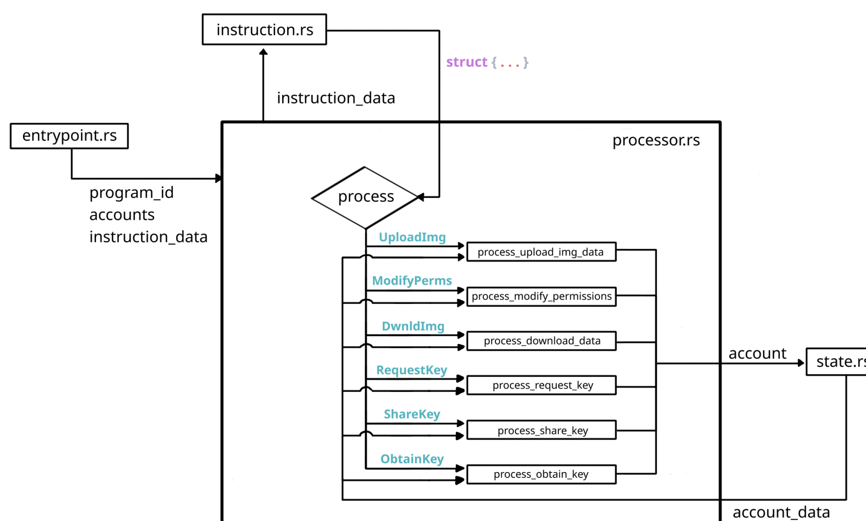

Figura 4.6: Representación del funcionamiento dentro de **processor.rs**

Cada función realiza las acciones necesarias para poder completar exitosamente la operación requerida. Todas estas tienen en común ciertas acciones que se llevan a cabo para evitar el mal uso del contrato inteligente. Dichas acciones son (1) el verificar que la persona interesada en realizar la transacción es quien está firmando la transacción y (2) que el contrato inteligente sea el único que tiene control sobre la cuenta que almacena información.

Cada vez que se ejecute una función, se deberá interactuar con cuentas para poder realizar la operación del linaje de la imagen con la que se está trabajando, para lograr esto, **state.rs** permite obtener información de la cuenta y también subirla a la misma.

4.2.4. state.rs

El almacenamiento de la información en la Blockchain se hace mediante cuentas. Como se mencionó anteriormente, dicha información es una secuencia de bytes que se encuentra serializada utilizando el formato de serialización Borsh. Cada estructura de datos tiene sus reglas de serialización y deserialización implementadas dentro de **state.rs** y llamadas `unpack` y `pack`. Estas funciones permitirán crear las estructuras definidas a partir del arreglo de bytes almacenado en la cuenta dentro de la Blockchain o almacenar dichas estructuras en la misma.

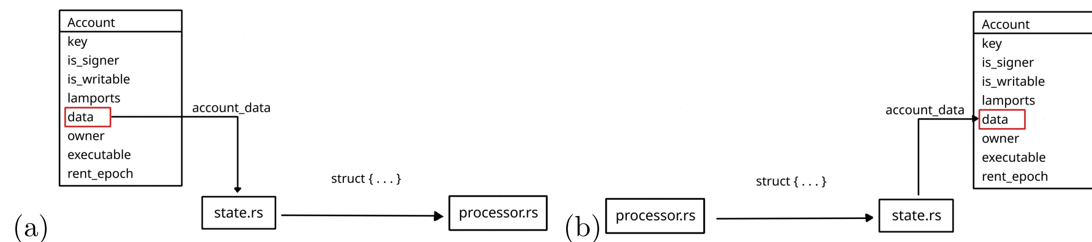

Figura 4.7: (a) Deserialización de la información (unpack) y (b) Serialización de la información (pack).

La función **unpack** (figura 4.7 a) consiste en obtener la información guardada en la cuenta y deserializarla de acuerdo a la estructura de datos que se necesita para ejecutar de manera exitosa la operación seleccionada. La deserialización consiste en obtener los valores de los campos de la estructura a partir de la partición del arreglo de bytes en varios segmentos siguiendo las reglas establecidas para esa estructura en específico, donde cada partición corresponde a un elemento de la estructura. Una vez obtenida la información de la cuenta, la estructura es enviada a **processor.rs** para continuar con la ejecución de la instrucción.

Por otro lado, la función pack (figura 4.7 b) consiste en almacenar la información dentro de la cuenta serializando la estructura de datos que se maneja en **processor.rs**. Para esto se construye el arreglo de bytes pasando cada elemento de la estructura a un arreglo de bytes siguiendo las especificaciones de Borsh y concatenando esos elementos en un solo arreglo. Dicho arreglo es el que se almacena en la cuenta correspondiente.

4.3. Ejemplos de uso

Para ilustrar lo anteriormente expuesto, a continuación se presentarán diferentes ejemplos de uso con finalidad de mostrar el ciclo de vida completo de una imagen y como las diferentes operaciones descritas anteriormente toman lugar.

4.3.1. Subir imagen

Alice desea subir una imagen y que ésta sea pública con la finalidad de compartirla con la comunidad, pero no desea que esta se pueda editar. Lo primero que hará será crear una cuenta para almacenar la información de la imagen a través de la función `createImgDataAccount`, como se muestra en la figura 4.8 (a). Para lograr esto, Alice ingresa tanto su llave pública (en color azul) como una semilla `image1` (la cual se muestra en color verde) y la llave de la cuenta que almacena el contratos inteligente (en color negro), obteniendo como resultado una cuenta (cuya llave pública está en color rojo) lista para recibir información.

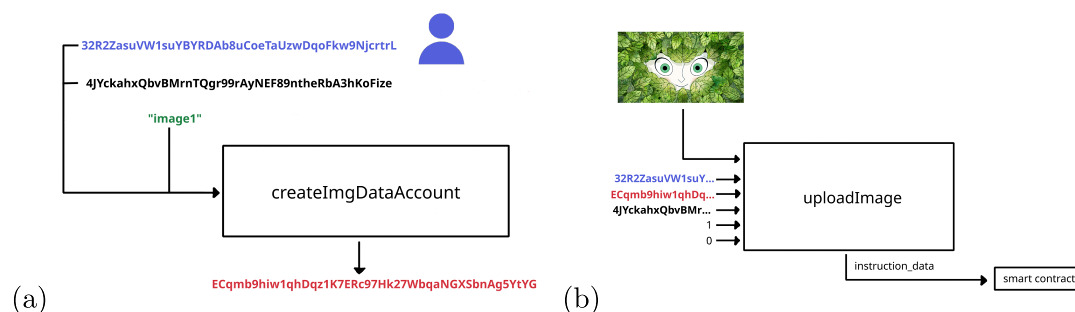

Figura 4.8: (a)Crear cuenta y (b) Subir imagen.

Una vez creada la cuenta de Alice, la imagen y su información se suben a IPFS y a la Blockchain, respectivamente. La figura 4.8 b muestra como Alice ejecuta la función `uploadImage` con su llave pública, la llave pública de la cuenta creada, la imagen y los valores que indican si la imagen es pública (valor 1) y editable (valor 0).

Dentro de esta función se construye un arreglo de bytes, el cual representa los datos de instrucción que se pasa al contrato inteligente. El primer paso es calcular el `cid` de la imagen para posteriormente definir el resto de los datos de instrucción de acuerdo a la operación deseada. Estos pueden consultarse en la tabla 4.2. Como se está subiendo una imagen, el byte que define la instrucción a ejecutar dentro del contratos inteligente corresponde al valor 0 (tabla 4.3). Además, dicha imagen no tiene un padre, por lo que el `cid` de la imagen padre es una cadena de caracteres compuesta por ceros. Los indicadores `child` y `diff` también tienen valores de 0 y, como se mencionó anteriormente, el indicador `public` y el indicador `editable` presentan los valores de 1 y 0, respectivamente. Ya que se cuenta con toda esta información se construye el arreglo de bytes, el cual se muestra en la figura 4.9.

En este arreglo, el primer byte define la instrucción (en color azul marino), seguido por el `cid` de la imagen (color naranja) y el `cid` de la imagen padre (en caso de que exista) que se presenta en color amarillo. Los últimos 4 bytes corresponden a los campos `child`, `diff`, `public` y `editable` que se presentan en los colores azul claro, negro, verde y morado, respectivamente.

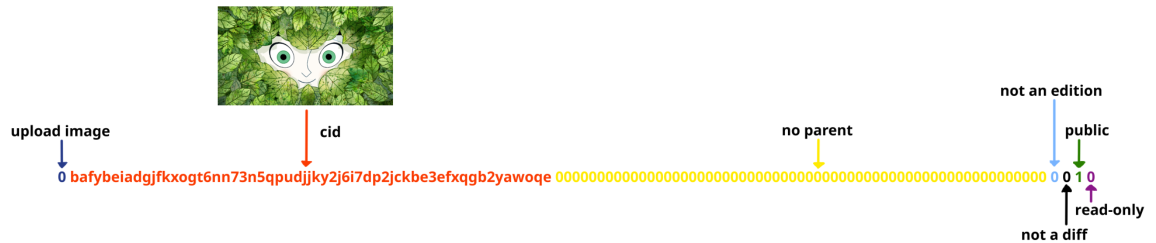

Figura 4.9: Datos de instrucción para la operación subir imagen.

Al momento de que el contrato inteligente recibe los datos de instrucción junto con el identificador de programa y las cuentas involucradas, estos son recibidos por el entry-point y pasan a processor.rs, donde los datos de instrucción se envían a instruction.rs para elegir la instrucción a ejecutar. En la figura 4.10 se muestra como el primer byte con valor igual a 0 indica que se subirá una imagen por lo que el resto del arreglo de bytes es acomodado dentro de la estructura **UploadImg**, la cual regresa a processor.rs para continuar con la ejecución.

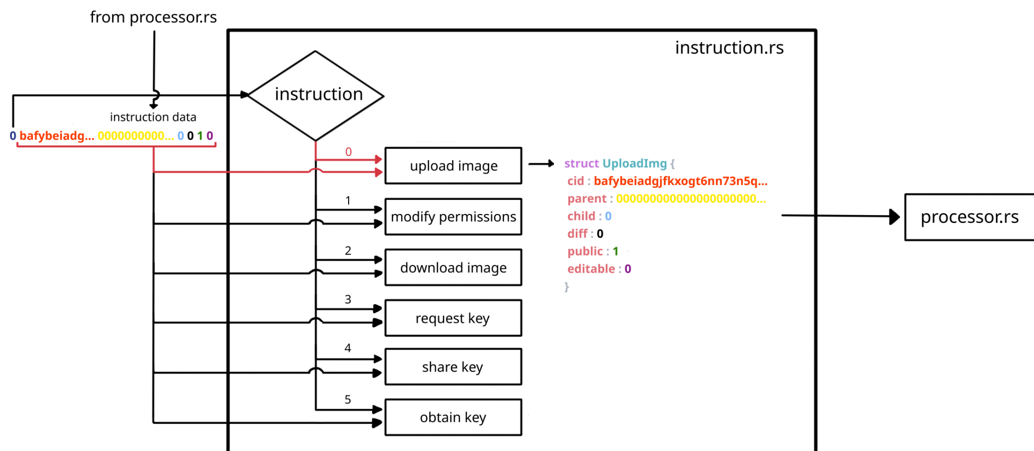

Figura 4.10: Selección de la instrucción upload image.

De nuevo en processor.rs (figura 4.11), se elige qué función se utilizará para ejecutar la instrucción a partir de la estructura proveniente de instruction.rs y como la estructura que se tiene es **UploadImg**, la función que se utilizará es `process_upload_img_data` donde se subirá la información a la cuenta creada para tal propósito. Para subir dicha información a la cuenta es necesario utilizar las reglas de serialización y deserialización adecuadas dentro de state.rs de acuerdo a la estructura de datos que se almacenará en la cuenta, en este caso la estructura **ImgData** mencionada con anterioridad.

Como la operación que se está realizando es la de subir una imagen, se creó una cuenta para subir la información y, por lo tanto, la información disponible al momento

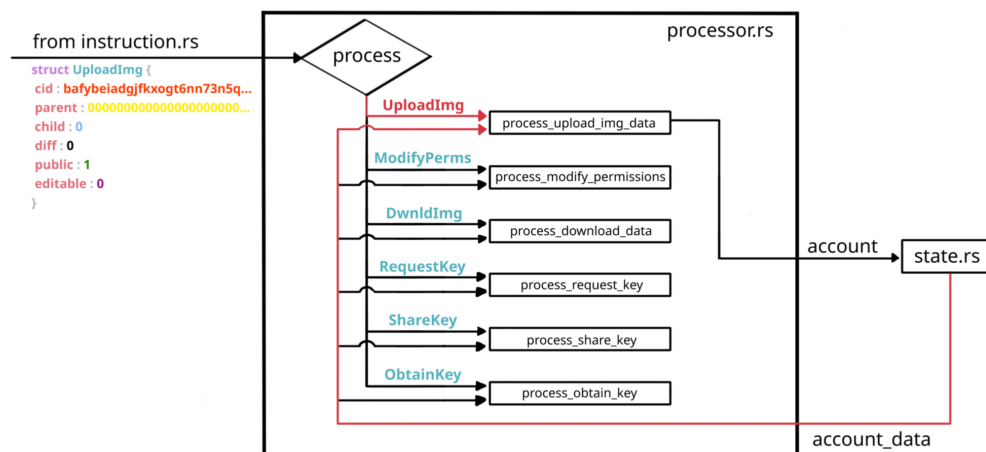

Figura 4.11: Selección de la función process_upload_img_data

es un arreglo de bytes cuyo contenido son ceros. Al ejecutar el método unpack (imagen 4.12 a), después de la deserialización se tendrá la estructura **ImgData** con los valores de sus campos iguales a cero. Posteriormente, en la función `process_upload_img_data` se sustituirán los ceros con la información correspondiente de la imagen para después ejecutar el método pack (imagen 4.12 b) con el fin de serializar y almacenar la información en la cuenta.

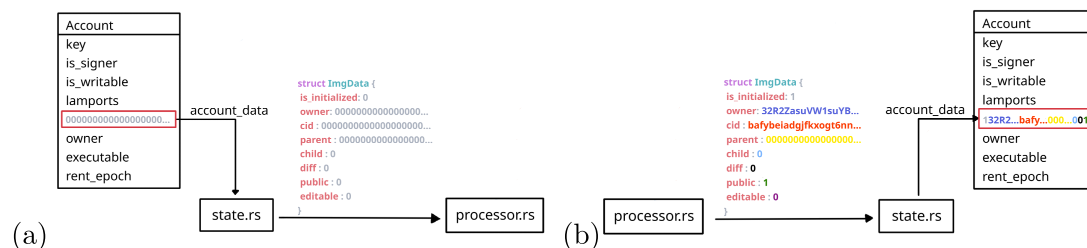

Figura 4.12: (a) Deserialización de la información en la cuenta y (b) Serialización de la información hacia la cuenta.

Al confirmarse la transacción en la Blockchain, de vuelta en la función `uploadImage` se llama a la función `uplaodToIpfs` donde la imagen se sube a IPFS para ser almacenada. Esto garantiza que la imagen sólo se subirá a IPFS sí y sólo sí la transacción se guardó exitosamente en la Blockchain.

4.3.2. Modificar permisos

Supongamos que después de almacenar su imagen en IPFS, Alice cambia de opinión y desea que esta imagen sea privada y hacerla editable por lo que se ejecutará una modificación de permisos. Para esto, Alice llama a la función `permissions` (figura 4.13) e ingresa como parámetros su llave pública, la cuenta que almacena la información de la imagen y los nuevos valores para los indicadores `public` (valor igual a 0) y `editable` (valor igual a 1). Como la imagen cambia de ser pública a ser privada, ésta deberá cifrarse. Para esto se obtiene la imagen desde IPFS con la función `getFromIpfs`, se genera una llave simétrica AES, se cifra la imagen y finalmente se calcula el `cid` de la imagen cifrada. Los datos de instrucción que pasarán ahora al contrato inteligente (figura 4.14) son el número de operación a ejecutar, el `cid` de la imagen cifrada y los campos `public` y `editable`.

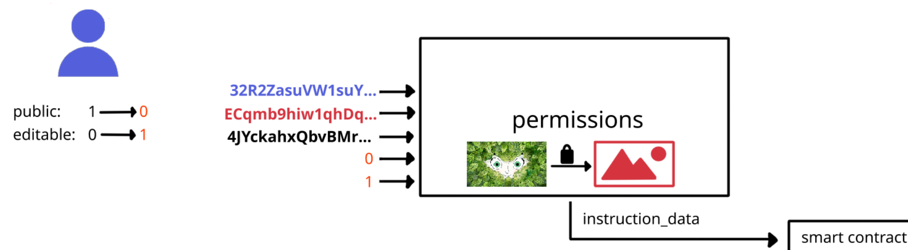

Figura 4.13: Modificación de permisos.

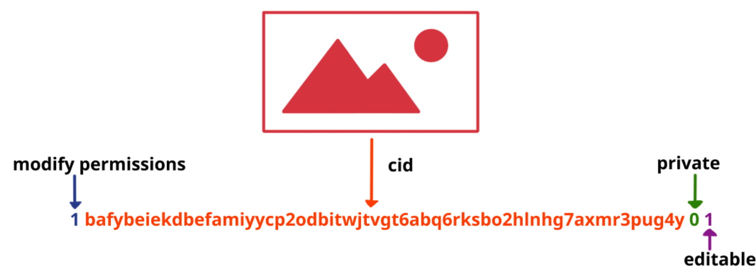

Figura 4.14: Datos de instrucción para la operación modificar permisos.

Dentro del contrato inteligente, el número de operación (cuyo valor es 1) define dentro de instruction.rs que se ejecutará una modificación de permisos generando la estructura **ModifyPerms** que pasará a processor.rs donde esta misma estructura indica que se utilizará la función `process_modify_permissions`.

Como únicamente Alice está modificando los permisos de la imagen, se trabaja con la información almacenada en la cuenta creada anteriormente por ella misma y la estructura que se necesita después de deserializar la información, que es nuevamente

ImgData. En este caso, luego de ejecutar el método unpack se tendrá la estructura con la información de la imagen en lugar de tener un arreglo de ceros como en el caso anterior y en la función `process_modify_permissions` se reemplazarán tanto el `cid` como los campos mencionados anteriormente (subrayados en la figura 4.15). Una vez finalizado esto, el método pack permitirá que la información actualizada de la imagen sea almacenada nuevamente en la cuenta.

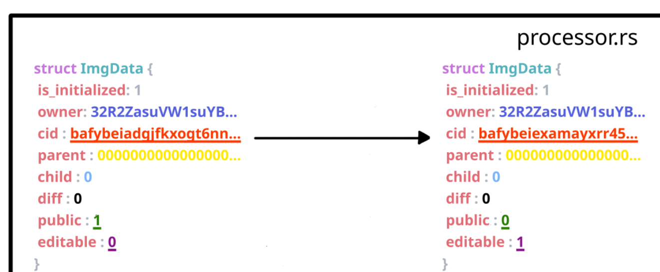

Figura 4.15: Modificación de los campos de interés para modificar permisos.

Al recibir una confirmación por parte de la Blockchain, de vuelta en la función `permissions` se procede a subir la imagen cifrada a IPFS, guardar la llave simétrica utilizada para el cifrado junto con el `cid` de dicho archivo y a eliminar la imagen pública de IPFS mediante la función `removeFromIpfs`.

4.3.3. Compartir llaves

Por su parte, Bob vio la imagen de Alice cuando ésta era pública y ahora que es privada no puede hacerlo, sin embargo quiere seguir teniendo acceso a ella. Para esto, Alice debe compartirle a Bob la llave simétrica con la que cifró la imagen para que él pueda verla mediante el conjunto de funciones `requestKey`, `shareKey` y `obtainKey`.

Primero Bob envia una solicitud de llave mediante la función `requestKey`, ingresando su llave pública, el `cid` de la imagen y la llave pública de una cuenta creada para almacenar la solicitud. Dentro de la función se prepara la llave pública RSA de Bob y se envían los datos de instrucción al contrato inteligente. En este caso el arreglo de bytes está compuesto por el número de operación, la llave pública RSA de Bob y el `cid` de la imagen.

Como el número de operación tiene un valor de 3, la estructura que será enviada a processor.rs es **RequestKey** que corresponde a la instrucción request key y que define la utilización de la función `process_request_key` dentro de processor.rs donde almacenará la información dentro de la cuenta en la Blockchain. En este caso la estructura que se utiliza para almacenar la información es **SecKeyRequest** y, al igual que en el caso anterior, tras ejecutar el método unpack se tendrá que los campos de dicha estructura tienen valores de cero y posteriormente estos valores son reemplazados por la información de la solicitud generada por Bob.

En el momento en que Alice recibe la petición y corrobora que la llave pública del solicitante almacenada en la Blockchain es la llave pública de Bob, ella llama a la función `shareKey` ingresando su llave pública, la llave pública de Bob y la cuenta de la solicitud. Dentro de esta función, se consulta la información de la solicitud almacenada en la Blockchain, de ahí se busca la llave simétrica asociada al `cid` de la imagen y se descifra para que después Alice cifre dicha llave con la llave pública RSA que Bob proporcionó. Con esto, Alice llama al contrato inteligente pasando el arreglo de bytes compuesto por el número de operación y la llave simétrica cifrada mediante RSA.

Dentro del contrato inteligente la instrucción y estructura que corresponden al número de operación 4 son share key y **ShareKey** respectivamente, dando como resultado la utilización de la función `process_share_key`, donde la llave cifrada mediante RSA se coloca en el campo `secKey` de la estructura **SecKeyRequest**.

Al momento en que Alice notifica a Bob que la llave simétrica ya se encuentra en la cuenta, Bob llama a la función `obtainKey`, donde ingresa su llave pública y la llave pública de la cuenta que almacena la información. La función obtiene la llave simétrica cifrada de la cuenta y la descifra con la llave privada de Bob y posteriormente la almacena junto con el `cid` asociado a la imagen cifrada. Al concluir este proceso, se llama al contrato inteligente con los datos de instrucción que son: el número de operación (5) y un byte de confirmación (1). Dentro del contrato inteligente se revisa que el byte de confirmación tenga el valor de 1 para confirmar que la llave fue obtenida y, de ser así, elimina la cuenta que almacena la solicitud al transferir los fondos de ésta a la cuenta de Bob y como dicha cuenta ya no tiene fondos para permanecer en la Blockchain, ésta es eliminada (como se explica a detalle en 2.4).

4.3.4. Descargar Imagen

Como Bob ya cuenta con la llave simétrica para descifrar la imagen de Alice, y dicha imagen cuenta con los permisos para edición (el campo editable de la información almacenada en la Blockchain tiene un valor de 1), Bob procede a acceder a la imagen realizando una descarga de la misma. Para realizar esto sube a la Blockchain la información referente a la descarga a través de la función `downloadImage` que tiene como entradas la llave pública de Bob, la llave pública de la cuenta creada para almacenar la información y la llave pública de la cuenta que almacena la información de la imagen.

Lo primero que se realiza dentro de la función es una verificación de si la imagen es editable o no, posteriormente se llama al contrato inteligente con los datos de instrucción que corresponden al número de instrucción (en este caso el número 2) y al `cid` de la imagen. El contrato inteligente ejecuta la instrucción download donde guarda la información correspondiente a la estructura **DwnldImgLog** en la cuenta creada anteriormente para tal propósito. Una vez que se confirme la transacción, de nuevo en la función `downloadImage` se obtiene el contenido de la imagen mediante la función `getFromIpfs`.

4.3.5. Editar Imagen

Después de descargar la imagen, Bob realiza una modificación a la imagen y desea registrar los cambios para que Alice pueda observarlo. Como se trata de una edición, se llama a la función `editImage` donde ingresa tanto la imagen de Alice como su edición, además de las cuentas que almacenan o almacenarán la información de dichas imágenes. La función crea la diferencia entre ambas imágenes y posteriormente llama dos veces a la función `uploadImage` para subir tanto la imagen editada como la imagen diferencia. De este modo, Bob sube tanto el resultado de la edición como la diferencia a IPFS y su información a la Blockchain para que cualquiera pueda ver que se realizó un cambió a la imagen de Alice.

4.4. Registro de mensajes durante transacciones

Dentro del contrato inteligente es posible generar mensajes que se imprimirán en el registro de la transacción después de que dicha transacción se ejecute y que pueden ser observados cada vez que ésta se consulte. Estos mensajes desplegarán información importante sobre la transacción que ocurrió como el tipo de operación y el `cid` de la imagen involucrada. De este modo será posible dar seguimiento a la historia de la imagen al rastrear todas las operaciones en las que estuvo involucrada dicha imagen. Esto se realiza mediante la consulta de las transacciones buscando que el `cid` de la imagen esté presente en los mensajes generados durante la transacción.

Subir imagen	Modificar permisos	Descargar imagen	Editar Imagen	Solicitar llave	Compartir llave	Obtener llave
Image: {}	Image: {}	Image: {}	Image: {}	Image: {}	Image: {}	Image: {}
Operation: upload	Operation: permissions	Operation: download	Operation: edition	Operation: request key	Operation: share key	Operation: obtain key
	Public: {} -> {}	Image owner: {}	Original Image Owner: {}			
	Editable: {} -> {}		Edition/Diff: {}			
	New Image: {}		Operation: upload			

Figura 4.16: Mensajes generados por tipo de operación.

En la figura 4.16 se presentan los mensajes generados por transacción dependiendo de la operación que se esté realizando (los campos en color rojo son opcionales). En el caso de una modificación de permisos donde se haga una imagen privada, pública o viceversa se registrará el `cid` de la nueva imagen, pero, si no se modifica el campo `public` entonces se omite dicho `cid` ya que este no cambia debido a que la imagen sigue siendo la misma. Por otro lado, al editar una imagen (como se ha mencionado) se subirán tanto una edición como una diferencia por lo que dependiendo del mensaje se tendrá el `cid` de dicha edición o diferencia.

Antes de finalizar la ejecución de alguna de las funciones antes mencionadas dentro de **processor.rs**, se despliegan estos mensajes para ser incluidos en el registro de la transacción. Como en cada transacción queda registrado quién pagó por la misma, no es necesario escribirlo en los mensajes ya que, al momento de consultar la transacción, puede obtenerse quién realizó la operación sobre la imagen al igual que la firma de dicha transacción.

Capítulo 5

Resultados

Como resultado de la implementación presentada en el capítulo anterior, además de la implementación de las operaciones a realizarse a las imágenes, se generó información referente a cualquier operación realizada a estas. Toda esta información se encuentra almacenada en la Blockchain. Al realizar consultas sobre esta información podemos observar los procesos a las que cada imagen ha sido sometida. Además, con el registro de mensajes mencionado en la Sección 4.4 es posible observar la historia de una sola imagen, esto es, su linaje electrónico o *provenance*. Este capítulo describe la forma en la que se pueden consultar los procesos y el linaje electrónico de las imágenes dentro de la implementación del marco de referencia.

5.1. Consulta de transacciones y visualización de imágenes

Existen varias formas de consultar la información que se encuentra dentro de una Blockchain. La mayoría de las redes públicas de Blockchain ofrecen los servicios de los llamados "exploradores". Estos son "software que permite a los usuarios explorar y visualizar tanto bloques como transacciones y proporciona métricas de actividad de la red como las tarifas promedio de transacciones, tamaño de bloques, entre otras" [55]. Blockchains como Cardano [56], Algorand [57, 58] o Ethereum [59] cuentan con exploradores para poder consultar la información antes mencionada. Para el caso de Solana, la red usada en esta implementación, su explorador permite de manera pública visualizar toda la información contenida en cualquiera de sus clusters (mainnet, devnet y testnet). Específicamente, este permite realizar consultas por transacción, cuentas, bloques, contratos inteligentes, entre otros. Por ejemplo, la figura 5.1 muestra los detalles de una transacción que involucra el caso de uso "Subir imagen"mencionado en el capítulo anterior. La primera sección muestra los datos generales de la transacción como la firma de la misma, la marca de tiempo, la tarifa, entre otros. Posteriormente, se muestran las cuentas involucradas, incluyendo también sus detalles y el balance de

SOL con el que cuenta cada una.

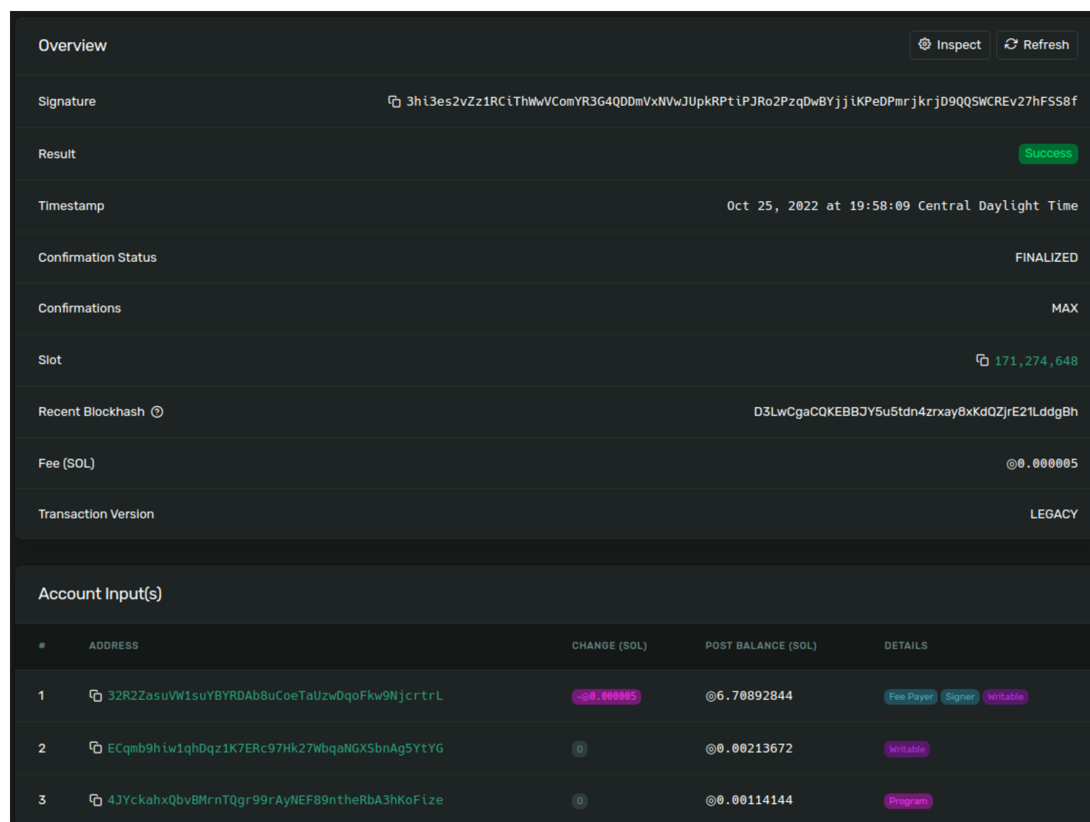

Figura 5.1: Visualización general de una transacción.

Lo siguiente que se observa al desplazarse hacia abajo en el explorador (figura 5.2) son los detalles del contrato inteligente. Nuevamente, se muestran las cuentas involucradas, siendo la primera la cuenta donde se almacena el contrato inteligente, la segunda es la del usuario que está llamando al contrato inteligente y la tercera es la cuenta que almacena la información generada por dicho contrato. Además de las cuentas, se muestran los datos de instrucción (tabla 4.2) que fueron enviados al contrato inteligente en formato hexadecimal. Por último, se muestran los registros generados durante la ejecución de este contrato inteligente. Estos registros muestran qué proceso se está realizando, cuántas unidades de cómputo de la red fueron utilizados y también incluyen el `cid` de la imagen y qué operación se realizó sobre ella.

Para consultar la información de una imagen específica almacenada en la Blockchain se necesita la llave pública de la cuenta. Con esta llave se tiene acceso a la cuenta y a su información que, después de ser deserializada, muestra la estructura de datos correspondiente. Esto se puede ver en la figura 5.3 donde se presenta toda la información almacenada en la Blockchain referente a una imagen.

Como ya se ha mencionado, la imagen como tal no se almacena en la Blockchain

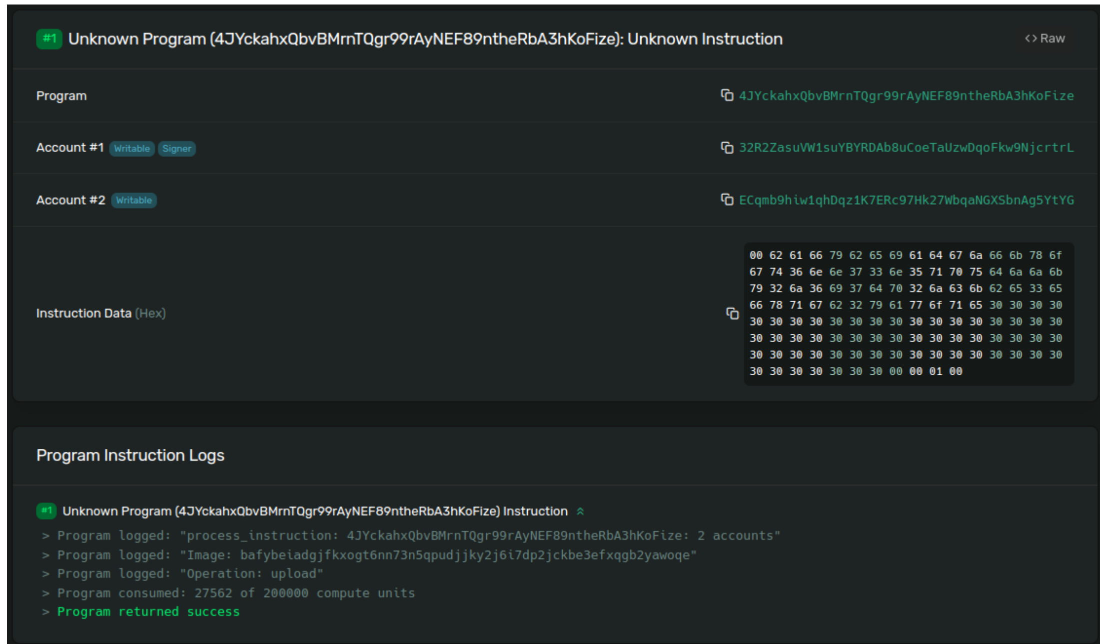

Figura 5.2: Detalles del contrato inteligente.

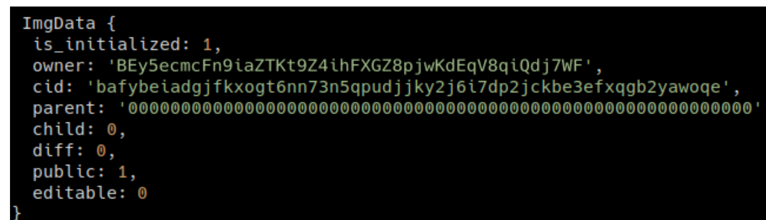

Figura 5.3: Visualización de la información referente a la imagen subida a IPFS

si no sólo su información, dentro la cual se encuentra el `cid`. Este `cid` es un identificador único dentro de IPFS, por lo que la consulta de esta imagen debe hacerse en la red de IPFS. Debido a que el marco de referencia no cuenta con un *front-end* terminado, el ejemplo de una consulta se realiza a través de un navegador con soporte para IPFS, en este caso se utilizó el navegador Brave. El `cid` de la imagen se transforma en una URL anteponiendo la cadena `ipfs://` (figura 2.17) para indicar que se está utilizando el protocolo `ipfs`. En el caso de la figura 5.4, el `cid` de la imagen es `bafybeiadgjfkxogt6nn73n5qpudjjky2j6i7dp2jckbe3efxqgb2yawoqe` y al utilizarlo como URL se puede acceder a la imagen.

Como se puede observar, tanto el explorador de Solana como el navegador con soporte para IPFS nos permiten visualizar la información que se ha creado durante la ejecución de la implementación descrita en el capítulo anterior.

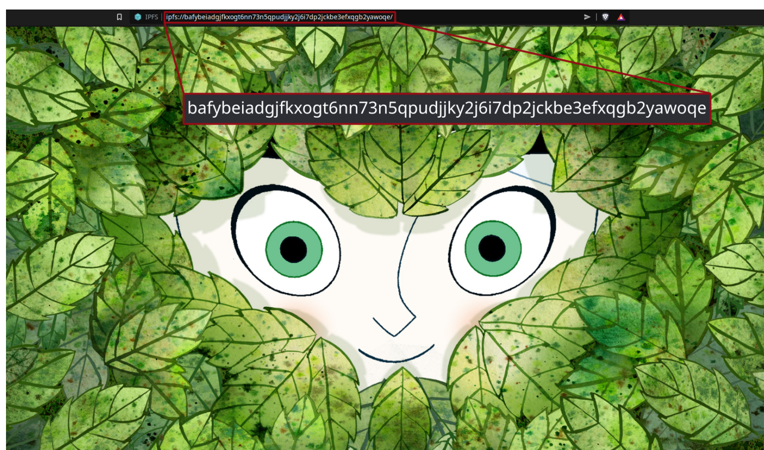

Figura 5.4: Visualización de la imagen subida a IPFS.

5.2. Consulta del linaje electrónico

Si lo que se desea es consultar el linaje electrónico de las imágenes, se debe hacer uso del registro de mensajes mencionado en la Sección 4.4. Estos mensajes generados durante las transacciones representan la documentación de las operaciones realizadas sobre las imágenes y permiten observar la historia de una sola imagen, es decir, su linaje electrónico.

La figura 5.5 muestra una imagen ya almacenada en IPFS y su información correspondiente almacenada en una cuenta dentro de Solana. Al observar con detenimiento la información, destaca el hecho de que el campo `parent` no es una cadena de caracteres compuesta por ceros sino que es un `cid` válido para un archivo y que el campo `child` es igual a 1, por lo que se puede concluir que dicha imagen es el resultado de la edición de otra imagen. Esta imagen se usará para mostrar un ejemplo de cómo consultar el linaje electrónico.

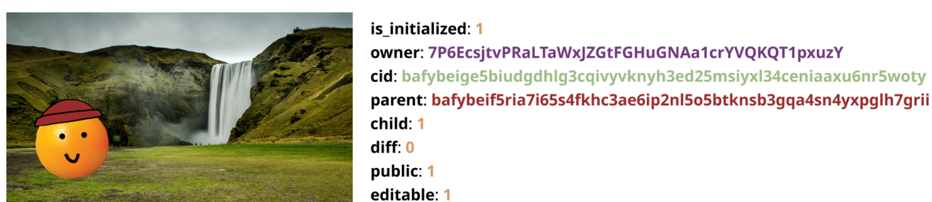

Figura 5.5: La imagen y su información almacenada en la cuenta.

Identificando esto, se realiza una consulta de las transacciones que involucran a la imagen padre a partir de su `cid` (resaltado en color rojo en la figura 5.5), el cual se registra durante la realización de la transacción en forma de un mensaje. La consulta anterior da como resultado un total de tres transacciones: dos ediciones y una descarga.

La figura 5.6 muestra estas transacciones en orden de ocurrencia, la secuencia de eventos se muestra de forma simplificada como una serie de nodos conectados entre sí donde cada color representa una operación diferente. El orden de eventos va de arriba hacia abajo, con el evento más antiguo en la parte superior y el evento más reciente en la parte inferior. Los nodos en color rojo indican la ocurrencia de una edición y los nodos en color morado una descarga. Una edición se representa como una vertiente donde se tienen ahora dos secuencias de eventos, los que se relacionan con la imagen original y los que se relacionan con la imagen editada. Para mostrar qué cambios se han realizado, se muestran tanto la imagen editada como la imagen de diferencia. Por último, se presenta qué usuario realizó la operación sobre la imagen, en este caso se puede ver que el color del usuario que realizó la edición que dio lugar a la imagen de interés coincide con el color de la llave pública en la figura 5.5.

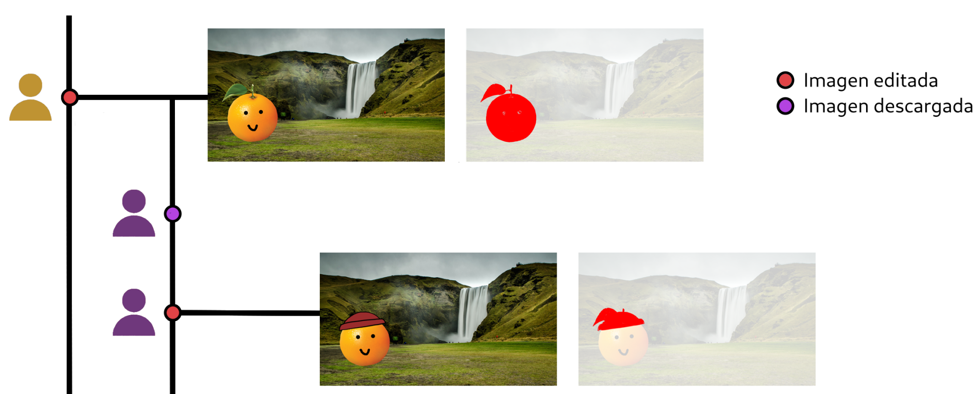

Figura 5.6: Transacciones en las que está involucrada la imagen padre.

Tomando en cuenta lo anterior y observando detenidamente, se tiene la siguiente secuencia de eventos:

1. Edición de una imagen que dio origen a la imagen padre.
2. Descarga de la imagen padre.
3. Edición de la imagen padre que dio origen a la imagen de interés.

Tanto la descarga como la edición que dio origen a la imagen de interés fueron realizadas por el mismo usuario (color morado) mientras que la edición que terminó en la creación de la imagen fue llevada a cabo por el usuario en color dorado.

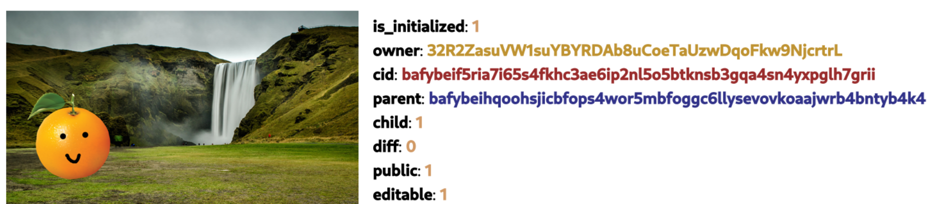

Figura 5.7: La imagen padre y su información almacenada en la cuenta.

Observando ahora la información de la imagen padre en la figura 5.7, se confirma que ésta es producto de una edición, ya que nuevamente presenta un `cid` válido en el campo `parent` y un valor igual a 1 en el campo `child`. La imagen tiene permiso de ser editada (esto debido a que el campo `editable` tiene un valor de 1), lo que muestra congruencia con lo obtenido hasta el momento ya que la imagen padre nunca hubiera podido ser descargada ni editada para dar lugar a la imagen de interés.

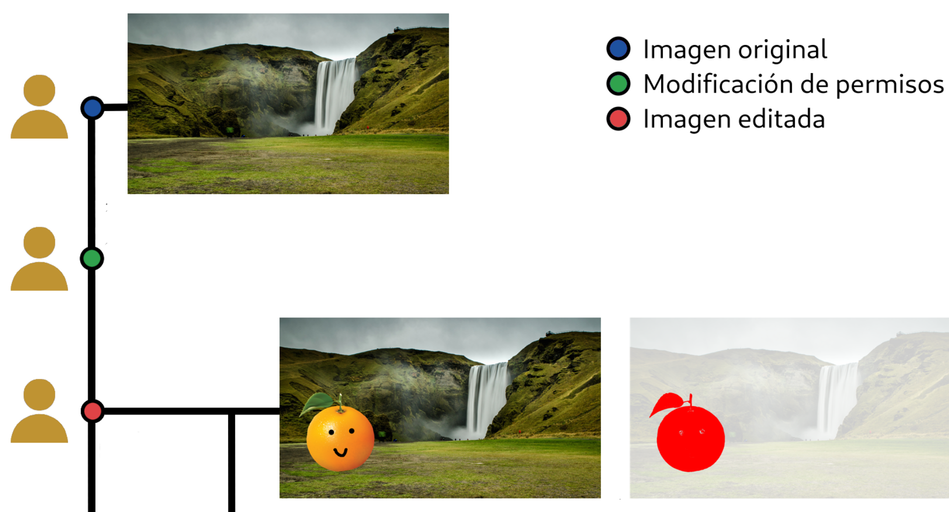

Figura 5.8: Transacciones en las que está involucrada la imagen original.

Consultando ahora las transacciones que involucran a la imagen padre con `cid` de color azul se da origen a la figura 5.8 donde se presentan tres transacciones y cuya secuencia de eventos es la siguiente:

1. Imagen subida al sistema.
2. Modificación de los permisos de la imagen.
3. Edición de la imagen, dando origen a la imagen padre de la imagen de interés.

En este caso se tienen dos nodos adicionales a los ya mencionados, el nodo azul representa cuando una imagen original es subida y el nodo verde indica cuando ocurre

una modificación de permisos. Como se tiene una transacción que corresponde a la operación de subir una imagen original, no hay necesidad de consultar más transacciones porque es en este punto donde inicia la secuencia de eventos que desencadenan en la imagen de interés. Originalmente la imagen fue subida sin un permiso de edición, esto se conoce una vez que se analiza la transacción del nodo verde donde los mensajes generados durante la transacción (figura 5.9) muestran que la modificación de permisos realizó un cambio en el valor del campo `editable`, cambiando del 0 (definido al momento en el que se subió la imagen como solo lectura) a un valor de 1 (editable). El campo `public` no es de interés ya que este no se modificó porque el usuario decidió que la imagen siguiera siendo pública.

Image: bafybeihqoohsjicbfops4wor5mbfoggc6llysevovkoaajwrb4bntyb4k4
Operation: permissions
Public: 1 **->** 1
Editable: 0 **->** 1

Figura 5.9: Mensajes generados durante la transacción de modificación de permisos.

is_initialized: 1
owner: 32R2ZasuVW1suYBYRDAb8uCoeTaUzwDqoFkw9NjcrtrL
cid: bafybeihqoohsjicbfops4wor5mbfoggc6llysevovkoaajwrb4bntyb4k4
parent: 0000000000000000000000000000000000000000000000000000000000000000
child: 0
diff: 0
public: 1
editable: 1

Figura 5.10: La imagen original y su información almacenada en la cuenta.

La figura 5.10 muestra la información correspondiente a la imagen original, aquí se observa como el campo `parent` contiene una cadena de caracteres conformada por ceros y el campo `child` tiene igualmente un valor de 0, confirmando así que esta es la imagen que inicia la secuencia de eventos que dieron origen a la imagen de interés. Como el campo `editable` fue modificado anteriormente aquí se observa el valor de 1 que es resultado de la modificación de permisos discutida previamente y, al igual que con la imagen padre, se muestra consistencia ya que el campo `editable` tiene un valor de 1 y permitió la edición que dio lugar a la secuencia de eventos en la figura 5.6.

Con todo lo explicado anteriormente en esta sección, es posible construir la historia completa (figura 5.11) que inicia con la imagen original y termina con la imagen de interés, dicha historia se dio de la siguiente forma:

1. La imagen original fue subida inicialmente por un usuario (color dorado) como una imagen pública y de solo lectura.

2. El mismo usuario, siendo el propietario de la imagen, decidió hacerla editable mediante una modificación de permisos.

3. El mismo propietario de la imagen realizó una edición, dando lugar a otra imagen.
4. Un segundo usuario (color morado) descarga la imagen editada con el fin de editarla nuevamente.
5. El mismo usuario edita la imagen descargada, dando como resultado a la imagen de interés.

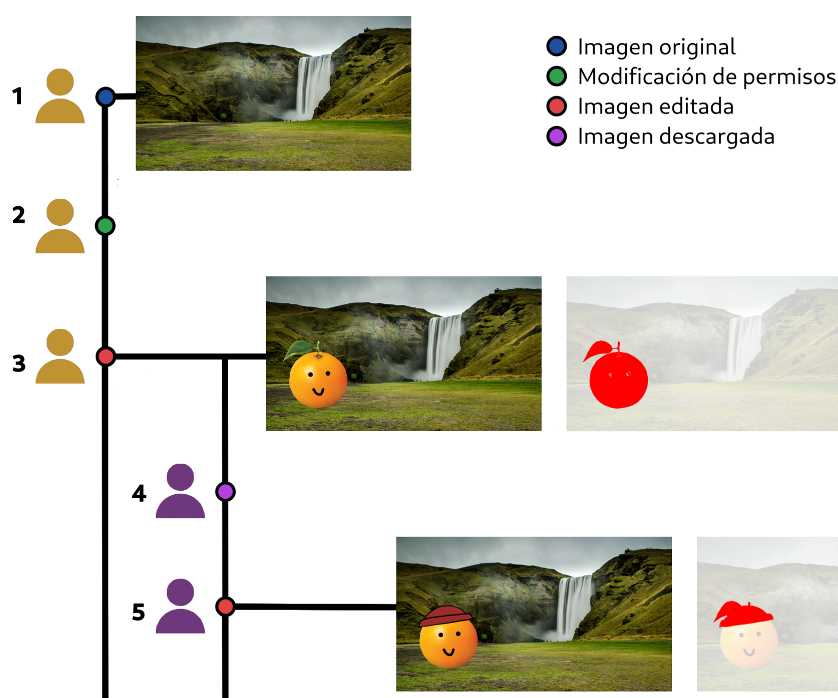

Figura 5.11: Linaje electrónico completo.

Este caso muestra cómo es posible tener trazabilidad y conocer todos los procesos por los que pasa una imagen mediante el uso de su identificador de contenido (`cid`), los registros creados durante las transacciones y los metadatos de imágenes almacenados en cuentas. La forma en la que se recrea la secuencia de eventos puede ser tanto de forma ascendente como descendente. En este caso, se hizo de forma ascendente ya que se partió de una imagen que es una edición y el objetivo era conocer las operaciones previas que dieron origen a dicha imagen pero, de haberse requerido, se podía partir de la imagen original e ir de forma descendente a través de todos los procesos que ocurrieron después de que esta imagen fuera subida al sistema. Las transacciones pueden consultarse mediante el explorador de Solana o a través de la biblioteca `@solana/web3.js` en Typescript y la imagen puede ser consultada en IPFS. De esta forma, es posible conocer los procesos ocurridos a través de Blockchain y, de ser el caso, ver los resultados de dichos procesos en IPFS.

Capítulo 6

Conclusiones

Las personas tienen necesidad de poder controlar las imágenes que comparten en Internet y así evitar daños personales o industriales como consecuencia del uso no autorizado de estas. Almacenar el linaje electrónico de imágenes mediante Blockchain ofrece integridad y no repudio de la información relacionada con la historia de dichas imágenes, con lo cual se le permite a cualquier persona conocer lo que ha ocurrido con su archivo. Sin embargo, conociendo las desventajas relacionadas con ciertas Blockchains, se requiere de una tecnología eficiente, sustentable y que permita el almacenamiento de archivos de gran tamaño.

Se presenta el uso de una Blockchain que utiliza un protocolo de consenso basado en Proof of Stake con la adición del protocolo Proof of History para almacenar el linaje electrónico, mientras que el almacenamiento de las imágenes corre por parte del protocolo de almacenamiento IPFS. Lo anterior permite documentar el historial de procesos de las imágenes con el uso de contratos inteligentes cuyos tiempos de transacción son cortos y su consumo de energía es reducido, además de almacenar dichas imágenes de forma distribuida. También, el utilizar tecnologías distribuidas brinda confianza a la información generada, garantizando el cumplimiento de las medidas de control y trazabilidad implementadas.

Se logró realizar una implementación del esquema propuesto haciendo uso de la Blockchain de Solana a través de los lenguajes de programación Rust para los contratos inteligentes y Typescript para el *front-end* y la comunicación con IPFS. Esta implementación logra registrar y ejecutar las operaciones de subir una imagen, modificar sus permisos, descargarla y editarla, además de proporcionar métodos para generar y compartir llaves para cifrar y descifrar simétricamente imágenes que son clasificadas como privadas. Finalmente, se presentó la consulta del linaje electrónico y su posterior análisis. Este análisis permite verificar de manera efectiva y transparente que es lo que sucedió con cada una de las imágenes dentro del prototipo presentado.

Una de las ideas principales de este marco de referencia es preservar la información de la imagen y registrar todos los procesos por los que dicha imagen pasa. Cualquier Blockchain puede realizar lo anterior por lo que este modelo podría implementarse en alguna otra red de Blockchain mostrando que el marco de referencia presentado en esta

tesis generaliza el problema planteado.

Si bien se muestra que es posible almacenar el linaje electrónico de imágenes y las mismas imágenes de manera distribuida, se asume el uso de este marco de referencia por parte de los usuarios. Si alguien quisiera pasar por alto el marco de referencia buscando algún tipo de ventaja, no habría modo de impedirlo. Las posibles medidas para evitar el comportamiento inadecuado de un usuario salen del alcance de esta investigación y se plantean como posible trabajo a futuro.

Dentro de las áreas de mejora para este trabajo se tiene el desarrollo de un *front-end* que se complemente con el ya realizado para que usuarios finales puedan tener una mejor interacción con el sistema gracias a una interfaz gráfica. Otro tema es el profundizar en la diferencia de imágenes ya que, hasta el momento, solo se tiene una representación visual de los cambios, una idea es el tener un registro de todos los píxeles modificados teniendo en cuenta el valor actual y el valor anterior. También es recomendable prestar atención a la gestión de las cuentas que almacenan la información y asociarlas con el `cid` de la imágenes.

Asimismo, el asegurar que la ejecución del contrato inteligente y la interacción con IPFS sucedan de forma atómica sería un trabajo adicional a futuro que puede realizarse mediante la aplicación de algún patrón o arquitectura ya existente de transacciones distribuidas.

Otros aspectos a considerar son los nodos de IPFS y de Solana, el primero serviría para evitarle a un usuario la responsabilidad de mantener su propio nodo de IPFS utilizando los servicios de terceros como Infura o Pinata aunque esto podría centralizar el almacenamiento de las imágenes. Para Solana existen limitantes de velocidad y solicitudes dependiendo del cluster al que se conecte uno. Estás limitantes podrían hacerse presentes en los clusters gratuitos proporcionados por la fundación Solana a medida que el número de usuarios crezca, por lo que vale la pena considerar la utilización de un nodo proporcionado por un tercero como Quicknode, Figment o Alchemy.

Finalmente, se hace de su conocimiento que este trabajo fue presentado en los congresos ENC2021 y CORE2022. El producto generado del primer evento puede encontrarse en [60], mientras que el segundo se encuentra en proceso de publicación en la revista *Research in Computing Science* que podrá encontrarse en [61].

Apéndice A

Código

A.1. Código Typescript

A.1.1. utils.ts

```
import path from 'path'
import os from 'os'
import fs from 'mz/fs.js'
import yaml from 'yaml'
import { Keypair } from '@solana/web3.js'

async function getConfig(): Promise<any> {
  // Path to Solana CLI config file
  const CONFIG_FILE_PATH = path.resolve(
    os.homedir(),
    '.config',
    'solana',
    'cli',
    'config.yml',
  );
  const configYml = await fs.readFile(CONFIG_FILE_PATH, {encoding: 'utf8'});
  return yaml.parse(configYml);
}

export async function getRpcUrl(): Promise<string> {
  try {
    const config = await getConfig();
    if (!config.json_rpc_url) throw new Error('Missing RPC URL');
    return config.json_rpc_url;
  } catch (err) {
    console.warn(
```

```
27        'Failed to read RPC url from CLI config file, falling back to localhost',
28      );
29      return 'http://localhost:8899';
30    }
31  }
32
33  export async function createKeypairFromFile(
34    filePath: string,
35  ): Promise<Keypair> {
36    const secretKeyString = await fs.readFile(filePath, {encoding: 'utf8'});
37    const secretKey = Uint8Array.from(JSON.parse(secretKeyString));
38    return Keypair.fromSecretKey(secretKey);
39  }
40
41  export async function getPayer(): Promise<Keypair> {
42    try {
43      const config = await getConfig();
44      if (!config.keypair_path) throw new Error('Missing keypair path');
45      return await createKeypairFromFile(config.keypair_path);
46    } catch (err) {
47      console.warn(
48        'Failed to create keypair from CLI config file, falling back to new random keypair',
49      );
50      return Keypair.generate();
51    }
52  }
```

A.1.2. cryptography.ts

```
import fs from 'mz/fs.js'
import os from 'os'
import path from 'path'
import crypto from 'crypto'
import inquirer from 'inquirer'

export async function keyGen(): Promise<Buffer> {
  return new Promise((resolve, reject) => {
    crypto.generateKey('aes', {length: 256}, (err, key) => {
      if(err) reject(err)
        resolve(key.export())
    })
  })
}

export async function requestPsswd(): Promise<string> {
  const answer = await inquirer.prompt({
    name: 'psswd',
    type: 'password',
    message: 'Password:',
    default() {
      return ''
    },
  })
  return answer.psswd
}

export async function configInit(): Promise<void> {
  const dirPath = path.resolve(os.homedir(), '.imgprov')
  fs.access(dirPath, (err) => {
    if (err) {
      console.log('Creating directory')
      fs.mkdir(dirPath, (err) => {
        if (err) throw err
      })
    } else {
      console.log('Directory already exists')
    }
  })

  let salt = new Uint8Array(16)
  console.log('Please enter a password in order to continue configuration')
  const psswdFilePath = path.resolve(dirPath,'user.psswd')
```

```
44    salt = await new Promise(async(resolve, reject) => {
45      crypto.randomFill(new Uint8Array(16), (err, salt) => {
46        if (err) reject(err)
47        resolve(salt)
48      })
49    })
50
51    fs.stat(psswdFilePath, async(exists) => {
52      if (exists == null) {
53        console.log('Password file already exists')
54      } else if (exists.code === 'ENOENT') {
55        await setPsswd(salt, psswdFilePath)
56        console.log('Password file created!')
57      }
58    })
59
60    const keyFilePath = path.resolve(dirPath, 'user.keys')
61    fs.stat(keyFilePath, async(exists) => {
62      if (exists == null) {
63        console.log('Keys file already exists')
64      } else if (exists.code === 'ENOENT'){
65        fs.open(keyFilePath, 'w', (err, fd) => {
66          if (err) throw err
67          fs.close(fd, (err) => {
68            if (err) throw err
69          })
70        })
71      }
72    })
73  }
74
75  export async function setPsswd(salt: Uint8Array, psswdFilePath: string): Promise<void> {
76    let condition = false
77    while (condition == false) {
78      const psswd = await requestPsswd()
79                  console.log('chi')
80      let firstPsswd = new Uint8Array(64)
81      firstPsswd = await new Promise((resolve, reject) => {
82        crypto.pbkdf2(psswd, salt, 100000, 64, 'sha512', (err, psswdHash) => {
83          if (err) reject(err)
84          resolve(psswdHash)
85        })
86      })
87
88      console.log('Confirm password')
```

```
89      const confirmPsswd = await requestPsswd()
90      let secondPsswd = new Uint8Array(64)
91      secondPsswd = await new Promise((resolve, reject) => {
92        crypto.pbkdf2(confirmPsswd, salt, 100000, 64, 'sha512', (err, psswdHash) => {
93          if (err) reject(err)
94          resolve(psswdHash)
95        })
96      })
97
98      if (firstPsswd.toString() === secondPsswd.toString()) {
99        let array: any[][] = [
100         [Buffer.from(salt), firstPsswd],
101       ]
102       fs.writeFile(psswdFilePath, JSON.stringify(array), (err) => {
103         if (err) throw err
104       })
105       condition = true
106     } else {
107       console.log("Passwords don't match, please try again")
108     }
109   }
110 }
111
112 export async function keyPairGen(): Promise <void> {
113   const keyFilePath = path.resolve(os.homedir(),'.imgprov','user.keys')
114   let keys: any[][] = [[]]
115   keys = await new Promise((resolve, reject) => {
116     crypto.generateKeyPair('rsa', {
117       modulusLength: 4096,
118       publicKeyEncoding: {
119         type: 'spki',
120         format: 'pem'
121       },
122       privateKeyEncoding: {
123         type: 'pkcs8',
124         format: 'pem'
125       }
126     }, (err, publicKey, privateKey) => {
127       if (err) reject(err)
128       let keys = [
129         ['public key', publicKey],
130         ['private key', privateKey]
131       ]
132       const jsonKeys = JSON.parse(Buffer.from(JSON.stringify(keys)).toString())
133       jsonKeys[0][1] = Buffer.from(jsonKeys[0][1])
```

```
134         jsonKeys[1][1] = Buffer.from(jsonKeys[1][1])
135         resolve(jsonKeys)
136       })
137     })
138     keys[0][1] = await encryptKey(keys[0][1])
139     keys[1][1] = await encryptKey(keys[1][1])
140 
141     fs.writeFile(keyFilePath, JSON.stringify(keys), (err) => {
142       if (err) throw err
143     })
144     console.log('Keypair generated!')
145   }
146 
147   export async function checkPsswd(psswd: Buffer, salt: Buffer): Promise<string> {
148     let correctPsswd = false
149     let password = ''
150     while (correctPsswd == false) {
151       password = await requestPsswd()
152       let firstPsswd = new Uint8Array(64)
153       firstPsswd = await new Promise((resolve, reject) => {
154         crypto.pbkdf2(password, salt, 100000, 64, 'sha512', (err, psswdHash) => {
155           if(err) reject(err)
156           resolve(psswdHash)
157         })
158       })
159       if (psswd.toString() === firstPsswd.toString()) {
160         correctPsswd = true
161       } else {
162         console.log('Incorrect password, please try again')
163       }
164     }
165     return password
166   }
167 
168   export async function saveKey(cid: string, key: Buffer): Promise<void> {
169     const keyFilePath = path.resolve(os.homedir(), '.imgprov', 'user.keys')
170 
171     let keys: any[][] = [[]]
172     keys = await new Promise((resolve, reject) => {
173       fs.readFile(keyFilePath, (err, data) => {
174         if(err) reject(err)
175         resolve(JSON.parse(data.toString()))
176       })
177     })
178     const encryptedKey = await encryptKey(key)
```

```
179   const entry = [cid, {type: 'Buffer', data: Array.from(encryptedKey)}]
180   keys.push(entry)
181
182   fs.writeFile(keyFilePath, JSON.stringify(keys), (err) => {
183     if(err) throw err
184   })
185 }
186
187 export async function retrieveKey(cid: string): Promise<Buffer> {
188   const keyFilePath = path.resolve(os.homedir(), '.imgprov', 'user.keys')
189
190   let keys: any[][] = [[]]
191   keys = await new Promise((resolve, reject) => {
192     fs.readFile(keyFilePath, (err, data) => {
193       if(err) reject(err)
194       resolve(JSON.parse(data.toString()))
195     })
196   })
197   let idx = 0
198   for (let i = 0; i < keys.length; i++) {
199     if(keys[i][0] == cid) {
200       idx = i
201     }
202   }
203   return Buffer.from(keys[idx][1].data)
204 }
205
206 export async function removeKey(cid: string): Promise<void> {
207   const keyFilePath = path.resolve(os.homedir(), ',imgprov', 'user.keys')
208
209   let keys: any[][] = [[]]
210   keys = await new Promise((resolve, reject) => {
211     fs.readFile(keyFilePath, (err, data) => {
212       if(err) reject(err)
213       resolve(JSON.parse(data.toString()))
214     })
215   })
216
217   let key = 0
218   while (keys[key][0] != cid) {
219     key = key + 1
220   }
221   keys.splice(key, 1)
222
223   fs.writeFile(keyFilePath, JSON.stringify(keys), (err) => {
```

```
224      if(err) throw err
225    })
226  }
227
228  export async function encryptImage(image: Buffer, key: Buffer): Promise<Buffer> {
229    const algorithm = 'aes-256-gcm'
230    return new Promise((resolve, reject) => {
231      crypto.randomFill(new Uint8Array(16), (err, iv) => {
232        if(err) reject(err)
233        const cipher = crypto.createCipheriv(algorithm, key, iv)
234        let encrypted = cipher.update(image)
235        cipher.final()
236        const encryptedImage = Buffer.concat([Buffer.from(iv), encrypted])
237        resolve(encryptedImage)
238      })
239    })
240  }
241
242  export async function encryptKey(plainKey: string | Buffer): Promise<Buffer> {
243    const psswdFilePath = path.resolve(os.homedir(), '.imgprov', 'user.psswd')
244    let sltPsswd = ''
245    sltPsswd = await new Promise((resolve, reject) => {
246      fs.readFile(psswdFilePath, (err, data) => {
247        if(err) reject(err)
248        resolve(data.toString())
249      })
250    })
251    const salt = Buffer.from(JSON.parse(sltPsswd)[0][0])
252    const psswd = Buffer.from(JSON.parse(sltPsswd)[0][1])
253
254    const password = await checkPsswd(psswd, salt)
255    const algorithm = 'aes-256-gcm'
256    return new Promise((resolve, reject) => {
257      crypto.scrypt(password, salt, 32, (err, key) => {
258        if(err) reject(err)
259        crypto.randomFill(new Uint8Array(16), (err, iv) => {
260          if(err) throw err
261          const cipher = crypto.createCipheriv(algorithm, key, iv)
262
263          let encrypted = cipher.update(plainKey)
264          cipher.final()
265          const encryptedKey = Buffer.concat([Buffer.from(iv), encrypted])
266          resolve(encryptedKey)
267        })
268      })
```

```
269    })
270  }
271
272  export async function decryptImage(encryptedImage: Buffer, key: Buffer): Promise<Buffer> {
273    const iv = encryptedImage.subarray(0, 16)
274    const encrypted = encryptedImage.subarray(16, encryptedImage.length)
275    const algorithm = 'aes-256-gcm'
276    const decipher = crypto.createDecipheriv(algorithm, key, iv)
277    const decrypted = decipher.update(encrypted)
278
279    return decrypted
280  }
281
282  export async function decryptKey(encryptedKey: Buffer): Promise<Buffer> {
283    const psswdFilePath = path.resolve(os.homedir(), '.imgprov', 'user.psswd')
284    let sltPsswd = ''
285    sltPsswd = await new Promise((resolve, reject) => {
286      fs.readFile(psswdFilePath, (err, data) => {
287        if(err) reject(err)
288        resolve(data.toString())
289      })
290    })
291    const salt = Buffer.from(JSON.parse(sltPsswd)[0][0])
292    const psswd = Buffer.from(JSON.parse(sltPsswd)[0][1])
293    const password = await checkPsswd(psswd, salt)
294    const iv = encryptedKey.subarray(0,16)
295    const encrypted = encryptedKey.subarray(16, encryptedKey.length)
296    const algorithm = 'aes-256-gcm'
297
298    return new Promise((resolve, reject) => {
299      crypto.scrypt(password, salt, 32, (err, key) => {
300        if(err) reject(err)
301        const decipher = crypto.createDecipheriv(algorithm, key, iv)
302        const decrypted = decipher.update(encrypted)
303        resolve(decrypted)
304      })
305    })
306  }
```

A.1.3. solana-com.ts

```
import path from 'path'
import os from 'os'
import fs from 'mz/fs.js'
import borsh from 'borsh'
import { Connection,
  Keypair,
  PublicKey,
  sendAndConfirmTransaction,
  SystemProgram,
  Transaction,
  TransactionInstruction
} from '@solana/web3.js'
import { createKeypairFromFile, getRpcUrl } from './utils.js'
import { decryptImage,
  decryptKey,
  encryptImage,
  keyGen,
  removeKey,
  retrieveKey,
  saveKey
} from './cryptography.js'
import * as IPFS from 'ipfs'
import {concat as uint8ArrayConcat} from 'uint8arrays/concat'
import all from 'it-all'
import { privateDecrypt, publicEncrypt } from 'crypto'
import jimp from 'jimp'

let connection: Connection
let payer: Keypair
let programId: PublicKey
let imgDataPubkey: PublicKey
let dwnldLogPubkey: PublicKey
let childImgDataPubkey: PublicKey
let diffImgDataPubkey: PublicKey
let secKeyRequestPubkey: PublicKey


const PROGRAM_PATH = path.resolve(path.relative('.','src/program-rust/target/deploy'))
const PROGRAM_KEYPAIR_PATH = path.join(PROGRAM_PATH, 'test-keypair.json')
const PROGRAM_SO_PATH = path.join(PROGRAM_PATH, 'test.so')

let ipfsReady = false
let ipfs: any
```

```
44  const DOWNLOADS_PATH = path.resolve(os.homedir(), 'Imágenes', 'imgprov_dwnlds')
45
46  class ImgData {
47    is_initialized = 1
48    owner = Keypair.generate().publicKey.toString()
49    cid = '00000000000000000000000000000000000000000000000000000000000'
50    parent = '00000000000000000000000000000000000000000000000000000000000'
51    child = 1
52    diff = 1
53    public = 1
54    editable = 1
55    constructor (fields: {
56      is_initialized: number,
57      owner: string,
58      cid: string,
59      parent: string,
60      child: number,
61      diff: number,
62      public: number,
63      editable: number
64    } | undefined = undefined) {
65      if (fields) {
66        this.is_initialized = fields.is_initialized
67        this.owner = fields.owner
68        this.cid = fields.cid
69        this.parent = fields.parent
70        this.child = fields.child
71        this.diff = fields.diff
72        this.public = fields.public
73        this.editable = fields.editable
74      }
75    }
76  }
77
78  const ImgDataSchema = new Map([
79    [ImgData, {
80      kind: 'struct',
81      fields: [
82        ['is_initialized', 'u8'],
83        ['owner', 'String'],
84        ['cid', 'String'],
85        ['parent', 'String'],
86        ['child', 'u8'],
87        ['diff', 'u8'],
88        ['public', 'u8'],
```

```
89         ['editable', 'u8']
90       ]
91     }],
92   ])
93
94   const IMGDATA_SIZE = borsh.serialize(
95     ImgDataSchema,
96     new ImgData(),
97   ).length
98
99   class DwnldLog {
100    is_initialized = 1;
101    downloader = Keypair.generate().publicKey.toString()
102    cid = '00000000000000000000000000000000000000000000000000000000000'
103    constructor (fields: {
104      is_initialized: number,
105      downloader: string,
106      cid: string
107    } | undefined = undefined) {
108      if (fields) {
109        this.is_initialized = fields.is_initialized
110        this.downloader = fields.downloader
111        this.cid = fields.cid
112      }
113    }
114  }
115
116  const DwnldLogSchema = new Map([
117    [DwnldLog, {
118      kind:'struct',
119      fields: [
120        ['is_initialized', 'u8'],
121        ['downloader', 'String'],
122        ['cid', 'String']
123      ]
124    }],
125  ])
126
127  const DWNLDLOG_SIZE = borsh.serialize(
128    DwnldLogSchema,
129    new DwnldLog(),
130  ).length
131
132  class SecKeyRequest {
133    is_initialized = 1
```

```
134    petitioner = Keypair.generate().publicKey.toString()
135    ecPubKey = '-----BEGIN PUBLIC KEY-----\n'+
136    'MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAzNwsfxa5QEtPOdNjRtua\n'+
137    'GtIsVJ6TJOYo3qTOx9fSGvBAznoau4KjjrUh5P1b15uSc71cGlzCFkJeGMOOhJJl\n'+
138    'f/zmlkhVwyfDwv1QKa6ABzJtTA7NOsSYSYqgDytaIS7e1u71qyI2M9QitS9dYc6p\n'+
139    '1OKrAk3OQOHN4ghuIoTRSlaaGBX1a7AOj/33un36tBpzuDH2NkPF7eDPms7Gj51M\n'+
140    'Sw0ehz9grA1JGwdmF02BCJV96Go7yrbM81JtIxZflnddMnspJzofajtTORxWZ7/4\n'+
141    'sHBaUllg71lPoXgZLjq8tP0ZezcX02cA+IqdGN4wAtYGDlfmSsqkeICuNPoQsg/o\n'+
142    'nncy13JA8VYIEZ4X3ILFC8JN12g3HeME1CkLGu9GAn8NTrQYSLIICdMCC6/3YJqZ\n'+
143    'qYgOXUzSa6hRXWMTLmy5wSw09/zQVdA0h//2bPjqCQBcvBrBRA5S9AY0wClxnee/\n'+
144    'Ho4xvDFfW7u3rx4YfgDkmrcwl7sMAgANM8eNTfNj37QTnbV8/WBurfYJScBcKn49\n'+
145    '3YG+Sb76fsbveeoZojq+S1cZU2p3NhvlHKxLNS+zjty/REP/ctnNRgHJCLgjoEHr\n'+
146    'KdcNOGMK5MUJaVipbkRAxdYztby9lRi9Q/5QxYKUzmTDH0+n73EJkj8K1/PYxS30\n'+
147    'fyK+zMrblJQ5/qHHDyh9QLOCAwEAAQ==\n'+
148    '-----END PUBLIC KEY-----\n'
149    cid = '00000000000000000000000000000000000000000000000000000000'
150    secKey = new Array(512)
151    constructor (fields: {
152      is_initialized: number,
153      petitioner: string,
154      ecPubKey: string,
155      cid: string,
156      secKey: [512]
157    } | undefined = undefined) {
158      if (fields) {
159        this.is_initialized = fields.is_initialized
160        this.petitioner = fields.petitioner
161        this.ecPubKey = fields.ecPubKey
162        this.cid = fields.cid
163        this.secKey = fields.secKey
164      }
165    }
166  }
167
168  const SecKeyRequestSchema = new Map([
169    [SecKeyRequest, {
170      kind: 'struct',
171      fields: [
172        ['is_initialized', 'u8'],
173        ['petitioner', 'String'],
174        ['ecPubKey', 'String'],
175        ['cid', 'String'],
176        ['secKey', [512]]
177      ]
178    }],
```

```
179 ])
180
181 const SECKEYREQUEST_SIZE = borsh.serialize(
182   SecKeyRequestSchema,
183   new SecKeyRequest(),
184 ).length
185
186 export async function establishConnection(): Promise<void> {
187   const rpcUrl = await getRpcUrl()
188   connection = new Connection(rpcUrl, 'confirmed')
189   const version = await connection.getVersion()
190   console.log('Connection to cluster established:', rpcUrl, version)
191 }
192
193 export async function checkProgram(): Promise<void> {
194   try {
195     const programKeypair = await createKeypairFromFile(PROGRAM_KEYPAIR_PATH)
196     programId = programKeypair.publicKey
197   } catch (err) {
198     const errMsg = (err as Error).message
199     throw new Error(
200       `Failed to read program keypair at '${PROGRAM_KEYPAIR_PATH}' due to error: ${errMsg}.`,
201     );
202   }
203
204   const programInfo = await connection.getAccountInfo(programId);
205   if (programInfo === null) {
206     if (fs.existsSync(PROGRAM_SO_PATH)) {
207       throw new Error(
208         'Program needs to be deployed with `solana program deploy`',
209       );
210     } else {
211       throw new Error('Program needs to be built and deployed');
212     }
213   } else if (!programInfo.executable) {
214     throw new Error('Program is not executable');
215   }
216   console.log(`Using program ${programId.toBase58()}`);
217 }
218
219 async function getCid(file: string): Promise<string>
220 async function getCid(file: Buffer): Promise<string>
221
222 async function getCid(file: string | Buffer): Promise<string> {
223     if (!ipfsReady ) {
```

```
224            ipfs = await IPFS.create()
225            ipfsReady = true
226        }
227        let fileBuffer: Buffer | undefined = undefined
228        if (typeof file === "string") {
229            fileBuffer = fs.readFileSync(file)
230        } else {
231            fileBuffer = file
232        }
233        const {cid} = await ipfs.add(
234            {content: fileBuffer},
235            {cidVersion: 1},
236            {onlyHash: true},
237        )
238        ipfs.stop()
239        return cid.toString()
240    }
241
242    async function uploadToIpfs(file: string): Promise<void>
243    async function uploadToIpfs(file: Buffer): Promise<void>
244
245    async function uploadToIpfs(file: string | Buffer): Promise<void> {
246        if (!ipfsReady) {
247            ipfs = await IPFS.create()
248            ipfsReady = true
249        }
250        let fileBuffer: Buffer | undefined = undefined
251        if (typeof file === "string") {
252            fileBuffer = fs.readFileSync(file)
253        } else {
254            fileBuffer = file
255        }
256        await ipfs.add({content: fileBuffer}, {cidVersion: 1}, {pin: true})
257        await ipfs.stop()
258        console.log('Image uploaded to ipfs!')
259    }
260
261    async function getFromIpfs(cid: string): Promise<Buffer> {
262        if (!ipfsReady) {
263            ipfs = await IPFS.create()
264            ipfsReady = true
265        }
266        const data = Buffer.from(uint8ArrayConcat(await all(ipfs.cat(cid))))
267        await ipfs.stop()
268        return data
```

```
269 }
270
271 async function removeFromIpfs(cid: string): Promise<void> {
272     if (!ipfsReady) {
273         ipfs = await IPFS.create()
274         ipfsReady = true
275     }
276     await ipfs.pin.rm(cid)
277     await ipfs.repo.gc()
278     await ipfs.stop()
279 }
280
281
282 export async function createAccount(
283   payer: Keypair,
284   seed: string,
285   operation:string
286 ): Promise<PublicKey> {
287   const accountPubkey = await PublicKey.createWithSeed(payer.publicKey, seed, programId)
288
289   let size = 0
290   let op = ''
291
292   if (operation === 'upload') {
293     size = IMGDATA_SIZE
294     op = 'to save image data'
295   } else if (operation === 'download') {
296     size = DWNLDLOG_SIZE
297     op = 'to save download log'
298   } else if (operation === 'request') {
299     size = SECKEYREQUEST_SIZE
300     op = 'to request for secret key'
301   }
302
303   const createdAccount = await connection.getAccountInfo(accountPubkey)
304   if (createdAccount === null) {
305     console.log('Creating account ', accountPubkey.toBase58(), op)
306     const lamports = await connection.getMinimumBalanceForRentExemption(size)
307
308     const transaction = new Transaction().add(
309       SystemProgram.createAccountWithSeed({
310         basePubkey: payer.publicKey,
311         fromPubkey: payer.publicKey,
312         lamports: lamports,
313         newAccountPubkey: accountPubkey,
```

```
314         programId: programId,
315         seed: seed,
316         space: size,
317       }),
318     )
319     await sendAndConfirmTransaction(connection, transaction, [payer])
320   }
321   return accountPubkey
322 }
323 
324 export async function uploadImage(
325   payer: Keypair,
326   imgDataPubkey: PublicKey,
327   file: string,
328   pub: number,
329   edit: number
330 ): Promise<void>
331 export async function uploadImage(
332   payer: Keypair,
333   imgDataPubkey: PublicKey,
334   file: string | Buffer,
335   pub: number,
336   edit: number,
337   parent: string,
338   child: number,
339   diff: number,
340   originalImgAccount: PublicKey
341 ): Promise <void>
342 export async function uploadImage(
343   payer: Keypair,
344   imgDataPubkey: PublicKey,
345   file: string | Buffer,
346   pub: number,
347   edit: number,
348   parent?: string,
349   child?: number,
350   diff?: number,
351   originalImgAccount?: PublicKey
352 ): Promise<void> {
353   console.log('Uploading image and saving image data into account', imgDataPubkey.toBase58())
354   let cid = ''
355   let encryptedImage = Buffer.from('')
356   let key = Buffer.from('')
357   let data = Buffer.from('')
358   let ogAccount: any = ''
```

```
359     if (pub === 1) {
360       if (typeof file === 'string') {
361         cid = await getCid(file)
362       } else {
363         cid = await getCid(file)
364       }
365     } else if (pub === 0) {
366       if (typeof file === 'string') {
367         data = fs.readFileSync(file)
368       } else {
369         data = file
370       }
371       key = await keyGen()
372       encryptedImage = await encryptImage(data, key)
373       cid = await getCid(encryptedImage)
374     }
375
376     let instruction: TransactionInstruction
377     if (parent) {
378       ogAccount = originalImgAccount
379       instruction = new TransactionInstruction({
380         keys: [
381           {pubkey: payer.publicKey, isSigner: true, isWritable: false},
382           {pubkey: imgDataPubkey, isSigner: false, isWritable: true},
383           {pubkey: ogAccount, isSigner: false, isWritable: false}
384         ],
385         programId: programId,
386         data: Buffer.from(Uint8Array.of(
387           0, ...new TextEncoder().encode(cid),
388           ... new TextEncoder().encode(parent), child!, diff!, pub, edit
389         ))
390       })
391     } else {
392       instruction = new TransactionInstruction({
393         keys: [
394           {pubkey: payer.publicKey, isSigner: true, isWritable: false},
395           {pubkey: imgDataPubkey, isSigner: false, isWritable: true}
396         ],
397         programId: programId,
398         data: Buffer.from(Uint8Array.of(
399           0, ...new TextEncoder().encode(cid),
400           ...new TextEncoder().encode(
401             '00000000000000000000000000000000000000000000000000000000000'
402           ), 0, 0, pub, edit
403         ))
```

```
404      })
405    }
406    const confirmation = await sendAndConfirmTransaction(
407      connection, new Transaction().add(instruction), [payer],
408    )
409
410    if (confirmation) {
411      if (pub === 1) {
412        if (typeof file === 'string') {
413          await uploadToIpfs(file)
414        } else {
415          await uploadToIpfs(file)
416        }
417      } else if (pub === 0) {
418        await uploadToIpfs(encryptedImage)
419        await saveKey(cid, key)
420      }
421    }
422  }
423
424  export async function permissions(
425    payer: Keypair,
426    imgDataPubkey: PublicKey,
427    pub: number,
428    edit: number
429  ): Promise <void> {
430    console.log('Granting or changing permissions for the image');
431    const imageData = await reportImgData(imgDataPubkey)
432    let cid = ''
433    let encryptedImage = Buffer.from('')
434    let decryptedImage = Buffer.from('')
435    let decryptedKey = Buffer.from('')
436    let key = Buffer.from('')
437    if (pub == imageData.public) {
438      cid = imageData.cid
439    } else if (pub == 0 && imageData.public == 1) {//becomes private
440      console.log('Encrypting image')
441      const data = await getFromIpfs(imageData.cid)
442      key = await keyGen()
443      encryptedImage = await encryptImage(data, key)
444      cid = await getCid(encryptedImage)
445    } else if (pub == 1 && imageData.public == 0) { //becomes public
446      console.log('Decrypting image')
447      const data = await getFromIpfs(imageData.cid)
448      const key = await retrieveKey(imageData.cid)
```

```
449       decryptedKey = await decryptKey(key)
450       decryptedImage = await decryptImage(data, decryptedKey)
451       cid = await getCid(decryptedImage)
452     }
453     const instruction = new TransactionInstruction({
454       keys: [
455         {pubkey: payer.publicKey, isSigner: true, isWritable: false},
456         {pubkey: imgDataPubkey, isSigner: false, isWritable: true}
457       ],
458       programId: programId,
459       data: Buffer.from(Uint8Array.of(1, ...new TextEncoder().encode(cid), pub, edit))
460     })
461     const confirmation = await sendAndConfirmTransaction(
462       connection,
463       new Transaction().add(instruction),
464       [payer],
465     );
466     if (confirmation) {
467       if (pub == 0 && imageData.public == 1){
468         await uploadToIpfs(encryptedImage)
469         await saveKey(cid, key)
470         await removeFromIpfs(imageData.cid)
471       } else if (pub == 1 && imageData.public == 0) {
472         await uploadToIpfs(decryptedImage)
473         await removeKey(imageData.cid)
474       }
475     }
476   }
477 
478   export async function downloadImage(
479     payer: Keypair,
480     dwnldLogPubkey: PublicKey,
481     imgDataPubkey: PublicKey
482   ): Promise<void> {
483     console.log('Downloading image and saving log into account ', dwnldLogPubkey.toBase58());
484     const imageData = await reportImgData(imgDataPubkey)
485     if (imageData.editable == 0) {
486       throw new Error("Image doesn't have permission to edit/download")
487     }
488 
489     const instruction = new TransactionInstruction({
490       keys: [
491         {pubkey: payer.publicKey, isSigner: true, isWritable: false},
492         {pubkey: dwnldLogPubkey, isSigner: false, isWritable: true}
493       ],
```

```
494         programId: programId,
495         data: Buffer.from(Uint8Array.of(2, ...new TextEncoder().encode(imageData.cid)))})
496         const confirmation = await sendAndConfirmTransaction(
497             connection,
498             new Transaction().add(instruction),
499             [payer],
500         )
501         if (confirmation) {
502             const downloadedImg = await getFromIpfs(imageData.cid)
503             fs.writeFile(path.resolve(DOWNLOADS_PATH,imageData.cid+'.png'), downloadedImg, (err) => {
504                 if(err) throw err
505             })
506         }
507     }
508 
509 
510     export async function editImage(
511       payer: Keypair,
512       originalImgPubkey: PublicKey,
513       childImgPubKey: PublicKey,
514       diffImgDataPubkey: PublicKey,
515       originalImg: string,
516       childImg: string,
517       childPub: number,
518       childEdit: number,
519       diffPub: number,
520       diffEdit: number
521     ): Promise<void> {
522       let childData = new jimp(256, 256, 0)
523       let diff = new jimp(256, 256, 0)
524       const originalData = await jimp.read(originalImg!)
525       childData = await jimp.read(childImg!)
526       diff = jimp.diff(originalData, childData).image
527       await uploadImage(
528         payer,
529         childImgPubKey!,
530         childImg!,
531         childPub!,
532         childEdit!,
533         await getCid(originalImg!),
534         1,
535         0,
536         originalImgPubkey
537       )
538       await uploadImage(
```

```
539       payer,
540       diffImgDataPubkey!,
541       await diff.getBufferAsync(jimp.MIME_PNG),
542       diffPub!,
543       diffEdit!,
544       await getCid(originalImg!),
545       0,
546       1,
547       originalImgPubkey
548     )
549   }
550
551   export async function requestKey(
552     payer: Keypair,
553     cid: string,
554     secKeyRequestPubkey: PublicKey
555   ): Promise<void> {
556     const key = await retrieveKey('public key')
557     const decryptedKey = await decryptKey(key)
558     const ecPubKey = decryptedKey.toString()
559     const instruction = new TransactionInstruction({
560       keys: [
561         {pubkey: payer.publicKey, isSigner: true, isWritable: false},
562         {pubkey: secKeyRequestPubkey, isSigner: false, isWritable: true}
563       ],
564       programId: programId,
565       data: Buffer.from(Uint8Array.of(
566         3, ... new TextEncoder().encode(ecPubKey),... new TextEncoder().encode(cid)
567       ))
568     })
569     await sendAndConfirmTransaction(
570       connection,
571       new Transaction().add(instruction),
572       [payer],
573     )
574   }
575
576   export async function shareKey(
577     payer: Keypair,
578     petitioner: PublicKey,
579     secKeyRequestPubkey: PublicKey
580   ): Promise<void> {
581     const secKeyRequest = await reportSecKeyRequest(secKeyRequestPubkey)
582     const key = await retrieveKey(secKeyRequest.cid)
583     const decryptedKey = await decryptKey(key)
```

```
584   const pubKey = secKeyRequest.ecPubKey
585   const pubEncKey = publicEncrypt(pubKey, decryptedKey)
586   const instruction = new TransactionInstruction({
587     keys: [
588       {pubkey: payer.publicKey, isSigner: true, isWritable: false},
589       {pubkey: petitioner, isSigner: false, isWritable: false},
590       {pubkey: secKeyRequestPubkey, isSigner: false, isWritable: true}
591     ],
592     programId: programId,
593     data: Buffer.concat([Buffer.from(Uint8Array.of(4)), pubEncKey])
594   })
595   await sendAndConfirmTransaction(
596     connection,
597     new Transaction().add(instruction),
598     [payer]
599   )
600 }
601 
602 export async function obtainKey(payer: Keypair, secKeyRequestPubkey: PublicKey): Promise<void> {
603   const secKeyRequest = await reportSecKeyRequest(secKeyRequestPubkey)
604   const cid = secKeyRequest.cid
605   const seckey = Buffer.from(secKeyRequest.secKey)
606   const privKey = await retrieveKey('private key')
607   const decPrivKey = await decryptKey(privKey)
608   const key = privateDecrypt(decPrivKey, seckey)
609   await saveKey(cid, key)
610   const instruction = new TransactionInstruction({
611     keys: [
612       {pubkey: payer.publicKey, isSigner: true, isWritable: false},
613       {pubkey: secKeyRequestPubkey, isSigner: false, isWritable: true}
614     ],
615     programId: programId,
616     data: Buffer.from(Uint8Array.of(5,1))
617   })
618   await sendAndConfirmTransaction(
619     connection,
620     new Transaction().add(instruction),
621     [payer]
622   )
623 }
624 
625 async function reportImgData(imgDataPubkey: PublicKey): Promise<ImgData> {
626   const accountInfo = await connection.getAccountInfo(imgDataPubkey);
627   if (accountInfo === null) {
628     throw 'Error: cannot find the image data account';
```

```
629     }
630     const imgData = borsh.deserialize(
631       ImgDataSchema,
632       ImgData,
633       accountInfo.data
634     );
635     console.log('\n\n', imgData, '\n\n');
636     console.log(
637      'Image',
638      imgData.cid,
639     `and it's info have been saved`
640     )
641       return imgData
642   }
643
644   async function reportDownloadLog(dwnldLogPubkey: PublicKey): Promise<void> {
645     console.log(dwnldLogPubkey);
646     const accountInfo = await connection.getAccountInfo(dwnldLogPubkey);
647     if (accountInfo === null) {
648       throw 'Error: cannot find the log account';
649     }
650     console.log(accountInfo.data);
651     const dwnldLog = borsh.deserialize(
652       DwnldLogSchema,
653       DwnldLog,
654       accountInfo.data,
655     );
656     console.log('\n\n', dwnldLog, '\n\n');
657     console.log(
658       'Image',
659       dwnldLog.cid,
660       'downloaded'
661     )
662   }
663
664   async function reportSecKeyRequest(secKeyRequestPubkey: PublicKey): Promise<SecKeyRequest> {
665     console.log(secKeyRequestPubkey);
666     const accountInfo = await connection.getAccountInfo(secKeyRequestPubkey);
667     if (accountInfo === null) {
668       throw 'Error: cannot find the log account';
669     }
670     console.log(accountInfo.data);
671     const secKeyRequest = borsh.deserialize(
672       SecKeyRequestSchema,
673       SecKeyRequest,
```

```
674       accountInfo.data,
675     )
676     console.log('\n\n', secKeyRequest, '\n\n');
677     console.log(
678       'Scret key for image',
679       secKeyRequest.cid,
680       'requested'
681     )
682     return secKeyRequest
683   }
```

A.2. Código Rust

A.2.1. entrypoint.ts

```
1  use solana_program::{
2      account_info::AccountInfo,
3      entrypoint,
4      entrypoint::ProgramResult,
5      msg,
6      pubkey::Pubkey,
7  };
8
9  use crate::processor::Processor;
10
11 entrypoint!(process_instruction);
12 fn process_instruction(
13     program_id: &Pubkey,
14     accounts: &[AccountInfo],
15     instruction_data: &[u8],
16 ) -> ProgramResult {
17     msg!(
18         "process_instruction: {}: {} accounts",
19         program_id,
20         accounts.len(),
21     );
22     Processor::process(program_id, accounts, instruction_data)
23 }
```

A.2.2. instruction.rs

```
use solana_program::program_error::ProgramError;

use crate::error::ImgError::InvalidInstruction;

pub enum ImgOperation {
    UploadImg {
        cid: String,
        parent: String,
        child: u8,
        diff: u8,
        public: u8,
        editable: u8,
    },
    ModifyPerms {
        cid: String,
        public: u8,
        editable: u8,
    },
    DwnldImg {
        cid: String,
    },
    RequestKey {
        pet_pub_key: String,
        cid: String,
    },
    ShareKey {
        pet_pub_key: [u8; 512],
    },
    ObtainKey {
        confirmation: u8,
    },
}

impl ImgOperation {
    pub fn unpack(input: &[u8]) -> Result<Self, ProgramError> {
        let (instruction, info) = input.split_first().ok_or(InvalidInstruction)?;
        Ok(match instruction {
            0 => Self::UploadImg {
                cid: String::from_utf8((info[0..59]).to_vec()).unwrap(),
                parent: String::from_utf8((info[59..118]).to_vec()).unwrap(),
                child: info[118],
                diff: info[119],
                public: info[120],
```

```
44                      editable: info[121],
45                  },
46                  1 => Self::ModifyPerms {
47                      cid: String::from_utf8((info[0..59]).to_vec()).unwrap(),
48                      public: info[59],
49                      editable: info[60],
50                  },
51                  2 => Self::DwnldImg {
52                      cid: String::from_utf8((info[..59]).to_vec()).unwrap(),
53                  },
54                  3 => Self::RequestKey {
55                      pet_pub_key: String::from_utf8((info[0..800]).to_vec()).unwrap(),
56                      cid: String::from_utf8((info[800..859]).to_vec()).unwrap(),
57                  },
58                  4 => Self::ShareKey {
59                      pet_pub_key: info[..].try_into().unwrap(),
60                  },
61                  5 => Self::ObtainKey {
62                      confirmation: info[0],
63                  },
64                  _ => return Err(InvalidInstruction.into()),
65              })
66          }
67  }
```

A.2.3. processor.rs

```
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    program_pack::{IsInitialized, Pack},
    pubkey::Pubkey,
};

use crate::{
    error::ImgError::{InvalidPetitioner, NoConfirmation},
    instruction::ImgOperation,
    state::{DwnldLog, ImgData, SecKeyRequest},
};

pub struct Processor;
impl Processor {
    pub fn process(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        instruction_data: &[u8],
    ) -> ProgramResult {
        let instruction = ImgOperation::unpack(instruction_data)?;

        match instruction {
            ImgOperation::UploadImg {
                cid,
                parent,
                child,
                diff,
                public,
                editable,
            } => {
                // msg!("Instruction: Upload Image");
                Self::process_upload_img_data(
                    accounts, cid, parent, child, diff, public, editable, program_id,
                )
            }
            ImgOperation::ModifyPerms {
                cid,
                public,
                editable,
            } => {
```

```
44                 // msg!("Instruction: Give Permissions");
45                 Self::process_modify_permissions(accounts, cid, public, editable, program_id)
46             }
47             ImgOperation::DwnldImg { cid } => {
48                 // msg!("Instruction: Download Image");
49                 Self::process_download_image(accounts, cid, program_id)
50             }
51             ImgOperation::RequestKey { pet_pub_key, cid } => {
52                 // msg!("Instruction: Request Key");
53                 Self::process_request_key(accounts, pet_pub_key, cid, program_id)
54             }
55             ImgOperation::ShareKey { pet_pub_key } => {
56                 // msg!("Instruction: Share Key");
57                 Self::process_share_key(accounts, pet_pub_key, program_id)
58             }
59             ImgOperation::ObtainKey { confirmation } => {
60                 // msg!("Instruction: Obtain Key");
61                 Self::process_obtain_key(accounts, confirmation, program_id)
62             }
63         }
64     }
65 
66     fn process_upload_img_data(
67         accounts: &[AccountInfo],
68         cid: String,
69         parent: String,
70         child: u8,
71         diff: u8,
72         public: u8,
73         editable: u8,
74         program_id: &Pubkey,
75     ) -> ProgramResult {
76         let accounts_iter = &mut accounts.iter();
77         let owner = next_account_info(accounts_iter)?;
78 
79         if !owner.is_signer {
80             return Err(ProgramError::MissingRequiredSignature);
81         }
82 
83         let img_data_account = next_account_info(accounts_iter)?;
84 
85         if img_data_account.owner != program_id {
86             msg!("Image data account does not have the correct program_id");
87             return Err(ProgramError::IncorrectProgramId);
88         }
```

```
89
90          let mut img_data = ImgData::unpack_unchecked(&img_data_account.try_borrow_data()?)?;
91
92          if img_data.is_initialized() {
93              return Err(ProgramError::AccountAlreadyInitialized);
94          }
95
96          img_data.is_initialized = true;
97          img_data.owner = owner.key.to_string();
98          img_data.cid = cid;
99          img_data.parent = parent;
100         img_data.child = child;
101         img_data.diff = diff;
102         img_data.public = public;
103         img_data.editable = editable;
104
105         if img_data.parent == "0000000000000000000000000000000000000000000000000000000000000" {
106             msg!("Image: {}", img_data.cid);
107             msg!("Operation: upload");
108             msg!("User: {}", img_data.owner);
109         } else {
110             let parent_img_data_account = next_account_info(accounts_iter)?;
111             if parent_img_data_account.owner != program_id {
112                 msg!("Image data account does not have correct program_id");
113                 return Err(ProgramError::IncorrectProgramId);
114             }
115             let parent_img_data = ImgData::unpack_unchecked(&parent_img_data_account.try_borrow_data()?)?;
116             msg!("Image: {}", img_data.parent);
117             msg!("User: {}", parent_img_data.owner);
118             msg!("Operation: edition");
119             msg!("User: {}", img_data.owner);
120             msg!("Image: {}", img_data.cid);
121             msg!("Operation: upload");
122         }
123
124         ImgData::pack(img_data, &mut img_data_account.try_borrow_mut_data()?)?;
125
126         Ok(())
127     }
128
129     fn process_modify_permissions(
130         accounts: &[AccountInfo],
131         cid: String,
132         public: u8,
133         editable: u8,
```

```
134         program_id: &Pubkey,
135     ) -> ProgramResult {
136         let accounts_iter = &mut accounts.iter();
137 
138         let owner = next_account_info(accounts_iter)?;
139 
140         if !owner.is_signer {
141             return Err(ProgramError::MissingRequiredSignature);
142         }
143 
144         let img_data_account = next_account_info(accounts_iter)?;
145 
146         if img_data_account.owner != program_id {
147             return Err(ProgramError::IncorrectProgramId);
148         }
149 
150         let mut img_data = ImgData::unpack_unchecked(&img_data_account.try_borrow_data()?)?;
151         if !img_data.is_initialized() {
152             return Err(ProgramError::UninitializedAccount);
153         }
154 
155         if img_data.public != public {
156             img_data.cid = cid;
157         }
158 
159         let prev_pub_status = img_data.public;
160         let prev_edit_status = img_data.editable;
161 
162         img_data.public = public;
163         img_data.editable = editable;
164 
165         msg!("Image: {}", img_data.cid);
166         msg!("Operation: permissions");
167         msg!("User: {}", img_data.owner);
168         msg!("Public: {} -> {}", prev_pub_status, public);
169         msg!("Editable: {} -> {}", prev_edit_status, editable);
170 
171         ImgData::pack(img_data, &mut img_data_account.try_borrow_mut_data()?)?;
172 
173         Ok(())
174     }
175 
176     fn process_download_image(
177         accounts: &[AccountInfo],
178         cid: String,
```

```
179             program_id: &Pubkey,
180         ) -> ProgramResult {
181             let accounts_iter = &mut accounts.iter();
182 
183             let downloader = next_account_info(accounts_iter)?;
184 
185             if !downloader.is_signer {
186                 return Err(ProgramError::MissingRequiredSignature);
187             }
188 
189             let download_log_account = next_account_info(accounts_iter)?;
190 
191             if download_log_account.owner != program_id {
192                 return Err(ProgramError::IncorrectProgramId);
193             }
194 
195             let mut dwnld_log = DwnldLog::unpack_unchecked(&download_log_account.try_borrow_data()?)?;
196 
197             if dwnld_log.is_initialized() {
198                 return Err(ProgramError::AccountAlreadyInitialized);
199             }
200 
201             dwnld_log.is_initialized = true;
202             dwnld_log.downloader = downloader.key.to_string();
203             dwnld_log.cid = cid;
204 
205             msg!("Image: {}", dwnld_log.cid);
206             msg!("Operation: download");
207             msg!("User: {}", dwnld_log.downloader);
208 
209             DwnldLog::pack(dwnld_log, &mut download_log_account.try_borrow_mut_data()?)?;
210 
211             Ok(())
212         }
213 
214         fn process_request_key(
215             accounts: &[AccountInfo],
216             pet_pub_key: String,
217             cid: String,
218             program_id: &Pubkey,
219         ) -> ProgramResult {
220             let accounts_iter = &mut accounts.iter();
221 
222             let petitioner = next_account_info(accounts_iter)?;
223 
```

```
224         if !petitioner.is_signer {
225             return Err(ProgramError::MissingRequiredSignature);
226         }
227 
228         let request_account = next_account_info(accounts_iter)?;
229 
230         if request_account.owner != program_id {
231             return Err(ProgramError::IncorrectProgramId);
232         }
233 
234         let mut sec_key_request =
235             SecKeyRequest::unpack_unchecked(&request_account.try_borrow_data()?)?;
236 
237         if sec_key_request.is_initialized() {
238             return Err(ProgramError::AccountAlreadyInitialized);
239         }
240 
241         sec_key_request.is_initialized = true;
242         sec_key_request.petitioner = petitioner.key.to_string();
243         sec_key_request.pet_pub_key = pet_pub_key;
244         sec_key_request.cid = cid;
245 
246         msg!("Image: {}", sec_key_request.cid);
247         msg!("Operation: request key");
248         msg!("User: {}", sec_key_request.petitioner);
249 
250         SecKeyRequest::pack(sec_key_request, &mut request_account.try_borrow_mut_data()?)?;
251 
252         Ok(())
253     }
254 
255     fn process_share_key(
256         accounts: &[AccountInfo],
257         ec_pub_key: [u8; 512],
258         program_id: &Pubkey,
259     ) -> ProgramResult {
260         let accounts_iter = &mut accounts.iter();
261 
262         let sender = next_account_info(accounts_iter)?;
263 
264         if !sender.is_signer {
265             return Err(ProgramError::MissingRequiredSignature);
266         }
267 
268         let petitioner = next_account_info(accounts_iter)?;
```

```
269
270         let request_account = next_account_info(accounts_iter)?;
271
272         if request_account.owner != program_id {
273             return Err(ProgramError::IncorrectProgramId);
274         }
275
276         let mut sec_key_request =
277             SecKeyRequest::unpack_unchecked(&request_account.try_borrow_data()?)?;
278
279         if !sec_key_request.is_initialized() {
280             return Err(ProgramError::UninitializedAccount);
281         }
282
283         if petitioner.key.to_string() != sec_key_request.petitioner {
284             return Err(InvalidPetitioner.into());
285         }
286
287         sec_key_request.sec_key = ec_pub_key;
288
289         msg!("Image: {}", sec_key_request.cid);
290         msg!("Operation: share key");
291         msg!("User: {}", sender.key);
292
293         SecKeyRequest::pack(sec_key_request, &mut request_account.try_borrow_mut_data()?)?;
294
295         Ok(())
296     }
297
298     fn process_obtain_key(
299         accounts: &[AccountInfo],
300         confirmation: u8,
301         program_id: &Pubkey,
302     ) -> ProgramResult {
303         let accounts_iter = &mut accounts.iter();
304
305         let petitioner = next_account_info(accounts_iter)?;
306
307         if !petitioner.is_signer {
308             return Err(ProgramError::MissingRequiredSignature);
309         }
310
311         let request_account = next_account_info(accounts_iter)?;
312
313         if request_account.owner != program_id {
```

```
314                 return Err(ProgramError::IncorrectProgramId);
315             }
316 
317             let sec_key_request = SecKeyRequest::unpack_unchecked(&request_account.try_borrow_data()?)?;
318 
319             if !sec_key_request.is_initialized() {
320                 return Err(ProgramError::UninitializedAccount);
321             }
322 
323             if confirmation != 1 {
324                 return Err(NoConfirmation.into());
325             }
326 
327             msg!("Image: {}", sec_key_request.cid);
328             msg!("Operation: obtain key");
329             msg!("User: {}", sec_key_request.petitioner);
330 
331             msg!("Closing request account");
332 
333             let lamp = request_account.lamports();
334             msg!("{}", lamp);
335             **request_account.try_borrow_mut_lamports()? -= lamp;
336             **petitioner.try_borrow_mut_lamports()? += lamp;
337 
338             msg!("Request account closed");
339 
340             Ok(())
341         }
342 
343 }
```

A.2.4. state.rs

```
use solana_program::{
    program_pack::{IsInitialized, Pack, Sealed},
    program_error::ProgramError,
};

use arrayref::{array_mut_ref, array_ref, array_refs, mut_array_refs};

pub struct ImgData{
    pub is_initialized: bool,
    pub owner: String,
    pub cid: String,
    pub parent: String,
    pub child: u8,
    pub diff: u8,
    pub public: u8,
    pub editable: u8,
}

pub struct DwnldLog {
    pub is_initialized: bool,
    pub downloader: String,
    pub cid: String,
}

pub struct SecKeyRequest {
    pub is_initialized: bool,
    pub petitioner: String,
    pub pet_pub_key: String,
    pub cid: String,
    pub sec_key: [u8; 512],
}

impl Sealed for ImgData {}

impl IsInitialized for ImgData {
    fn is_initialized(&self) -> bool {
        self.is_initialized
    }
}

impl Pack for ImgData {
    const LEN: usize = 179;
    fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
```

```
44          let src = array_ref![src, 0, ImgData::LEN];
45          let (
46              is_initialized,
47              _key_size,
48              owner,
49              _cid_size,
50              cid,
51              _parent_size,
52              parent,
53              child,
54              diff,
55              public,
56              editable,
57          ) = array_refs![src, 1, 4, 44, 4, 59, 4, 59, 1, 1, 1, 1];
58          let is_initialized = match is_initialized {
59              [0] => false,
60              [1] => true,
61              _ => return Err(ProgramError::InvalidAccountData),
62          };
63          Ok(ImgData {
64              is_initialized,
65              owner: String::from_utf8(owner.to_vec()).unwrap(),
66              cid: String::from_utf8(cid.to_vec()).unwrap(),
67              parent: String::from_utf8(parent.to_vec()).unwrap(),
68              child: child[0],
69              diff: diff[0],
70              public: public[0],
71              editable: editable[0],
72          })
73      }
74
75      fn pack_into_slice(&self, dst: &mut [u8]) {
76          let dst = array_mut_ref![dst, 0, ImgData::LEN];
77          let (
78              is_initialized_dst,
79              key_size_dst,
80              owner_dst,
81              cid_size_dst,
82              cid_dst,
83              parent_size_dst,
84              parent_dst,
85              child_dst,
86              diff_dst,
87              public_dst,
88              editable_dst,
```

```
89          ) = mut_array_refs![dst, 1, 4, 44, 4, 59, 4, 59, 1, 1, 1, 1];
90          let ImgData {
91              is_initialized,
92              owner,
93              cid,
94              parent,
95              child,
96              diff,
97              public,
98              editable,
99          } = self;
100         is_initialized_dst[0] = *is_initialized as u8;
101         let tmp = owner_dst.len() as u32;
102         *key_size_dst = tmp.to_le_bytes();
103         owner_dst.copy_from_slice(owner.as_ref());
104         let tmp = cid_dst.len() as u32;
105         *cid_size_dst = tmp.to_le_bytes();
106         cid_dst.copy_from_slice(cid.as_ref());
107         let tmp = parent_dst.len() as u32;
108         *parent_size_dst = tmp.to_le_bytes();
109         parent_dst.copy_from_slice(parent.as_ref());
110         *child_dst = child.to_le_bytes();
111         *diff_dst = diff.to_le_bytes();
112         *public_dst = public.to_le_bytes();
113         *editable_dst = editable.to_le_bytes();
114     }
115 }
116
117 impl Sealed for DwnldLog {}
118
119 impl IsInitialized for DwnldLog {
120     fn is_initialized(&self) -> bool {
121         self.is_initialized
122     }
123 }
124
125 impl Pack for DwnldLog {
126     const LEN: usize = 112;
127     fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
128         let src = array_ref![src, 0, DwnldLog::LEN];
129         let(
130             is_initialized,
131             _key_size,
132             downloader,
133             _cid_size,
```

```
134                 cid,
135             ) = array_refs![src, 1, 4, 44, 4, 59];
136             let is_initialized = match is_initialized {
137                 [0] => false,
138                 [1] => true,
139                 _ => return Err(ProgramError::InvalidAccountData),
140             };
141             Ok(DwnldLog {
142                 is_initialized,
143                 downloader: String::from_utf8(downloader.to_vec()).unwrap(),
144                 cid : String::from_utf8(cid.to_vec()).unwrap(),
145             })
146         }
147 
148         fn pack_into_slice(&self, dst: &mut [u8]) {
149             let dst = array_mut_ref![dst, 0, DwnldLog::LEN];
150             let(
151                 is_initialized_dst,
152                 key_size_dst,
153                 downloader_dst,
154                 cid_size_dst,
155                 cid_dst,
156             ) = mut_array_refs![dst, 1, 4, 44, 4, 59];
157             let DwnldLog {
158                 is_initialized,
159                 downloader,
160                 cid
161             } = self;
162             is_initialized_dst[0] = *is_initialized as u8;
163             let tmp = downloader_dst.len() as u32;
164             *key_size_dst = tmp.to_le_bytes();
165             downloader_dst.copy_from_slice(downloader.as_ref());
166             let tmp = cid_dst.len() as u32;
167             *cid_size_dst = tmp.to_le_bytes();
168             cid_dst.copy_from_slice(cid.as_ref());
169         }
170     }
171 
172     impl Sealed for SecKeyRequest {}
173 
174     impl IsInitialized for SecKeyRequest {
175         fn is_initialized(&self) -> bool {
176             self.is_initialized
177         }
178     }
```

```
179
180 impl Pack for SecKeyRequest {
181     const LEN: usize = 1428;
182     fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
183         let src = array_ref![src, 0, SecKeyRequest::LEN];
184         let (
185             is_initialized,
186             _petitioner_key_size,
187             petitioner,
188             _key_size,
189             pet_pub_key,
190             _cid_size,
191             cid,
192             sec_key,
193             ) = array_refs![src, 1, 4, 44, 4, 800, 4, 59, 512];
194         let is_initialized = match is_initialized {
195             [0] => false,
196             [1] => true,
197             _ => return Err(ProgramError::InvalidAccountData)
198         };
199         Ok(SecKeyRequest {
200             is_initialized,
201             petitioner: String::from_utf8(petitioner.to_vec()).unwrap(),
202             pet_pub_key: String::from_utf8(pet_pub_key.to_vec()).unwrap(),
203             cid: String::from_utf8(cid.to_vec()).unwrap(),
204             sec_key: *sec_key
205         })
206     }
207
208     fn pack_into_slice(&self, dst: &mut [u8]) {
209         let dst = array_mut_ref![dst, 0, SecKeyRequest::LEN];
210         let(
211             is_initialized_dst,
212             petitioner_key_size_dst,
213             petitioner_dst,
214             key_size_dst,
215             pet_pub_key_dst,
216             cid_size_dst,
217             cid_dst,
218             sec_key_dst,
219             ) = mut_array_refs![dst, 1, 4, 44, 4, 800, 4, 59, 512];
220         let SecKeyRequest {
221             is_initialized,
222             petitioner,
223             pet_pub_key,
```

```
224             cid,
225             sec_key,
226         } = self;
227         is_initialized_dst[0] = *is_initialized as u8;
228         let tmp = petitioner_dst.len() as u32;
229         *petitioner_key_size_dst = tmp.to_le_bytes();
230         petitioner_dst.copy_from_slice(petitioner.as_ref());
231         let tmp = pet_pub_key_dst.len() as u32;
232         *key_size_dst = tmp.to_le_bytes();
233         pet_pub_key_dst.copy_from_slice(pet_pub_key.as_ref());
234         let tmp = cid_dst.len() as u32;
235         *cid_size_dst = tmp.to_le_bytes();
236         cid_dst.copy_from_slice(cid.as_ref());
237         sec_key_dst.copy_from_slice(sec_key);
238     }
239 }
```

A.2.5. error.rs

```
1  use thiserror::Error;
2
3  use solana_program::program_error::ProgramError;
4
5  #[derive(Error, Debug, Copy, Clone)]
6  pub enum ImgError {
7      #[error("Invalid Instruction")]
8      InvalidInstruction,
9      #[error("Invalid Petitioner")]
10     InvalidPetitioner,
11     #[error("No Confirmation")]
12     NoConfirmation,
13 }
14
15 impl From<ImgError> for ProgramError {
16     fn from(e: ImgError) -> Self {
17         ProgramError::Custom(e as u32)
18     }
19 }
```

Bibliografía

[1] "How many files are there in the world?" accessed: 2023-01-09. [Online]. Available: https://www.cloudfiles.io/blog/how-many-files-are-there-in-the-world 1

[2] M. Pixel, "México sigue teniendo problemas con las fake news: algunas de estas fotos son reales pero no son de cancún," 2017. [Online]. Available: https://www.xataka.com.mx/otros-1/mexico-sigue-teniendo-problemas-con-las-fake-news-algunas-de-estas-fotos-son-reales-pero-no-so
1

[3] T. Wen, "The hidden signs that can reveal a fake photo," 2020. [Online]. Available: https://www.bbc.com/future/article/20170629-the-hidden-signs-that-can-reveal-if-a-photo-is-fake 1

[4] B. Yañez, M. Galván, and S. Ramírez, "El ABC de la 'Ley Olimpia': sus alcances y retos," 2022. [Online]. Available: https://politica.expansion.mx/sociedad/2022/21/25/el-abc-de-la-ley-olimpia-sus-alcances-y-retos 1

[5] L. Moreau, P. Groth, S. Miles, J. Vázquez-Salceda, J. Ibbotson, J. Sheng, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga, "The provenance of electronic data," *Commun. ACM*, vol. 51, pp. 52–58, 04 2008. 1, 8, 9

[6] A. Joshi, M. Han, and Y. Wang, "A survey on security and privacy issues of blockchain technology," *Mathematical Foundations of Computing*, vol. 1, pp. 121–147, 01 2018. 2, 12, 13, 14

[7] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0.8.13," Tech. Rep., 2020. 2, 17

[8] J. Benet, "Ipfs - content addressed, versioned, p2p file system," 2014. 2

[9] J.-P. Aumasson, *Serious cryptography: A practical introduction to modern encryption.* San Francisco, CA: No Starch Press, 2017. 3, 6

[10] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* Boca Raton, FL: CRC Press, 1996. 3, 4, 5, 8

[11] C. Paar and J. Pelzl, *Understanding cryptography: A textbook for students and practitioners*, 2010th ed. Berlin, Germany: Springer, 2009. 5, 6, 8

[12] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *Advances in Cryptology — CRYPTO ’87*, C. Pomerance, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. 6

[13] B. Carminati, *Merkle Trees*. New York, NY: Springer New York, 2018, pp. 2231–2232. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_1492 6

[14] Kerala Blockchain Academy, “Merkle tree: A beginners guide,” https://medium.com/blockchain-stories/merkle-tree-a-beginners-guide-5c53a7defeb9, Sep. 2021, accessed: 2022-11-18. 8

[15] R. Aldeco-Pérez and M. Leon Chavez, “Evaluar la Calidad de los Objetos de Aprendizaje Mediante Linaje Electrónico,” in *El Desarrollo de los Recursos Digitales para la Educación en México*, 1st ed. Benemérita Universidad Autónoma de Puebla, 2013, p. 240. [Online]. Available: https://tinyurl.com/2p8akmch 9

[16] P. McDaniel, “Data provenance and security,” *IEEE Security Privacy*, vol. 9, no. 2, pp. 83–85, 2011. 9

[17] N. Gupta, “Chapter 4 - a deep dive into security and privacy issues of blockchain technologies,” in *Handbook of Research on Blockchain Technology*, S. Krishnan, V. E. Balas, E. G. Julie, Y. H. Robinson, S. Balaji, and R. Kumar, Eds. Academic Press, 2020, pp. 95–112. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128198162000046 10

[18] K. Sivleen, C. Sheetal, S. Aabha, and K. Jayaprakash, “A research survey on applications of consensus protocols in blockchain,” *Security and Communication Networks*, vol. 2021, 01 2021. 10, 11, 12, 13, 14, 16, 17

[19] C. D. Retamal, J. B. Roig, and J. L. G.-C. Tapia, “La blockchain : fundamentos, aplicaciones y relación con otras tecnologías disruptivas.” vol. 405, 2017, pp. 33–40. 10

[20] J. Mattila, “The blockchain phenomenon – the disruptive potential of distributed consensus architectures,” Helsinki, ETLA Working Papers 38, 2016. [Online]. Available: http://hdl.handle.net/10419/201253 10

[21] C. G. Akcora, Y. R. Gel, and M. Kantarcioglu, “Blockchain: A graph primer,” 2017. [Online]. Available: https://arxiv.org/abs/1708.08749 10

[22] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, “A survey of distributed consensus protocols for blockchain networks,” *IEEE Communications Surveys Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020. 13, 15

[23] F. Rebollar Castelan, M. Ramos Corchado, and R. Valdovinos, “La adopción del blockchain por la nueva era digital como un sistema descentralizado para el uso y creación de nuevos servicios digitales,” 10 2018. 13, 14

[24] C. Zozaya, J. Incera, and A. Velázquez, “Blockchain : un tutorial,” *Estudios: filosofía, historia, letras*, vol. 17, p. 113, 01 2019. 16

[25] B. K. Mohanta, S. S. Panda, and D. Jena, “An overview of smart contract and use cases in blockchain technology,” in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018, pp. 1–4. 18

[26] K. Upadhyay, R. Dantu, Y. He, A. Salau, and S. Badruddoja, “Paradigm shift from paper contracts to smart contracts,” in *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, 2021, pp. 261–268. 19

[27] “Introduction: Solana docs,” accessed: 2022-11-09. [Online]. Available: https://docs.solana.com/introduction 19, 20

[28] “Solana wallet guide: Solana docs,” accessed: 2022-11-09. [Online]. Available: https://docs.solana.com/wallet-guide 20

[29] “Turbine block propagation: Solana docs,” accessed: 2022-11-09. [Online]. Available: https://docs.solana.com/cluster/turbine-block-propagation 21

[30] “Accounts: Solana docs,” accessed: 2022-11-09. [Online]. Available: https://docs.solana.com/developing/programming-model/accounts 21

[31] “Solana cookbook: Accounts,” accessed: 2022-11-09. [Online]. Available: https://solanacookbook.com/core-concepts/accounts.html 21

[32] “Transactions: Solana docs,” accessed: 2022-11-09. [Online]. Available: https://docs.solana.com/developing/programming-model/transactions 24

[33] “Solana cookbook: Transactions,” accessed: 2022-11-09. [Online]. Available: https://solanacookbook.com/core-concepts/transactions.html 24

[34] “What are solana programs?: Solana docs,” accessed: 2022-11-09. [Online]. Available: https://docs.solana.com/developing/intro/programs 24

[35] “ what is ipfs?” accessed: 2022-11-09. [Online]. Available: https://docs.ipfs.tech/concepts/what-is-ipfs/ 24

[36] “Multiformats tutorial: Anatomy of a cid,” accessed: 2022-11-09. [Online]. Available: https://proto.school/anatomy-of-a-cid 25

[37] Ipld, "Specs/dag-pb.md at master · ipld/specs," accessed: 2022-12-05. [Online]. Available: https://github.com/ipld/specs/blob/master/block-layer/codecs/dag-pb.md 25

[38] "Multibase/multibase.csv at master · multiformats/multibase," accessed: 2022-11-09. [Online]. Available: https://github.com/multiformats/multibase/blob/master/multibase.csv 26

[39] Multiformats, "Multicodec/table.csv at master · multiformats/multicodec," accessed: 2022-11-09. [Online]. Available: https://github.com/multiformats/multicodec/blob/master/table.csv 26

[40] " merkle directed acyclic graphs (dags)," accessed: 2022-11-09. [Online]. Available: https://docs.ipfs.tech/concepts/merkle-dag/ 26

[41] "Ipld tutorial: Merkle dags: Structuring data for the distributed web," accessed: 2022-11-09. [Online]. Available: https://proto.school/merkle-dags 27

[42] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *2016 2nd International Conference on Open and Big Data (OBD)*, 2016, pp. 25–30. 28

[43] V. Mani, P. Manickam, Y. Alotaibi, S. Alghamdi, and O. I. Khalaf, "Hyperledger healthchain: Patient-centric ipfs-based storage of health records," *Electronics*, vol. 10, no. 23, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/23/3003 28

[44] E. B. Sifah, Q. Xia, K. O.-B. O. Agyekum, H. Xia, A. Smahi, and J. Gao, "A blockchain approach to ensuring provenance to outsourced cloud data in a sharing ecosystem," *IEEE Systems Journal*, pp. 1–12, 2021. 28

[45] P. Abhishek, Y. Akash, and D. G. Narayan, "A scalable data provenance mechanism for cloud environment using ethereum blockchain," in *2021 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, 2021, pp. 1–6. 29

[46] S. Khatal, J. Rane, D. Patel, P. Patel, and Y. Busnel, "Fileshare: A blockchain and ipfs framework for secure file sharing and data provenance," in *Advances in Machine Learning and Computational Intelligence*, S. Patnaik, X.-S. Yang, and I. K. Sethi, Eds. Singapore: Springer Singapore, 2021, pp. 825–833. 29

[47] T. Igarashi, T. Kazuhiko, Y. Kobayashi, H. Kuno, and E. Diehl, "Photrace: A blockchain-based traceability system for photographs on the internet," in *2021 IEEE International Conference on Blockchain (Blockchain)*, 2021, pp. 590–596. 29

[48] R. Sharma, S. Pawar, S. Gurav, and P. Bhavathankar, "A unique approach towards image publication and provenance using blockchain," in *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2020, pp. 311–314. 29

[49] Maldeadora, "Qué es frontend y backend: Diferencias y características," Feb 2018, accessed: 2022-12-02. [Online]. Available: https://platzi.com/blog/que-es-frontend-y-backend/ 31

[50] "Node.js v19.2.0 documentation," accessed: 2022-12-05. [Online]. Available: https://nodejs.org/api/crypto.html 44

[51] Oliver-Moran, "Oliver-moran/jimp: An image processing library written entirely in javascript for node, with zero external or native dependencies." accessed: 2022-12-05. [Online]. Available: https://github.com/oliver-moran/jimp 45

[52] Ipfs, "Ipfs/js-ipfs: Ipfs implementation in javascript," accessed: 2022-12-05. [Online]. Available: https://github.com/ipfs/js-ipfs 46

[53] "Solana javascript api," accessed: 2022-12-05. [Online]. Available: https://solana-labs.github.io/solana-web3.js/ 47

[54] Accessed: 2022-11-09. [Online]. Available: https://borsh.io/ 48

[55] L. Lesavre, P. Varin, and D. Yaga, *Blockchain Networks: Token Design and Management Overview*, Feb 2021. [Online]. Available: http://dx.doi.org/10.6028/NIST.IR.8301 63

[56] "Cardano explorer," accessed: 2022-12-02. [Online]. Available: https://cardanoscan.io/ 63

[57] "Algorand (algo) blockchain explorer," accessed: 2022-12-02. [Online]. Available: https://algoexplorer.io/ 63

[58] "Goalseeker by purestake: Algorand block explorer," accessed: 2022-12-02. [Online]. Available: https://goalseeker.purestake.io/ 63

[59] "Ethereum (eth) blockchain explorer," accessed: 2022-12-02. [Online]. Available: https://etherscan.io/ 63

[60] Accessed: 2022-12-19. [Online]. Available: http://computo.fismat.umich.mx/smcc/ENC2021_CLQ.pdf 72

[61] J. A. Miramontes and R. Aldeco-Perez, "Trazabilidad de imágenes digitales usando blockchain," *Research in Computing Science*, vol. 151, 09 2022 (en prensa). [Online]. Available: https://www.researchgate.net/publication/366569984_Trazabilidad_de_imagenes_digitales_usando_Blockchain 72