



**UNIVERSIDAD
DE GRANADA**

TRABAJO FIN DE GRADO

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Implementación de una blockchain resistente a ataques criptográficos cuánticos

Autor

María Victoria Granados Pozo

Directores

Gabriel Maciá Fernández

Francisco Javier Lobillo Borrero



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN**



Facultad de Ciencias

FACULTAD DE CIENCIAS

Granada, 18 de noviembre de 2020

Implementación de una blockchain resistente a ataques criptográficos cuánticos



BLOCKCHAIN

ALGORITMO CRIPTOGRÁFICO UOV

Autor

María Victoria Granados Pozo

Directores

Gabriel Maciá Fernández

Francisco Javier Lobillo Borrero

Granada, 18 de noviembre de 2020

Implementación de una blockchain resistente a ataques criptográficos cuánticos

María Victoria Granados Pozo

Palabras clave: algoritmo criptográfico, cadena de bloques, computación cuántica, firma, verificación

Resumen

Debido al auge de la tecnología en nuestra sociedad cabe pensar cuanto de seguras son las actividades que realizamos con el móvil o un ordenador, por ejemplo, las transacciones bancarias. Por otra parte, los avances tecnológicos nos advierten sobre la seguridad de los algoritmos criptográficos actuales, puesto que estos avances permiten una mayor capacidad de cómputo. De aquí surge la idea de este proyecto, marcando como objetivo evitar que las transacciones almacenadas en una *blockchain* se mantengan a salvo cuando esté disponible la computación cuántica.

Los algoritmos criptográficos que se utilizan hoy día para la firma y verificación de mensajes, basan su seguridad en la hipótesis de que no se pueden encontrar las claves por fuerza bruta. Así con un computador cuántico la seguridad de dichos algoritmos se vería perjudicada. De esta forma surgen los algoritmos criptográficos post-cuánticos, o dicho de otra manera, algoritmos criptográficos resistentes a un criptoanálisis mediante algoritmos implementables en ordenadores cuánticos.

En este proyecto el algoritmo que se va a utilizar es el algoritmo **UOV** (*Unbalanced Oil and Vinegar*). Este algoritmo es resistente a ataques cuánticos puesto que si se considera el problema de la creación y validación de firmas es necesario resolver un sistema con m ecuaciones y n variables, siendo esto un problema NP-duro. De este algoritmo hay que resaltar la simplicidad de las operaciones utilizadas, ya que se firman y verifican los bloques con operaciones de suma y multiplicación de valores pequeños. Para llevar a cabo la implementación del algoritmo UOV ha sido necesario implementar la aritmética del cuerpo finito de 128 elementos.

Además de la computación cuántica, la otra tecnología que se ha utilizado son las *blockchain*. Una *blockchain* es una cadena formada por bloques, donde cada bloque almacena información como transacciones o el *hash* del bloque anterior. Este tipo de estructura de almacenamiento permite verificar validar y rastrear todo tipo de información, y cada bloque tiene una única posición en la cadena sin poder ser alterada. Los árboles Merkle se utilizan para realizar una

búsqueda eficiente en la *blockchain*. Son árboles donde el *hash* de cada hijo es una combinación de los *hash* de los nodos padre.

La única línea de defensa de las *blockchain* es el algoritmo de firma de los bloques, de cara a los computadores cuánticos. Ya que en la actualidad son seguros debido a que los ordenadores clásicos no tienen la capacidad de cómputo necesaria para descifrar cada bloque y obtener la información sin dejar huella.

En este proyecto se ha seleccionado la *blockchain* ARK. Debido a sus propiedades de código abierto y su arquitectura modular. Al ser de código abierto cualquier persona puede modificar dicho código y pueden aportar algunas ideas de cambios. Mientras su arquitectura modular permite personalizar la *blockchain* según las necesidades de cada usuario. En nuestro caso cambiar los algoritmos de firma y verificación que vienen por defecto en la *blockchain*, algoritmo ECD-SA y Schnorr, por el algoritmo post-cuántico UOV.

Para la realización del proyecto se ha usado un contenedor, docker, de manera que las instalaciones de la *blockchain* no se vean reflejadas en la máquina local, aprovechando la propiedad de aislamiento de docker. Por tanto dentro del docker se encontrará el entorno necesario para la instalación de la *blockchain* modificada con los nuevos algoritmos de firma y verificación. Concluyendo en la *blockchain* de ARK menos vulnerable frente ataques cuánticos, puesto que se ha modificado el algoritmo de firma por el algoritmo UOV. Como valor añadido se podrá integrar en otra *blockchain*, gracias a las propiedades de ARK.

Implementation of a blockchain resistant to quantum cryptographic attacks

María Victoria Granados Pozo

Keywords: blockchain, cryptographic algorithm, quantum attacks, quantum computing, sign, verify

Abstract

Due to the rise of technology in our society, it is possible to think how safe the activities we carry out with the mobile phone or a computer are, for example, banking transactions. On the other hand, technological advances warn us about the security of current cryptographic algorithms, since these advances allow for greater computing capacity. This is where the idea of this project arises, with the objective of preventing transactions stored in a blockchain from being safe where quantum computer is available.

The cryptographic algorithms that are currently used for the signature and verification of messages, base their security on the hypothesis that the keys cannot be found by brute force. With a quantum computer, the security of these algorithms will be broken, mainly due to the calculation capacity of these computers. In this way, post-quantum cryptographic algorithms emerge, i. e., cryptographic algorithms resistant to cryptanalysis using algorithms that can be implemented in quantum computers.

In this project the algorithm considered is the [UOV](#) (Unbalance Oil and Vinegar) algorithm. This algorithm is resistant to quantum attacks. If we consider the problem of creation and verification of signatures, we will need to solve a system with m equations and n variables, this being an NP-hard problem. In this algorithm, the simplicity of the operations used must be highlighted, since the blocks are signed and verified with operations of addition and multiplication of small values. To carry put the implementation of the UOV algorithm, it has been necessary to implement the finite body arithmetic of 128 element.

Besides quantum computing, the other technology that has been used is blockchain. A blockchain is a chain made up of blocks, where each block store information like transaction or the hash of the previous block. This type of storage structure allows to verify, validate and track all types of information, and each block has just one place. Merkle trees are used to perform an efficient search on the blockchain. They are trees where each child node hash is a combination of the hashes of the parent node.

The main line of defense for blockchains is the block signature algorithm. Thus, they are currently safe because classic computers do not have the necessary computing capacity to decipher each block and obtain the information without leaving a trace.

In this project, the ARK blockchain has been selected. Because of its open source properties and its modular architecture. Being open source, anyone can modify the code and can contribute some ideas for changes. Its modular architecture allows to customize the blockchain according to the needs of each user. In this case, we change the signature and verification algorithms that come by default in the blockchain, ECDSA and Schnorr algorithms, for the UOV post-quantum algorithm.

To implement the project, a docker container, has been used so that the blockchain installations are not reflected in the local machine, taking advantage of the isolation property of docker. Therefore, within the docker you will find the necessary environment for the installation of the modified blockchain with the new signature and verification algorithms. Concluding in the ARK blockchain is less vulnerable to quantum attacks, since the signature algorithm. As an added value, it can be integrated into another blockchain, thanks to the ARK properties.

Yo, **María Victoria Granados Pozo**, alumna de la titulación Doble Grado de Ingeniería Informática y Matemáticas de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación y Facultad de Ciencias de la Universidad de Granada**, con DNI 77137043, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: María Victoria Granados Pozo

Granada a 18 de noviembre de 2020 .

D. **Gabriel Maciá Fernández**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

D. **Francisco Javier Lobillo Borrero**, Profesor del Área de Matemáticas del Departamento Álgebra de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Implementación de una blockchain resistente a ataques criptográficos cuánticos***, ha sido realizado bajo su supervisión por **María Victoria Granados Pozo**, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 18 de noviembre de 2020 .

Los directores:

Gabriel Maciá Fernández Francisco Javier Lobillo Borrero

Agradecimientos

En especial agradezco a mis tutores Grabiél Maciá y Javier Lobillo, por el apoyo y la paciencia que han tenido conmigo a lo largos de todos estos meses. También a Javier Tallón por orientarme a elegir el tema de este trabajo.

A mis padres, Miguel y Esther, que me han soportado y animado en los momentos más difíciles. A mi hermano, Miguel, por darme fuerza en el día a día en estos momentos tan complicados de la pandemia que nos ha tocado vivir. A mi pareja, Alfonso, por animarme a seguir adelante en este largo camino.

Por último, agradecer a los profesores y compañeros que me he encontrado a lo largo de estos cinco años de carrera, que tanto me han enseñado y tantos momentos he compartido con ellos.

Índice general

1. Introducción	1
1.1. Motivación y contexto del proyecto	1
1.2. Objetivos del proyecto y logros conseguidos	5
1.3. Estructura de la memoria	5
2. Contenidos teóricos	7
2.1. Computación cuántica	7
2.1.1. Historia	7
2.1.2. Estructura de los cúbits	9
2.1.3. Propiedades	11
2.2. Blockchain	11
2.2.1. Blockchain ARK	16
2.3. Algoritmo UOV	17
2.3.1. Cuerpos finitos	17
2.3.2. Parámetros y fórmula	18
2.3.3. Generación de la clave privada	19
2.3.4. Generación de la clave pública	19
2.3.5. Algoritmo de firma	20
2.3.6. Algoritmo de verificación	21
2.3.7. Ejemplo	21
3. Planificación y costes	27
3.1. Planificación	27
3.2. Costes	30
4. Análisis del problema	35
4.1. Especificación de requisitos	36
4.1.1. Requisitos funcionales	36
4.1.2. Requisitos no funcionales	38
4.2. Análisis	40
5. Diseño	41

6. Implementación	45
6.1. Ecosistema ARK	45
6.1.1. Directorio deployer	46
6.1.2. Directorio core-bridgechain	46
6.2. Algoritmo criptográfico UOV	53
6.2.1. Funciones del cuerpo de 128 elementos	53
6.2.2. Funciones con matrices	57
6.2.3. Funciones algoritmo UOV	61
7. Evaluación y pruebas	67
7.1. Pruebas algoritmo criptográfico	67
7.2. Pruebas de integración	68
8. Conclusiones	69
Bibliografía	74
Glosario de siglas	75
A. Manual de usuario	77
A.1. Instalación de Docker	77
A.2. Instalación Blockchain ARK	79
A.3. Instalación de la aplicación ARK Desktop Wallet	85
A.4. Visualización de los datos con el Explorer	96

Índice de figuras

1.1. Comparativa de la capacidad de cómputo de un ordenador clásico con un ordenador cuántico[1]	3
2.1. Estados de un bit y de cúbit [1]	10
2.2. Estructura cúbit, esfera de Bloch [2]	10
2.3. Estructura de un árbol Merkle [3]	12
3.1. Diagrama de Gantt inicial	28
3.2. Diagrama de Gantt real. Parte I	29
3.3. Diagrama de Gantt real. Parte II	29
3.4. Diagrama de Gantt real. Parte III	29
5.1. Diagrama de bloques para el prototipo	43
5.2. Diagrama de bloques. Ejemplo de configuración 1	43
5.3. Diagrama de bloques. Ejemplo de configuración 2	44
6.1. Árbol de directorios de deployer	46
6.2. Árbol de directorios de core-bridgechain	47
A.1. Salida tras la instalación del <i>core</i>	83
A.2. Salida tras iniciar del <i>explorer</i>	84
A.3. Primeras transacciones en el <i>explorer</i>	84
A.4. Inicio de Wallet	85
A.5. Detalles del perfil	86
A.6. Configuración de la nueva red	86
A.7. Detalles de la nueva red	87
A.8. Selección de la red	88
A.9. Personalización del diseño de la interfaz	88
A.10. Configuración de la conexión peer	89
A.11. Importar monedero	89
A.12. Encriptación el monedero	90
A.13. Confirmación para crear el monedero	90
A.14. Selección de la dirección del nuevo monedero	91
A.15. <i>Passphrase</i> o clave privada del monedero	91

A.16.Verificación de la <i>passphrase</i>	92
A.17.Encriptación del monedero	92
A.18.Confirmación para crear el segundo monedero	93
A.19.Página principal con los monederos	93
A.20.Botón enviar	94
A.21.Realizar transferencia	94
A.22.Confirmación de la transferencia	95
A.23.Comprobación de que se ha enviado la transferencia	95
A.24.Logs de la cadena de bloques	96
A.25.Vista del <i>explorer</i> con los últimos bloques creados	96
A.26.Bloques anterior y posterior del bloque con ID 63372384328353607	97
A.27.Información del bloque con ID 63372384328353607, visto en el <i>ex- plorer</i>	97
A.28.Información del bloque con ID 63372384328353607, visto en la API	98
A.29.Información del bloque anterior al bloque 63372384328353607, vis- to desde la API	98

Índice de tablas

1.1. Niveles de seguridad de ordenadores clásicos y cuánticos [4]	4
2.1. Representación de los elementos no nulos de \mathbb{F}_{128}	18
3.1. Desglose de los costes en recursos humanos de los tutores	30
3.2. Desglose de los costes en recursos humanos de la alumna	31
3.3. Desglose de los costes directos	32
3.4. Presupuesto gastos previstos desglosado	32
3.5. Presupuesto total desglosado	33
4.1. Requisitos funcionales del programa	36
4.2. Requisitos funcionales del docker	37
4.3. Requisitos funcionales del <i>Explorer</i>	37
4.4. Requisitos funcionales de la aplicación Wallet	38
4.5. Requisitos no funcionales del sistema	39
4.6. Requisitos no funcionales personales	39
6.1. Parámetros de la función <code>suma</code>	53
6.2. Parámetros de la función <code>product</code>	54
6.3. Parámetros de la función <code>F128</code>	55
6.4. Parámetros de la función <code>inverse</code>	55
6.5. Parámetros de la función <code>mayor</code>	56
6.6. Parámetros de la función <code>matrix_F2tol28</code>	56
6.7. Parámetros de la función <code>matrix3d_F2tol28</code>	57
6.8. Parámetros de la función <code>matrix_sum</code>	58
6.9. Parámetros de la función <code>matrix_product</code>	58
6.10. Parámetros de la función <code>matrix_product_F2</code>	59
6.11. Parámetros de la función <code>matrix_transpose</code>	59
6.12. Parámetros de la función <code>matrix_identity</code>	60
6.13. Parámetros de la función <code>matrix_rref</code>	61
6.14. Parámetros de la función <code>clavePrivada</code>	62
6.15. Parámetros de la función <code>generacionT</code>	62
6.16. Parámetros de la función <code>clavePublica</code>	63
6.17. Parámetros de la función <code>signature</code>	64

6.18. Parámetros de la función <code>verify</code>	65
7.1. Tiempos de generación de la clave pública	68

Código

6.1. Línea 108 app-core.sh	46
6.2. Modificación archivo hash.ts funciones ECDSA	48
6.3. Modificación archivo hash.ts función signUOV	48
6.4. Modificación archivo hash.ts función verifyUOV	49
6.5. Archivo signature.py	50
6.6. Archivo verify.py	51
6.7. Estructura archivo data.json	52
6.8. Estructura archivo signature.json	52
6.9. Tabla para calcular la potencia en \mathbb{F}_{128}	53
6.10. Tabla para calcular el logaritmo en \mathbb{F}_{128}	53
6.11. Suma de dos elementos del cuerpo	54
6.12. Producto de dos elementos del cuerpo	54
6.13. Convierte un entero en un elemento del cuerpo	55
6.14. Inverso de un elemento del cuerpo	55
6.15. Compara dos elementos del cuerpo	56
6.16. Matriz de \mathbb{F}_2 a un elemento del cuerpo 128 elementos	56
6.17. Vector de matrices de \mathbb{F}_2 a una matriz de \mathbb{F}_{128}	57
6.18. Suma de dos matrices con elementos en el cuerpo	58
6.19. Producto de matrices con elementos del cuerpo	58
6.20. Producto de matrices con elementos en \mathbb{F}_2	59
6.21. Matriz transpuesta	59
6.22. Matriz identidad del cuerpo	60
6.23. Método de Gauss-Jordan	61
6.24. Generación clave privada	62
6.25. Generación matriz T	62
6.26. Generación clave pública	63
6.27. Firma del mensaje	64
6.28. Verificación de la firma	65
A.1. Instalación Docker. Parte I	77
A.2. Instalación Docker. Parte II	77
A.3. Instalación Docker. Parte III	77
A.4. Instalación Docker. Parte IV	78
A.5. Instalación Docker. Parte V	78
A.6. Instalación Docker. Parte VI	78

A.7. Instalación Docker. Parte VII	78
A.8. Comandos útiles de Docker	78
A.9. Instalación <i>blockchain</i> . Parte I	79
A.10.Instalación <i>blockchain</i> . Parte II	79
A.11.Instalación <i>blockchain</i> . Parte III	79
A.12.Instalación <i>blockchain</i> . Parte IV	80
A.13.Instalación <i>blockchain</i> . Parte V	80
A.14.Instalación <i>blockchain</i> . Parte VI	80
A.15.Instalación <i>blockchain</i> . Parte VII	80
A.16.Instalación <i>blockchain</i> . Parte VIII	80
A.17.Comandos útiles de pm2	80
A.18.Instalación <i>blockchain</i> . Parte IX	81
A.19.Línea 108 app-core.sh	81
A.20.Instalación <i>blockchain</i> . Parte X	81
A.21.Clonar nuevo <i>core</i>	81
A.22.Ejemplo fichero <i>data.json</i>	82
A.23.Ejemplo fichero <i>signature.json</i>	82
A.24.Instalación <i>blockchain</i> . Parte XI	83
A.25.Instalación <i>blockchain</i> . Parte XII	83
A.26.Instalación <i>blockchain</i> . Parte XIII	84
A.27.Instalaciones previas a la aplicación ARK Wallet	85
A.28.Instalación de la aplicación ARK Wallet	85

Capítulo 1

Introducción

El objetivo de este proyecto es evitar que un sistema *blockchain* sea vulnerable a futuros ataques cuánticos. Para ello se ha implementado un algoritmo criptográfico resistente a ordenadores cuánticos, denominado [UOV](#), para la firma de bloques, y posteriormente se ha integrado en la *blockchain* ARK.

1.1. Motivación y contexto del proyecto

La tecnología ha transformado nuestra sociedad en una sociedad digitalizada, donde actualmente, los dispositivos digitales comportan la mayor parte de nuestras actividades diarias en distintos ámbitos, como económicas, organizativas o sociales. En el proceso de digitalización de la sociedad podemos distinguir las siguientes cinco fases [\[5\]](#).

La primera fase o era del Internet, corresponde a mediados de los 90. En esta fase se comenzaron a crear páginas web para que los medios de comunicación y las empresas pudieran publicar y compartir información.

La segunda fase o era de las redes sociales, tuvo mayor auge a partir de 2005. Plataformas de bajo o ningún coste, se utilizaban en las empresas para poder llegar mejor a los clientes.

La tercera fase o era de la economía colaborativa, nació con la crisis de 2008 cuando las empresas tenían pocos recursos. Surgieron plataformas para conectar a las personas, y poder obtener lo que necesitasen unas de otras. Por ejemplo pagos online, ver recomendaciones y reseñas de un alojamiento o pedir un taxi. Además se da un gran paso ya que estas aplicaciones pasan de estar alojadas en ordenadores a teléfonos inteligentes.

La cuarta fase o era del mundo autónomo, se ha desarrollado durante décadas. Se desarrollan tecnología con inteligencia artificial, es decir, que simulan la

inteligencia de los humanos para poder resolver problemas más complejos.

Quinta fase o era del bienestar moderno, comienza con las pulseras inteligentes como *Fitbit* o *Fuelband* de Nike. Estas pulseras son el impulso de la tecnología para facilitar la vida de los clientes y poder integrar la tecnología en la vida de los mismos.

La digitalización debe de venir acompañada de mecanismos que aporten seguridad a los datos. Los pilares de la seguridad de la información son los conocidos como la tríada CIA (confidencialidad, integridad y disponibilidad) [6].

La **confidencialidad** es la propiedad que impide que la información pueda ser accesible por entidades no autorizadas. Un sistema garantiza la confidencialidad cuando un tercero entra en posesión de la información intercambiada entre el remitente y el destinatario, no es capaz de extraer ningún contenido legible. Para asegurar la confidencialidad se utilizan mecanismos de cifrado y ocultación de la comunicación.

La **integridad** busca mantener la exactitud de los datos, es decir, que no hayan sido modificados durante su envío. La integridad se obtiene adjuntando al mensaje otro conjunto de datos de comprobación de la integridad, un ejemplo es la firma digital.

La **disponibilidad** es la cualidad de la información de encontrarse a disposición de quienes deben acceder a ella, ya sean personas, procesos o aplicaciones, en los momentos que así lo quieran. Los mecanismos para asegurar la disponibilidad se implementan con la infraestructura tecnológica.

Además de estos tres pilares hay otro principio, la **autenticación**, que es la propiedad que permite identificar al generador de la información. Trata de comprobar si un mensaje enviado por un usuario, ha sido verdaderamente firmado por él mismo. Esto se consigue con el uso de cuentas de usuario y contraseñas de acceso.

Para garantizar estos servicios de seguridad se hace uso de protocolos de seguridad de la información entre los que se encuentra la criptografía, la lógica y la autenticación.

La criptografía se ocupa de cifrar ciertos mensajes con el fin de hacerlos ilegibles a receptores no autorizados, una vez que llega a su destino y sea descifrado, el receptor obtendrá el mensaje original [7]. Además dota de seguridad a las comunicaciones, a la información y a las entidades que se comunican.

Podemos diferenciar dos tipos de criptografía, la criptografía simétrica y la

asimétrica. La criptografía simétrica utiliza la misma clave para cifrar y descifrar un mensaje, esta clave la ha de conocer tanto el emisor como el receptor. Mientras que la asimétrica utiliza dos claves la pública y la privada.

En la criptografía asimétrica podemos diferenciar dos ramas, el cifrado de clave pública y las firmas digitales [8]. En el cifrado de clave pública, el emisor cifra el mensaje con la clave pública del destinatario y el receptor lo descifra con su propia clave privada. En las firmas digitales, el emisor firma el mensaje con su clave privada y el receptor puede verificar el mensaje con su propia clave pública, además cualquier manipulación del mensaje se refleja en su resumen o *hash*.

Este tipo de criptografía basa su seguridad en la hipótesis de que no se pueden encontrar las claves por fuerza bruta con la tecnología existente en la actualidad. Los ataques de fuerza bruta tratan de recuperar las claves probando todas las posibles combinaciones hasta encontrar la que permite el acceso, a partir del algoritmo de cifrado y del texto cifrado con su original [9]. Para que la búsqueda tenga éxito se deberán de realizar $10^n - 1$ operaciones donde n es la longitud de la clave.

Otro factor importante, en la seguridad, es si en la clave aparecen números, caracteres o la combinación de ambos, aumentando así el coste de encontrar las claves, llegando a alcanzar tiempos de cálculo logarítmicos, es decir, que podrían tardar siglos en encontrar una contraseña compleja pero también depende de la capacidad de operación del ordenador.

En este contexto, la aparición de la futura computación cuántica permitirá el cálculo de operaciones a una velocidad mucho mayor. En la figura 1.1 podemos observar la capacidad de cómputo del peor ordenador cuántico, con la línea continua, que sigue la gráfica de una función exponencial, frente a la capacidad del mejor ordenador clásico, la línea discontinua, que sigue una función lineal.

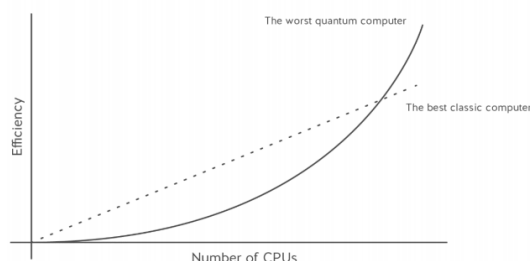


Figura 1.1: Comparativa de la capacidad de cómputo de un ordenador clásico con un ordenador cuántico[1]

La comparativa también nos muestra que para operaciones pequeñas como, por ejemplo, editar un documento de texto un ordenador cuántico sería probablemente ineficiente. Por tanto lo mejor sería un ordenador híbrido, que mezclas computación clásica, para cálculos pequeños, y computación cuántica, para operaciones de mayor tamaño.

Cuando esté desarrollado el ordenador cuántico no serán válidos los actuales algoritmos criptográficos de clave pública, como [RSA](#), Diffie-Hellman y [ECDSA](#), ya que se basan en los problemas del logaritmo discreto y factorización de enteros, resolubles fácilmente por un ordenador cuántico. Las primeras ideas de la criptografía cuántica aparecieron en los años 70, destacando los algoritmos de Shor y Grover.

Veamos la tabla comparativa [1.1](#). Esta nos indica el tipo de algoritmo criptográfico, el algoritmo con la longitud de la clave y a continuación el nivel de seguridad tanto en un ordenador clásico como en uno cuántico. El nivel de seguridad de un algoritmo nos indica el número de operaciones necesarias para romper dicho algoritmo, por ejemplo, si tiene un nivel de seguridad n entonces se requieren 2^n operaciones para romper el algoritmo [\[10\]](#). Observamos que hay una diferencia considerable en los niveles de seguridad de los algoritmos asimétricos, puesto que con un ordenador clásico al menos se necesitan 2^{112} operaciones mientras que con computación cuántica solo una.

Tipo	Algoritmo-Longitud clave	Nivel seguridad (ordenador clásico)	Nivel seguridad (ordenador cuántico)	Ataque cuántico
Asimétrico	RSA-2048	112	0	Algoritmo de Shor
	RSA-3072	128	0	Algoritmo de Shor
	ECC-521	128	0	Algoritmo de Shor
	ECC-521	256	0	Algoritmo de Shor
Simétrico	AES-128	128	64	Algoritmo de Grover
	AES-256	256	128	Algoritmo de Grover

Tabla 1.1: Niveles de seguridad de ordenadores clásicos y cuánticos [\[4\]](#)

En la actualidad, se están desarrollando muchos algoritmos para que sean resistentes a ataques de tipo cuántico, denominados algoritmos de criptografía post-cuántica [\[11\]](#). Estos ataques afectan principalmente a los algoritmos de clave pública o asimétrica, puesto que para la criptografía simétrica duplicar el tamaño de clave empleada es suficiente para hacerlos seguros y hacer inservible el algoritmo de Grover.

Por otro lado, también en la actualidad está siendo muy relevante la adopción de las *blockchain* como tecnología para ofrecer diversos servicios. Las *blockchain* o cadenas de bloques son listas de transacciones, denominadas bloques, firmadas y unidas con algoritmos criptográficos. Además cada bloque contiene el hash del bloque anterior, lo que se explicará con más detalle en la sección 2.2.

Esta tecnología se ha integrado en diferentes áreas, donde resalta el uso en los servicios financieros o criptomonedas, que aumenta la eficiencia y disminuye los costes. Otro uso de las *blockchain* es en las cadenas de suministro, algunos restaurantes, como Fogo de Chão [12], están empezando a utilizar las *blockchain* para poder rastrear el origen de sus alimentos hasta llegar al propio restaurante, una gran ventaja para encontrar fácilmente si hay algún producto contaminado o en mal estado.

Un ejemplo del uso de las *blockchain* queda reflejado en este proyecto en el que se ha implementado un algoritmo resistente a ataques cuánticos, UOV [13] y se ha adaptado a la *blockchain* ARK [14] para que se utilice dicho algoritmo de firma.

1.2. Objetivos del proyecto y logros conseguidos

El objetivo de este proyecto es modificar el algoritmo de firma y verificación de los bloques de la *blockchain* ARK, para hacerla resistente a ataques cuánticos.

- Implementación del algoritmo UOV: Se ha implementado las funciones de generación de claves tanto públicas como privadas, la función de firma a partir de la clave privada y la función de verificación de la misma con la clave pública. Además ha sido necesario implementar la aritmética de cuerpo finito de 2^7 elementos.
- Integración del algoritmo UOV en la *blockchain* ARK para comprobar su funcionamiento: Se ha modificado el algoritmo de firma dado en la *blockchain* por el algoritmo UOV para aumentar la seguridad.

1.3. Estructura de la memoria

A continuación se muestran los capítulos que presenta la memoria junto con una breve descripción de lo que contiene cada uno.

1. **Introducción:** Presenta la motivación del proyecto y el contexto en el que surge. En este capítulo también se encuentran los objetivos que se persiguen con este trabajo.
2. **Contenidos teóricos:** Incluye una breve reseña introduciendo las dos tecnologías que se han utilizado computación cuántica y la *blockchain*, aparte de la explicación matemática del algoritmo utilizado.
3. **Planificación y costes:** Contiene dos diagramas de Gantt que reflejan el seguimiento del proyecto, el primero muestra la planificación inicial y el segundo la planificación realista. Incluye también el presupuesto del proyecto.
4. **Análisis del problema:** Descripción de las funcionalidades y requisitos, y análisis de los objetivos que se muestran en la sección 1.2.
5. **Diseño:** Se encuentra el diseño de la implementación del algoritmo **UOV** y el diseño del ecosistema ARK, donde se integrará el algoritmo de firma.
6. **Implementación:** Contiene la explicación de la implementación del algoritmo **UOV** y la aritmética del cuerpo finito de 2^7 elementos.
7. **Evaluación y pruebas:** Abarca pruebas sobre los tiempos de ejecución del algoritmo y pruebas de integración comprobando que se ejecuta sin errores.
8. **Conclusiones:** Se explica que se ha conseguido con la realización del proyecto y algunas líneas de investigación para continuar con desarrollo del mismo.
9. **Apéndice:** Muestra el manual de usuario con los pasos que se han seguido para realizar este proyecto.

Capítulo 2

Contenidos teóricos

En los siguientes apartados se explican los contenidos claves de este proyecto, que son la computación cuántica [2.1](#), la tecnología *blockchain* [2.2](#) y el algoritmo *UOV* [2.3](#).

2.1. Computación cuántica

La evolución de la tecnología se ha basado principalmente en la reducción de los transistores para aumentar la velocidad, llegando a escalas de tan solo algunas decenas de nanómetros. Esto tiene un límite y es la eficiencia, puesto que al seguir disminuyendo el tamaño podrían dejar de funcionar correctamente. De ahí surge la necesidad de descubrir nuevas tecnologías, la computación cuántica [\[15\]](#). Así, la computación cuántica constituye un nuevo paradigma de la informática basado en los principios de la teoría cuántica.

Las tecnologías cuánticas nacieron del estudio de algunos fenómenos físicos que aún no se entendían bien, entre los años 1900 y 1930, dando lugar a una nueva teoría en la física, la Mecánica Cuántica. La Mecánica Cuántica es la rama de la física que estudia del mundo microscópico, los sistemas atómicos y subatómicos y su interacción con la radiación electromagnética [\[16\]](#).

2.1.1. Historia

La computación cuántica tuvo sus inicios en los años 50 cuando algunos físicos, como Richard Feynman, fueron pioneros en mencionar posibilidad de utilizar efectos cuánticos para realizar cálculos computacionales [\[15\]](#). En la charla de Richard Feynman titulada “Simulación de la física con computadoras”, a principio de la década de los 80, expuso algunos cálculos complejos que se podrían realizar más rápido con un ordenador cuántico.

A finales de los años 60, Stephen Wiesner escribe un artículo titulado “Conjugate Coding”, donde expone un primer acercamiento a la criptografía cuántica. El artículo fue publicado en los años 80 [17].

En 1981 Paul Benioff expone las ideas esenciales de la computación cuántica acompañada de su teoría, en la que propuso que un ordenador clásico trabajara con algunos principios de la mecánica cuántica, y aprovechar así las leyes cuánticas.

En la década de 1990 ya empezaron a poner en práctica algunas teorías, apareciendo los primeros algoritmos cuánticos, primeras aplicaciones cuánticas y las primeras máquinas diseñadas para realizar cálculos cuánticos. Así en 1991, Artur Ekert desarrolla una aproximación diferente a la distribución de claves cuántica (QKD) basado en el entrelazamiento cuántico.

En 1993 hubo varios acontecimientos, por un lado, Dan Simon comparó el modelo de probabilidad clásica con el cuántico, esto se utilizó para el desarrollo de futuros algoritmos cuánticos. Por otro, Charles Bennett acuñó el término del teletransporte cuántico, abriendo una vía de investigación para las comunicaciones cuánticas. Además Ekert organizó la primera conferencia internacional de criptografía cuántica en Inglaterra, primer evento de gran alcance dedicado a este área.

Peter W. Shor definió un algoritmo cuántico, el algoritmo de Shor, que permite calcular los factores primos de números muy grandes en tiempo polinomial, resolviendo el problema de la factorización de enteros como el problema del logaritmo discreto. Como consecuencia, el algoritmo de Shor permite romper muchos sistemas criptográficos actuales. Un año más tarde, propuso un sistema de corrección de errores en el cálculo cuántico.

Lov Grover, en 1996, expone un algoritmo de búsqueda en una secuencia de datos no ordenada con N elementos, denominado algoritmo de Grover, que tiene una complejidad en tiempo de $O(\sqrt{n})$.

En 1997, tiene lugar el primer experimento de comunicación con criptografía cuántica a una distancia de 23km. Además del primer teletransporte cuántico de un fotón.

A finales de los 90, los laboratorios IBM-Almadén crearon la primera máquina con 3 cúbits y ejecutó el algoritmo de Grover. Y en 2001, IBM junto con la Universidad de Stanford ejecutaron el algoritmo de Shor en un computador cuántico con 7 cúbits, se calcularon los factores primos del número quince.

En 2004, sale a la luz el primer criptosistema cuántico comercial (QKD), creado por la ID Quantique.

En 2007, D-Wave fabricó una máquina que utilizaba mecánica cuántica con 16 cúbits sin llegar a ser un computador cuántico, especializado en la optimización de problemas a través de algoritmo de temple cuántico. En septiembre de ese mismo año, consiguieron unir componentes cuánticos a través de superconductores, apareciendo el primer bus cuántico capaz de retener información cuántica durante un corte espacio de tiempo antes de volver a ser transferida. Un año después se consiguió almacenar un cúbit en el interior del núcleo de un átomo de fósforo y hacer que la información permaneciera intacta durante 1.75 segundos.

Pasaron varios años hasta que se vendió la primera computadora cuántica comercial, en 2011, por la empresa D-Wave Systems por 10 millones de dólares.

En 2018, la Universidad de Innsbruck consiguen un entrelazamiento estable de 20 cúbits, marcando el récord actual. El 18 de septiembre de 2019, IBM anunció que pronto lanzará un ordenador cuántico de 53 cúbits, el más grande y potente hasta la fecha.

En la actualidad, Google ha logrado aplicar supercomputadores al mundo real, simulando con éxito una reacción química simple. Marcando el camino hacia la química cuántica. Esto podría ayudar a los científicos a comprender mejor las reacciones moleculares, dando lugar a descubrimientos útiles como mejores baterías, nuevas formas de producir fertilizante y métodos para eliminar el dióxido de carbono del aire [18].

2.1.2. Estructura de los cúbits

La computación clásica funciona con bits cuyos valores pueden ser 0 o 1, mientras que la computación cuántica funciona con bit cuánticos o cúbits, una combinación de 0 y 1, pudiendo tomar ambos valores a la vez. Esto se denomina la superposición cuántica de los estados [19]. Se entrará en detalle más adelante.

La figura 2.1 muestra los estados de un bit y los posibles estados que puede tomar un cúbit.

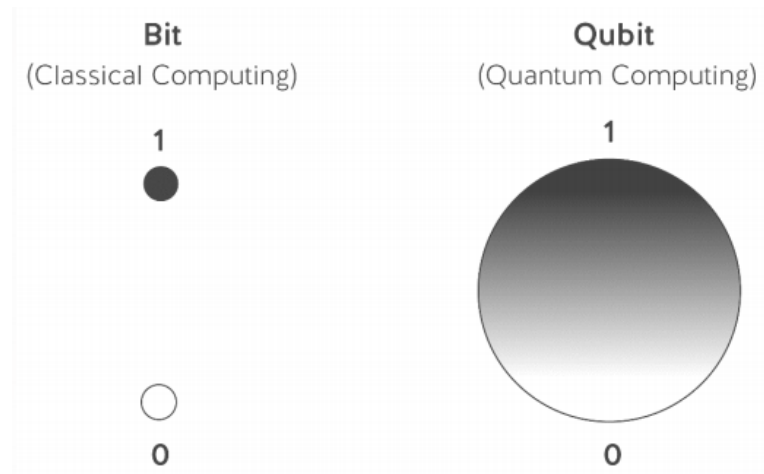


Figura 2.1: Estados de un bit y de cúbit [1]

El espacio de estados de un cúbit se puede representar mediante un espacio vectorial complejo bidimensional, al no ser práctico, se aprovecha el homeomorfismo entre la superficie de una esfera y el plano complejo cerrado con un punto en el infinito, dando lugar a lo que se conoce como la esfera de Bloch.

Una esfera de Bloch es una representación geométrica del espacio de estados puros de un sistema cuántico de dos niveles. Además se representa en el espacio \mathbb{R}^3 por la esfera de radio unidad como se observa en la figura 2.2, donde cada punto de la esfera es un posible estado del cúbit.

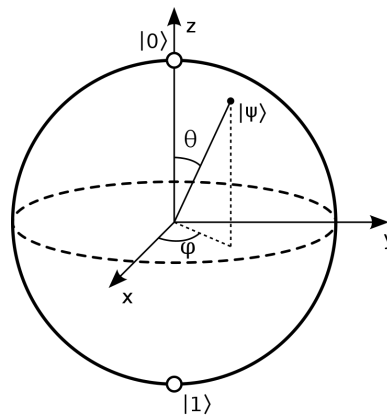


Figura 2.2: Estructura cúbit, esfera de Bloch [2]

Un cúbit se puede representar como una combinación lineal de los estados $|0\rangle$ y $|1\rangle$, ecuación (2.1).

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad (2.1)$$

Como α y β son números complejos, la ecuación (2.1) se puede escribir en forma exponencial, ecuación (2.2).

$$|\psi\rangle = r_\alpha e^{i\phi_\alpha} |0\rangle + r_\beta e^{i\phi_\beta} |1\rangle \quad (2.2)$$

2.1.3. Propiedades

Entre las propiedades cuánticas destacan la superposición cuántica, el entrelazamiento cuántico y el teletransporte cuántico.

La **superposición cuántica** describe cómo una partícula puede estar en diferentes estados al mismo tiempo. Esto aporta gran capacidad de procesamiento, lo que hace posible resolver de manera eficiente problemas de mayor complejidad como la factorización de enteros, el algoritmo discreto y la simulación cuántica, que a día de hoy con los ordenadores clásicos son difíciles de romper.

Otro aspecto importante de la física cuántica relacionado con la superposición es el **entrelazamiento cuántico** de las partículas[20]. Esto es, si dos partículas en algún instante han interactuado retienen un tipo de conexión y pueden entrelazarse formando pares, de forma que al interactuar con una de las partículas, por muy separadas que estén, la otra se entera. Esto permite que aunque los cúbits estén separados interactúen entre sí. Con estos dos aspectos la capacidad de procesamiento aumenta considerablemente, cuántos más cúbits la capacidad de procesamiento aumenta considerablemente.

Por último, el **teletransporte cuántico** utiliza el entrelazamiento para enviar información de un lugar del espacio a otro sin necesidad de viajar a través de él.

2.2. Blockchain

Blockchain es un sistema de almacenamiento de información que se divide en bloques de datos enlazados mediante hash. A cada bloque se le asocia un hash a partir del bloque anterior, creando una lista enlazada, la búsqueda de información no es muy óptima si hay un número elevado de bloques. Para la

búsqueda eficiente en *blockchain* se usan los árboles de Merkle.

Los datos que almacena cada bloque son transacciones válidas, información referente a ese bloque y la relación con el bloque anterior mediante el *hash*, por tanto el bloque tiene un lugar específico dentro de la cadena. De esta forma si hay una alteración en un determinado bloque se verá reflejado en su *hash* y en el de los bloques posteriores, haciendo que la información de la cadena no se pueda perder, modificar o eliminar.

Los árboles de Merkle [21] son una estructura de datos en árbol en el que cada nodo que no es hoja está etiquetado con el *hash* que surge de la combinación de los valores o etiquetas de sus nodos hijo. Esta estructura permite que aunque los datos estén separados puedan ser ligados a un único valor de *hash*, el *hash* del nodo raíz del árbol. El *hash* de este nodo va firmado para asegurar la integridad y hacer que la verificación sea fiable.

De esta forma se asegura que los datos son recibidos sin daños y sin ser alterados, además permite que los datos puedan ser entregados por partes, ya que un nodo puede obtener solo la cabecera de un bloque desde una fuente y otra pequeña parte del árbol desde otra fuente, pudiendo asegurar que los datos son correctos. Esto funciona porque si un usuario intenta hacer un cambio en una transacción falsa en la parte inferior del árbol en seguida se verá reflejado en la parte superior del árbol, es decir, en el nodo raíz. Esta propiedad se ve clara en la estructura del árbol de Merkle 2.3, donde los *hash* de los nodos superiores se calculan a partir de los *hash* de los nodos hijos.

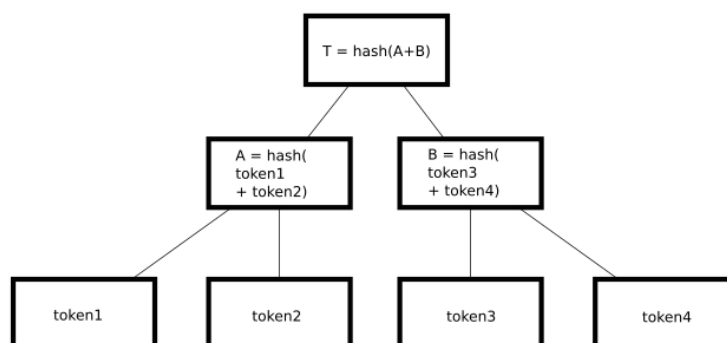


Figura 2.3: Estructura de un árbol de Merkle [3]

La idea de la tecnología *blockchain* surge a comienzos de 1991 cuando los científicos Stuart Haber y W. Scott Stornetta introducen una solución computacional para la firma de documentos digitales y que no pudieran ser modificados con el tiempo. Usaron cadenas de bloque para almacenar los documentos

con sello de tiempo y en 1992 se incorporaron los árboles de Merkle, que podían recopilar varios documentos en un bloque haciendo el diseño más eficiente. Sin embargo, esta tecnología no se utilizó y la patente caducó en 2004 [22].

En 1998, Nick Szabo trabaja en una moneda digital descentralizada, “bit gold”. Dos años después Stefan Konst publica su teoría sobre la seguridad criptográfica en las cadenas de bloques junto con algunas ideas de implementación [23].

En 2004, el informático y criptógrafo Harold Thomas Finney introdujo el sistema **RPoW** (prueba de trabajo reutilizable). El sistema se basa en *HashCash* pero los tokens de prueba no están ligados a una aplicación sino que pueden ser gastados libremente como una moneda. Los clientes pueden crear tokens e intercambiarlos sin necesidad de regenerarlos [24]. **RPoW** resolvió el problema del doble gasto registrando los tokens en un servidor fiable diseñado para permitir a los usuarios verificar su exactitud e integridad en tiempo real. Este sistema puede considerarse como un prototipo de las criptomonedas.

A finales de 2008, un grupo de desarrolladores bajo el nombre de Satoshi Nakamoto publican un documento técnico en que se establece un modelo para *blockchain*. Está basado en el algoritmo **RPoW** pero en lugar de usar dicho hardware, se utiliza un protocolo descentralizado peer-to-peer para verificar y rastrear las transacciones. En otras palabras los “mineros” extraen bitcoins para obtener una recompensa mediante pruebas de trabajo y posteriormente los nodos los verifican. Bitcoin nació el 3 de enero de 2009 cuando Satoshi Nakamoto extrajo el primer bloque de bitcoin con una recompensa de 50 bitcoins. Y el 12 de enero de 2009 tuvo lugar la primera transacción entre Satoshi Nakamoto y Hal Finney que obtuvo 10 bitcoins.

A partir de 2014, se comienzan a explorar el potencial de las cadenas de bloque y a buscar otras aplicaciones fuera de su uso en las transacciones financieras.

Ethereum introduce programas informáticos que se ejecutan en la *blockchain*, se pueden utilizar para realizar una transacción cumpliendo ciertas condiciones como los contratos inteligentes.

Los contratos inteligentes se tratan de contratos que tienen la capacidad de cumplirse de forma automática. Un contrato inteligente está constituido por un protocolo de códigos que permiten a un dispositivo ejecutar de forma automatizada las sentencias previamente programadas, prescindiendo de la intervención humana [25].

Además de los contratos inteligentes, Ethereum tiene su propia criptomoneda llamada Ether, se puede transferir entre cuentas y se utiliza para pagar las tarifas por la ejecución de los contratos inteligentes.

Otro proyecto de *blockchain* a destacar es Hyperledger[26], iniciado en diciembre de 2015 por la Fundación Linux. Hyperledger es una plataforma de código abierto centrada en el desarrollo de un conjunto de *frameworks*, herramientas y bibliotecas para implementaciones de cadenas de bloques a nivel empresarial.

Hyperledger contiene varios *frameworks* de contabilidad distribuidos, en los que se incluye Hyperledger Fabric, Sawtooth, Indy, así como herramientas como Hyperledger Caliper. Y bibliotecas como Hyperledger Ursa.

El proyecto Hyperledger pretende la mejora de aspectos como el rendimiento y fiabilidad de las empresas, además de agilizar los procesos comerciales de las industrias. Para ello ha de contar con el apoyo de las diferentes empresas (como Accenture, Fujitsu, IBM o Intel) formando así un proyecto colaborativo[27].

Dentro de Hyperledger, el proyecto más destacado es Hyperledger Fabric. Este es uno de los dos proyectos originales de Hyperledger en el que colaboraba Digital Asset Holding, Blockstream e IBM. Es una *blockchain* privada, de las más conocidas, que aspira a facilitar la implementación de cualquier modelo de uso. Además permite el despliegue de contratos inteligentes desarrollados en el lenguaje de programación de Google, “Golang”. Dicha *blockchain* posee un diseño muy flexible ya que permite crear contratos inteligentes en cualquier lenguaje o decidir qué algoritmo de consenso utilizar en la red (por defecto usa Practical Byzantine Fault Tolerance, [PBFT](#))[28].

Las cadenas de bloques o *blockchain* permiten verificar, validar, rastrear todo tipo de información, ya sean contratos inteligentes, transacciones financieras, certificados digitales o firmas [29], siendo estas últimas el centro de este trabajo. También permiten impulsar modificaciones orientadas a crear soluciones más robustas, por ejemplo en centros de salud o notarías que se explicarán más adelante.

Las *blockchain* son vulnerables a futuros ataques cuánticos ya que su única línea de defensa es el algoritmo de firma de los bloques. Aunque, actualmente las cadenas de bloques son seguras, puesto que un ordenador clásico no tiene la capacidad de cómputo necesaria para descifrar cada bloque, obtener la información y volver a firmar todos los bloques sin dejar huella. Por eso para hacer una *blockchain* resistente es necesario tener un criptosistema que no se pueda romper con computación cuántica, como por ejemplo el algoritmo [UOV](#), ver apartado 2.3.

Los tres pilares de la tecnología *blockchain* son la descentralización, transparencia e inmutabilidad [30].

Un sistema centralizado almacena todos los datos en una misma entidad y habría que interactuar con la misma para obtener la información necesaria. Un ejemplo de un sistema centralizado son los bancos que almacenan todo el dinero y la única forma de pagar a alguien es a través de un banco. Es similar a la arquitectura cliente-servidor donde los clientes se comunican entre ellos mediante el servidor. Pero tener un único sitio para almacenar todos los datos es vulnerable a los ataques, informáticos, por otra parte si el nodo central se corrompe o tiene una actualización, los datos serán incorrectos o no se podrán acceder a ellos. De los contras de los sistemas centralizados surge la idea de los sistemas **descentralizados**, la información no la tiene un único nodo sino que todos los usuarios son dueños de la información. La principal ideología de las *blockchain* es poder interactuar usuario con usuario sin tener que pasar por un tercero.

El concepto **transparencia** se refiere a la transparencia de los datos no de las identidades. Esto es la identidad de la persona se oculta a través de la criptografía y lo único que se ve es su dirección pública, pero podemos ver todas las transacciones que se han realizado en su dirección pública. En el historial de transacciones no vemos “Antonio envió 1BTC” sino que aparece “1MF1bhsFLkBzzz9vpFYEmvwT2TbyCt7NZJ envió un 1BTC”. Este nivel de transparencia nunca antes había existido en el sistema financiero, lo que exige más responsabilidad a las grandes empresas. De la misma forma podemos trasladar este concepto fuera del sistema financiero por ejemplo a las cadenas de suministro, y saber exactamente de donde provienen los alimentos de un restaurante.

La **inmutabilidad** en el contexto de las cadenas de bloques significa que una vez introducida una transacción en la *blockchain* ya no se puede alterar. De esta forma aplicando esta tecnología a los bancos se evitarían casos de malversación de fondos. Esta propiedad se obtiene gracias a la función criptográfica *hash*.

La función *hash* es el resultado de aplicar una función que transforma un mensaje de longitud variable en uno de longitud fija. Esto es calcular el resto módulo n con n la longitud fija. Al aplicar la función hash a un fichero, si se modifica algún dato del mismo cambiará su hash y por tanto se sabrá si ha sido manipulado desde que se envió, consiguiendo la integridad del mensaje.

De la misma forma si hay un cambio en una de las transacciones de un bloque se reflejará en el *hash* del bloque, afectado a todos los bloques anteriores. Así si el atacante quiere preservar la integridad deberá de modificar todos los

bloques siendo una tarea imposible. De esta forma se obtiene la inmutabilidad de los datos.

Hoy en día, la tecnología *blockchain* está ganando mucha atención, no limitándose solo al uso en las criptomonedas. Así las cadenas de bloques tienen diversas aplicaciones entre ellas se encuentra la salud o la firma de documentos en las notarías. En el primer caso, cada centro de salud podría tener el historial médico de cualquier paciente, de una forma segura y evitando falsificaciones, estos historiales se encontrarían en nodos distribuidos de forma descentralizada así se obtendría un acceso rápido y seguro. El segundo caso será en el que nos centraremos a lo largo de este proyecto. Hoy día la firma de documentos o transacciones por parte de un usuario es un problema puesto que se pueden copiar con facilidad, pero con *blockchain* no podrían ser falsificadas debido a la propiedad de validación y rastreo de los datos.

2.2.1. Blockchain ARK

ARK es un ecosistema de criptomonedas descentralizado diseñado para aumentar la adopción de la tecnología *blockchain* por parte de los usuarios. Se busca que la implementación de la cadena de bloques sea eficiente y fácil, y de esta forma poder usar dicha plataforma como puente inteligente entre varias cadenas de bloques no necesariamente de ARK. Cualquier desarrollador puede programar ARK, puesto que admite varios lenguajes de programación como Python, Elixir, Java o API TypeScript entre otros. El *core* de la *blockchain* que se ha utilizado se encuentra en TypeScript.

Algunas ventajas de la *blockchain* ARK son, tener una interfaz de usuario intuitiva para extender el uso a la vida cotidiana, crear puentes inteligentes para posibilitar la comunicación entre diferentes *blockchain*, admitir varios lenguajes de programación, facilidad para duplicar la cadena de bloques para permitir que se utilice para necesidades personales y por último, los coste de red son bastante bajos[31].

Hay dos motivos principales por los que se ha elegido esta *blockchain* ARK y no otra. En primer lugar, la cadena de bloques es completamente de código abierto, permite a cualquier persona poder contribuir. Y en segundo lugar, su arquitectura modular permitiendo personalizar la aplicación para alcanzar nuestro objetivo (modificar el algoritmo de firma y verificación).

Un logro significativo del ecosistema ARK es brindar soporte a los contratos inteligentes directamente desde la cadena de bloques ARK utilizando puentes inteligentes. La capacidad de crear, lanzar y administrar contratos inteligentes entre dos tecnologías *blockchain* diferentes sin ningún problema, abre el campo

de posibilidades para crear soluciones comerciales.

Hyperledger Fabric agrega soporte al núcleo ARK v2 para contratos inteligentes. Dicho núcleo de ARK con toda la integración se ejecuta en los nodos conectados a la cadena de bloques principal de ARK. También la cadena Hyperledger personalizada puede agregarse a los mismos nodos o a un conjunto de nodos independientes[32].

2.3. Algoritmo UOV

El algoritmo aceite y vinagre desequilibrado[13] es una versión simplificada del algoritmo aceite y vinagre, ambos algoritmos de firma digital. Para crear las firmas y validarlas es necesario resolver un sistema con m ecuaciones y n variables, que es un problema NP-duro[33], lo que significa que si fuésemos capaces de resolverlo con un ordenador cuántico, todos los problemas considerados en la actualidad serían vulnerables. Si m y n son casi iguales o iguales será más difícil resolver el sistema, obteniendo de esta forma un algoritmo de firma resistente a ataques cuánticos.

La principal ventaja del algoritmo UOV es que es un algoritmo post-cuántico, a diferencia de otros esquemas de firmas como RSA, DSA, basado en el logaritmo discreto, y su variante para curvas elípticas ECDSA que no permanecerían seguros ante un ordenador cuántico. Esto se debe a que en la actualidad no existe un algoritmo eficiente para la resolución de sistemas multivariados de ecuaciones en ordenador cuántico. Otra ventaja es la simplicidad de las operaciones utilizadas, ya que las firmas se crean y validan con operaciones de suma y multiplicaciones de valores pequeños, lo que requiere bajos recursos *hardware*.

Aunque el algoritmo usa sistemas pequeños y la longitud de las firmas son pequeñas, se necesitan claves públicas bastante más grandes en comparación con otros algoritmos de firma como ECDSA, pudiendo ocupar dicha clave pública varios kilobytes de almacenamiento. Por otro lado ya se conocen algunos métodos de ataque, probablemente aparecerán más si se empieza a comercializar.

2.3.1. Cuerpos finitos

Se trabajará con el cuerpo finito de 128 elementos, \mathbb{F}_{2^7} , extensión de grado 7 del cuerpo \mathbb{F}_2 de los enteros módulo 2

$$\mathbb{F}_{128} = \frac{\mathbb{F}_2[x]}{\langle x^7 + x + 1 \rangle} \quad (2.3)$$

Además el orden del cuerpo de las unidades es 127, que es primo. Entonces todo elemento del cuerpo distinto de 1 es un elemento primitivo, es decir, un generador.

La tabla 2.1 muestra una representación de los elementos no nulos del cuerpo. En la implementación se ha utilizado la representación como cadena de bits, puesto que a la hora de trabajar es más fácil con una cadena de bits que con los polinomios.

Polinomio	Bits	\log_a
1	[0, 0, 0, 0, 0, 0, 1]	0
a	[0, 0, 0, 0, 0, 1, 0]	1
a^2	[0, 0, 0, 0, 1, 0, 0]	2
\vdots	\vdots	\vdots
$a^6 + a^5 + a^4 + 1$	[1, 1, 1, 0, 0, 0, 1]	124
$a^6 + a^5 + 1$	[1, 1, 0, 0, 0, 0, 1]	125
$a^6 + 1$	[1, 0, 0, 0, 0, 0, 1]	126

Tabla 2.1: Representación de los elementos no nulos de \mathbb{F}_{128}

La implementación del cuerpo finito de 2^7 elementos no se ha realizado de forma genérica sino para que sea específica para el algoritmo UOV, de esta forma es mucho más sencillo implementar la aritmética del cuerpo. Para la suma en \mathbb{F}_2 sólo tenemos que fijarnos que es lo mismo que el operador lógico XOR, mientras que para el producto, al encontrarnos en un cuerpo como un orden pequeño, se usarán unas tablas que contienen las correspondencias entre los elementos no nulos del cuerpo y sus logaritmos en base a , por lo que el producto se convierte en una suma módulo 127.

2.3.2. Parámetros y fórmula

Para empezar se indican los parámetros que serán de utilidad para entender el algoritmo.

- r : Grado del cuerpo extendido, $\mathbb{F}_2 \subset \mathbb{F}_{2^r}$. En la implementación se va tomar r igual a 7, pero se puede realizar con cualquier valor de r sin un esfuerzo adicional.
- m : Tamaño de la clave pública, además del número de variables de aceite.
- v : Número de variables vinagre.

- n : Número total de variables, las de aceite más las de vinagre.
- x : Vector de n componentes, denominando a las primeras v componentes x_1, \dots, x_v vinagre y al resto aceites.

El esquema de la firma UOV utiliza la función unidireccional $\mathcal{P} : \mathbb{F}_{2^r}^n \rightarrow \mathbb{F}_{2^r}^m$, que es una función cuadrática multivariante en $n = m + v$ variables. Esta función se puede descomponer como $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$, donde $\mathcal{T} : \mathbb{F}_{2^r}^n \rightarrow \mathbb{F}_{2^r}^n$ es linear invertible, y $\mathcal{F} : \mathbb{F}_{2^r}^n \rightarrow \mathbb{F}_{2^r}^m$ función cuadrática cuyas m componentes son de la forma:

$$f_k(x) = \sum_{i=1}^v \sum_{j=i}^n \alpha_{i,j,k} x_i x_j + \sum_{i=1}^n \beta_{i,k} x_i \quad (2.4)$$

donde $\alpha_{i,j,k}$ y $\beta_{i,k}$ se toman aleatoriamente en \mathbb{F}_2 siendo $(\alpha_{i,j,k})_{\substack{1 \leq i \leq v \\ 1 \leq j \leq n}}$ un vector de matrices triangulares superiores. De esta manera será más eficiente y no afectará a la seguridad del algoritmo. Se entiende por matriz triangular superior, una matriz no necesariamente cuadrada que tiene debajo de la primera diagonal valores nulos. Las primeras v variables, x_1, \dots, x_v son las variables vinagres y las m variables restantes son las variables aceite.

2.3.3. Generación de la clave privada

La clave privada viene dada para cada una de las m ecuaciones, es decir, para cada k se tiene una matriz de dimensiones $v \times n$, $(\alpha_{i,j,k})_{\substack{1 \leq i \leq v \\ 1 \leq j \leq n}}$, y un vector de v componentes, $(\beta_{i,k})_{1 \leq i \leq v}$, cuyos valores son elegidos de forma aleatoria en \mathbb{F}_2 .

2.3.4. Generación de la clave pública

La clave pública se va a definir para cada $k \in \{1, \dots, m\}$ de la siguiente manera,

- $\alpha_{pub_k} = \left(\frac{I_v}{T'_{v \times m}} \right) (\alpha_{i,j,k})_{\substack{1 \leq i \leq v \\ 1 \leq j \leq n}}^T$
- $\beta_{pub_k} = (\beta_{j,k})_{1 \leq j \leq n}^T$

Ahora se va a justificar porqué se toman estos valores de la clave pública. Primero se pasan las m ecuaciones (2.4) a forma matricial, ecuación (2.5). La notación a seguir para las matrices transpuestas es X^T , con X una matriz.

$$f_k(x) = x^v (\alpha_{i,j,k})_{\substack{1 \leq i \leq v \\ 1 \leq j \leq n}}^T (x^v, x^m)^T + (\beta_{i,k})_{1 \leq i \leq v} (x^v, x^m)^T \quad (2.5)$$

siendo x^v los vinagres y x^m los aceites, por tanto x se puede expresar como $(x^v, x^m)^T$.

Para generar la clave pública es necesario incluir una nueva matriz T con la forma que muestra la ecuación (2.6), a esta matriz se le denomina matriz de distorsión. La matriz de distorsión se incluye para aumentar la seguridad del algoritmo y así sea más complejo calcular la función inversa \mathcal{P} .

$$T = \left(\begin{array}{c|c} I_v & T_{v \times m} \\ \hline 0 & I_m \end{array} \right) \quad (2.6)$$

Además la matriz de distorsión cumple la igualdad $T \cdot s^T = x^T$, donde s es la firma del mensaje, y despejando x se obtiene la ecuación (2.7).

$$x = s \cdot T^T = s \left(\begin{array}{c|c} I_v & 0 \\ \hline T_{v \times m}^T & I_m \end{array} \right) = (s^v, s^m) \left(\begin{array}{c|c} I_v & 0 \\ \hline T_{v \times m}^T & I_m \end{array} \right) = (s^v + s^m T_{v \times m}^T, s^m) \quad (2.7)$$

Sustituyendo la anterior definición de x en la ecuación (2.5), se llega a la ecuación (2.8).

$$f_k(x) = s \left(\begin{array}{c|c} I_v & \\ \hline T_{v \times m}^T & \end{array} \right) (\alpha_{i,j,k})_{\substack{1 \leq i \leq v \\ 1 \leq j \leq n}} T s^T + (\beta_{j,k})_{1 \leq j \leq n} T s^T \quad (2.8)$$

donde $k \in \{1, \dots, m\}$

2.3.5. Algoritmo de firma

Para calcular la firma es necesario conocer los valores de la clave privada α y β , tomar de forma aleatoria los del vinagre x^v , puesto que estos son fijos y no variables pasarán a denominarse como a^v , y coger los m primeros bits del *hash* del mensaje h_k .

Para facilitar los cálculos se multiplican los vinagres por α y se obtiene $A_k = a^v (\alpha_{i,j,k})_{\substack{1 \leq i \leq v \\ 1 \leq j \leq n}} = (A_k^v, A_k^m)$. Si se sustituye este resultado en la ecuación (2.5) y se despejan los aceites, se llega a la ecuación (2.11). Este despeje se puede realizar sin problema, debido a que el sistema de ecuaciones se vuelve lineal cuando las variables de vinagre son fijas y ninguna variable de aceite se multiplica por otra de aceite. Por tanto se puede resolver usando, por ejemplo, el algoritmo de reducción gaussiano.

$$h_k = A_k^v (a^v)^T + A_k^m (x^m)^T + \beta_k^v (a^v)^T + \beta_k^m (x^m)^T \quad (2.9)$$

$$(A_k^m + \beta_k^m)(x^m)^T = h_k - (A_k^v + \beta_k^v)(a^v)^T \quad (2.10)$$

$$(x^m)^T = (A_k^m + \beta_k^m)^{-1} (h_k - (A_k^v + \beta_k^v)(a^v)^T) \quad (2.11)$$

Si $(A_k^m + \beta_k^m)$ fuese una matriz singular, entonces se tomarían otros valores de vinagres.

De esta forma se han calculado los aceites, esto es, las últimas m componentes del vector x . Para calcular la firma, s , se hace uso de la definición de x dada

en la ecuación (2.7), puesto que solo es necesario realizar un simple despeje. Por otro lado, se tiene que T es igual a su inversa, gracias a la definición de T y teniendo en cuenta que la matriz toma valores en \mathbb{F}_2 .

$$s = x \cdot T^{T-1} = x \cdot T^T \quad (2.12)$$

donde $x = (x^v, x^m)$ con x^v son los vinagres aleatorios y x^m los aceites.

2.3.6. Algoritmo de verificación

Para comprobar que el mensaje es correcto y que no ha sufrido ninguna transformación durante el envío del mismo, se utiliza una versión del sistema utilizado para la firma. Se hace modificación para que el atacante no pueda obtener la clave privada ni las variables de aceite y vinagre. Para cada $k \in \{1, \dots, m\}$ se tiene que cumplir la igualdad (2.13).

$$h_k = \alpha_{pub_k} s^T + \beta_{pub_k} s^T \quad (2.13)$$

2.3.7. Ejemplo

Se muestra un ejemplo para mejor comprensión del algoritmo UOV utilizando valores de m y v pequeños, ambos con valor 3, y dejando fijo el valor de r igual a 7.

Generamos la clave privada con valores aleatorios. Notamos que las matrices del vector α_{priv} , ecuación (2.14), son matrices triangulares superiores de dimensión 3×6 , esto es dimensión $m \times n$. Además el número de matrices es 3 el valor de v .

$$\alpha_{priv} = \left(\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \right) \quad (2.14)$$

La segunda parte de la clave privada es β_{priv} , consta de un vector de 3 vectores, donde cada uno tiene n componentes, esto es 6 componentes, ecuación (2.15).

$$\beta_{priv} = ((1 \ 0 \ 0 \ 1 \ 0 \ 0) (0 \ 0 \ 1 \ 0 \ 1 \ 0) (1 \ 1 \ 0 \ 1 \ 0 \ 1)) \quad (2.15)$$

La forma de la matriz T viene dada por la ecuación 2.6, la del ejemplo se muestra en la ecuación (2.16)

$$T = \left(\begin{array}{c|c} I_3 & T_{3 \times 3} \\ \hline 0 & I_3 \end{array} \right) = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right) \quad (2.16)$$

Clave pública consta de dos partes α_{pub} y β_{pub} . Cada componente de α_{pub} se calculan multiplicando tres matrices, una matriz, que tiene dos partes la I_3 y $T'_{3 \times 3}$, por la k -ésima componente de α_{priv} y por T .

$$\begin{aligned}
 \alpha_{pub_1} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \cdot \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right) \\
 &= \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\
 &= \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}
 \end{aligned} \tag{2.17}$$

De la misma forma se obtiene α_{pub_2} y α_{pub_3} , así la ecuación (2.18) muestra el resultado de α_{pub} .

$$\alpha_{pub} = \left(\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \right) \tag{2.18}$$

Ahora se calcula β_{pub} para ello se va a realizar el cálculo de la primera componente del vector β_{pub_1} a modo de ejemplo, multiplicando la primera componente de β_{priv} por T , ecuación (2.19).

$$\beta_{pub_1} = (1 \ 0 \ 0 \ 1 \ 0 \ 0) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = (1 \ 0 \ 0 \ 1 \ 0 \ 0) \tag{2.19}$$

De manera análoga se calculan el resto de componentes, β_{pub_2} y β_{pub_3} , para obtener el vector completo β_{pub} , ecuación (2.20).

$$\beta_{pub} = (1 \ 0 \ 0 \ 1 \ 0 \ 0) (0 \ 0 \ 1 \ 1 \ 1 \ 1) (1 \ 1 \ 0 \ 1 \ 0 \ 0) \tag{2.20}$$

Llegados a este punto comienza el proceso de firma del mensaje, en este caso el mensaje al que se le va a realizar la firma es "Este mensaje es un mensaje de prueba. Quiero se sea un poco largo para que se aprecie el efecto de la función hash", el hash de dicho mensaje calculado con la función sha256 es "c2f", la ecuación (2.21) contiene el *hash* en \mathbb{F}_{128} .

$$hash = ((0001100)(0000010)(0001111)) \quad (2.21)$$

Los vinagres se han tomado de forma aleatoria, ecuación (2.22).

$$a^v = ((0110100)(1111100)(1111001)) \quad (2.22)$$

La k -ésima matriz del vector A es el resultado de multiplicar el vinagre por la matriz α_{priv_k} donde k se mueve entre 1 y 3, para lo cual, es necesario extender los componentes de las matrices de α_{priv} , que se encuentran en \mathbb{F}_2 , al cuerpo \mathbb{F}_{128} . En el ejemplo no se ha puesto la matriz extendida para entender mejor la multiplicación y no perderse con los vectores, finalmente la multiplicación será el vector nulo si se multiplica por 0 o el mismo vector cuando se multiplique por 1.

$$\begin{aligned} A_1 &= ((0110100)(1111100)(1111001)) \cdot \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \\ &= ((0110100)(0000000)(0110100)(1001101)(0110001)(0000101)) \end{aligned} \quad (2.23)$$

Calculando el resto de las componentes se obtiene la matriz A , ecuación (2.24).

$$\begin{aligned} A &= \left(((0110100)(0000000)(0110100)(1001101)(0110001)(0000101)) \right. \\ &\quad \left. ((0110100)(0110100)(0000101)(0110001)(0110001)(1111001)) \right. \\ &\quad \left. ((0000000)(0000000)(1111100)(1001000)(1111001)(0000101)) \right) \end{aligned} \quad (2.24)$$

Los coeficientes se obtienen de la suma de las últimas m componentes, en este caso las 3 últimas, de cada A_k con β_{priv_k} . Pero tenemos el mismo problema de antes, puesto que los coeficientes de A_k pertenecen a \mathbb{F}_{128} y los coeficientes de β_{priv_k} se encuentra en \mathbb{F}_2 , por tanto lo que vamos a sumar es 0 o 1, o lo que es lo mismo realizar la operación lógica *XOR* con los elementos (0000000) o (0000001), ecuación (2.25).

$$coef = \left(\begin{pmatrix} (1001100) \\ (0110001) \\ (0000101) \end{pmatrix} \begin{pmatrix} (0110001) \\ (0110000) \\ (1111001) \end{pmatrix} \begin{pmatrix} (1001001) \\ (1111001) \\ (0000100) \end{pmatrix} \right) \quad (2.25)$$

Los términos corresponden a la parte de la derecha de la ecuación (2.10), a continuación se va a realizar el cálculo para k igual 1. La ecuación (2.26) corresponde al primer vector de β , $(1\ 0\ 0\ 1\ 0\ 0)$, donde se han transformado los elementos de \mathbb{F}_2 a \mathbb{F}_{128} y se han tomado las 3 primeras componentes.

$$\beta_1^v = ((0\ 0\ 0\ 0\ 0\ 0\ 1)(0\ 0\ 0\ 0\ 0\ 0\ 0)(0\ 0\ 0\ 0\ 0\ 0\ 0)) \quad (2.26)$$

De la misma forma se han tomado las 3 primeras componentes del primer vector de A , ecuación (2.27).

$$A_1^v = ((0\ 1\ 1\ 0\ 1\ 0\ 0)(0\ 0\ 0\ 0\ 0\ 0\ 0)(0\ 1\ 1\ 0\ 1\ 0\ 0)) \quad (2.27)$$

El vector *sum* contiene la suma de A_1^v y β_1^v , ecuación (2.28).

$$sum = ((0\ 1\ 1\ 0\ 1\ 0\ 1)(0\ 0\ 0\ 0\ 0\ 0\ 0)(0\ 1\ 1\ 0\ 1\ 0\ 0)) \quad (2.28)$$

Finalmente, para calcular *term* hay que multiplicar *sum* por a^v y restárselo a la primera componente de *hash*. Al realizar las operaciones módulo 2, la resta y la suma son lo mismo, por tanto se realizará la operación lógica XOR con el *hash*, ecuación (2.29).

$$\begin{aligned} term_1 &= ((0\ 1\ 1\ 0\ 1\ 0\ 1)(0\ 0\ 0\ 0\ 0\ 0\ 0)(0\ 1\ 1\ 0\ 1\ 0\ 0)) \cdot \begin{pmatrix} (0\ 1\ 1\ 0\ 1\ 0\ 0) \\ (1\ 1\ 1\ 1\ 1\ 0\ 0) \\ (1\ 1\ 1\ 1\ 0\ 0\ 1) \end{pmatrix} + (1\ 0\ 1\ 1\ 0\ 0\ 0) \\ &= (0\ 0\ 0\ 1\ 1\ 0\ 0) + (1\ 0\ 1\ 1\ 0\ 0\ 0) \\ &= (1\ 0\ 1\ 0\ 1\ 0\ 0) \end{aligned} \quad (2.29)$$

De la misma forma se calculan el resto de los términos llegando al vector *term*, ecuación (2.30).

$$term = ((1\ 0\ 1\ 0\ 1\ 0\ 0)(1\ 1\ 0\ 1\ 0\ 0\ 0)(1\ 1\ 0\ 0\ 0\ 0\ 0)) \quad (2.30)$$

Tras la resolución del sistema $coef \cdot oil = term$ se han obtenido los aceites, ecuación (2.31).

$$oil = ((0\ 0\ 0\ 0\ 1\ 1\ 0)(0\ 0\ 1\ 1\ 0\ 1\ 1)(0\ 1\ 0\ 0\ 1\ 1\ 0)) \quad (2.31)$$

Ahora se forma el vector *aux* uniendo los vinagres con los aceites, ecuación (2.32).

$$\begin{aligned} aux &= ((0\ 0\ 0\ 0\ 1\ 1\ 0)(0\ 0\ 1\ 1\ 0\ 1\ 1)(0\ 1\ 0\ 0\ 1\ 1\ 0) \\ &\quad (0\ 1\ 1\ 0\ 1\ 0\ 0)(1\ 1\ 1\ 1\ 1\ 0\ 0)(1\ 1\ 1\ 1\ 0\ 0\ 1)) \end{aligned} \quad (2.32)$$

La firma del mensaje se obtiene multiplicando el vector aux por la inversa de T , que de la forma que está construida dicha matriz T coincide con su inversa, ecuación (2.33).

$$firma = ((0110100)(1011010)(1011001) \\ (0000110)(0011011)(0100110)) \quad (2.33)$$

Para realizar la confirmación del mensaje es necesario calcular dos variables auxiliares, α_{aux} y β_{aux} .

Para calcular α_{aux1} hay que multiplicar los vectores $firma$, α_{pub1} y $firma$ transpuesta, ecuación (2.34).

$$\alpha_{aux1} = (firma) \cdot \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot (firma)^T \quad (2.34) \\ = (0111110)$$

La segunda variable β_{auxk} es el producto de la k -ésima componente del vector β_{pub} , donde sus componentes se encuentran en \mathbb{F}_2 y habrá que transformarlos en componentes de \mathbb{F}_{128} , por el vector $firma$ transpuesta, ecuación (2.35).

$$\beta_{aux1} = (100100) \cdot (firma)^T \\ = (0111110) \quad (2.35)$$

Calculando todas la componentes de α_{aux} y β_{aux} se obtienen los vectores las ecuaciones (2.36) y (2.37), respectivamente.

$$\alpha_{aux} = ((0111110)(1100000)(1100111)) \quad (2.36)$$

$$\beta_{aux} = ((0110010)(1100010)(1101000)) \quad (2.37)$$

Sumando los vectores componente a componente, ecuación (2.38), anteriormente calculados, se obtiene vector a comparar con el *hash* del mensaje.

$$((0001100)(0000010)(0001111)) \quad (2.38)$$

Si el mensaje no ha sido corrompido habrá de ser igual que el *hash* del mensaje, ecuación (2.39).

$$((0001100)(0000010)(0001111)) = ((0001100)(0000010)(0001111)) \quad (2.39)$$

Capítulo 3

Planificación y costes

En este capítulo se van a definir las diferentes etapas del proyecto mediante diagramas de Gantt realizados con la aplicación *OpenProj*. Además se incluye el presupuesto necesario para la realización de dicho proyecto.

3.1. Planificación

La figura 3.1 muestra la propuesta inicial de las etapas del proyecto, donde se diferencian varios bloques, el primero sería el de estudio e introducción a lo que se va a realizar en el proyecto, que comprendería desde septiembre hasta finales de noviembre, el segundo bloque o implementación del algoritmo, desde mediados de noviembre hasta finales de diciembre, el tercer bloque contiene todo lo referente a la *blockchain* ARK siendo el grueso del proyecto, comienza a finales de enero hasta mayo. Y por último la parte de las pruebas, son 10 días en mayo. La memoria se ha redactado durante todo el proyecto.

Pero no todo ha sido como se había planificado, puesto que han surgido algunos imprevistos. A la hora de realizar la implementación del algoritmo [UOV](#) en `python` no existe una biblioteca para trabajar con matrices y cuerpos finitos al mismo tiempo, así ha aumentado el tiempo que se iba a dedicar al algoritmo. Además el trabajo con ARK ha sido más tedioso del esperado, retrasando los tiempos programados.

El diagrama de Gantt real se ha dividido en aproximadamente cuatrimestres para que se visualice mejor, la figura 3.2 muestra las fases de estudio e implementación hasta febrero, la figura 3.3 incluye el tiempo dedicado a terminar la implementación y realizar el trabajo con ARK hasta julio y la figura 3.4 contiene el tiempo restante de trabajo con ARK desde julio hasta noviembre, además del periodo de pruebas.

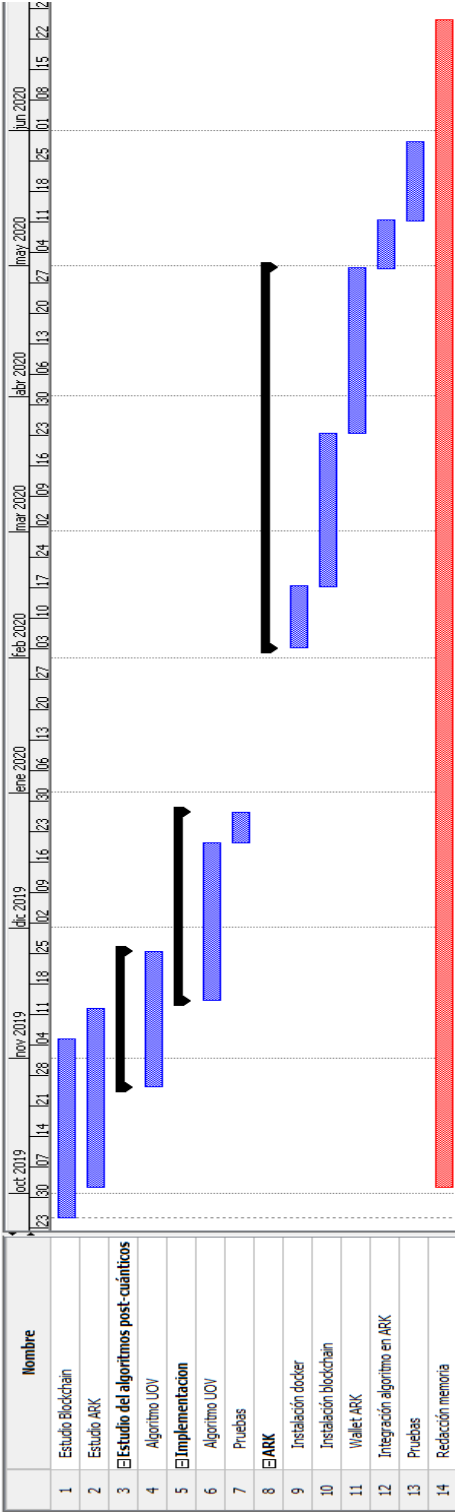


Figura 3.1: Diagrama de Gantt inicial

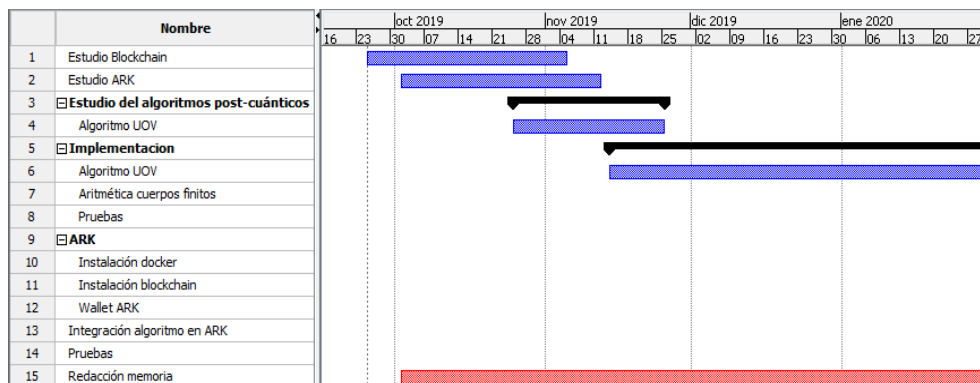


Figura 3.2: Diagrama de Gantt real. Parte I

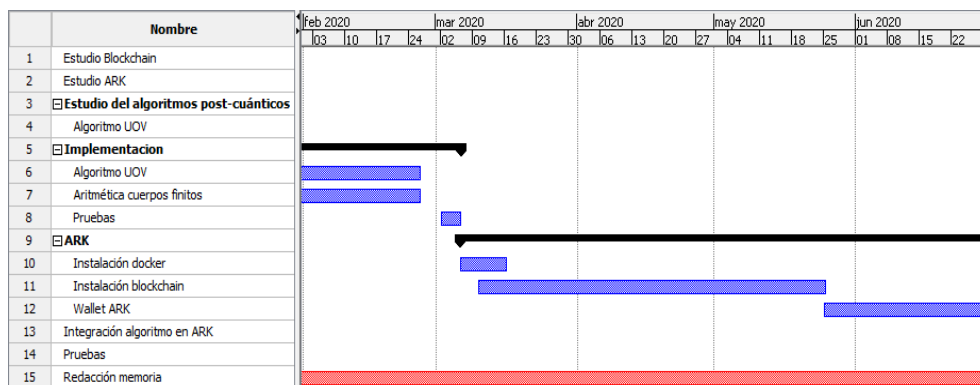


Figura 3.3: Diagrama de Gantt real. Parte II

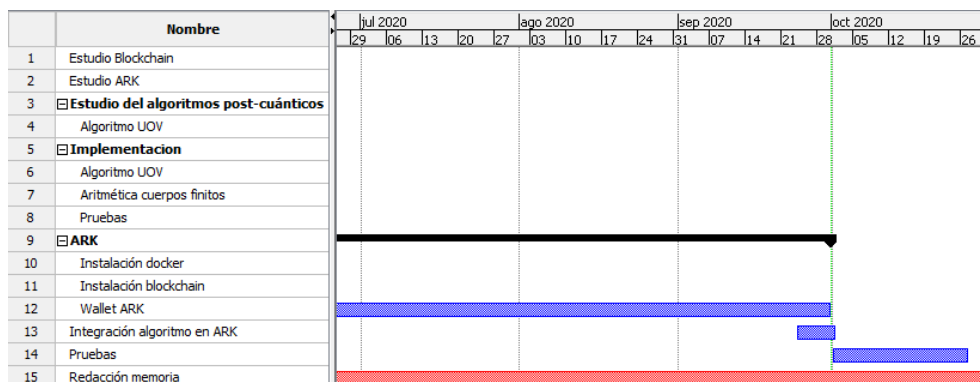


Figura 3.4: Diagrama de Gantt real. Parte III

3.2. Costes

A continuación se va a detallar el presupuesto invertido en el desarrollo del proyecto. Los costes del mismo se van a desglosar en recursos humanos, costes indirectos, costes directos y viajes.

Los costes en recursos humanos se van a dividir en dos tablas, una para los tutores, tabla 3.1, y otra para la alumna, tabla 3.2.

La tabla 3.1 muestra los costes de recursos humanos de los tutores. Para ello se han calculado las horas dedicadas a cada apartado y se ha estimado que el precio de cada hora de trabajo de los tutores es de 30€. Las horas se han dividido en tres grandes bloques, estudio, desarrollo y uno general, este último corresponde al tiempo dedicado a la corrección de la memoria y las tutorías con la alumna.

Descripción del coste	Cantidad (horas)	Coste total (€)
Bloque de estudio		
Selección del proyecto	8	240
Estudio ARK	10	300
Estudio algoritmo UOV	23	690
Bloque de desarrollo		
Algoritmo UOV	28	840
Instalación <i>blockchain</i>	7	210
Integración algoritmo	20	600
Bloque general		
Corrección memoria	12	360
Reuniones	53	1.590
TOTAL (€)		4.830

Tabla 3.1: Desglose de los costes en recursos humanos de los tutores

La tabla 3.2 muestra los costes de recursos humanos de la alumna de la misma forma se ha dividido en tres bloques, estudio, desarrollo y uno general, el último corresponde al tiempo dedicado a la memoria y a las reuniones con ambos tutores. Se ha estimado que el precio por cada hora de trabajo de la alumna es de 20€.

Descripción del coste	Cantidad (horas)	Coste total (€)
Bloque de estudio		
Estudio tecnología <i>blockchain</i>	37	740
Estudio ARK	50	1.000
Estudio algoritmos post-cuánticos	31	620
Bloque de desarrollo		
Algoritmo UOV	41	820
Aritmética cuerpos finitos	26	520
Instalación docker	20	400
Instalación <i>blockchain</i>	29	580
Wallet ARK	25	500
Integración algoritmo	92	1.840
Pruebas	23	460
Bloque general		
Redacción memoria	83	1.660
Corrección memoria	26	520
Reuniones	53	1.060
TOTAL (€)		10.720

Tabla 3.2: Desglose de los costes en recursos humanos de la alumna

Los costes indirectos corresponden a la electricidad y alquiler de la oficina como estos gastos no se pueden estimar correctamente se va a destinar un 10% del total del proyecto. En este caso corresponde a 1.578,24€.

En los costes directos se van a meter los gastos en material de oficina, material informático y servicio de mantenimiento, véase la tabla 3.3. Este último incluye el precio de un nuevo disco de memoria interno, ya que ha sido necesario aumentar la memoria del portátil durante el desarrollo del proyecto.

El material informático utilizado ha sido únicamente el equipo de trabajo, un portátil ASUS TP300L. Teniendo en cuenta que el precio de adquisición del ordenador fue de 750,00€ y suponiendo que la vida media de un ordenador es de unos 6 años, entonces el gasto que corresponde durante los 13 meses del desarrollo del proyecto es de 135,41€.

Descripción del coste	Cantidad
Material de oficina	30€
Material informático	135,41€
Servicio de mantenimiento	45€
TOTAL (€)	210,4 €

Tabla 3.3: Desglose de los costes directos

En los costes en viajes solo ha sido necesario incluir los viajes a la escuela de informática para las reuniones con el tutor, puesto que a la facultad de ciencias no era necesario coger un transporte. Solo se han realizado viajes en el primer cuatrimestre del curso 2019-2020, ya que el resto de las reuniones han sido de forma virtual, obteniendo un total de costes de 22€.

Tipo de costes	Cantidad
Recursos humanos tutores	4.830€
Recursos humanos alumna	10.720€
Indirectos	1.578,24€
Directos	210,40€
Viajes	22€
TOTAL (€)	17.360,64€

Tabla 3.4: Presupuesto gastos previstos desglosado

Una vez que hemos calculado todos los gastos posibles, tabla 3.5, vamos a añadir un margen de contingencia del 5% para posibles imprevistos del proyecto. Hemos obtenido que los gastos previstos serán de 16.537,64€, por tanto el margen de contingencia será de 868,03€, así el presupuesto final será de 18.228,67€.

Tipo de costes	Cantidad
Recursos humanos tutores, tabla 3.1	4.830€
Recursos humanos alumna, tabla 3.2	10.720€
Indirectos, párrafo 3.2	1.578,24€
Directos, tabla 3.3	210,40€
Viajes, párrafo 3.2	22€
Gastos imprevistos, párrafo 3.2	868,03€
TOTAL (€)	18.228,67€

Tabla 3.5: Presupuesto total desglosado

Capítulo 4

Análisis del problema

En este cuarto capítulo se aportan las opciones que se han elegido para el desarrollo del proyecto. Además se incluye una descripción de las funcionalidades que se pretenden alcanzar, así como de los problemas que han podido surgir para no llegar al diseño propuesto inicialmente. Así como una especificación de tanto los requisitos funcionales como los no funcionales.

Se ha optado por el uso de una máquina Docker, en lugar de hacerlo en local, para el desarrollo del proyecto debido al auge de DevOps en la ciberseguridad.

DevOps es la combinación de desarrollo (*development*) y operaciones (*operations*)[34]. DevOps incluye entre otras características sistemas de seguridad, maneras de trabajar en grupo y análisis de datos. Además describe los enfoques para agilizar los procesos con los que una idea, como corrección de errores, pasa del desarrollo a la implementación, generando valor para el usuario. Para llevar acabo estas ideas, son necesarios equipos de desarrollo y operaciones comunicados con frecuencia, así como escalabilidad y aprovisionamiento flexible.

Los principios de DevOps se encuentran reflejados en el proceso de compilación de Docker, garantizando una imagen Docker totalmente rastreable y documentada. Las imágenes de Docker son portátiles e independientes del sistema operativo subyacente, esto permite desarrollar código en computadoras locales sin preocuparse por la arquitectura de implementación. Además Docker proporciona un marco modular y extensible, obteniendo un procesamiento de datos actualizado con los últimos desarrollos algorítmicos[35].

Por último trabajar con Docker ha permitido tener un control de las versiones generadas. De esta forma si alguna de las nuevas imágenes guardadas tenía un error, no había problema en volver a versiones anteriores y empezar a trabajar desde una máquina sin errores.

En segundo lugar se ha elegido cuál sería la *blockchain* en donde integrar el algoritmo de firma y verificación, en este caso ARK. Gracias a las funciones intuitivas e innovadoras de ARK, han hecho posible que trabajar con dicha cadena de bloques sea fácil.

Su diseño modular y el código fuente abierto, aporta a la *blockchain* grandes capacidades de personalización para satisfacer las necesidades de cada usuario. Así desde el principio, fue sencillo encontrar que archivos había que modificar, para cumplir los objetivos del proyecto, y entender que hacía la parte del código que nos interesaba.

Otras propiedades interesantes que han hecho decantarse por esta cadena de bloques, han sido la escalabilidad y la compatibilidad con otras *blockchain* de cara al futuro. Puesto que esta *blockchain* modificada podría integrarse en otra cambiando así los algoritmos de firma y verificación de los bloques.

4.1. Especificación de requisitos

4.1.1. Requisitos funcionales

La especificación de los requisitos funcionales se ha dividido en varios apartados según cada parte del proyecto. Se distinguen, el programa en `python` con la implementación del algoritmo UOV, tabla 4.1, el `docker` que contiene la interacción con la base de datos y la *blockchain*, tabla 4.2, el *explorer* para visualizar las transacciones realizadas y los bloques generados, tabla 4.3, y por último, la aplicación *wallet* desde donde se realizarán las transacciones, tabla 4.4.

Requisito	Descripción
RF 1.1	El programa deberá tener la implementación de la aritmética de \mathbb{F}_{128}
RF 1.2	El programa deberá de generar la clave pública y privada de cada usuario
RF 1.3	El programa deberá de firmar correctamente cada <i>hash</i> de un bloque
RF 1.4	El programa deberá de realizar correctamente la verificación de un <i>hash</i> con una firma

Tabla 4.1: Requisitos funcionales del programa

Requisito	Descripción
RF 2.1	La base de datos local del docker deberá almacenar la claves del usuario
RF 2.2	La base de datos local del docker deberá almacenar la información del usuario
RF 2.3	La base de datos local del docker deberá almacenar los monederos así como el saldo de cada usuario
RF 2.4	El docker deberá de almacenar la <i>blockchain</i>
RF 2.5	El docker deberá mantener activa la ejecución de la <i>blockchain</i>
RF 2.6	El docker deberá mantener activa la ejecución del <i>explorer</i>
RF 2.7	El docker deberá mantener abiertos los puertos del <i>explorer</i> y API
RF 2.8	El docker tendrá integrado la <i>blockchain</i> con el nuevo código de firma
RF 2.9	El docker tendrá integrado la <i>blockchain</i> con el nuevo código de verificación

Tabla 4.2: Requisitos funcionales del docker

Requisito	Descripción
RF 4.1	En el <i>explorer</i> se podrá realizar búsqueda por bloques
RF 4.2	En el <i>explorer</i> se podrá realizar búsqueda por transacciones
RF 4.3	El <i>explorer</i> deberá mostrar la firma de cada bloque
RF 4.4	El <i>explorer</i> deberá mostrar el ID de cada bloque
RF 4.5	El <i>explorer</i> deberá mostrar la firma de cada transacción
RF 4.6	El <i>explorer</i> deberá mostrar el usuario que ha realizado la transacción además del receptor
RF 4.7	Los datos que muestre el <i>explorer</i> deberán de ser a tiempo real

Tabla 4.3: Requisitos funcionales del *Explorer*

Requisito	Descripción
RF 3.1	La aplicación deberá dar la opción al usuario de crear un perfil
RF 3.2	La aplicación deberá dar la opción de crear una nueva red <i>testnet</i>
RF 3.2	La aplicación deberá dar la opción al usuario de conectarse a la red <i>testnet</i>
RF 3.3	La aplicación deberá dar la opción al usuario de importar el monedero génesis
RF 3.4	La aplicación deberá dar la opción al usuario de crear monederos
RF 3.5	La aplicación deberá dar la opción al usuario realizar transacciones entre diferentes monederos
RF 3.6	La aplicación deberá dar la opción al usuario de firmar mensajes
RF 3.7	La aplicación deberá dar la opción al usuario de validar mensajes

Tabla 4.4: Requisitos funcionales de la aplicación Wallet

4.1.2. Requisitos no funcionales

En los requisitos no funcionales podemos encontrar dos bloques, los referentes al sistema como los tiempos de ejecución o a la seguridad, tabla [4.5](#), y los personales, que logros esperaba conseguir tras la finalización del proyecto, tabla [4.6](#)

Requisito	Descripción
RNF 1.1	El programa no deberá de tardar más de medio minuto en generar las claves públicas y privadas
RNF 1.2	El programa deberá de tardar unos segundos en firmar un mensaje
RNF 1.3	El programa deberá de tardar unos segundos en verificar la firma de un <i>hash</i>
RNF 1.4	En cualquier momento se podrá realizar la verificación de una firma
RNF 1.5	El programa deberá de ser correctamente integrado en el sistema <i>blockchain</i>
RNF 1.6	El programa deberá de ser compatible con cualquier sistema compatible con <code>python</code>
RNF 1.7	La aplicación deberá de realizar transacciones de forma segura
RNF 1.8	El proyecto deberá de contar con un manual de usuario claro y conciso

Tabla 4.5: Requisitos no funcionales del sistema

Requisito	Descripción
RNF 2.1	Aprender a gestionar los tiempos de un proyecto
RNF 2.2	Aprender a solucionar de la manera más eficiente los problemas que surjan
RNF 2.3	Aprender otros lenguajes de programación como <code>python</code>
RNF 2.4	Entender como funcionan la tecnología <i>blockchain</i>
RNF 2.5	Entender el algoritmo UOV
RNF 2.6	Aprender a realizar una memoria definiendo un problemas y los objetivos
RNF 2.7	

Tabla 4.6: Requisitos no funcionales personales

4.2. Análisis

Con la realización de este proyecto se persiguen principalmente dos funcionalidades, la implementación del algoritmo UOV y la integración del mismo en la *blockchain*.

Para la implementación del algoritmo ha sido necesario implementar la aritmética de \mathbb{F}_{128} puesto que no se ha encontrado ninguna librería de `python` que trabaje conjuntamente las matrices con cuerpos finitos. Se ha optado por el lenguaje de programación `python` para el algoritmo porque inicialmente se pensaba trabajar con la *blockchain* implementada en `python` y no tener problemas a la hora de llamar a las funciones del algoritmo. Sin embargo tras un proceso de investigación se vio mejor opción utilizar la *blockchain* en `typescript` para hacer uso de la parte gráfica como es la aplicación *wallet* y el *explorer*. De esta forma se ha visto afectado el diseño inicial puesto que han sido necesarios dos ficheros más de transición entre los ficheros de `typescript` y `python`, se entrará en detalle en el capítulo 6.

A la hora de la integración, el primer problema que se ha obtenido es la generación y almacenamiento de las claves públicas y privadas. Para no tener problemas de compatibilidad con el formato entre las claves que se necesitan para el algoritmo UOV y las que genera la *blockchain*, se ha optado por añadir un fichero con extensión `json` que almacena conjuntos de claves tanto las propias de la *blockchain* como las de UOV. De esta forma el algoritmo de firma crea una nueva entrada en el fichero `data.json` añadiendo las claves pasadas como parámetro y las claves del algoritmo UOV (los α y β públicos y privados).

Las claves generadas por el algoritmo UOV se hacen a partir de la clave privada de la creada por la *blockchain*, esto es, los valores aleatorios de la clave privada de UOV (α y β) se calculan tomando como semilla la clave privada de la *blockchain*. Dando lugar a la unicidad entre claves, ya que el mismo par de claves de la cadena de bloques generará el mismo par de claves con el algoritmo UOV.

Un segundo problema se ha obtenido, en la integración, ha sido que a la hora de realizar la verificación de la firma, se ha notado que la firma del *hash* pasada como parámetro se corta, sin ser esta truncada en la función firma. Al no encontrar el lugar donde se trunca ha sido necesario cambiar el diseño e incluir un nuevo fichero `signature.json`, donde se almacena la firma en hexadecimal y en vector. Dicha firma en hexadecimal es la que devuelve la función firma, aunque la función de verificación no reciba el valor completo en hexadecimal no hay problema puesto que se busca en el fichero la firma que comience por el valor recibido.

Capítulo 5

Diseño

Este capítulo de diseño abarca una explicación de los bloques que se encuentran en la parte práctica del proyecto. Además del diseño, reflejado en un diagrama de bloques, que se ha seguido a la hora de implementar el prototipo de la *blockchain*.

ARK *deployer* [36] da la posibilidad a cualquier usuario de crear una cadena de bloques independiente y funcional, que se puede personalizar de forma intuitiva. Dentro a del *deployer* existen tres posibles configuraciones de la red (*testnet*, *devnet* y *mainnet*). La red *testnet* permite realizar cambios en una red local y privada, se puede personalizar entre otras cosas la moneda, la interfaz o en nuestro caso el algoritmo de firma. La red *devnet* es una red de desarrollo no privada sino que cualquier persona puede acceder a ella para realizar cambios, la moneda que se utiliza se llama DARK. Y finalmente, la red *mainnet* puedes personalizar la interfaz.

El *core* gestiona todo lo referente a la creación de bloques y al almacenamiento de las transacciones en el mismo. Por tanto será la parte en la que se introduzca el algoritmo de firma y verificación la firma de los bloques. A causa de modularidad de las funciones del *core*, se puede modificar una parte código y ajustarla a las necesidades del usuario, sin tocar otras funciones. El *core* como su propio nombre indica es el centro del prototipo, puesto que será el que interactúe con la base de datos, con la aplicación Wallet y con el *explorer* de ARK. Tratará con la base datos para almacenar la información actualizada de los bloques. Con el Wallet para admitir las nuevas transacciones y guardarlas en un bloque. Y en último lugar, con el *explorer* para mostrar cualquier tipo de información.

La base datos se encarga de almacenar y servir datos de las transacciones dentro de un nodo[37]. Algunos datos que recoge son el usuario que realiza la transacción, el bloque en el que se almacena o la cantidad enviada en la transacción.

La aplicación Wallet de ARK [38] es un monedero de criptomonedas construida para proporcionar la realización de transacciones y firma de mensajes de forma segura, puesto que nadie excepto el usuario podrá acceder a las claves privadas del monedero. Una ventaja de esta aplicación es que se conecta con la *blockchain* de forma instantánea, manteniendo siempre la *blockchain* actualizada. Se puede descargar para ordenadores, versión Desktop Wallet, está disponible para los principales sistemas operativos como MacOS, Windows y Linux. También se encuentra una versión para móviles, Mobile Wallet, compatible con *smartphones* de iOS o android.

El *explorer* de ARK [39] es una interfaz muy intuitiva que recoge toda la información referente a las transacción. Los usuarios pueden acceder al estado de una transacción, realizada con el Wallet de ARK, para comprobar si ha llegado a su destino o si la han recibido. Pero no solo los usuarios que realizan dichas transacciones pueden visualizar el estado, sino que cualquier persona tiene acceso a este buscador de bloques, obteniendo cualquier información de la *blockchain*. Las principales acciones que se pueden realizar en el *explorer* de ARK son la búsqueda de un bloque creado, ver cada transacción realizada a una dirección, listar las direcciones de los monederos de ARK o incluso la instantánea del estado de la red.

El prototipo de instalación de la *blockchain* modificada que se ha realizado en el docker, recoge todos bloques anteriormente citados, estos son el *deployer*, el *core*, la base de datos, la aplicación Wallet y el *explorer* de ARK.

La figura 5.1 muestra un diagrama con los bloques con el diseño general que se ha implementado. En el docker se incluye el *deployer* que instala el *core* de la cadena de bloques. Además, se encuentra la base de datos *postgresql*, que estará en interacción con el *core*.

Por otra parte, fuera del docker, se encuentra el *explorer* y la aplicación Desktop Wallet. El *explorer* estará conectado al *core* mediante el puerto API 4200, al *explorer* se podrá acceder mediante un navegador con la dirección `http://NODE_GENESIS:4200`. Sin embargo, para acceder a la aplicación Desktop Wallet será necesario descargarla, véase apéndice manual de usuario A.3, dicha aplicación se conectará con el *core* mediante el puerto API 4103.

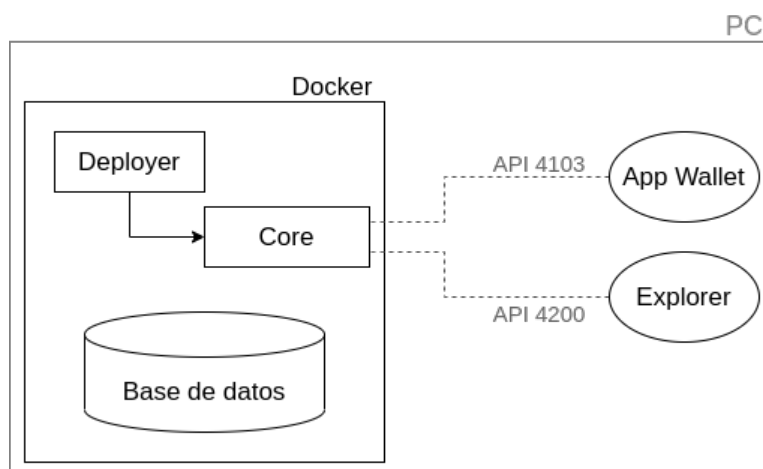


Figura 5.1: Diagrama de bloques para el prototipo

No hay una única configuración de los bloques. A modo de ejemplo se incluyen dos configuraciones diferentes. La primera, en la figura 5.2, muestra que no es necesario que la aplicación Wallet y el *explorer* de ARK se encuentren en una misma máquina que el docker, es decir, que pueden alojarse en diferentes máquinas distribuidas y mediante una conexión por la red podrían comunicarse.

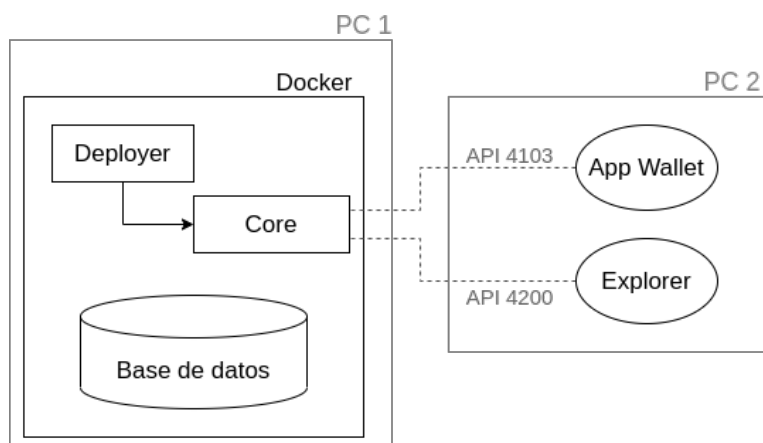


Figura 5.2: Diagrama de bloques. Ejemplo de configuración 1

La segunda configuración, en la figura 5.3, muestra que tampoco tienen por qué, la aplicación Wallet y el *explorer*, establecerse en la misma máquina. Dicha aplicación wallet se encuentra instalada en un *smartphone*, puesto que dicha aplicación es compatible con dispositivos cuyo sistema operativo sea android o [iOS](#).

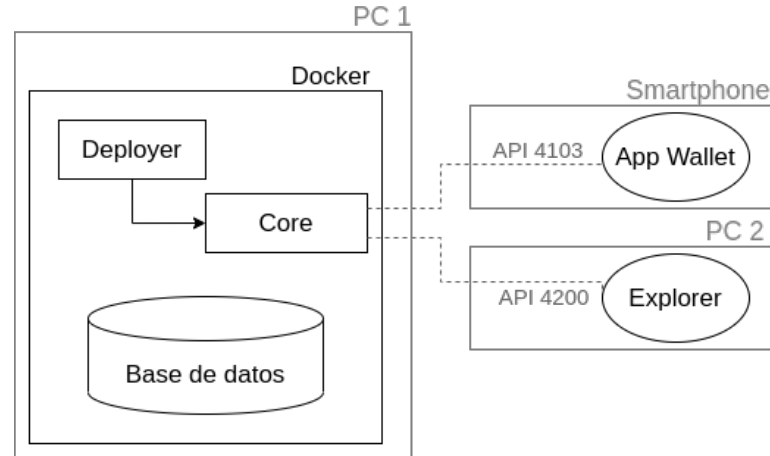


Figura 5.3: Diagrama de bloques. Ejemplo de configuración 2

Finalmente, la estructura del algoritmo viene dada por un fichero escrito en python, denominado `uov.py`, donde se encuentran las funciones que se explicarán en detalle en el capítulo 6. Para continuar con la idea de modularidad de la arquitectura de ARK, la integración del algoritmo en la *blockchain* se ha hecho en un fichero independiente. Esto es desde el archivo original donde se encuentran las funciones de firma y verificación de los bloques se creará un proceso que llame a las distintas funciones del algoritmo UOV.

El fichero `uov.py` incorpora main con un ejemplo de uso del algoritmo independientemente de la *blockchain*. Dicho fichero se encuentra alojado en el *core* de la cadena de bloques para utilizarlo en el algoritmo de firma y verificación.

Capítulo 6

Implementación

En este capítulo se va a detallar cómo se ha llevado a la práctica el diseño del capítulo anterior. Se distinguen dos grandes bloques el ecosistema ARK, sección 6.1, y el algoritmo criptográfico, sección 6.2.

En el primer bloque se muestra la organización de los archivos de los diferentes directorios alojados en el docker. También se aporta el código de los nuevos archivos generados para llevar a cabo la integración del algoritmo UOV en la *blockchain*.

Mientras en el segundo bloque, hay una clara explicación de las funciones necesarias para la implementación del algoritmo UOV en el lenguaje de programación `python`. Además se incluye el código y los parámetros necesarios para la ejecución de las mismas.

Durante el desarrollo del proyecto se ha utilizado Git, para tener un control de las versiones generadas y reflejar los cambios realizados, por si en algún momento fuese necesario deshacerlos. Se han utilizado dos repositorios uno para las modificaciones en el *core* de la *blockchain*[40] y otro para la memoria y el código en `python`[41].

6.1. Ecosistema ARK

El ecosistema ark se ha instalado en un docker `ubuntu:xenial`. En el *home* del mismo, nos encontramos con las tres carpetas necesarias para la ejecución e instalación tanto de la *blockchain* y como del *explorer*, estas son `deployer`, `core-bridgechain` y `core-explorer`. A continuación se van a explicar para que sirven las dos primeras, puesto que del `core-explorer` no ha habido ninguna modificación y se ha usado con todos los valores por defecto.

6.1.1. Directorio deployer

La carpeta `deployer` contiene varios directorios y ficheros siendo los más importantes, `app`, `bridgechain.sh`, `setup.sh`, ver árbol de directorio 6.1.

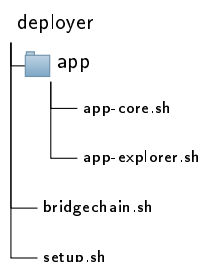


Figura 6.1: Árbol de directorios de deployer

- El directorio `app` contiene a su vez dos ficheros, `app-core.sh` que clona e instala el repositorio de GitHub `mvictoria1997/core` puesto que se ha cambiado la línea 108 para tal fin, código 6.1, en su defecto instalaría la *blockchain* de ArkEcosystem. Este cambio se hace para obtener en la *blockchain* las modificaciones en el algoritmo de firma.

```
1 git clone https://github.com/mvictoria1997/core.git --single-
2 branch "$BRIDGECHAIN_PATH"
```

Código 6.1: Línea 108 `app-core.sh`

El segundo archivo, `app-explorer.sh`, clona e instala el repositorio de GitHub `ArkEcosystem/explorer` no se hace ningún cambio puesto que se va a utilizar todo por defecto.

- El archivo `bridgechain.sh` ejecuta los archivos `app-core.sh` y `app-explorer.sh` para instalar la *blockchain* y el *explorer* respectivamente.
- El archivo `setup.sh` realiza una instalación inicial del repositorio.

6.1.2. Directorio core-bridgechain

El directorio `core-bridgechain` contiene cinco directorios importantes, `blocks`, `crypto`, `identities`, `interfaces` y `transactions`, ver árbol de directorios 6.2.

core-bridgechain/packages/crypto/src

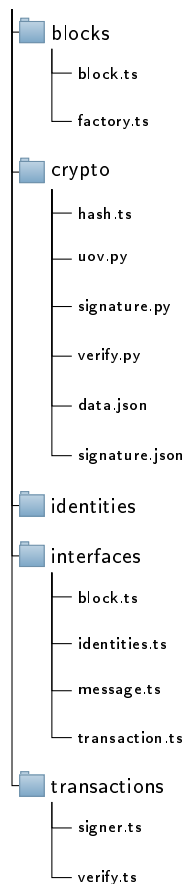


Figura 6.2: Árbol de directorios de core-bridgechain

- Directorio `blocks` gestiona los bloques de la *blockchain*, en este directorio se encuentran los ficheros, `block.ts` que engloba las funciones de verificación de la firma de los bloques y `factory.ts` que administra la creación y firma de los bloques.
- En el directorio `crypto` es donde se han realizado los cambios, puesto que se encuentran los ficheros con los algoritmos de firma y verificación (objetivo de nuestro proyecto). Podemos distinguir tres tipos de ficheros, los escritos en `typescript`, los escritos en `python` y por último los de almacenamiento con extensión `json`.

De los escritos en `typescript` destaca el fichero `hash.ts`, en él se encuentran las funciones de firma y verificación de tres algoritmos, [ECD-SA](#), código 6.2, Schnorr[42] y [UOV](#) (la añadida para este trabajo), código 6.3 y código 6.4. Para no tener que modificar las funciones desde donde

se realizan las llamadas a los distintos algoritmos, lo que se ha hecho es llamar desde la función de firma y verificación del algoritmo ECDSA a las funciones de firma y verificación del algoritmo UOV, respectivamente.

```

1  public static signECDSA(hash: Buffer, keys: IKeyPair): string {
2      return Hash.signUOV(hash, keys)
3  }
4
5  public static verifyECDSA(hash: Buffer, signature: Buffer |
6  string, publicKey: Buffer | string): boolean {
7      return Hash.verifyUOV(hash, signature, publicKey);
8  }

```

Código 6.2: Modificación archivo hash.ts funciones ECDSA

```

1  public static signUOV(hash: Buffer, keys: IKeyPair): string {
2
3      //Transforma el Buffer en un array legible en python
4      var hash_string = new Array();
5      for (let i=0; i < hash.length; i++){
6          hash_string.push(hash[i].toString());
7      }
8
9      //Llamada al proceso que ejecuta el fichero signature.py
10     const { execSync } = require('child_process');
11     var cmd = 'python ' + __dirname + '/../../src/crypto/signature.
12     py ' + hash_string + ' ' + keys.publicKey + ' ' + keys.privateKey
13     ;
14     var signature = execSync(cmd);
15
16     //Almacena la firma en forma de vector y en hexadecimal para la
17     verificación
18     const data = readFileSync(__dirname + '/../../src/crypto/
19     signature.json');
20     var data_json = JSON.parse(data.toString());
21     let new_sign = {
22         hex: signature.toString("hex").slice(0, -2),
23         vector: signature.toString().trim()
24     };
25     data_json.push(new_sign);
26     var fs = require('fs');
27     fs.writeFileSync(__dirname + '/../../src/crypto/signature.json',
28     JSON.stringify(data_json));
29
30     return signature.toString("hex");
31 }

```

Código 6.3: Modificación archivo hash.ts función signUOV


```

1  public static verifyUOV(hash: Buffer, signature: Buffer | string,
   publicKey: Buffer | string): boolean {
2
3      var hash_string = new Array();
4      for (let i=0; i < hash.length; i++){
5          hash_string.push(hash[i].toString());
6      }
7
8      //Lectura de la firma en signature.json y encuentra la que
   empieza igual
9      var hex_signature = signature.toString("hex");
10     let data = readFileSync(__dirname + '/../../src/crypto/
   signature.json');
11     let data_json = JSON.parse(data.toString());
12     let encontrado : boolean = false;
13     var i=0, signature_string;
14     var regular_expression = new RegExp(hex_signature + '[^]*', 'i'
   );
15     while (i < data_json.length && !encontrado){
16         if (regular_expression.test(data_json[i]['hex'])){
17             signature_string = data_json[i]['vector'];
18             encontrado = true;
19         }
20         ++i;
21     }
22
23     //Quita los espacios sino toma hasta cada espacio como un pará
   metro
24     while (signature_string.search('\ ') != -1){
25         signature_string = signature_string.replace('\ ', '');
26     }
27     signature_string.replace('\n', '');
28
29     //Llamada al proceso que ejecuta el fichero verify.py
30     const { execSync } = require('child_process');
31     var cmd = 'python ' + __dirname + '/../../src/crypto/verify.py
   ' + hash_string + ' ' + signature_string + ' ' + publicKey.
   toString();
32     var verify = execSync(cmd);
33
34     if (verify.toString().trim() == 'True')
35         return true;
36     else
37         return false;
38 }
39

```

Código 6.4: Modificación archivo hash.ts función verifyUOV

En los archivos de python se encuentran uov.py, signature.py, código 6.5, y verify.py, código 6.6. El archivo uov.py contiene las funciones del algoritmo UOV, estas se explicarán en el capítulo 6. Los otros dos archivos son de transición, esto es, se llaman desde la función de firma a signature.py y desde la función de verificación a verify.py, para pasar los parámetros del lenguaje typescript al lenguaje python, a continuación se realizan llamadas a las funciones de firma o verificación, volviendo a pasar los resultados a las funciones de typescript.

```

1  import sys
2  from luov import *
3  import json
4  def main():
5      m, v = 3, 3
6      hash_schnorr, pub_schnorr, priv_schnorr = sys.argv[1], sys.argv
7      [2], sys.argv[3]
8      alpha, beta, hash, hashed_message = [], [], [], []
9      number = ''
10
11     for i in range(len(hash_schnorr)):
12         if hash_schnorr[i] != ',':
13             number += hash_schnorr[i]
14         else:
15             aux = int (number, 16) % 128
16             hash += [aux]
17             number = ''
18     aux = int (number, 16) % 128
19     hash += [aux]
20
21     T = generacionT (m, v)
22
23     encontrado = False
24     with open("/home/deployer/core-bridgechain/packages/crypto/src/
25     crypto/data.json", "r") as jsonread:
26         data = json.load(jsonread)
27         for i in range(len(data)):
28             if data[i]['pub_schnorr'] == pub_schnorr:
29                 alpha = data[i]['priv_alpha_UOV']
30                 beta = data[i]['priv_beta_UOV']
31                 encontrado = True
32                 break
33
34     if not encontrado:
35         alpha, beta = clavePrivada(m, v, priv_schnorr)
36         alpha_pub, beta_pub = clavePublica(m, v, alpha, beta, T)
37         with open("/home/deployer/core-bridgechain/packages/crypto/
38         src/crypto/data.json", "w") as jsonwrite:
39             nuevo = {"id": len(data),
40                     "pub_schnorr": pub_schnorr,
41                     "priv_schnorr": priv_schnorr,
42                     "priv_alpha_UOV": alpha,
43                     "priv_beta_UOV": beta,
44                     "pub_alpha_UOV": alpha_pub,
45                     "pub_beta_UOV": beta_pub}
46             data.append(nuevo)
47             jsonwrite.seek(0)
48             json.dump(data, jsonwrite)
49
50     hash = hash[0:m]
51     for i in range(len(hash)):
52         hashed_message += [bin(hash[i])[2:]] #Pasa a binario el hash
53
54     for i in range (len(hashed_message)):
55         aux = [int(d) for d in (hashed_message[i])]
56         for k in range(7-len(aux)):
57             aux.insert(0, 0)
58         hashed_message [i] = aux
59
60     firma = signature (hashed_message, alpha, beta, m, v, T)
61     print (firma)

```

```

59     sys.stdout.flush()
60
61     #print (alpha)
62     #verif = verificacion (hashed_message, firma, alpha_pub,
        beta_pub, m)
63
64     if __name__=="__main__":
65         main()
66

```

Código 6.5: Archivo signature.py

```

1  import sys
2  from luov import *
3  import os
4  import json
5  from ast import literal_eval
6  def main():
7      m, v = 3, 3
8      hash_schnorr, signature, pub_schnorr = sys.argv[1],
        literal_eval(sys.argv[2]), sys.argv[3]
9      alpha, beta, hash, hashed_message = [], [], [], []
10     number = ''
11
12     for i in range(len(hash_schnorr)):
13         if hash_schnorr[i] != ',':
14             number +=hash_schnorr[i]
15         else:
16             aux = int (number, 16)%128
17             hash += [aux]
18             number = ''
19     aux = int (number, 16)%128
20     hash += [aux]
21
22     with open("/home/deployer/core-bridgechain/packages/crypto/src/
        crypto/data.json", "r") as jsonread:
23         data = json.load(jsonread)
24         for i in range(len(data)):
25             if data[i]['pub_schnorr'] == pub_schnorr:
26                 alpha = data[i]['pub_alpha_UOV']
27                 beta = data[i]['pub_beta_UOV']
28                 break
29
30     hash = hash[0:m]
31     for j in range(len(hash)):
32         hashed_message += [bin(hash[j])[2:]] #Pasa a binario el hash
33
34     for i in range (len(hashed_message)):
35         aux = [int(d) for d in (hashed_message[i])]
36         for k in range(7-len(aux)):
37             aux.insert(0, 0)
38         hashed_message [i] = aux
39
40     print (verificacion (hashed_message, signature, alpha, beta, m)
        )
41     sys.stdout.flush()
42
43     if __name__=="__main__":
44         main()
45

```

Código 6.6: Archivo verify.py

Por último han de destacarse los ficheros de almacenamiento `data.json` y `signature.json`. En `data.json` podemos encontrar cada una de las claves públicas y privadas generadas para el algoritmo de Schnorr y las correspondientes para el algoritmo UOV (los α y β privados y públicos), cada conjunto de claves se ha almacenado en un diccionario con la estructura que muestra el código A.22. Debido a los problemas explicados en el apartado de análisis 4.2, el archivo `signature.json` contiene las firmas en hexadecimal y sus correspondientes en forma de vector, tal y como muestra el código 6.8.

```
1  {
2    "id":
3    "pub_schnorr":
4    "priv_schnorr":
5    "priv_alpha_UOV":
6    "priv_beta_UOV":
7    "pub_alpha_UOV":
8    "pub_beta_UOV":
9  }
```

Código 6.7: Estructura archivo `data.json`

```
1  {
2    "hex":
3    "vector":
4  }
```

Código 6.8: Estructura archivo `signature.json`

- Directorio `identities` abarca la creación y almacenamiento de las claves pública y privada.
- Directorio `interfaces` incluye las interfaces, esto es, los campos que se requieren para la creación de cada objeto o los que pueden añadirse posteriormente. Los objetos más destacados y con los que se han trabajado son bloques, identidades, mensajes y transacciones.

Los bloques se muestran en el archivo `blocks.ts`, las identidades (con las claves pública y privada) en el archivo `identities.ts`, los mensajes en el archivo `message.ts`, y las transacciones en el archivo `transactions.ts`.

- Entre los ficheros del directorio `transactions` podemos encontrar `signer.ts` y `verifier.ts`. El primero contiene la gestión de la firma de transacciones y el segundo la verificación de la firma.

6.2. Algoritmo criptográfico UOV

Aquí se proporcionan los detalles de las funciones implementadas para la ejecución algoritmo UOV, además de la implementación de la aritmética en el cuerpo de 128 elementos.

6.2.1. Funciones del cuerpo de 128 elementos

Las funciones referentes a esta sección son la suma, el producto y conversiones de elementos del cuerpo de 2 elementos a elementos del cuerpo de 128 elementos.

Los elementos del cuerpo se representarán con vectores de siete componentes para poder facilitar la implementación de la suma y del producto. Esto se refleja en las dos tablas denominadas `exp` y `log`, que tienen la estructura que se muestra en los códigos 6.9 y 6.10, respectivamente. Para la implementación se han usado las variables diccionario de `python`, puesto que tienen fácil acceso a todas las componentes.

```

1 exp = {
2     0 : [0, 0, 0, 0, 0, 0, 1],
3     1 : [0, 0, 0, 0, 0, 1, 0],
4     125 : [1, 1, 0, 0, 0, 0, 1],
5     126 : [1, 0, 0, 0, 0, 0, 1]
6 }
```

Código 6.9: Tabla para calcular la potencia en \mathbb{F}_{128}

```

1 log = {
2     tuple( [0, 0, 0, 0, 0, 0, 1] ) : ' 0 ',
3     tuple( [0, 0, 0, 0, 0, 1, 0] ) : ' 1 ',
4     tuple( [1, 1, 0, 0, 0, 0, 1] ) : ' 125 ',
5     tuple( [1, 0, 0, 0, 0, 0, 1] ) : ' 126 ',
6 }
```

Código 6.10: Tabla para calcular el logaritmo en \mathbb{F}_{128}

La suma de dos elementos del cuerpo se ha implementando la puerta *xor*, esto es si las *iésimas*-componentes son iguales entonces la suma vale 0, en caso contrario vale 1, código 6.11.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Elemento para sumar de \mathbb{F}_{128}
int vector	n2	Input	Elemento para sumar de \mathbb{F}_{128}
int vector	suma	Output	Elemento del cuerpo que almacena la suma de n1 y n2

Tabla 6.1: Parámetros de la función `suma`

```

1 def suma(n1=[], n2=[]):
2
3     suma = []
4     for i in range(len(n1)):
5         if n1[i] == n2[i]:
6             suma += [0]
7         else:
8             suma += [1]
9     return suma

```

Código 6.11: Suma de dos elementos del cuerpo

Para el producto de dos elementos del cuerpo se ha diferenciado el caso en el que uno de los vectores sea 0 en dicho caso el producto vale 0. Si ninguno de los vectores es 0 entonces se hace uso de las tablas 6.9 y 6.10, para trabajar con enteros módulo 127, código 6.12.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Elemento para multiplicar de \mathbb{F}_{128}
int vector	n2	Input	Elemento para multiplicar de \mathbb{F}_{128}
int	suma	Inner	Almacena la suma de los logaritmos de n1 y n2 módulo 127
int vector	product	Output	Elemento del cuerpo que almacena el producto de n1 y n2

Tabla 6.2: Parámetros de la función `product`

```

1 def product(n1=[], n2=[]):
2
3     product = []
4     if (n1 == [0,0,0,0,0,0,0,0]) or (n2 == [0,0,0,0,0,0,0,0]):
5         product = [0,0,0,0,0,0,0,0]
6     else:
7         suma = (int(log.get(tuple(n1))) + int(log.get(tuple(n2)))) % 127
8         product = exp.get(suma)
9     return product

```

Código 6.12: Producto de dos elementos del cuerpo

Las siguientes funciones son necesarias para la resolución del sistema de ecuaciones.

Calcula el vector correspondiente vector en el cuerpo de un entero, código 6.13.

Tipo	Nombre	Variable	Descripción
int	n1	Input	Entero del que se desea calcular su vector en el cuerpo
int vector		Output	Elemento de \mathbb{F}_{128}

Tabla 6.3: Parámetros de la función `F128`

```

1 def F128(n1):
2
3     if n1 == 127:
4         return [0,0,0,0,0,0,0,0]
5     else:
6         return exp.get(n1)

```

Código 6.13: Convierte un entero en un elemento del cuerpo

Al estar en un cuerpo todo elementos tiene inverso y tiene sentido hacer la función `inverso` de un elemento del cuerpo, la implementación que se ha hecho ha sido hacer la multiplicación por los restantes elementos y comprobando que de como resultado el elemento unidad, en este caso $[0,0,0,0,0,0,1]$, código 6.14.

Tipo	Nombre	Variable	Descripción
int vector	n	Input	Elemento para calcular su inverso en \mathbb{F}_{128}
int vector		Output	Inverso de n

Tabla 6.4: Parámetros de la función `inverse`

```

1 def inverse(n=[]):
2
3     i = 0
4     while product(n, F128(i)) != [0,0,0,0,0,0,1]:
5         i += 1
6     return F128(i)

```

Código 6.14: Inverso de un elemento del cuerpo

La función `mayor` compara si el elemento $n1$ es mayor que el elemento $n2$, para ello se convierten los vectores a enteros con la tabla `log` y se compara que entero es mayor, código 6.15.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Elemento a comparar con n2 en \mathbb{F}_{128}
int vector	n2	Input	Elemento a comparar con n1 en \mathbb{F}_{128}
int	log1	Inner	Almacena el logaritmo de n1
int	log2	Inner	Almacena el logaritmo de n2
boolean		Output	True si n1 es mayor False en otro caso

Tabla 6.5: Parámetros de la función `mayor`

```

1 def mayor (n1=[],n2=[]):
2
3     if n1 == [0,0,0,0,0,0,0]:
4         return False
5     else:
6         log1, log2 = log.get(tuple(n1)), log.get(tuple(n2))
7         return int(log1) > int(log2)

```

Código 6.15: Compara dos elementos del cuerpo

Convierte una matriz $n1$ del cuerpo \mathbb{F}_2 en un vector n de \mathbb{F}_{128} . Como los elementos de la matriz son solo 0 y 1 la conversión se reduce a, si en la matriz hay un 0 añade al vector n el vector $[0,0,0,0,0,0,0]$ y en otro caso $[0,0,0,0,0,0,1]$, código 6.16.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Matriz con elementos en \mathbb{F}_2
int	row	Inner	Almacena las filas generadas de la matriz n
int vector	n	Output	Matriz en \mathbb{F}_{128}

Tabla 6.6: Parámetros de la función `matrix_F2to128`

```

1 def matrix_F2to128(n1=[]):
2
3     n=[]
4     for i in range(len(n1)):
5         row=[]
6         for j in range(len(n1[0])):
7             if n1[i][j] == 0:
8                 row += [[0,0,0,0,0,0,0]]
9             else:
10                row += [[0,0,0,0,0,0,1]]
11        n += [row]
12    return n

```

Código 6.16: Matriz de \mathbb{F}_2 a un elemento del cuerpo 128 elementos

Convierte un vector de matrices $n1$ del cuerpo \mathbb{F}_2 en una matriz *matrix* del cuerpo de 128 elementos. Como en la función anterior los elementos de la matriz son solo 0 y 1, se aplica el mismo cambio, código 6.17.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Vector de matriz con elementos en \mathbb{F}_2
int	n	Inner	Almacena las matrices del vector matrix
int	row	Inner	Almacena las filas generadas de la matriz n
int vector	matrix	Output	Vector de matrices con elementos en \mathbb{F}_{128}

Tabla 6.7: Parámetros de la función `matrix3d_F2to128`

```

1 def matrix3d_F2to128(n1=[]):
2
3     matrix=[]
4     for i in range(len(n1)):
5         n=[]
6         for j in range(len(n1[0])):
7             row=[]
8             for k in range(len(n1[0][0])):
9                 if n1[i][j][k]==0:
10                    row+= [[0,0,0,0,0,0,0]]
11
12                else:
13                    row+= [[0,0,0,0,0,0,1]]
14            n+= [row]
15        matrix+= [n]
16    return matrix

```

Código 6.17: Vector de matrices de \mathbb{F}_2 a una matriz de \mathbb{F}_{128}

6.2.2. Funciones con matrices

En esta sección se explicarán funciones como la suma de matrices, cálculo de la matriz transpuesta, matriz identidad y el producto de matrices tanto en \mathbb{F}_2 como en \mathbb{F}_{128} . Además incluye la función que resuelve el sistema de ecuaciones con el método de Gauss-Jordan.

Suma de dos matrices $m1$ y $m2$ del cuerpo de 128 elementos, código 6.18.

Tipo	Nombre	Variable	Descripción
int vector	m1	Input	Matriz a sumar en \mathbb{F}_{128}
int vector	m2	Input	Matriz a sumar en \mathbb{F}_{128}
int	row	Inner	Almacena las filas de n
int vector	m_suma	Output	Suma de las matrices m1 y m2 en \mathbb{F}_{128}

Tabla 6.8: Parámetros de la función `matrix_sum`

```

1 def matrix_sum (m1, m2):
2
3     m_suma = []
4
5     if (len(m1) == len(m2)) and (len(m1[0]) == len(m2[0])):
6         for i in range(len(m1)):
7             row = []
8             for j in range(len(m1[0])):
9                 row += [suma (m1[i][j], m2[i][j])]
10            m_suma += [row]
11    return m_suma

```

Código 6.18: Suma de dos matrices con elementos en el cuerpo

Producto de dos matrices con elementos en el cuerpo de 128 elementos, código 6.19.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Matriz a multiplicar en \mathbb{F}_{128}
int vector	n2	Input	Matriz a multiplicar en \mathbb{F}_{128}
int	p_suma	Inner	Almacena la suma de los productos de cada elemento de la fila de n1 y de la columna de n2
int	row	Inner	Almacena las filas de prod
int vector	prod	Output	Producto de las matrices m1 y m2 en \mathbb{F}_{128}

Tabla 6.9: Parámetros de la función `matrix_product`

```

1 def matrix_product (n1=[], n2=[]):
2
3     prod=[]
4     if len(n1[0]) == len(n2):
5         for i1 in range(len(n1)):
6             row = []
7             for j in range(len(n2[0])):
8                 p_suma = [0,0,0,0,0,0,0]
9                 for i2 in range(len(n2)):
10                    p_suma = suma (product(n1[i1][i2], n2[i2][j]), p_suma)
11                row += [p_suma]
12            prod += [row]

```

```
13 return prod
```

Código 6.19: Producto de matrices con elementos del cuerpo

Producto de dos matrices con elementos en el cuerpo \mathbb{F}_2 , como los elementos de la matriz son 0 y 1 la suma acumulada se hace con la operación lógica `xor`, código 6.20.

Tipo	Nombre	Variable	Descripción
int vector	n1	Input	Matriz a multiplicar en \mathbb{F}_2
int vector	n2	Input	Matriz a multiplicar en \mathbb{F}_2
int	p_suma	Inner	Almacena la suma de los productos de cada elemento de la fila de n1 y de la columna de n2
int	row	Inner	Almacena las filas de prod
int vector	prod	Output	Producto de las matrices m1 y m2 en \mathbb{F}_2

Tabla 6.10: Parámetros de la función `matrix_product_F2`

```
1 def matrix_product_F2 (n1=[], n2=[]):
2
3     prod=[]
4     if len(n1[0]) == len(n2):
5         for i1 in range(len(n1)):
6             row = []
7             for j in range(len(n2[0])):
8                 p_suma = 0
9                 for i2 in range(len(n2)):
10                     p_suma = (n1[i1][i2] * n2[i2][j]) ^ p_suma
11                 row += [p_suma]
12             prod += [row]
13     return prod
```

Código 6.20: Producto de matrices con elementos en \mathbb{F}_2

Calcula la matriz transpuesta de la matriz dada como parámetro, código 6.21.

Tipo	Nombre	Variable	Descripción
int vector	m	Input	Matriz para calcular su transpuesta
int	row	Inner	Almacena las filas de trans
int vector	trans	Output	Matriz transpuesta

Tabla 6.11: Parámetros de la función `matrix_transpose`

```
1 def matrix_transpose(m):
2
```

```

3  trans = []
4  for j in range(len(m[0])):
5      row = []
6      for i in range(len(m)):
7          row += [m[i][j]]
8      trans += [row]
9  return trans

```

Código 6.21: Matriz transpuesta

Calcula la matriz identidad con una determinada dimensión, *dim*, que se pasa como parámetro, código 6.22.

Tipo	Nombre	Variable	Descripción
int	dim	Input	Dimensión de la que calcular la matriz identidad en \mathbb{F}_{128}
int	row	Inner	Almacena las filas de matrix
int vector	matrix	Output	Matriz identidad

Tabla 6.12: Parámetros de la función `matrix_identity`

```

1  def matrix_identity(dim):
2
3      matrix = []
4
5      for i in range(dim):
6          row = []
7          for j in range(dim):
8              if i == j:
9                  row += [[0, 0, 0, 0, 0, 0, 1]]
10             else:
11                 row += [[0, 0, 0, 0, 0, 0, 0]]
12             matrix += [row]
13  return matrix

```

Código 6.22: Matriz identidad del cuerpo

La función `matrix_rref` resuelve de un sistema de ecuaciones con el método de Gauss-Jordan, código 6.23. El sistema de ecuaciones es de la forma de la ecuación 6.1.

$$A \cdot x = b \quad (6.1)$$

Tipo	Nombre	Variable	Descripción
int vector	A	Input	Matriz de la ecuación
int vector	b	Input	Vector de la ecuación
int vector	M	Inner	Matriz aumentada, combinación de A y b
int vector	x	Output	Solución del sistema

Tabla 6.13: Parámetros de la función `matrix_rref`

```

1 def matrix_rref(A, b):
2
3     r = 0
4     row = []
5     n = len(A)
6
7     #Matriz aumentada, añadir b como una columna
8     M = A
9     for i in range(len(M)):
10         M[i] += b[i]
11
12     for k in range(n):
13         #Intercambio de filas para que quede arriba la de menor valor
14         for i in range(k, n):
15             if mayor(M[i][k], M[k][k]):
16                 row = M[k]
17                 M[k] = M[i]
18                 M[i] = row
19
20         #Hacer ceros
21         for j in range(k+1, n):
22             q = product(M[j][k], inverse(M[k][k]))
23             for m in range(k, n+1):
24                 M[j][m] = suma(M[j][m], product(q, M[k][m]))
25
26     #Calcular la solución x de abajo arriba
27     x = [[0,0,0,0,0,0,0]] for i in range(n)
28
29     x[n-1] = [product(M[n-1][n], inverse(M[n-1][n-1]))]
30     for i in range(n-1, -1, -1):
31         z = [0,0,0,0,0,0,0]
32         for j in range(i+1, n):
33             z = suma(z, product(M[i][j], x[j][0]))
34         x[i] = [product(suma(M[i][n], z), inverse(M[i][i]))]
35
36     return x

```

Código 6.23: Método de Gauss-Jordan

6.2.3. Funciones algoritmo UOV

La función `clavePrivada` genera la clave privada de un usuario, para ello genera una matriz triangular superior, α , en \mathbb{F}_2 con valores aleatorios, y un vector, β , en \mathbb{F}_2 . Los parámetros m y v son el número de variables de aceite y vinagre, respectivamente, código 6.24.

Tipo	Nombre	Variable	Descripción
int vector	m	Input	Número de aceites
int vector	v	Input	Número de vinagres
int vector	alpha	Output	Vector de matrices triangulares superiores aleatorias en \mathbb{F}_2 , parte de la clave privada
int vector	beta	Output	Matriz aleatoria en \mathbb{F}_2 , parte de la clave privada

Tabla 6.14: Parámetros de la función `clavePrivada`

```

1 def clavePrivada (m, v):
2
3     alpha, beta = [], []
4     n = m+v
5
6     for k in range(m):
7         #alpha matriz triangular superior
8         aux = []
9         for i in range (v):
10             row = []
11             for j in range (n):
12                 if i > j:
13                     row += [0]
14                 else:
15                     row += [randint(0,1)]
16             aux += [row]
17         alpha += [aux]
18
19         #beta
20         row = []
21         for i in range(n):
22             if randint(0,1) == 0:
23                 row += [randint(0,1)]
24
25         beta += [row]
26     return alpha, beta

```

Código 6.24: Generación clave privada

La función `generacionT` genera la matriz de distorsión T a partir de los valores m y v , número de variables de aceite y vinagre, código 6.25. La forma de la matriz de distorsión viene dada por la ecuación 2.6.

Tipo	Nombre	Variable	Descripción
int vector	m	Input	Número aceites
int vector	v	Input	Número vinagres
int matrix	T	Output	Matriz en \mathbb{F}_2 de dimensión nxn

Tabla 6.15: Parámetros de la función `generacionT`

```

1 def generacionT (v, m):

```

```

2  T = []
3  n = v + m
4  for i in range(n):
5      row = []
6      if i < v:
7          for k in range(n):
8              if k < v: #Matriz identidad dimension v
9                  if i == k:
10                     row += [1]
11                 else:
12                     row += [0]
13
14             else: #Matriz aleatoria vxm
15                 if randint(0,2) == 1:
16                     row += [1]
17                 else:
18                     row += [0]
19
20         else:
21             for k in range(n):
22                 if k < v: #Matriz nula dimension v
23                     row += [0]
24
25                 else: #Matriz identidad dimension m
26                     if i == k:
27                         row += [1]
28                     else:
29                         row += [0]
30     T += [row]
31 return T

```

Código 6.25: Generación matriz T

La función `clavePublica` genera la clave pública a partir de la clave privada, calculada con la función anterior, los valores m y v el número de variables de aceite y vinagre, además de una matriz de distorsión, código 6.26.

Tipo	Nombre	Variable	Descripción
int vector	m	Input	Número aceites
int vector	v	Input	Número vinagres
int vector	alpha	Input	Vector de matrices triangulares superiores en \mathbb{F}_2 , parte de la clave privada
int vector	beta	Input	Matriz en \mathbb{F}_2 , parte de la clave privada
int vector	T	Input	Matriz de distorsión de la forma 2.6
int vector	alpha_pub	Output	Vector de matrices en \mathbb{F}_2 , parte de la clave pública
int vector	beta_pub	Output	Matriz en \mathbb{F}_2 , parte de la clave pública

Tabla 6.16: Parámetros de la función `clavePublica`

```

1 def clavePublica(m, v, alpha, beta, T):
2
3     alpha_pub = []
4     beta_pub = []

```

```

5  T_trans = matrix_transpose(T[0:v])
6
7  for k in range(len(alpha)):
8      aux = matrix_product_F2(T_trans , alpha [k])
9      alpha_pub += [matrix_product_F2(aux, T)]
10
11
12      beta_pub += [matrix_product_F2([beta[k]], T)]
13
14  return alpha_pub, beta_pub

```

Código 6.26: Generación clave pública

La función `signature` calcula la firma del hash de un mensaje, *hashed*, con las claves privadas del usuario, *alpha_F2* y *beta_F2*, el número de aceites y vinagres, *m* y *v*, y con la matriz de transición, *T*. Se toman los vinagres aleatorios, se resuelve el sistema para calcular los aceites, y con la ecuación 2.12 obtenemos la firma del mensaje, código 6.27.

Tipo	Nombre	Variable	Descripción
int vector	hashed	Input	Vector <i>hash</i>
int vector	alpha_F2	Input	Vector de matrices triangulares superiores en \mathbb{F}_2 , parte de la clave privada
int vector	beta_F2	Input	Matriz en \mathbb{F}_2 , parte de la clave privada
int	m	Input	Número aceites
int	v	Input	Número vinagres
int vector	T	Input	Matriz de distorsión de la forma 2.6
int vector	alpha	Inner	Almecena los correspondientes valores de alpha_F2 en \mathbb{F}_{128}
int vector	beta	Inner	Almecena los correspondientes valores de beta_F2 en \mathbb{F}_{128}
int vector	vinagre	Inner	Vector aleatorio de tamaño v, contiene los vinagres
int vector	coef	Inner	Almacena los coeficientes del sistema
int vector	term	Inner	Almacena los términos del sistema
int vector	oil	Inner	Almacena la solución del sistema, son los aceites del algoritmo
int vector	firma	Output	Contiene la firma de <i>hashed</i>

Tabla 6.17: Parámetros de la función `signature`

```

1  def signature(hashed, alpha_F2, beta_F2, m, v, T):
2
3      alpha = matrix3d_F2to128(alpha_F2)
4      beta = matrix_F2to128(beta_F2)
5
6      vinagre = []
7      for k in range(v):
8          aux = randint(0, 127)

```



```

9     vinagre += [F128(aux)]
10    coef = []
11    term = []
12
13    n= m + v
14    for k in range(m):
15        A = matrix_product([vinagre], alpha[k])
16        coef += matrix_sum([A[0][v:n]], [beta[k][v:n]])
17        v_suma = suma(matrix_product([A[0][0:v]], matrix_transpose([vinagre
18        ])) [0][0], matrix_product([beta[k][0:v]], matrix_transpose([vinagre]))
19        [0][0])
20        term += [suma(hashcd[k], v_suma)]
21
22    oil = matrix_rref(coef, matrix_transpose([term]))
23
24    aux = []
25    aux += vinagre + matrix_transpose(oil)[0]
26    firma = matrix_product([aux], matrix_transpose(matrix_F2to128(T))) #T =
    T.inverse()
    return firma[0]

```

Código 6.27: Firma del mensaje

Por último la función `verify` comprueba si la firma, *firma*, de un mensaje, *m*, es correcta, para ello utiliza las claves públicas del usuario y el número de variables de aceite, se va comprobando una a una si se cumple la igualdad 2.13, código 6.28.

Tipo	Nombre	Variable	Descripción
int vector	hashed	Input	Vector <i>hash</i>
int vector	firma	Input	Vector con la firma del <i>hash</i>
int vector	alpha_F2	Input	Vector de matrices triangulares superiores en \mathbb{F}_2 , parte de la clave privada
int vector	beta_F2	Input	Matriz en \mathbb{F}_2 , parte de la clave privada
int	m	Input	Número aceites
int vector	alpha	Inner	Almacena los correspondientes valores de alpha_F2 en \mathbb{F}_{128}
int vector	beta	Inner	Almacena los correspondientes valores de beta_F2 en \mathbb{F}_{128}
boolean	verif	Output	True si la firma es válida False en otro caso

Tabla 6.18: Parámetros de la función `verify`

```

1 def verify(hashed, firma, alpha_pub_F2, beta_pub_F2, m):
2
3     verif = True
4     alpha_pub = matrix3d_F2to128(alpha_pub_F2)
5     beta_pub = matrix3d_F2to128(beta_pub_F2)
6     for k in range(m):

```

```
7      aux_alpha = matrix_product(matrix_product ([firma], alpha_pub[k]),  
      matrix_transpose([firma]))  
8      aux_beta = matrix_product (beta_pub[k], matrix_transpose([firma]))  
9      verif = verif and (hashed[k] == matrix_sum(aux_alpha, aux_beta)  
      [0][0])  
10  
11     return verif
```

Código 6.28: Verificación de la firma

Capítulo 7

Evaluación y pruebas

Se han llevado a cabo pruebas sobre los tiempos de ejecución del algoritmo criptográfico UOV. Se han comparado dichos tiempos en dos lenguajes de programación `python` y `SageMath`. Por otra parte las pruebas de la integración han sido realizar una transacción y ver que esa transacción se encuentra en un bloque.

7.1. Pruebas algoritmo criptográfico

El tiempo de ejecución del algoritmo se lo lleva en gran parte la generación de la clave pública. En esta función para cada α_{pub_k} , con $k \in \{1, \dots, v\}$, es necesario multiplicar tres matrices de órdenes $n \times n$, $m \times n$, y $n \times n$, respectivamente. De esta forma la complejidad de la multiplicación de matrices es de $\mathcal{O}(n^3)$ por v , se obtiene que la complejidad de la generación de α_{pub} es de $\mathcal{O}(n^4)$.

En el caso de β_{pub} , hay que multiplicar v un vector por de tamaño n por un matriz de dimensiones $n \times n$. Así la complejidad del cálculo de β_{pub} es de $\mathcal{O}(n^3)$.

Entonces la complejidad de dicha función es de orden $\mathcal{O}(n^4)$.

La tabla 7.1 muestra algunos tiempos de ejecución al calcular la clave pública. Se han comparado los tiempos de ejecución en `python` y en un lenguaje de programación matemático, `SageMath`. Vemos que los tiempos en `python` son considerablemente mayores, más aún con valores de m y v grandes, una razón es por que las funciones de `python` han sido implementadas y no son de una biblioteca como las de `SageMath` que siempre son más óptimas.

m	v	Tiempo (mm' ss") python	Tiempo (mm' ss") SageMath
57	197	3'34.65"	0.012"
44	151	1'15.61"	0.007"
35	191	29.61"	0.009"
19	65	2.77"	0.006"
10	30	0.17"	0.004"
3	3	0.0004"	0.003"

Tabla 7.1: Tiempos de generación de la clave pública

Para la implementación en la *blockchain* se ha optado por los valores de m y v , 3 y 3, respectivamente. De esta forma los bloques se firmarán rápidamente y las firmas serán más pequeñas. Así para será más didáctico ya que dichas firmas podrán visualizarse mejor y entender que significa.

La ventaja al integrar el algoritmo en *blockchain* será que hará la cadena de bloques más segura, sobre todo cuanto mayores sean los valores de m y v . Por contra disminuirá el rendimiento, ya que para esos valores de m y v los tiempos serán más elevados.

7.2. Pruebas de integración

Para ver que la integración funciona correctamente se han seguido unos pasos.

En primer lugar se ha instalado e iniciado el *core*, y se han mostrado los logs del mismo para comprobar que no haya ningún error durante la firma de los bloques que se vayan generando.

Posteriormente, se ha abierto la aplicación Wallet para realizar una transacción, en los mismos logs del *core* se ha mostrado que la transacción es válida y que se ha almacenado en un bloque. Esta parte no se ha podido complimentar, ya que los bloques con una transacción no se firman correctamente.

Por último, en el *explorer* de ARK se buscan la transacción y el bloque en el que se ha almacenado, se puede ver la información referente al bloque como quién lo ha firmado, cuándo se ha creado el bloque o el número de transacciones. Si se desea visualizar la firma de un bloque hay que acceder a la API y buscar el bloque en cuestión, por la ID.

Capítulo 8

Conclusiones

Este proyecto parte de la idea de hacer más seguras las cadenas de bloques, puesto que por si mismas no lo son. Para ello es necesario tener un algoritmo de firma de los bloques y evitar dichas vulnerabilidades. Además el algoritmo implementado es resistente a ataques cuánticos, de esta forma no solo hará la *blockchain* más segura antes un ordenador clásico sino también ante un ordenador cuántico.

El inicio del proyecto fue el estudio de algunos conceptos fundamentales como son la computación cuántica, cadenas de bloques y el algoritmo UOV. A continuación, se implementó el algoritmo criptográfico UOV, junto con la aritmética de cuerpos finitos. La implementación se ha hecho en dos lenguajes para comparar los tiempos de ejecución, en SageMath y python. Este último será el que se utilice en la *blockchain*. Seguidamente, se ha creado el entorno de trabajo para instalar la *blockchain*, utilizando la tecnología docker. También, se ha procedido a la integración del algoritmo en la *blockchain*.

Con todos estos pasos se ha conseguido una cadena de bloques resistentes a ataques cuánticos que servirá para almacenar las transacciones generadas con la aplicación Wallet. Todo este proceso de creación y firma tanto de bloques como de transacciones se pueden visualizar con el *explorer* de ARK.

Para complementar este proyecto de cara al futuro, se podría trabajar con la base de datos, que incluya la información de los archivos *json* generados para almacenar la firma, en lugar de tenerlos en archivos independientes.

Otra línea de investigación que podría ser interesante, sería integrar esta *blockchain* en otra para hacerla más segura frente a ataques cuánticos gracias a las propiedades de la cadena de bloques de ARK.

Bibliografía

- [1] “Qilimanjaro: Next computing generation, quantum computation at your fingertips,” 2018, <https://neironix.io/documents/whitepaper/4514/whitepaper.pdf>.
- [2] Wikipedia, “Esfera de Bloch,” https://es.wikipedia.org/wiki/Esfera_de_Bloch.
- [3] A. Palau, “Tokenización & Árbol de Merkle,” 2018, <https://medium.com/@albpalau/tokenizaci%C3%B3n-%C3%A1rbol-de-merkle-1276820a1d60>.
- [4] L. Wilson, “The Rise of Quantum Computers – The Current State of Cryptographic Affairs,” 2016, <https://www.activecyber.net/rise-quantum-computers-current-state-cryptographic-affairs/>.
- [5] J. V. of Jeremiah Owyang, “Roadmap: Five Phases of Digital Eras,” 2019, <https://web-strategist.com/blog/2019/01/04/roadmap-five-phases-of-digital-eras/>.
- [6] F. G. Pacheco and H. Jara, *Hackers al descubierto*. USERSHOP, 2010.
- [7] J. C. G. Díaz and J. C. G. Díaz, *Criptografía: historia de la escritura cifrada*. Editorial Complutense, 1995.
- [8] F. J. Lobillo, “Parte II Criptografía. Criptosistema de RSA y criptosistemas basados en el logaritmo discreto.”
- [9] I. Gallego Sagastume, “Estructuras algebraicas aplicables en criptografía,” in *XVIII Workshop de Investigadores en Ciencias de la Computación (WICC 2016, Entre Ríos, Argentina)*, 2016.
- [10] D. Mahto, D. A. Khan, and D. K. Yadav, “Security analysis of elliptic curve cryptography and rsa,” in *Proceedings of the world congress on engineering*, vol. 1, 2016, pp. 419–422.
- [11] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on post-quantum cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016, vol. 12.

- [12] R. Tonon, "The story of Fogo de Chão's rise to fame," 2016, <https://www.eater.com/2016/10/6/13168942/fogo-de-chao-brazilian-steakhouse>.
- [13] A. Szeponiec, B. Preneel, F. Vercauteren, and W. Beullens, "LUOV: Signature Scheme proposal for NIST PQC Project (Round 2 version)," 2018.
- [14] "Ark," <https://ark.io/>.
- [15] F. Luis, "Computación cuántica con imanes moleculares," 2014.
- [16] L. De La Peña, *Introducción a la mecánica cuántica*. Fondo de Cultura económica, 2014.
- [17] Álvaro Rodrigo Reyes, "Estado de la criptografía post-cuántica y simulaciones de algoritmos post-cuánticos," <http://openaccess.uoc.edu/webapps/o2/bitstream/10609/89026/6/alvaroreyesTFM1218memoria.pdf>.
- [18] Redacción Vivir, "La "nueva frontera" alcanzada por el computador cuántico de Google esta semana," 08 septiembre 2020, <https://www.elspectador.com/noticias/ciencia/la-nueva-frontera-de-la-computacion-cuantica/>.
- [19] A. Muñoz and J. I. Escribano, "La computación cuántica y el "futuro de la criptografía": la criptografía post-cuántica," 2020, <https://www.bbvanexttechnologies.com/la-computacion-cuantica-y-el-futuro-de-la-criptografia-la-criptografia-post-cuantica/>.
- [20] A. Banafa, "Computación cuántica y blockchain, mitos y realidades," 2019, <https://www.bbvaopenmind.com/tecnologia/mundo-digital/computacion-cuantica-y-blockchain-mitos-y-realidades/>.
- [21] V. J. Cuadros Choez, "Análisis comparativo de un sistema de auditoría tradicional y un sistema de auditoría blockchain e ipes." B.S. thesis, 2020.
- [22] "History of Blockchain," <https://academy.binance.com/blockchain/history-of-blockchain>.
- [23] "A brief history of Blockchain," <https://www.icaew.com/technical/technology/blockchain/blockchain-articles/what-is-blockchain/history>.
- [24] F. Tschorsch and B. Scheuermann, *Bitcoin and Beyond: A technical survey on decentralized digital currencies*. Humboldt University of Berlin.
- [25] "Contrato inteligente," <https://elderecho.com/los-contratos-inteligentes-smart-contracts-contratos-inteligentes>.
- [26] "Web oficial de Hyperledger," <https://www.hyperledger.org/>.

- [27] Hyperledger Project, "Linux Foundation's Hyperledger Project announces 30 founding members and code proposals to advance Blockchain technology," <https://web.archive.org/web/20160225023123/https://www.hyperledger.org/news/announcement/2016/02/hyperledger-project-announces-30-founding-members>.
- [28] A. Preukschar, "Hyperledger: la Blockchain privada que todos tenemos que conocer," 2018, <https://www.eleconomista.es/economia/noticias/8899454/01/18/Hyperledger-la-Blockchain-privada-que-todos-tenemos-que-conocer.html>.
- [29] C. Pastorino, "Blockchain: qué es, cómo funciona y cómo se está usando en el mercado," 2018, <https://www.welivesecurity.com/la-es/2018/09/04/blockchain-que-es-como-funciona-y-como-se-esta-usando-en-el-mercado/>.
- [30] A. Rosic, "What is blockchain Technology? A Step-by-Step Guide For Beginners," https://blockgeeks.com/guides/what-is-blockchain-technology/#The_Three_Pillars_of_Blockchain_Technology.
- [31] B. Zieli, "ARK Crypto Project," 2020, <https://www.quora.com/q/cryptoplanet360/Ark-ARK-Crypto-Project-Pros-and-Cons-of-the-Ark-ecosystem-my-opinion>.
- [32] R. Cernec, "Making Cross-Chain Smart Contracts on ARK via Hyperledger Fabric," 2018.
- [33] A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced oil and vinegar signature schemes," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 206–222.
- [34] RedHat, "Concepto de DevOps," <https://www.redhat.com/es/topics/devops>.
- [35] S. Metzger, D. Durden, C. Sturtevant, H. Luo, N. Pingintha-Durden, T. Sachs, A. Serafimovich, J. Hartmann, J. Li, K. Xu *et al.*, "eddy4r 0.2. 0: a devops model for community-extensible processing and analysis of eddy-covariance data based on r, git, docker, and hdf5," *Geoscientific Model Development*, vol. 10, pp. 3193–3194, 2017.
- [36] J. Renken, "Ark deployer launch: Creating your own blockchain in three simple steps," 2019.
- [37] DocumentationARK, "Glosario de ARK – Base de datos," <https://ark.dev/docs/glossary/glossary#database>.
- [38] DocumentationARK, "Desktop Wallet ARK," <https://ark.io/desktop-wallet>.
- [39] DocumentationARK, "What is a block explorer?" <https://arkdoc-23.docs.uns.network/tutorials/usage-guides/how-to-use-ark-explorer.html>.

- [40] M. V. Granados and ARK, “Core Blockchain ARK,” 2020, <https://github.com/mvictoria1997/core.git>.
- [41] M. V. Granados, “Repositorio Trabajo de Fin de Grado,” 2020, <https://github.com/mvictoria1997/TFG.git>.
- [42] M. Beunardeau, A. Connolly, H. Ferradi, R. Géraud, D. Naccache, and D. Vergnaud, “Reusing nonces in schnorr signatures,” in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 224–241.
- [43] “Install docker engine on ubuntu,” <https://docs.docker.com/engine/install/ubuntu/>.
- [44] Github, “Repositorio ARKEcosystem/core,” <https://github.com/ArkEcosystem/core>.
- [45] Github, “Versiones para descargar de ARK Wallet,” <https://github.com/ArkEcosystem/desktop-wallet/releases>.

Siglas

DSA Digital Signature Algorithm.

ECDSA Elliptic Curve Digital Signature Algorithm.

iOS iPhone Operative System.

PBFT Practical Byzantine Fault Tolerance.

RPoW Reusable Proof-Of-Work.

RSA Rivest, Shamir y Adleman.

UOV Unbalanced Oil and Vinegar.

Apéndice A

Manual de usuario

El manual de usuario se va a realizar para una máquina ubuntu.

A.1. Instalación de Docker

Para guardar mejor las modificaciones que se vayan realizando es mejor instalar una máquina docker en lugar de hacerlo en local. Así el primer paso es instalar la última versión de docker, en esta parte se seguirá el tutorial oficial de Docker [43]. Antes de nada se va a desinstalar cualquier versión anterior de Docker, código A.1

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

Código A.1: Instalación Docker. Parte I

Es necesario actualizar los paquetes apt para tener acceso a las últimas actualizaciones e instalar los paquetes que permiten al sistema operativo acceder a los repositorios de Docker a través de HTTPS, código A.2.

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates curl gnupg-
agent software-properties-common
```

Código A.2: Instalación Docker. Parte II

Se añade la clave GPG oficial de Docker, código A.3. La clave GPG es una característica de seguridad para asegurar que el software que se va instalar es auténtico.

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
OK
```

Código A.3: Instalación Docker. Parte III

Verificar que obtenemos la clave con la siguiente huella, para ello hay que buscar la huella con los últimos 8 dígitos, de la misma, código [A.4](#).

```
9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
```

```
$ sudo apt-key fingerprint 0EBFCD88

pub  rsa4096 2017-02-22 [SCEA]
     9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
uid          [ unknown] Docker Release (CE deb) <docker@docker.com>
sub  rsa4096 2017-02-22 [S]
```

Código A.4: Instalación Docker. Parte IV

La instalación del repositorio de Docker se hace mediante el código [A.5](#).

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/
linux/ubuntu $(lsb_release -cs) stable"
```

Código A.5: Instalación Docker. Parte V

Actualizar los repositorios que se acaban de agregar, e instalar la última versión de Docker Engine y Docker Containerd, código [A.6](#).

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Código A.6: Instalación Docker. Parte VI

Verificar que se ha instalado correctamente comprobando la versión de Docker, código [A.7](#).

```
$ docker --version

Docker version 19.03.13, build 4484c46d9d
```

Código A.7: Instalación Docker. Parte VII

Algunos comandos útiles para el trabajo con Docker se muestran en el código [A.8](#).

```
1 #Muestra los contenedores
2 $ sudo docker ps
3
4 #Lista los contenedores con los IDs
5 $ sudo docker container ls --all
6
7 #Lista las imagenes con los IDs
8 $ sudo docker images ls --all
9
10 #Guarda los cambios del docker
11 $ sudo docker commit <ID-CONTAINER> <NOMBRE-NUEVO:ETIQUETA>
12
13 #Corre un contenedor, abriendo los puertos indicados
```

```
14 $ sudo docker run -it -p <PUERTO:PUERTO> <NOMBRE:ETIQUETA>
15
16 #Elimina un contenedor
17 $ sudo docker rm <ID-CONTAINER>
```

Código A.8: Comandos útiles de Docker

A.2. Instalación Blockchain ARK

En Docker, hay que iniciar una imagen de ubuntu xenial, código [A.9](#). Además es necesario abrir algunos puertos como 4103, para la API pública, 4102, conexión P2P API y 4200, para el *explorer*.

```
$ sudo docker run -ti -p 4102:4102 -p 4103:4103 -p 4200:4200 ubuntu:
xenial
```

Código A.9: Instalación *blockchain*. Parte I

Una vez dentro de la máquina Docker hay que instalar `sudo`, para poder trabajar en modo administrador desde el usuario que se va a crear, código [A.10](#).

```
$ apt-get update
$ apt-get install sudo
$ adduser deployer

Añadiendo el usuario 'deployer' ...
Añadiendo el nuevo grupo 'deployer' (1001) ...
Añadiendo el nuevo usuario 'deployer' (1001) con grupo 'deployer' ...
Creando el directorio personal '/home/deployer' ...
Copiando los ficheros desde '/etc/skel' ...
Introduzca la nueva contraseña de UNIX: *****
Vuelva a escribir la nueva contraseña de UNIX: *****
passwd: contraseña actualizada correctamente
Cambiando la información de usuario para deployer
Introduzca el nuevo valor, o presione INTRO para el predeterminado
Nombre completo []:
Número de habitación []:
Teléfono del trabajo []:
Teléfono de casa []:
Otro []:
¿Es correcta la información? [S/n] S
```

Código A.10: Instalación *blockchain*. Parte II

Cambiar el modo del usuario “deployer”, incluyéndolo en el grupo `sudo`, para que sea un superusuario y finalmente, entrar al usuario `deployer`, código [A.11](#).

```
$ usermod -aG sudo deployer
$ su - deployer
```

Código A.11: Instalación *blockchain*. Parte III

Actualizar los paquete e instalar algunos nuevos como `git`, `curl` y `yarn`, código [A.12](#).

```
$ sudo apt-get update
$ sudo apt-get install git curl jq apt-transport-https nano
```

Código A.12: Instalación *blockchain*. Parte IV

Instalar las dependencias `nvm`, código [A.13](#), es necesario para instalar posteriormente el paquete `pm2`.

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.8/
install.sh | bash
```

Código A.13: Instalación *blockchain*. Parte V

Para comprobar que se ha instalado correctamente hay que salir y volver a entrar en el usuario "deployer", código [A.14](#).

```
$ exit
$ su - deployer
$ command -v nvm

nvm
```

Código A.14: Instalación *blockchain*. Parte VI

Si en algún momento se desea desinstalar la dependencia `nvm` habrá que seguir los comandos [A.15](#).

```
$ nvm use system
$ npm uninstall -g a_module
$ sudo npm install -g npm
```

Código A.15: Instalación *blockchain*. Parte VII

Para instalar `pm2` ver el código [A.16](#).

```
$ sudo apt-get install npm
$ sudo npm i -g pm2
$ sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Código A.16: Instalación *blockchain*. Parte VIII

Algunos comandos interesantes para trabajar con `pm2` se muestran en el código [A.17](#). Servirán para observar si la *blockchain* y el *explorer* están levantados.

```
#Lista los demonios de pm2
$ pm2 list

#Se obtienen los estados de los demonios de pm2
$ pm2 status
```

Código A.17: Comandos útiles de `pm2`

Hasta aquí se han instalado los paquetes y dependencias para poder empezar a instalar la *blockchain*. Ahora hay que clonar el directorio `deployer` de @ArkEcosystem, código A.18.

```
$ git clone https://github.com/ArkEcosystem/deployer.git
$ chmod 764 deployer/setup.sh
$ ./deployer/setup.sh
```

Código A.18: Instalación *blockchain*. Parte IX

Una vez instalado el `deployer` hay que proceder a realizar una modificación en el archivo `/home/<USUARIO>/deployer/app/app-core.sh`, código A.19. Para que en lugar de instalar el repositorio de GitHub sin modificaciones, se instale la *blockchain* con el nuevo algoritmo UOV.

```
1 git clone https://github.com/mvictoria1997/core.git --single-branch "
  $BRIDGECHAIN_PATH"
```

Código A.19: Línea 108 `app-core.sh`

La *blockchain* va a utilizar la base de datos local del Docker, así pues hay que iniciarla con el código A.20. Además va a ser necesario instalar la librería `libjemalloc` para no tener problemas de memoria.

```
$ sudo service postgresql start
$ sudo apt-get update -y
$ sudo apt-get install -y libjemalloc-dev
```

Código A.20: Instalación *blockchain*. Parte X

Durante la instalación de la *blockchain* se descargará el código con los algoritmos de firma, por tanto en este punto debemos de modificar los archivos. Para ello hay que hacer un `fork` del repositorio `ArkEcosystem/core` [44] en nuestro usuario de Github. A continuación se descarga el código con el comando A.21.

```
$ git clone https://github.com/<USUARIO>/core.git
```

Código A.21: Clonar nuevo *core*

Los archivos que se van a modificar se encuentran en el directorio `core/packages/src/crypto/hash.ts`. En el archivo `hash.ts` se modifican las funciones de firma y verificación con el algoritmo ECDSA quedando de la forma que muestra el código 6.2.

Se añaden, en el archivo `hash.ts`, las funciones de firma y validación de la misma con el algoritmo UOV. Ver código 6.3 para la firma y código 6.4 para la verificación.

Hay que añadir los archivos en `python` a donde se realizarán las llamadas desde `typescript`. Estos son los ficheros `signature.py` para la firma, código 6.5 y `verify.py` para la validación, código 6.6.

Finalmente se crean los dos ficheros `.json`, en ellos se puede incluir un ejemplo para ver que estructura van a seguir dichos archivos.

El código A.22 muestra un ejemplo de la estructura del fichero `data.json` que almacena las claves privadas y públicas tanto para el algoritmo de Schnorr como para el de UOV.

```

1  [
2    { "id": 1,
3      "pub_schnorr": "D2",
4      "priv_schnorr": "C6",
5      "priv_alpha_UOV": [[1, 0], [1, 1]],
6      "priv_beta_UOV": [1, 0, 1],
7      "pub_alpha_UOV": [[1, 1], [0, 1]],
8      "pub_beta_UOV": [[1, 0, 1, 1, 1]]
9    }
10 ]

```

Código A.22: Ejemplo fichero `data.json`

El código A.23 muestra un ejemplo de la estructura del fichero `signature.json` que almacena las firmas en dos formatos, hexadecimal y en vector.

```

1  [
2    { "hex": "54ga24",
3      "vector": [[1, 0, 0, 0, 1], [1, 0, 1, 0, 1, 0]]
4    }
5  ]

```

Código A.23: Ejemplo fichero `signature.json`

Llegados a este punto hay que instalar el `core` de la *blockchain* con el algoritmo UOV, código A.24, la instalación tardará unos diez minutos. Una vez que instalado se obtiene en la salida la dirección y la *passphrase* del wallet Génesis, ambas serán necesarias para poder realizar transacciones por tanto puede resultar útil almacenarlas en un archivo, ver figura A.1. De todas formas se puede encontrar la *passphrase* junto con la dirección en el archivo `/home/<USUARIO>/bridgechain/testenet/<NOMBRE-BRIDGECHAIN>/genesisWallet.json`.

El archivo de configuración `/home/<USUARIO>/deployer/config.sample.conf` se puede modificar para cambiar por ejemplo el nombre de la *blockchain*, en este caso se le ha puesto “victoria-bridgechain”, variable `chainName`. Otros parámetros a destacar para modificar son `databaseName`, `totalPremine`, `bridgechainPath` y `explorerPath`.

- `databaseName`: Nombre de la base de datos.
- `totalPremine`: El valor total de tokens o monedas que tendrá la red local.
- `bridgechainPath`: *path* del directorio de instalación de la *blockchain*.
- `explorerPath`: *path* del directorio de instalación del *explorer*.

```
$ ./deployer/bridgechain.sh install-core --config deployer/config.sample
.json --autoinstall-deps --non-interactive
```

Código A.24: Instalación *blockchain*. Parte XI

```
-----
Passphrase Details
-----
Your MAINNET Genesis Details are:
  Passphrase: "rely faith vessel enable other tide lab marble include acquire plastic special"
  Address: "MQeLHEKRZTfGMDCKWCee9o22eSQta66CC1"

You can find the genesis wallet passphrase in '/home/deployer/.bridgechain/mainnet/bridgechain/genesisWallet.json'
You can find the delegates.json passphrase file at '/home/deployer/.bridgechain/mainnet/bridgechain/delegates.json'
-----
Your DEVNET Genesis Details are:
  Passphrase: "warrior category crime bone much culture excite test agent vendor unique hood"
  Address: "D5wawkANcMLigBjTSL45uyfkRvA2gQ7ZGn"

You can find the genesis wallet passphrase in '/home/deployer/.bridgechain/devnet/bridgechain/genesisWallet.json'
You can find the delegates.json passphrase file at '/home/deployer/.bridgechain/devnet/bridgechain/delegates.json'
-----
Your TESTNET Genesis Details are:
  Passphrase: "soccer abandon assume bleak feed unable sphere deliver connect stay liquid diet"
  Address: "TVUTRhCeDYmmLM56VPgCBmVedfkXyQ14m3"

You can find the genesis wallet passphrase in '/home/deployer/.bridgechain/testnet/bridgechain/genesisWallet.json'
You can find the delegates.json passphrase file at '/home/deployer/.bridgechain/testnet/bridgechain/delegates.json'
or '/home/deployer/core-bridgechain/packages/core/bin/config/testnet/delegates.json'
-----
==> Installing [mainnet] configuration to /home/deployer/.config/bridgechain-core/mainnet...
==> [mainnet] configuration Installed!
==> Installing [devnet] configuration to /home/deployer/.config/bridgechain-core/devnet...
==> [devnet] configuration Installed!
==> Installing [testnet] configuration to /home/deployer/.config/bridgechain-core/testnet...
==> [testnet] configuration Installed!
==> Bridgechain Installed!
```

Figura A.1: Salida tras la instalación del *core*

A continuación se instalará el *explorer* para poder ver las transacciones y bloques generados, código [A.25](#).

```
$ ./deployer/bridgechain.sh install-explorer --config deployer/config.
sample.json --skip-deps --non-interactive
```

Código A.25: Instalación *blockchain*. Parte XII

El último paso es iniciar la *blockchain* y el *explorer*, código A.26. La salida tras iniciar el *explorer* la muestra la figura A.2, que son los estados de la *blockchain* y del *explorer*. Esto mismo se visualiza con el comando `pm2 status`.

```
$ ./deployer/bridgechain.sh start-core --network testnet

==> Starting...
Starting victoriabridgechain-relay... done
Starting victoriabridgechain-forger... done
==> Start OK!

$ ./deployer/bridgechain.sh start-explorer --network testnet
```

Código A.26: Instalación *blockchain*. Parte XIII

id	name	namespace	version	mode	pid	uptime	U	status	cpu	mem	user	watching
2	explorer	default	3.0.0	fork	4733	0s	0	online	0%	26.4mb	deployer	disabled
1	victoriabridgechain-forger	default	2.6.49	fork	4523	85s	0	online	0%	28.1mb	deployer	disabled
0	victoriabridgechain-relay	default	2.6.49	fork	4392	89s	0	online	0%	54.3mb	deployer	disabled

==> Start OK!

Figura A.2: Salida tras iniciar del *explorer*

Para visualizar las transacciones en el *explorer*, debemos de abrir un navegador y acceder a la url `http://NODE_GENESIS_IP:EXPLORER_PORT`, donde el `EXPLORER_PORT` es 4200. La figura A.3 muestra las primeras transacciones realizadas, entre ellas se encuentra la transacción inicial al monedero génesis, además del registro de los delegados.


<div>  Menu Find a block, transaction, address or delegate </div>						
<div> Transactions Height: 467 Network: Testnet Local Supply: 21000.934 M </div>						
<div> Transaction type: All Type: All </div>						
ID	Timestamp	Sender	Recipient	Smartbridge	Amount	Fee
ea160...1c80b	29/09/2020 10:17:40	genesis_3	Delegate Registration		0 M	0 M
08224...a7d25	29/09/2020 10:17:40	genesis_2	Delegate Registration		0 M	0 M
104e2...b5c43	29/09/2020 10:17:40	genesis_1	Delegate Registration		0 M	0 M
d6c90...61f35	29/09/2020 10:17:40	TAndj...9rgxw	TShpH...wrWWA		21.000.000 M	0 M
<div> Previous 1 2 3 </div>						
<div> © ARK.io 2020. All rights reserved Version: ac16056 Date: 2020-09-19 </div>						

Figura A.3: Primeras transacciones en el *explorer*

A.3. Instalación de la aplicación ARK Desktop Wallet

Antes de descargar la aplicación ARK Desktop Wallet, en el ordenador local, son necesarias algunas instalaciones previas, como algunos archivos de desarrollo de `libudev`, `node 12` y `yarn`, código A.28.

```
$ sudo apt-get install libudev-dev libusb-1.0-0-dev

$ npm install -g n
$ sudo n 12

# Para comprobar que se ha instalado node 12
$ n --version

$ npm install -g yarn
```

Código A.27: Instalaciones previas a la aplicación ARK Wallet

La descarga de la aplicación se realiza desde el repositorio de GitHub de `@ArkEcosystem/desktop-wallet` [45], para el proyecto se ha descargado la última versión del archivo `.deb`.

```
$ sudo dpkg -i ark-desktop-wallet-linux-amd64-<VERSION>.deb

#Para desinstalarlo
$ sudo apt-get remove ark-desktop-wallet
```

Código A.28: Instalación de la aplicación ARK Wallet

Una vez que se ha instalado la aplicación, la abrimos y nos encontramos con la figura A.4. A continuación, vamos siguiendo los pasos que nos salen en la parte de la derecha de la pantalla.

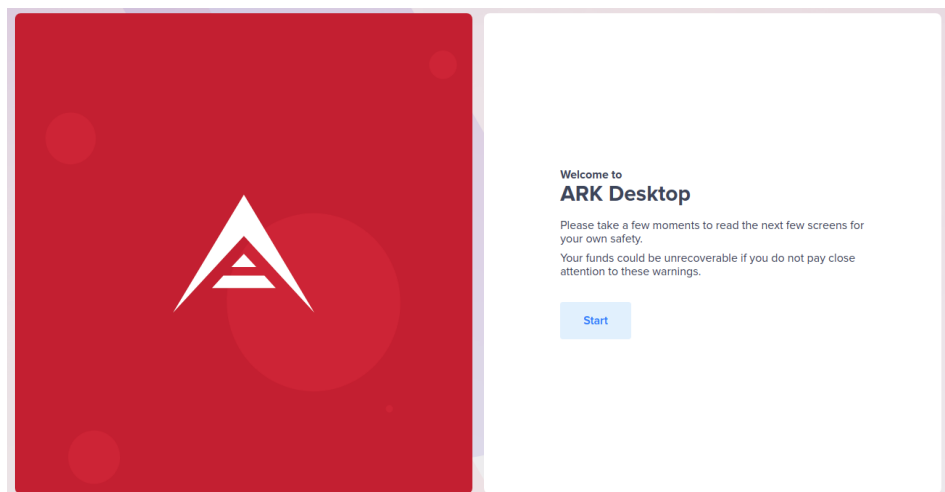
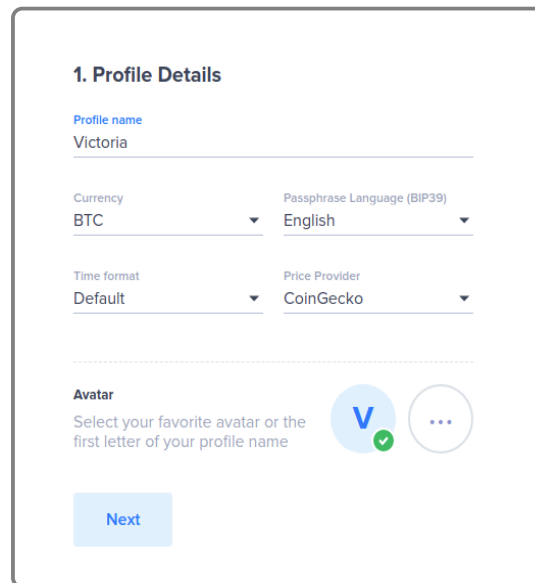


Figura A.4: Inicio de Wallet

Así se llega al primer paso, crear un perfil, en el que hay que indicar el nombre y la moneda con la que se quiere trabajar, en este caso se dejan los valores por defecto, además existe la opción de cambiar el avatar del usuario, figura A.5.



1. Profile Details

Profile name
Victoria

Currency
BTC

Passphrase Language (BIP39)
English

Time format
Default

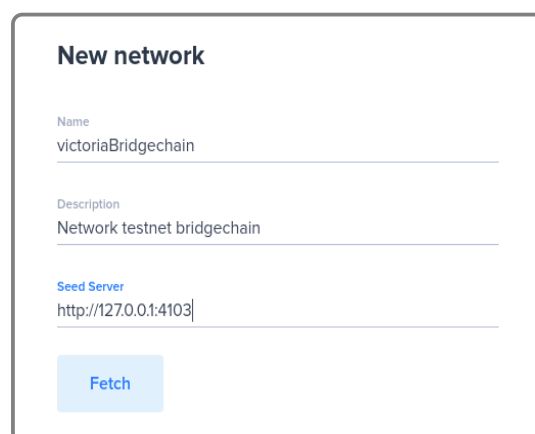
Price Provider
CoinGecko

Avatar
Select your favorite avatar or the first letter of your profile name

Next

Figura A.5: Detalles del perfil

En el siguiente paso aparece un menú con las posibles redes, *mainnet* y *devnet*. Sin embargo, si se desea usar la red *testnet* es necesario crearla pulsando nueva red. Los campos a rellenar son el nombre, una breve descripción y la dirección del servidor con el puerto de la API, `http://GENESIS_NODE_IP:API_PORT`, figura A.6.



New network

Name
victoriaBridgechain

Description
Network testnet bridgechain

Seed Server
http://127.0.0.1:4103

Fetch

Figura A.6: Configuración de la nueva red

Cuando se ha creado la red aparece una pantalla con los detalles de la misma donde debemos de cambiar la dirección por defecto del *explorer* y poner `http://GENESIS_NODE_IP:EXPLORER_PORT`. También se pueden cambiar el nombre de los Tokens y el símbolo. Una vez realizados los cambios, guardamos la red, figura A.7.

New network

Basic Advanced

Name
victoriaBridgechain

Description
Network testnet bridgechain

Seed Server
http://127.0.0.1:4103

Nethash
ce7e827b340e43251930587de8dcdee1a98c96077f91779

Token
MVToken

Symbol
MV

Version
65

Epoch
2020-10-02T07:29:30.947Z

Explorer
http://127.0.0.1:4200

Known Wallets URL

Market Ticker (Optional)
MINE

Save

Figura A.7: Detalles de la nueva red

Se selecciona la red recién creada, victoriaBridgechain y se avanza al siguiente paso, figura A.8.

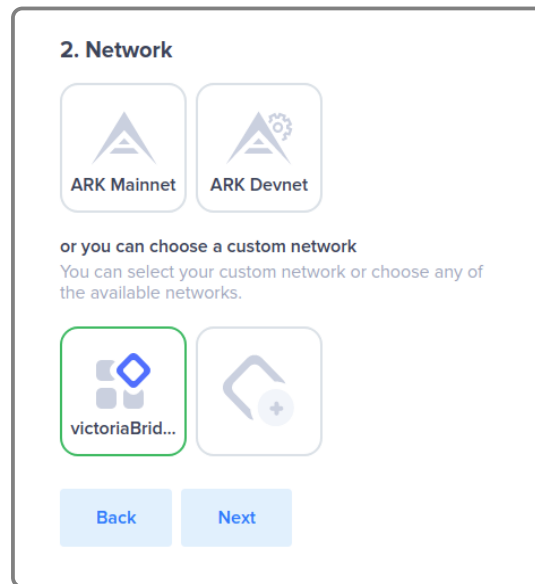


Figura A.8: Selección de la red

Para acabar con la creación del usuario, se tiene la opción de personalizar los colores de la interfaz de usuario o cambiar al tema de noche, figura A.9.

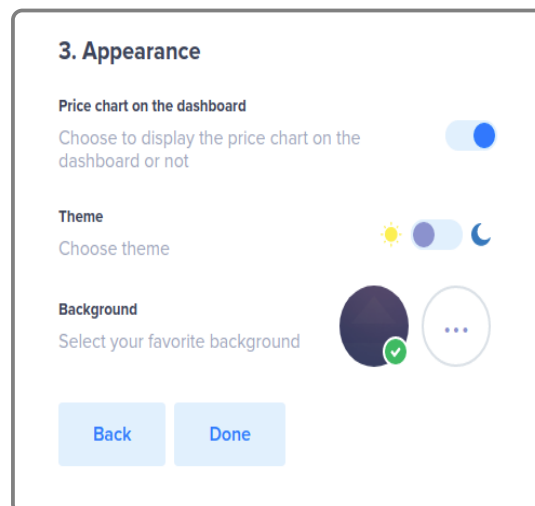


Figura A.9: Personalización del diseño de la interfaz

Es necesario conectar la aplicación con la *blockchain*, para que salga la cantidad de dinero inicial. Pulsando en el panel lateral izquierdo en *Network*, se accede a la configuración de la red y pulsando en *Connect custom peer* obtenemos la figura A.10. Rellenamos los campos con `http://GENESIS_NODE_IP` y con el `API_PORT`. Cuando se conecte hay que refrescar la página para que se actualice el monedero con el dinero.

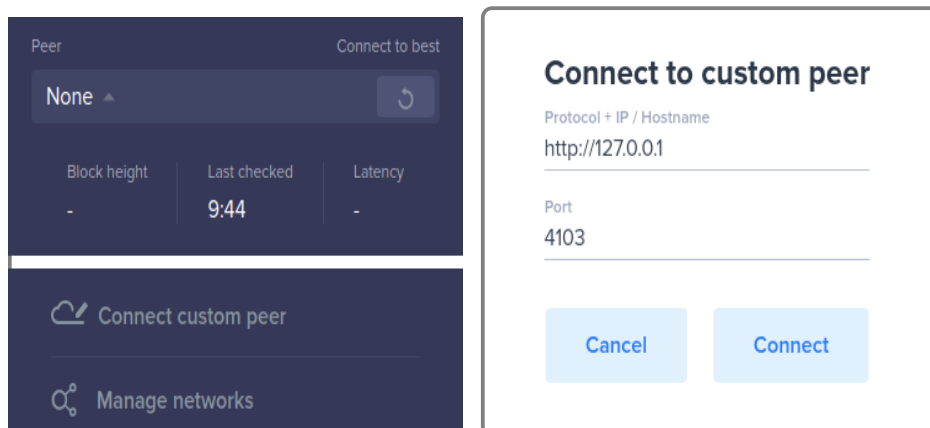


Figura A.10: Configuración de la conexión peer

La primera vez que se entra al usuario es necesario importar el monedero que se ha creado durante la instalación de la *blockchain*, pulsando en la esquina superior derecha en *Import Wallet*. No es necesario rellenar los dos campos, se puede importar el monedero introduciendo solamente la *passphrase*, figura A.11.

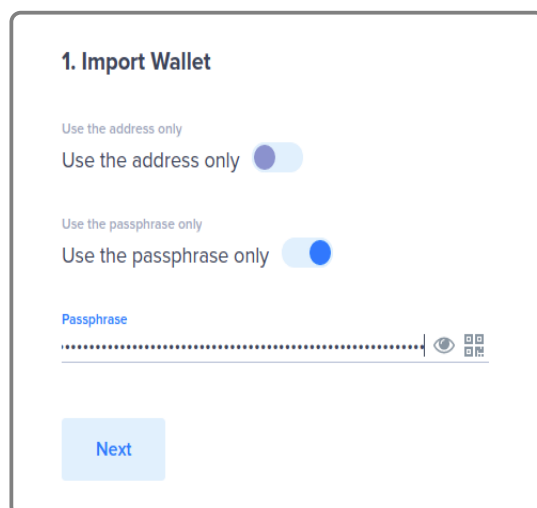


Figura A.11: Importar monedero

Si se desea se puede poner contraseña al monedero, haciéndolo más seguro, en el ejemplo no se van a poner contraseña a ninguno de los monederos, figura A.12.

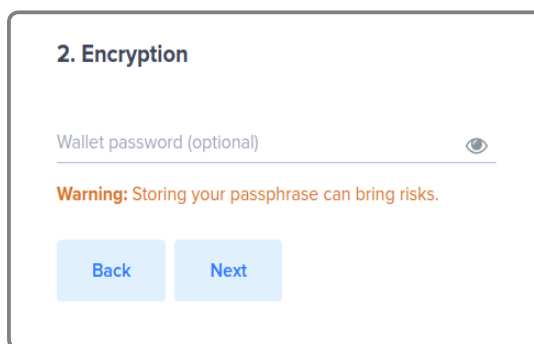


Figura A.12: Encriptación el monedero

Finalmente, hay que ponerle nombre al monedero, para diferenciarlo de otros monederos y saber cual es el que contiene la cantidad inicial, el nombre que se le va a poner es MainWallet, figura A.13.

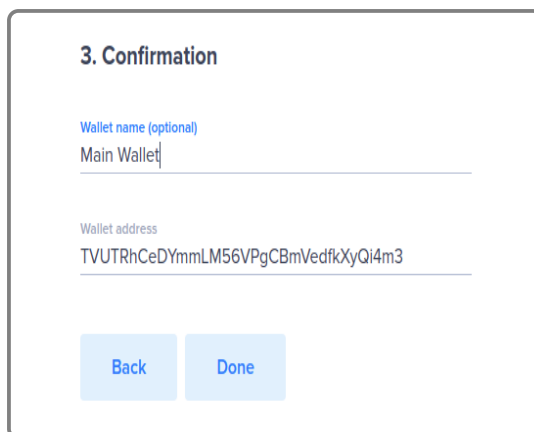
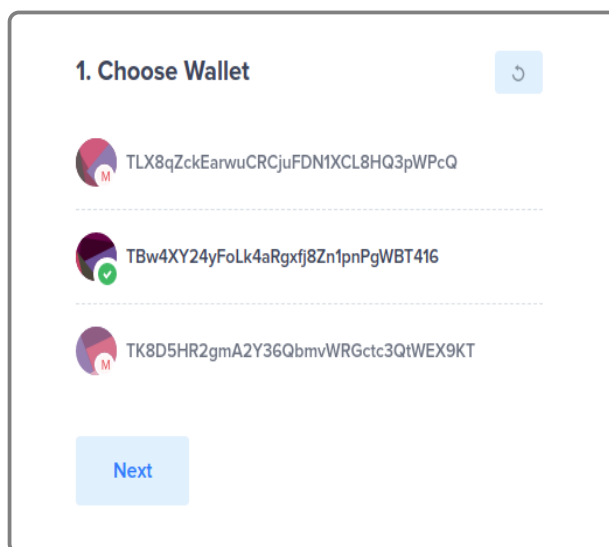


Figura A.13: Confirmación para crear el monedero

Para realizar transacciones y poder probar el algoritmo, es necesario crear otro monedero y realizar transacciones entre ambos. De esta forma, hay que acceder a la sección *My wallets* y pulsar *Create Wallet*. El primer paso es introducir el nombre del monedero, figura A.14.



1. Choose Wallet

TLX8qZckEarwuCRCjuFDN1XCL8HQ3pWPcQ

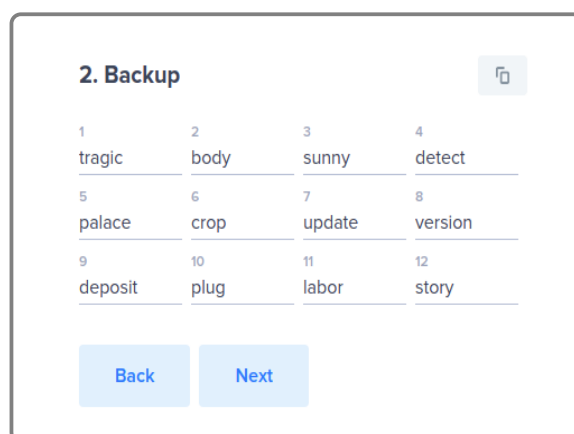
TBw4XY24yFoLk4aRgxfj8Zn1pnPgWBT416

TK8D5HR2gmA2Y36QbmVWRGctc3QtWEX9KT

Next

Figura A.14: Selección de la dirección del nuevo monedero

La figura A.15 muestra la *passphrase* del monedero. Esta *passphrase* hay que guardarla, tal y como se hizo con la *passphrase* de monedero importado, pues será necesaria para realizar transacciones posteriormente.



2. Backup

1 tragic	2 body	3 sunny	4 detect
5 palace	6 crop	7 update	8 version
9 deposit	10 plug	11 labor	12 story

Back Next

Figura A.15: *Passphrase* o clave privada del monedero

La verificación de la *passphrase* se hace introduciendo tres palabras, la número tres, seis y nueve, si se desea se puede realizar la verificación introduciendo todas las palabras, figura A.16.

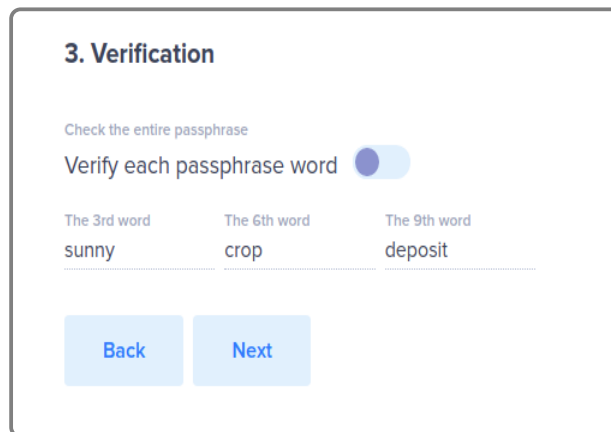


Figura A.16: Verificación de la *passphrase*

Si se desea se puede poner contraseña al monedero, en el ejemplo no es necesario, figura A.17.

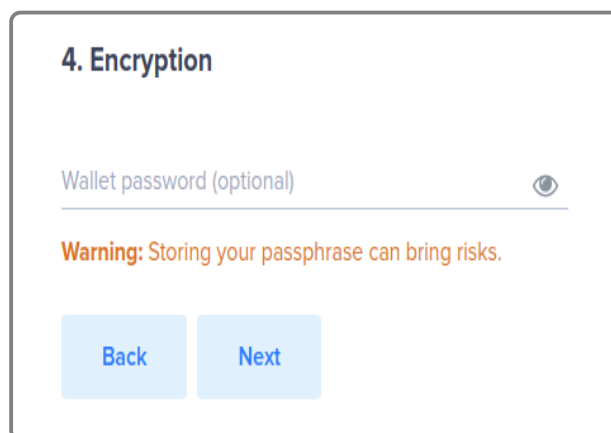
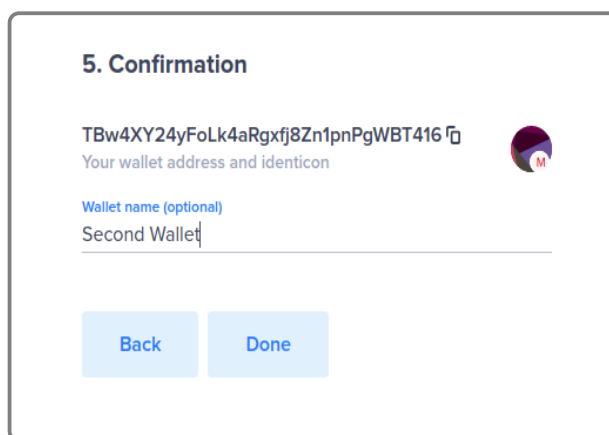



Figura A.17: Encriptación del monedero

Antes de confirmar la creación del monedero, hay que introducir el nombre del mismo, *Second Wallet*, para hacer referencia a que no tendrá ningún token hasta que no reciba una transferencia, figura A.18.



5. Confirmation

TBw4XY24yFoLk4aRgxfj8Zn1pnPgWBT416 

Your wallet address and identicon

Wallet name (optional)

Second Wallet

Back Done

Figura A.18: Confirmación para crear el segundo monedero

A la página principal, donde aparecen los monederos del usuario, se accede pulsando en el panel de la izquierda el símbolo del monedero, sección *My Wallets*, figura A.19. Se puede observar que el usuario en el que nos encontramos se llama “Victoria”, este contiene dos monederos *Main Wallet*, con 21.000.000,00MV y *Second Wallet*, con 0,00MV.

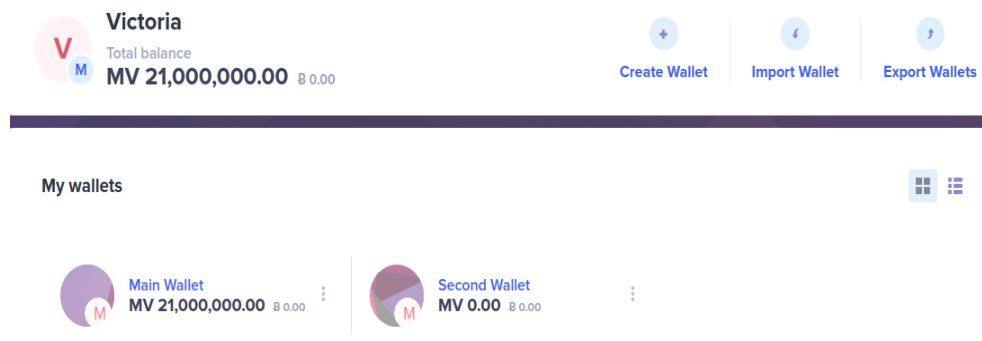


Figura A.19: Página principal con los monederos

Pulsando en cada uno de los monederos se accede a dicho monedero. Para realizar la primera transacción hay que pulsar en *Main Wallet* y posteriormente pulsar en la esquina superior derecha sent, figura A.20.

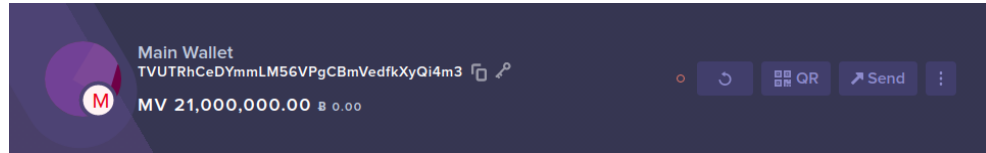


Figura A.20: Botón enviar

Entonces aparecerá un pantalla emergente para introducir los campos necesarios para realizar la transacción, figura A.21. Algunos de esos campos son el receptor de la transacción, en el ejemplo se ha enviado al otro monedero anteriormente creado, la cantidad de MVTokens a enviar, un mensaje para el receptor y la clave privada del monedero.

Transfer

Select a Single or Multiple Recipient Transaction

Single Multiple ?

Sender
Main Wallet TVUTRhCeDYmmLM56VPgCBmVedfkXyQi4m3

Recipient
TBw4XY24yFoLk4aRgxfj8Zn1pnPgWBT416

This wallet is known as "Second Wallet"

Amount
MV 8,592.00 B 0.00 Send All ☒

Smartbridge - Max 255
Sent 8592 MVToken

238/255 Remaining

Transaction fee
MV 0.10 Minimum **Average** Maximum Input Advanced

The network fee has been set to the static value of 0.1

Passphrase
.....

Next Load

Figura A.21: Realizar transferencia

Después hay que confirmar la transacción, además se pueden comprobar los datos de la transacción y si fuese necesario es pueden modificar, figura A.22. También existe la opción de almacenar los datos de la transacción por si en el futuro se desee volver a hacer la misma transacción y no tener que poner otra

vez todos los datos. Así en el paso anterior habrá que pulsar `load` y cargar el archivo con la información de la transacción.

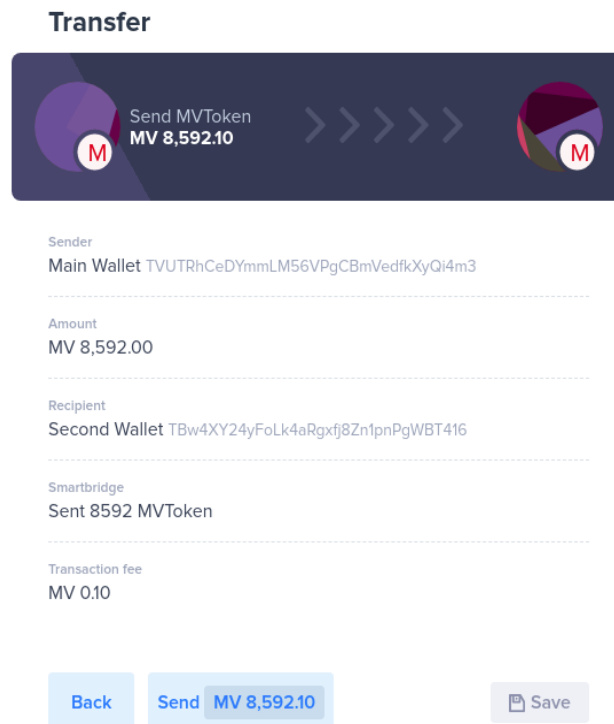


Figura A.22: Confirmación de la transferencia

En el monedero desde donde se ha realizado la transacción, se puede ver que se ha creado una nueva entrada en la lista de transacciones, figura A.23.

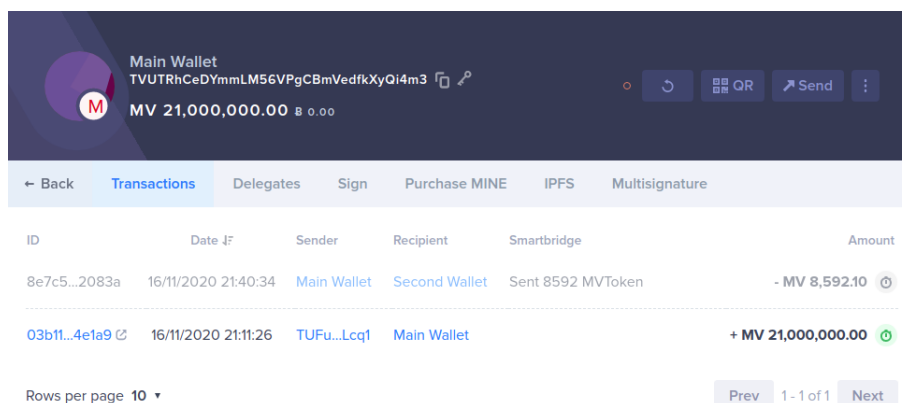



Figura A.23: Comprobación de que se ha enviado la transferencia

Se accede a los logs de la cadena de bloques, a través de la terminal con el comando `pm2 log`. En la misma terminal se obtendrá una vista similar a la siguiente figura A.24 donde en primer lugar se muestran las salidas de las funciones `firma` y `verificación de UOV`, con la firma y el resultado de la validación de la misma, respectivamente. Además del log de la propia *blockchain* indicando que el bloque se ha creado con éxito, en este caso se ha creado el bloque 63372384328353607. Ha de notarse que la firma del bloque es un vector de vectores con componentes en \mathbb{F}_2 y que lo que se envía es dicho vector en hexadecimal, por tanto la firma modificada siempre empezará por `5b5b` que corresponde a los caracteres `[]`.

Figura A.24: Logs de la cadena de bloques



Menu

Find a block, transaction, address or delegate

Q

👤

Latest transactions and blocks

Height: 7

Network: Testnet Local

Supply: 21,000.014 M

Latest transactions

Latest blocks

ID	Height	Timestamp	Transactions	Generated by	Total forged	Fees
15242...96468	8	16/11/2020 21:29:50	0	genesis_42	2 M	0 M
30801...99097	7	16/11/2020 21:29:42	0	genesis_36	2 M	0 M
63372...53607	6	16/11/2020 21:29:34	0	genesis_19	2 M	0 M
35430...47460	5	16/11/2020 21:29:26	0	genesis_40	2 M	0 M
93449...63789	4	16/11/2020 21:29:18	0	genesis_41	2 M	0 M
38137...13698	3	16/11/2020 21:29:10	0	genesis_21	2 M	0 M
63710...41921	2	16/11/2020 21:29:02	0	genesis_12	2 M	0 M
98561...96001	1	16/11/2020 21:11:26	52	TVUTR...Q4m3	0 M	0 M

Figura A.25: Vista del *explorer* con los últimos bloques creados

En la misma página se busca el bloque deseado por el ID 63372384328353607 para ver más información sobre él, figura A.26.

30801...99097	7	16/11/2020 21:29:42	0	genesis_36	2 M	0 M
63372...53607	6	16/11/2020 21:29:34	0	genesis_19	2 M	0 M
35430...47460	5	16/11/2020 21:29:26	0	genesis_40	2 M	0 M

Figura A.26: Bloques anterior y posterior del bloque con ID 63372384328353607



La figura A.27 muestra toda la información referente al bloque, como el número de transacciones que almacena, el número de confirmaciones por parte de los delegados, la hora a la que se ha generado el bloque o el nombre del delegado que lo ha generado. En la parte superior derecha hay dos botones Previous block y Next block, para acceder al bloque anterior y al bloque siguiente, respectivamente.

Block

Height: 10

Network: Testnet Local

Supply: 21,000,020 M

Block ID
63372384328353607 

[< Previous block](#)

[Next block >](#)

Transactions	0
Confirmations	4
Height	6
Reward	2 M
Fees	0 M
Total forged	2 M
Processed amount	0 M
Timestamp	16/11/2020 21:29:34
Generated by	genesis_19

Figura A.27: Información del bloque con ID 63372384328353607, visto en el *explorer*

Mientras la información que se puede ver del bloque desde la API es un poco más detallada, el ID del bloque, el ID del bloque anterior, el *hash* del bloque, que delegado ha generado el bloque tanto el nombre como la dirección, la firma del bloque, las confirmaciones y la hora a la que se ha generado, figura A.28. Se puede comprobar que la firma del bloque comienza por 5b5b que es la firma modificada.

[illegible]

Figura A.28: Información del bloque con ID 63372384328353607, visto en la API

Finalmente, se puede comprobar que coinciden los bloques mostrados en el *explorer*, figura A.26, con los de la API. Esto es, ver en la información del siguiente bloque 3080127205067599097 que se ha creado, que el identificador del bloque anterior es 63372384328353607, figura A.29.

```
{ "id": "3080127205067599097", "version": 0, "height": 7, "previous": "63372384328353607", "forged":
```

Figura A.29: Información del bloque anterior al bloque 63372384328353607, visto desde la API