Pasado, Presente y Futuro del *Blockchain*

Grado en Tecnologías de Telecomunicaciones mención en Telemática

10 de Junio de 2018

Estudiante: Carlos Ruiz de Mendoza
Director: Félix Freitag
Tutor: Joan Manuel Marquès Puig

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Common

FICHA DEL TRABAJO FINAL

Título del trabajo:	Pasado, Presente y Futuro del *Blockchain*
Nombre del autor:	Carlos Ruiz de Mendoza
Nombre del consultor:	Félix Freitag
Fecha de entrega (mm/aaaa):	06/2018
Área del Trabajo Final:	Aplicaciones y Sistemas Distribuidos
Titulación:	*Grado de Tecnologías de Telecomunicación*

Resumen del Trabajo (máximo 250 palabras):

“*Pasado, Presente y Futuro del Blockchain*” es un trabajo de investigación y desarrollo con el objetivo de justificar, a partir de los acontecimientos socioeconómicos y tecnológicos del pasado, el nacimiento de la tecnología de la cadena de bloques (*Blockchain)* y su proyección en el futuro.

La criptografía es uno de los pilares del *Blockchain* y por ello se tratan los sistemas de criptografía más relevantes, tanto de la época clásica como de la moderna, haciendo un especial énfasis en aquellos sistemas adoptados por la tecnología de la cadena de bloques.

En el presente trabajo se ilustra el desarrollo al completo de una aplicación descentralizada (DApp) para la plataforma Ethereum. El desarrollo de la DApp se implementa en lenguaje Solidity y se realizan las pruebas de funcionamiento, tanto en una red virtual como *online* en la red de pruebas de Ethereum Ropsten.

Previamente, se realiza una explicación al funcionamiento del *Blockchain* 1.0 de Bitcoin, precursor del sistema de la cadena de bloques. Una vez explicados los fundamentos de la red Bitcoin, se profundiza en el *Blockchain* 2.0 de Ethereum y sus contratos inteligentes que permiten asentar las bases para el desarrollo de una DApp.

Finalmente se extraen las conclusiones acerca del potencial de la tecnología *Blockchain* y si realmente se trata de la revolución de Internet o no y cuál es su proyección en el futuro.

Abstract (in English, 250 words or less):

"Pasado, Presente y Futuro del *Blockchain*" is a research and development work with the aim of justifying, from socio-economic and technological past events, the *Blockchain* technology birth and its future projection.

A major pillar of *Blockchain* is cryptography, therefore the most relevant systems are treated, from classical to modern cryptography, with special emphasis on those adopted by *Blockchain* technology.

The present study illustrates the development of a full decentralized application (DApp) for the Ethereum platform. A DApp development is implemented in Solidity language and tests performances are done, both, in virtual and online networks, in the Ethereum Ropsten test network.

Previously, the *Blockchain* 1.0 operation for Bitcoin, the *Blockchain* system precursor, explanation is given. Once the Bitcoin network basics have been explained, Ethereum's *Blockchain* 2.0 and its intelligent contracts concepts are deepened, which allow laying the foundations for a DApp development.

Finally, conclusions are drawn about the *Blockchain* technology potential and whether its or not the Internet revolution and what is its future projection.

Palabras clave (entre 4 y 8):

Bitcoin, Ethereum, Blockchain, Criptografía, DApp, Proof-of-Work, Solidity

Agradecimientos

Quiero agradecer la paciencia que han tenido mi mujer Bea y mis hijas Lucía y Clara por todos esos fines de semana que no han podido disfrutar de mí. También quiero agradecer a mis padres por todas las oportunidades que me han brindado y han permitido ser quien soy.

Por último agradecer a la comunidad de Ethereum y Bitcoin por la dedicación y el tiempo que invierten para documentar y mantener dichas plataformas.

Índice

Índice de ilustraciones **9**

1. Introducción **12**
Contexto y justificación del Trabajo 12
Objetivos del Trabajo 12
Enfoque y método seguido 13
Planificación del Trabajo 14
Breve sumario de productos obtenidos 15
Breve descripción de los otros capítulos de la memoria 16

2. Criptografía **17**
La importancia de la criptografía 17
Historia de la criptografía 19
Criptología clásica 19
Cifrado por transposición 19
Escítala 20
Cifrado por sustitución 21
Sustitución monoalfabética 21
Cifrado César 21
Tablero de Polibio 22
Sustitución homofónica 23
Sustitución polialfabética 23
Disco de Alberti 24
Cifrado de Vigenère 24
Máquina de rotores 26
Criptología moderna 28
Cifrado simétrico (Clave secreta) 28
Cifrado de flujo 28
Postulados de Golomb 29
Cifrado de bloque 30
Electronic Code Book (ECB) 31
Cipher Block Chaining (CBC) 31
Cipher Feedback (CFB) 32
Output Feedback (OFB) 33
Cifrado asimétrico (Clave pública) 34
Algoritmo RSA 35
Función Hash 37
Algoritmos Hash 38
Message-Digest Algorithm (MD) 38
Secure Hash Algorithm (SHA) 38

3. Blockchain 1.0 **39**
Génesis del Bitcoin 39
¿Qué es el Bitcoin? 41
La red Bitcoin 43

Tipos de nodos 43
Las transacciones 45
Mineros 47
Proof-of-Work 49

4. Blockchain 2.0 **54**
DApps (Decentralized applications) 54
¿Qué son las DApps? 54
Las criptomonedas 54
Ventajas y desventajas de una DApp 55
Ethereum 56
¿Qué es Ethereum? 56
Smart Contracts 57
Cuentas de usuario 58
Transacciones 58
Proof-of-Work 59
Forking 61

5. Desarrollo DApp **63**
Arquitectura DApp en Ethereum 63
Entorno de trabajo 64
Diseño del sistema 65
Introducción de la DApp 65
Arquitectura del sistema 66
Diseño del programa 67
Codificación 68
Pruebas 80
Red virtual 80
Verificación 87
Online en red Ropsten 87

6. Conclusiones **101**

Bibliografía **103**

Índice de ilustraciones

Imagen 1. Gráfico del método cascada empleado - Carlos Ruiz pág. 11

Imagen 2. Planificación temporal - Carlos Ruiz pág. 13

Imagen 3. Estructura de la historia de la criptografía - Carlos Ruiz pág. 16

Imagen 4. Escítala - http://themindmuseumbgc.blogspot.com.es/2015/09/classic-cryptography-making-and.html pág. 17

Imagen 5. Disco de Alberti - http://quhist.com/disco-alberti-criptografia/ pág. 21

Imagen 6. Funcionamiento máquina Enigma - https://www.tispain.com/2011/05/origen-e-historia-de-la-maquina-de.html pág. 24

Imagen 7. Diagrama de un criptosistema de flujo - http://criptosec.unizar.es/doc/tema_c3_criptosec.pdf pág. 25

Imagen 8. Modo ECB de cifrado - http://dlerch.blogspot.com.es/2007/07/modos-de-cifrado-ecb-cbc-ctr-ofb-y-cfb.html pág. 28

Imagen 9. Modo CBC de cifrado - Xifres de clau compartida: xifres de bloc (Jordi Herrera Joancomartí) P05/05024/00978 pág. 28

Imagen 10. Modo CFB de cifrado - Xifres de clau compartida: xifres de bloc (Jordi Herrera Joancomartí) P05/05024/00978 pág. 29

Imagen 11. Modo OFB de cifrado - Xifres de clau compartida: xifres de bloc (Jordi Herrera Joancomartí) P05/05024/00978 pág. 30

Imagen 12. Cifrado asimétrico - Carlos Ruiz pág. 31

Imagen 13. Circulación de bitcoins https://*Blockchain*.info/es/charts/total-bitcoins?timespan=all&showDataPoints=true pág. 39

Imagen 14. Distribución de los nodos de Bitcoin en el mundo - https://bitnodes.earn.com/ pág. 41

Imagen 15. Anatomía de una transacción de bitcoins - Carlos Ruiz pág. 42

Imagen 16. Proceso de una transacción de bitcoins - Carlos Ruiz pág. 43

Imagen 17. Información de una transacción en Bitcoin - https://*Blockchain*.info pág. 44

Imagen 18. Empresa china de minado de bloques de Bitcoin - https://www.trustnodes.com/2017/09/19/rumors-china-block-bitcoin-nodes-will-criminalize-crypto-mining pág. 44

Imagen 19. Encabezado de un bloque de Bitcoin - http://www.righto.com/2014/02/bitcoin-mining-hard-way-algorithms.html pág. 48

Imagen 20. Merkle Root - pág. 49

https://www.safaribooksonline.com/library/view/mastering-bitcoin/9781491902639/ch07.html

Imagen 21. Histórico de la dificultad en los bloques de Ethereum - https://etherscan.io/chart/difficulty pág. 58

Imagen 22. Forks de Ethereum - https://www.forks.net/list/Ethereum/ pág. 59

Imagen 23. Forks de Bitcoin - https://www.forks.net/list/Bitcoin/Forked/1/2017-01-01/2020-01-01 pág. 60

Imagen 24. Arquitectura de una DApp en Ethereum - Carlos Ruiz pág. 60

Imagen 25. Arquitectura del sistema - Carlos Ruiz pág. 63

Imagen 26. Estructura de datos de la DApp - Carlos Ruiz pág. 64

Imagen 27. Cuenta de usuarios en Solidity Remix - Carlos Ruiz pág. 77

Imagen 28. Cuenta de usuarios en Solidity Remix - Carlos Ruiz pág. 78

Imagen 29. Transacción compilación contrato en Solidity Remix pág. 78

Imagen 30. Funciones del contrato en Solidity Remix - Carlos Ruiz pág. 79

Imagen 31. Transferencia de ethers al contrato - Carlos Ruiz pág. 79

Imagen 32. Cuenta de usuario propietario - Carlos Ruiz pág. 80

Imagen 33. Transacción de transferencia de ethers al contrato - Carlos Ruiz pág. 80

Imagen 34. Cuenta de usuario empleado - Carlos Ruiz pág. 81

Imagen 35. Horas trabajadas por el empleado - Carlos Ruiz pág. 81

Imagen 36. Transacción del empleado al fichar - Carlos Ruiz pág. 82

Imagen 37. Saldo en la cuenta del empleado - Carlos Ruiz pág. 83

Imagen 38. Saldo en el contrato - Carlos Ruiz pág. 83

Imagen 39. Llamada a la función getEmpleado() - Carlos Ruiz pág.83

Imagen 40. Cuentas de usuarios en Metamask - Carlos Ruiz pág. 84

Imagen 41. Puesta en marcha de lite-server - Carlos Ruiz pág. 85

Imagen 42. Interfaz de usuario (UI) - Frontend - Carlos Ruiz pág.85

Imagen 43. Entorno de Solidity Remix - Carlos Ruiz pág. 86

Imagen 44. Sincronización de cuentas entre Metamask y Solidity Remix - Carlos Ruiz pág. 86

Imagen 45. Precio de gas - Carlos Ruiz pág. 87

Imagen 46. Enlace a la transacción en la red Ropsten - Carlos Ruiz pág. 87

Imagen 47. Transacción de subir el contrato al *Blockchain* - Carlos Ruiz pág. 88

Imagen 48. Confirmación de la transacción - Carlos Ruiz	pág. 88
Imagen 49. Transferencia de fondos al contrato y pago de gas - Carlos Ruiz	pág. 89
Imagen 50. transacciones del contrato - Carlos Ruiz	pág. 89
Imagen 51. ABI del contrato - Carlos Ruiz	pág. 90
Imagen 52. Ficha el empleado 1 - Carlos Ruiz	pág. 91
Imagen 53. La UI en proceso de carga de datos al contrato - Carlos Ruiz	pág. 91
Imagen 54. Enlace a la transacción desde la cuenta de Metamask - Carlos Ruiz	pág. 92
Imagen 55. Información de la transacción en Ropsten - Carlos Ruiz	pág. 92
Imagen 56. Transacción interna del contrato - Carlos Ruiz	pág. 93
Imagen 57. Información del bloque que agrega la transacción - Carlos Ruiz	pág. 93
Imagen 58. Resumen de datos en la UI - Carlos Ruiz	pág. 94
Imagen 58. Saldo resultante en la cuenta de usuario en Metamask - Carlos Ruiz	pág. 94
Imagen 59. Llamada a la función getBalance() del contrato - Carlos Ruiz	pág. 95
Imagen 60. Ficha el empleado 2 - Carlos Ruiz	pág. 95
Imagen 61. Fallo en la transacción del empleado 2	pág. 96
Imagen 62. Llamada a la función getEmpleado() - Carlos Ruiz	pág. 97
Imagen 63. Bloques de Bitcoin - http://guillermoboss.com/*Blockchain*-la-tecnologia-posible-bitcoin/	pág. 51

1. Introducción

Contexto y justificación del Trabajo

Actualmente la palabra *Blockchain*, representa un movimiento tecnológico que impulsa una revolución digital en la industria financiera, conocida como Fintech. El objetivo de esta revolución es desafiar las estructuras tradicionales de los mercados financieros que no han sabido o no han querido adaptarse a las necesidades del mercado después de la crisis financiera mundial de 2008. Muchos son los que apuestan que el Fintech cambiará de arriba a abajo el sector de las finanzas tradicionales dado el éxito y la implicación de la tecnología de Bitcoin.

No obstante, a pesar de la relevancia que está adquiriendo en el mundo financiero, resulta que sus capacidades son más atractivas en el *Internet-of-Things* (IoT). El rápido crecimiento del IoT puede adoptar en el *Blockchain* un sistema que permita el registro inalterable y seguimiento, ya no sólo de los dispositivos sino de cómo interactúan entre ellos. Las capacidades de la tecnología *Blockchain* pueden resultar en un crecimiento exponencial de las funciones de IoT de alto valor que deban ser protegidas de manera eficaz en cuanto se instaure el 5G, por ejemplo en el sector del *smart vehicle, smart city,* etc.

Otro campo que puede verse beneficiada de adoptar el *Blockchain* es el de las *Enterprise of Things* (EoT), disponer de pruebas de datos inalterados, puede ser una misión crítica y marcar la diferencia en situaciones de vida o muerte, o en operaciones de alto valor.

A toda innovación tecnológica le siguen sus detractores y para el *Blockchain* no podía ser menos. La falta de conocimiento o el conocimiento de su potencial, a desinteresados, hace que muchos cataloguen al *Blockchain* de burbuja tecnológica. Existe una gran opinión que sigue relacionando el *Blockchain* con la especulación de criptomonedas.

En el presente trabajo se pretende ofrecer una visión mucho más amplia del *Blockchain* y de sus capacidades. No sólo como tecnología adscrita a Bitcoin sino como una revolución de registro digital programable que puede facilitar la colaboración y el seguimiento de todo tipo de transacciones e interacciones, además de descentralizar la figura del intermediario.

Objetivos del Trabajo

Los objetivos que “Pasado, Presente y Futuro del *Blockchain*” pretende alcanzar son:

- Conocer y profundizar en los sistemas de criptografía clásicos y modernos más relevantes de la historia, relacionados con aquellos que han adoptado las plataformas de *Blockchain* más populares.

- Conocer los factores socioeconómicos que dieron lugar a la aparición del *Blockchain* 1.0 de Bitcoin.

- Conocer la arquitectura de Bitcoin y su funcionamiento.

- Conocer qué son las DApps (*Decentralized Applications*) y su arquitectura dentro de una plataforma como Ethereum.

- Conocer la arquitectura de Ethereum y el funcionamiento de sus *smart contracts*.

- Implementar una solución completa DApp, tanto el *frontend* como el *backend*, y ponerlo en funcionamiento *online*.

Enfoque y método seguido

El presente trabajo está dividido en cinco bloques principales que contienen:

1. Bloque 1: Criptografía
2. Bloque 2: *Blockchain* 1.0 (Bitcoin)
3. Bloque 3: *Blockchain* 2.0 (Ethereum)
4. Bloque 4: Desarrollo y puesta en marcha de una DApp
5. Bloque 5: Conclusiones y proyección futura del *Blockchain*

Uno de los objetivos del presente trabajo es el desarrollo de una DApp al completo, por lo que se ha adoptado un método de cascada para llevarla a cabo. Se ha aprovechado este método para desarrollar las partes de teoría e investigación de la siguiente manera:

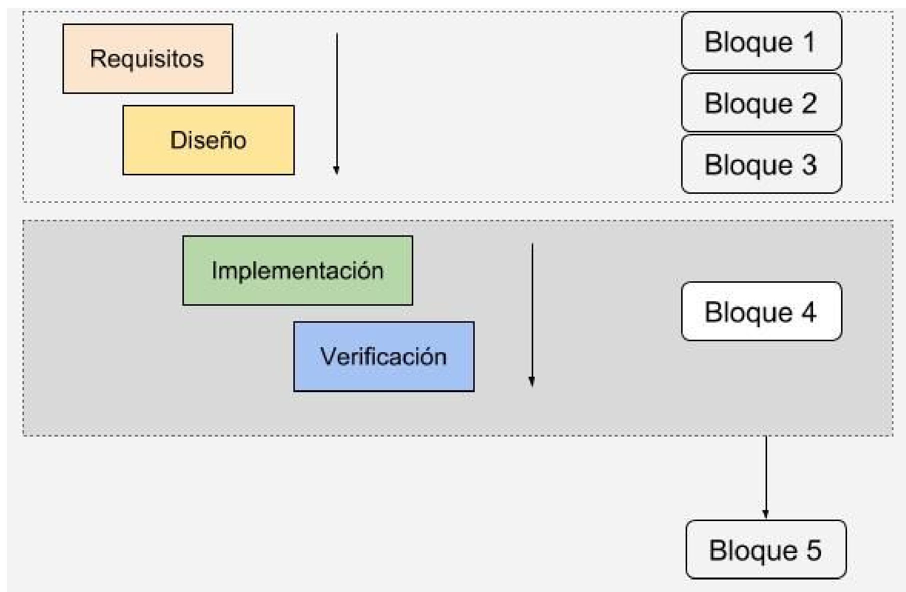
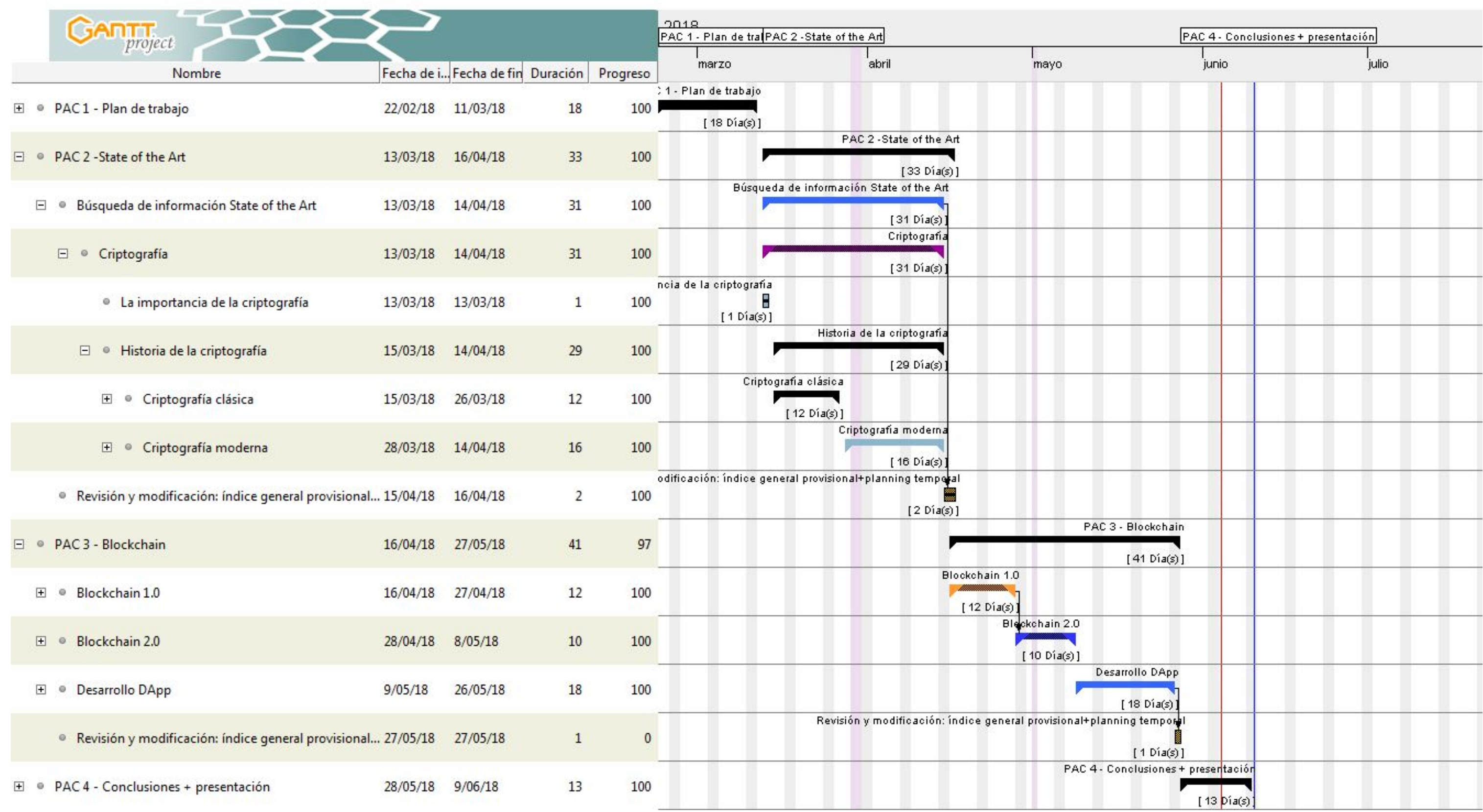


Imagen 1. Gráfico del método cascada empleado

El modelo en cascada proporciona las bases teóricas necesarias para poder adentrarse en el diseño e implementación de una DApp y asentar las conclusiones que confirmarán si la tecnología *Blockchain* es tan sólo una moda pasajera o realmente se trata de la revolución de Internet.

Para ello, antes de poder desarrollar la DApp es necesario profundizar en todos los aspectos técnicos de la plataforma donde se va a ejecutar dicha aplicación y para ello hay que profundizar previamente en la plataforma predecesora. Así mismo, la criptografía es uno de los pilares del *Blockchain* por lo que antes de nada es necesario entender el papel que juega y cómo lo juega en la tecnología *Blockchain*.

Planificación del Trabajo



Nombre	Fecha de i...	Fecha de fin	Duración	Progreso
PAC 1 - Plan de trabajo	22/02/18	11/03/18	18	100
PAC 2 -State of the Art	13/03/18	16/04/18	33	100
Búsqueda de información State of the Art	13/03/18	14/04/18	31	100
Criptografía	13/03/18	14/04/18	31	100
La importancia de la criptografía	13/03/18	13/03/18	1	100
Historia de la criptografía	15/03/18	14/04/18	29	100
Criptografía clásica	15/03/18	26/03/18	12	100
Criptografía moderna	28/03/18	14/04/18	16	100
Revisión y modificación: índice general provisional...	15/04/18	16/04/18	2	100
PAC 3 - Blockchain	16/04/18	27/05/18	41	97
Blockchain 1.0	16/04/18	27/04/18	12	100
Blockchain 2.0	28/04/18	8/05/18	10	100
Desarrollo DApp	9/05/18	26/05/18	18	100
Revisión y modificación: índice general provisional...	27/05/18	27/05/18	1	0
PAC 4 - Conclusiones + presentación	28/05/18	9/06/18	13	100

Plan Temporal TFG

04-jun-2018

Tarea

2

Nombre	Fecha de inicio	Fecha de fin	Duración	Progreso
PAC 1 - Plan de trabajo	22/02/18	11/03/18	18	100
Propuesta tema del TFG	22/02/18	28/02/18	7	100
Búsqueda información	22/02/18	27/02/18	6	100
Borrador contenido propuesta TFG	26/02/18	28/02/18	3	100
Plan de trabajo	1/03/18	11/03/18	11	100
Borrador descripción del TFG	5/03/18	7/03/18	3	100
PAC 2 -State of the Art	13/03/18	16/04/18	33	100
Búsqueda de información State of the Art	13/03/18	14/04/18	31	100
Criptografía	13/03/18	14/04/18	31	100
La importancia de la criptografía	13/03/18	13/03/18	1	100
Historia de la criptografía	15/03/18	14/04/18	29	100
Criptografía clásica	15/03/18	26/03/18	12	100
Cifrado por transposición	15/03/18	15/03/18	1	100
Cifrado por sustitución	17/03/18	26/03/18	10	100
Sustitución monoalfabética	17/03/18	19/03/18	3	100
Sustitución homofónica	21/03/18	21/03/18	1	100
Sustitución Polialfabética	22/03/18	26/03/18	5	100
Criptografía moderna	28/03/18	14/04/18	16	100

Nombre	Fecha de inicio	Fecha de fin	Duración	Progreso
Cifrado simétrico	28/03/18	3/04/18	5	100
Cifrado de flujo	28/03/18	28/03/18	1	100
Postulados de Golumb	28/03/18	28/03/18	1	100
Cifrado de bloque	31/03/18	3/04/18	4	100
Cifrado asimétrico	6/04/18	14/04/18	9	100
Funciones Hash	6/04/18	14/04/18	9	100
Revisión y modificación: índice general provisional+planning temporal	15/04/18	16/04/18	2	100
PAC 3 - Blockchain	16/04/18	27/05/18	41	97

Plan Temporal TFG 04-jun-2018

Tarea 3

Nombre	Fecha de inicio	Fecha de fin	Duración	Progreso
Blockchain 1.0	16/04/18	27/04/18	12	100
Génesis del Bitcoin	16/04/18	17/04/18	2	100
¿Qué es el Bitcoin?	17/04/18	19/04/18	3	100
La red Bitcoin	20/04/18	22/04/18	3	100
Minería y el consenso	23/04/18	25/04/18	3	100
Estructura de un bloque	26/04/18	26/04/18	1	100
Transacciones	27/04/18	27/04/18	1	100
Blockchain 2.0	28/04/18	8/05/18	10	100
¿Qué es una DApp?	28/04/18	29/04/18	2	100
¿Cómo funciona la red Etherum?	29/04/18	2/05/18	3	100
Qué es un smartcontract	3/05/18	4/05/18	2	100
Transacciones en Etherum	5/05/18	5/05/18	1	100
Consenso	6/05/18	8/05/18	3	100
Desarrollo DApp	9/05/18	26/05/18	18	100
Entorno de trabajo	9/05/18	11/05/18	3	100
Implementación y pruebas	12/05/18	26/05/18	15	100
Revisión y modificación: índice general provisional+planning temporal	27/05/18	27/05/18	1	0
PAC 4 - Conclusiones + presentación	28/05/18	9/06/18	13	100
Conclusiones de la DApp	28/05/18	29/05/18	2	100
Proyecciones futuras del Blockchain	2/06/18	3/06/18	2	100
Presentación virtual TFG	4/06/18	6/06/18	3	100
Revisión final TFG	7/06/18	9/06/18	3	100

Imagen 2. Planificación temporal

Breve sumario de productos obtenidos

El presente trabajo ha obtenido el diseño e implementación de una aplicación descentralizada para ser ejecutada en la plataforma de código abierto Ethereum. Se ha implementado la interfaz de usuario (*frontend*) y la parte responsable de procesar los datos (*backend*).

Breve descripción de los otros capítulos de la memoria

Capítulo 1: Introducción - En este capítulo se hace una breve explicación de la justificación para realizar el presente trabajo y el método que se ha empleado para completar todos los conceptos que aborda el trabajo. También se especifica el plan temporal que se ha llevado a cabo para completar el trabajo.

Capítulo 2: Criptografía - En este capítulo se recorre cada uno de los sistemas más relevantes a lo largo de la historia de la criptografía, desde los sistemas de la época clásica, anterior a los ordenadores, hasta la época moderna donde los sistemas ya están totalmente digitalizados. Se hace una explicación en profundidad de los sistemas criptográficos empleados en la tecnología *Blockchain*.

Capítulo 3: *Blockchain* 1.0 - En este capítulo se aborda los motivos socioeconómicos que motivaron a la aparición del primer sistema de *Blockchain*, el Bitcoin. Se explica el funcionamiento de la red Bitcoin, profundizando en aquellos conceptos que se heredan en el *Blockchain* 2.0.

Capítulo 4: *Blockchain* 2.0 - En este capítulo se introduce el concepto de aplicación descentralizada (DApp) y de las criptomonedas para a continuación profundizar en la tecnología *Blockchain* 2.0 de la plataforma Ethereum. Se explican cada uno de los conceptos imprescindibles, entre ellos los *smart contracts*, para poder desarrollar una DApp en Ethereum del capítulo 5.

Capítulo 5: Desarrollo DApp - En este capítulo se desarrolla una DApp al completo, desde el frontend hasta el backend. Una vez implementada la DApp se realizan pruebas de test tanto en red virtual como online en la red de pruebas de Ethereum, Ropsten.

Capítulo 6: Conclusiones - En este capítulo se llevan a cabo las conclusiones extraídas del presente trabajo y se hace una valoración de la proyección futura del *Blockchain*.

2. Criptografía

La importancia de la criptografía

Etimológicamente la palabra criptografía procede del griego “κρυπτός” (kryptós) 'oculto' y “γραφή” (graphḗ) 'escritura'.

Se entiende por criptografía la ciencia o arte de escribir con clave, dicho de otro modo, la criptografía se ocupa de cifrar textos, imágenes, audio, etc para garantizar la privacidad en el intercambio de la información entre dos partes y a su vez, poder descifrarlos de nuevo.

La información original que debe ser cifrada se denomina texto o mensaje en claro y el proceso por el cual se convierte el texto en claro en un criptograma (texto cifrado), se reconoce como encriptar o cifrar. La palabra cifra es de origen arábigo, utilizada para designar el número cero. Antiguamente en Europa, con el cambio del sistema de numeración romano al arábico, se desconocía la figura del número cero por lo que resultaba si más no, un tanto misterioso y de ahí que cifrado sea sinónimo de enigmático.

La criptografía es tan antigua como lo es la propia escritura y aunque los egipcios y mesopotámicos ya hicieron uso de métodos jeroglíficos, griegos y romanos fueron los primeros en aplicarse de lleno, culturas belicosas para las cuales, mantener confidencialidad absoluta en las comunicaciones, era la clave para el éxito militar. Con ellos nace un nuevo conflicto, el que se declara entre los criptógrafos y los criptoanalistas, aquellos que intentan romper las encriptaciones para desvelar los mensajes en claro. En el siglo VIII, el sabio árabe Al-Kindi (801-866/873) ideó una herramienta de descifrado mediante el análisis de frecuencias y parecía que iba a poner en jaque las esperanzas de los criptógrafos de la época. La respuesta de estos segundos fue la encriptación cifra polialfabética que tardó siglos en llegar. Por aquel entonces parecía una solución definitiva hasta que una versión más sofisticada de criptoanálisis, ideada por un genio inglés en la intimidad de su despacho, volvía a dar ventaja a los espías. Ya por aquel entonces el arma principal empleada por unos y otros fueron las matemáticas; la estadística, la aritmética modular y la teoría de números.

Este conflicto entre las partes de la criptología -conjunto que engloba la criptografía y el criptoanálisis- vivió un punto de inflexión con la aparición de las primeras máquinas de encriptación, a las que siguieron, poco tiempo después las que realizarían la operación inversa. De estas últimas surgió el primer ordenador, el *Colossus* creado por los británicos para descifrar los mensajes de Enigma, el perspicaz cifrado alemán. Con la eclosión de la computación, los códigos adquirieron un papel fundamental en la transmisión de la información además de las relativas al secreto o a la confidencialidad.

Hoy en día la criptografía permite que el mundo digital en el que vivimos pueda ejecutarse con cierta seguridad. Cada vez que se inicia una llamada por teléfono móvil, se realiza una compra con tarjeta de crédito, o se realiza una gestión telemática, la encriptación concede a esa transacción la confidencialidad y seguridad necesaria para evitar cualquier intento de fraude sobre la misma.

Toda encriptación sobre un mensaje en claro debe cumplir con éxito los siguientes principios:

- **Autenticación**: La autenticación es un proceso criptográfico que permite el uso de certificados, los denominados “ID digitales”, como solución para verificar la identidad de una persona a la hora de enviar o recibir información.

- **Integridad**: La integridad trata la protección de los datos para garantizar que no han sido manipulados durante la transmisión. Para ello, se hace uso de la firma digital que garantiza la asociación entre el ID digital de una persona o equipo informático y el mensaje transmitido.

- **Confidencialidad**: La encriptación de los mensajes otorga confidencialidad al contenido del mensaje cuando es transmitido. Sólo el destinatario y el remitente de dicho mensaje debe poder entender su contenido y debe ser ininteligible para cualquier otra persona no autorizada.

Existe la ficción popular que cualquier sistema puede ser “pirateado”, siempre y cuando exista un *hacker* con los conocimientos adecuados. La realidad es otra, un hacker debe encontrar una vulnerabilidad en un sistema para poder explotarla, como un fácil acceso a una sala de servidores, una contraseña fácil de adivinar o un puerto de red desprotegido para obtener acceso no autorizado.

Si bien es cierto que no se puede garantizar la invulnerabilidad de un sistema, desde los años 90, existen técnicas criptográficas que son simplemente inmunes a ser pirateadas (*hackeadas*). La criptografía por sí sola no puede ser corrompida para generar una firma digital falsificada, del mismo modo que las matemáticas no pueden ser mutables para obtener cinco como resultado de dos más dos. No obstante, tanto la criptografía como la matemática pueden ser utilizadas de manera incorrecta. Si un sistema que hace uso de la criptografía falla, es porque el diseñador del sistema lo implementó de forma incorrecta. No es porque la criptografía fallara, o porque alguien pirateó la criptografía. De la misma manera que no es culpa de las matemáticas si el cajero automático del banco, dispone de manera incorrecta la cantidad de dinero solicitada. Es importante la distinción de esto último ya que la tecnología de *Blockchain* 1.0 (Bitcoin), del que trataremos posteriormente en el presente trabajo final de grado, está directamente relacionada con la criptografía moderna establecida.

La criptografía no es una técnica reciente que acaba de ser descubierta. Todos los procedimientos criptográficos utilizados por *Blockchain* 1.0 y 2.0, han sido utilizados desde el comienzo de Internet, y son esenciales para muchos de los protocolos de Internet que se usan a diario.

Historia de la criptografía

Desde tiempos memorables, la criptografía ha sido utilizada en ámbitos militares, religiosos o comerciales. La historia de la criptografía se puede dividir en dos bloques, la criptografía clásica que se compone de los cifrados por transposición y los cifrados por sustitución, y la criptografía moderna, compuesta por sistemas simétricos, asimétricos e híbridos.

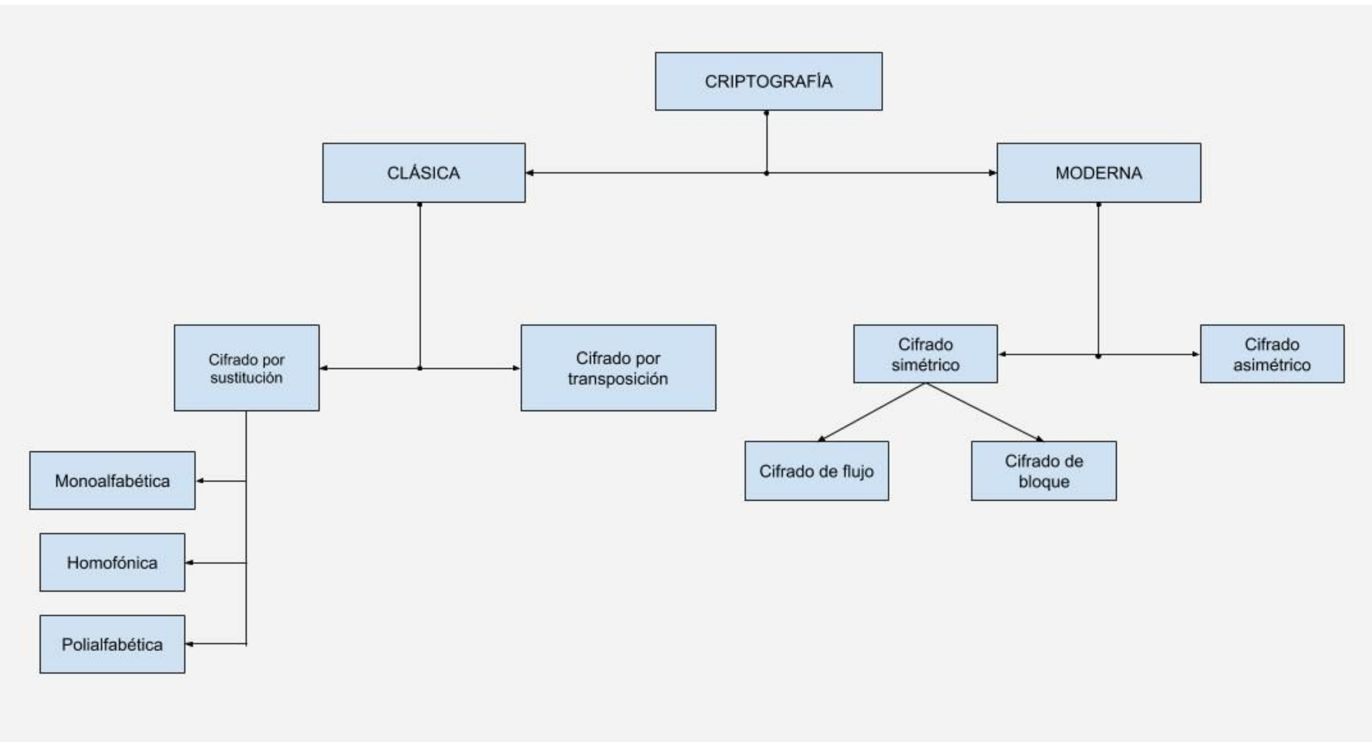


Imagen 3. Estructura de la historia de la criptografía

Criptología clásica

Se entiende por criptografía clásica aquellos sistemas de cifrado anteriores a la aparición de los ordenadores.

Cifrado por transposición

Los cifrados por transposición reordenan los caracteres del mensaje en función de ciertas reglas. Están basados en permutaciones de los caracteres del texto en claro con un periodo fijo d.

Sea:

$$Z_d \in [1,\ d]$$

$$f : Z_d \rightarrow Z_d$$

La clave de cifrado está definida por el par:

$$K = (d, f)$$

Por lo tanto, si:

$$d = 4$$

Y consideramos la permutación:

$$f \ tal\ que\ f(1) = 2,\ f(2) = 4, f(3) = 1, f(4) = 3$$

El mensaje en claro M , quedará cifrado, $E_k(M)$, de la siguiente manera:

$M =$	E	S	C	I	T	A	L	A
$E_k(M) =$	C	E	I	S	L	T	A	A

Escítala

El primer registro formal de sistema de cifrado por transposición data del siglo V a.C. Durante la guerra del Peloponeso entre atenienses y espartanos, éstos últimos hacían uso de lo que se conoce como escítala. Consistía en dos listones de madera del mismo diámetro y longitud que sólo los participantes de la comunicación conocían. Sobre una tira de cuero o papiro que se enrollaba alrededor del listón, se escribía el mensaje longitudinalmente al eje del listón. Cuando se desenrollaba la tira o papiro del listón, el mensaje se mostraba como un texto incoherente. Sólo el receptor del mensaje que poseía un listón del mismo diámetro y longitud, podía descifrar el mensaje volviendo a enrollar la tira de cuero o papiro alrededor del listón. En este tipo de sistema, la clave viene definida por el diámetro del listón.

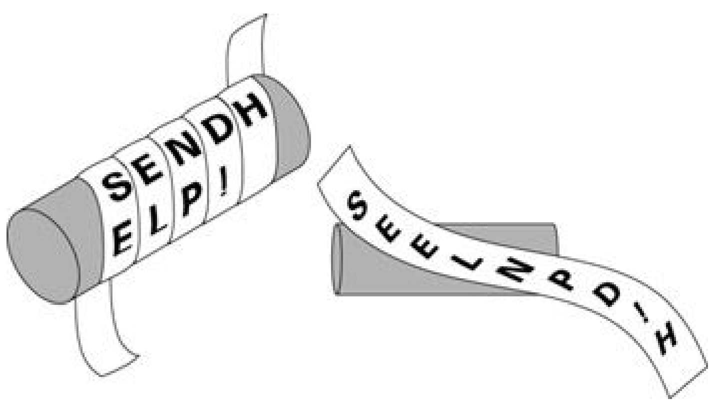


Imagen 4. Escítala

Cifrado por sustitución

El cifrado por sustitución, se basa en reemplazar los caracteres del texto en claro por otras letras o símbolos, como por ejemplo en la obra *Adventure of the Dancing Men* (Sir Arthur Conan Doyle, 1903), donde Sherlock Holmes rompe un cifrado de sustitución monoalfabético en el que los símbolos del texto cifrado son figuras de un ser humano en distintas posturas de baile.

Dentro de los cifrados de sustitución existen cuatros subgrupos; los cifrados de sustitución monoalfabética, homofónica, polialfabética y poligráfica.

Sustitución monoalfabética

Los cifrados por sustitución monoalfabética reemplazan los caracteres del texto en claro de un alfabeto α por la letra correspondiente de un alfabeto cifrado β.

$$\alpha = \{a_0,\ a_1,\ a_2,\ ...,\ a_{n-1}\}$$

$$\beta = \{f(a_0),\ f(a_1),\ f(a_2),\ ...,\ f(a_{n-1})\}$$

donde:

$$f : \alpha \rightarrow \beta$$

Por lo que un mensajes quedaría cifrado de la siguiente manera:

$$M = \{m_0,\ m_1,\ ...,\ m_{n-1}\}$$

$$E_k(M) = \{f(m_0),\ f(m_1),\ ...,\ f(m_{n-1})\}$$

Siendo f la clave de cifrado.

Cifrado César

Un ejemplo de cifrado por sustitución monoalfabética es el “cifrado César” del 100 a.C que sirvió al ejercito de Julio César para ocultar información durante sus campañas belicosas.

La técnica utilizada para cifrar un mensaje en el cifrado César era sustituir cada una de las letras del mensaje en claro por aquella que ocupaba tres posiciones más en el alfabeto $(k = 3)$, es decir, se desplaza el alfabeto tantas posiciones como indique k.

$\alpha =$	A	B	C	D	E	F	G	...	Z
$\beta =$	D	E	F	G	H	I	J	...	C

$M =$	R	E	T	I	R	A	R	L	A	S	T	R	O	P	A	S
$E_k(m_n) =$	U	H	W	L	U	D	U	O	D	V	W	U	R	S	D	V

El cifrado César es un ejemplo claro en la utilización de la aritmética modular para garantizar la confidencialidad de la información. Matemáticamente, podemos describir el método usado por Julio César como una función lineal del tipo:

$$E_k(x_n) = m_n + 3 \ (mod\ 27)$$

donde m_n indica la posición del carácter del texto en claro que ocupa en el alfabeto M, el 3 es la clave (k) que define el desplazamiento sobre el alfabeto M, siendo M un alfabeto de 27 caracteres. Finalmente, x_n indica la letra cifrada del alfabeto cifrado $E_k(m_n)$.

La operación inversa que permite descifrar el texto cifrado se obtiene a partir de la función modular:

$$M(m_n) = x_n - 3 \ (mod\ 27)$$

La vulnerabilidad del cifrado de César recae en la frecuencia relativa de aparición de los caracteres cifrados, ya que mantienen la misma estadística que las letras del alfabeto de los textos en claro. Por lo que mediante un criptoanálisis no muy complejo se puede llegar a adivinar la clave de desplazamiento (k). Por ejemplo, si se sabe que la letra A tiene un 12% en la distribución de frecuencias de letras en español para un texto literario, también lo tendrá la letra D en el texto cifrado, por lo que hallar la relación entre ambos caracteres no resulta muy complicado.

Tablero de Polibio

Otro ejemplo de cifrado por sustitución monoalfabética, aún más antiguo que el cifrado César, es el tablero de Polibio, del 200 a.C. Dicho sistema consta de un tablero de 6x6 en donde la primera fila y columna, codifican mediante coordenadas la letra del alfabeto del texto en claro.

	A	**B**	**C**	**D**	**E**
A	A	B	C	D	E
B	F	G	H	I	J
C	K	L	M	N/Ñ	O
D	P	Q	R	S	T
E	U/V	W	X	Y	Z

$M =$	T	O	P	S	E	C	R	E	T
$E_k(M) =$	DE	CE	DA	DD	AE	AC	DC	AE	DE

Sustitución homofónica

En los cifrados de sustitución monoalfabética se ha observado cierta debilidad y que son susceptibles de ser rotos por métodos estadísticos simples, debido a la relación que existe de la frecuencia relativa de aparición, entre los caracteres del alfabeto plano y del cifrado en mensajes encriptados.

El cifrado por sustitución homofónica consigue evitar dicha frecuencia relativa de aparición de los caracteres haciendo corresponder a cada letra del mensaje en claro una variedad de caracteres sustitutos, y el número de éstos es proporcional a la frecuencia relativa de aparición de la letra. Por ejemplo, la letra ‘a’ supone el 13% de frecuencia de aparición de todas las letras del alfabeto español, de manera que se asignarían 13 símbolos para cifrarla. Cada vez que apareciese una ‘a’ en el texto en claro, ésta sería reemplazada en el texto cifrado por uno de los trece símbolos elegidos al azar, de forma que al final del cifrado, cada símbolo tan sólo representa aproximadamente el 1% del texto codificado.

Un ejemplo de cifrado de sustitución homofónica lo encontramos en los papeles de Beale. A principios del siglo XIX, Thomas Jefferson Beale encabezando un grupo de 30 hombres, descubrió una mina con grandes cantidades de oro y plata. Presuntamente, Beale transportó el tesoro a Virginia y lo enterró en el condado de Bedford. Luego escribió tres mensajes encriptados, el primero indicando la ubicación del tesoro, el segundo en el que describe el contenido del tesoro, y el tercero donde indica quiénes eran los propietarios.

A día de hoy, sólo el segundo de los textos se ha conseguido descifrar. Beale enumeró las palabras que aparecen en la Declaración de Independencia de Estados Unidos y cifró cada letra del texto en claro sustituyendola por el número de alguna palabra que empezara por la letra que había que cifrar.

Sustitución polialfabética

Los cifrados por sustitución polialfabética, al igual que los homofónicos, buscan evitar las frecuencias relativas de aparición de los caracteres, no obstante, en los cifrados polialfabéticos la sustitución de cada carácter en el texto en claro, varía en función de la posición que ocupe éste en el propio texto. Se trata de un cifrado de sustitución con periodo d, dando como resultado una aplicación cíclica de n cifrados de sustitución monoalfabética.

Por lo tanto, sean $ς_1, ..., ς_d$ alfabetos del texto cifrado y $f_i : Ấ \rightarrow Ļ_i$ la aplicación del alfabeto del texto en claro $Ấ$ a la i-ésima alfabeto cifrado $Ļ_i$ para $1 \leq i \leq d$.

Sea:

$$M = \{m_1,\ m_d\ m_{d+1},\ ...\ m_{2d}\}$$

El cifrado se lleva a cabo repitiendo la secuencia de aplicaciones $f_1, ..., f_d$ cada d caracteres:

$$E_k(M) = \{f_1(m_1)\ ...\ f_d(m_d) f_1(m_{d+1}),\ ... f_d(m_{2d})\}$$

Disco de Alberti

El primer registro de cifrado por sustitución polialfabética que existe fue ideado en el siglo XV por León Bautista Alberti. Compuesto por un disco el cual permitía cifrar textos sin que hubiera una correspondencia única entre el alfabeto en claro y el alfabeto cifrado. Cada letra del texto en claro se corresponde con un carácter distinto en función de la clave utilizada.

El sistema se basaba en un disco de dos anillos concéntricos, el exterior fijo, compuesto por 24 casillas que contenían los 20 caracteres del latín (excluyendo la H, J, K, Ñ, U, W, Y) y los números 1, 2, 3, y 4 del alfabeto del texto en claro. El interno, móvil disponía de los 24 caracteres en latín además del signo '&' para cifrar los textos en claro. La clave del sistema viene por la situación inicial relativa de los dos anillos.

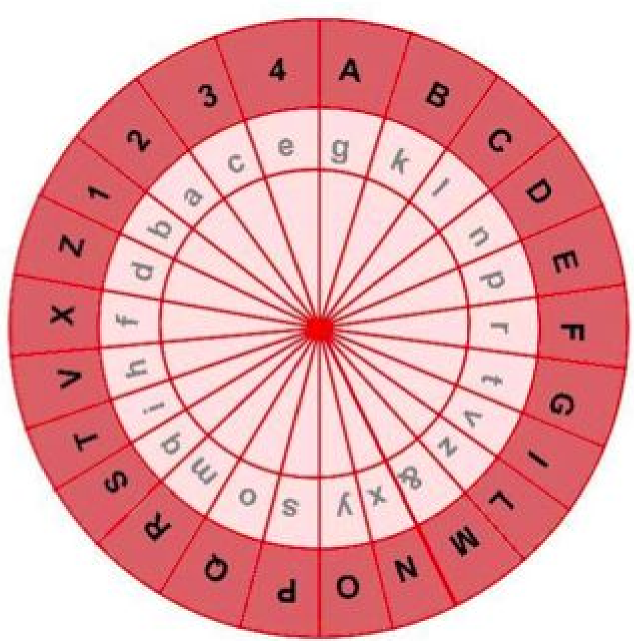


Imagen 5. Disco de Alberti

Cifrado de Vigenère

El cifrado de Vigenère data del siglo XVI, originalmente creado por Giovan Battista Belasso, a pesar de que más tarde dicho cifrado fue erróneamente atribuido a Blaise de Vigenère hasta el siglo XIX, fecha en la que se dejó de reconocerse como invención de Vigènere.

El cifrado de Vigenere es una sustitución periódica basada en alfabetos desplazados. La clave determina que alfabeto es el que se utiliza para cifrar cada letra del mensaje a partir de la operación modular:

Sea el alfabeto a utilizar:

$$K = \{k_0,\ k_1,\ ...,\ k_{d-1}\}$$

$$E_k(m_i) = (m_i + k_i)\ mod\ n$$

donde m_i es la posición en el alfabeto K, de la letra a cifrar y k_i corresponde con la posición de la letra de la palabra clave en el alfabeto K.

El proceso de cifrado es el siguiente:

1. Se elige una palabra clave que resulte fácil de recordar.

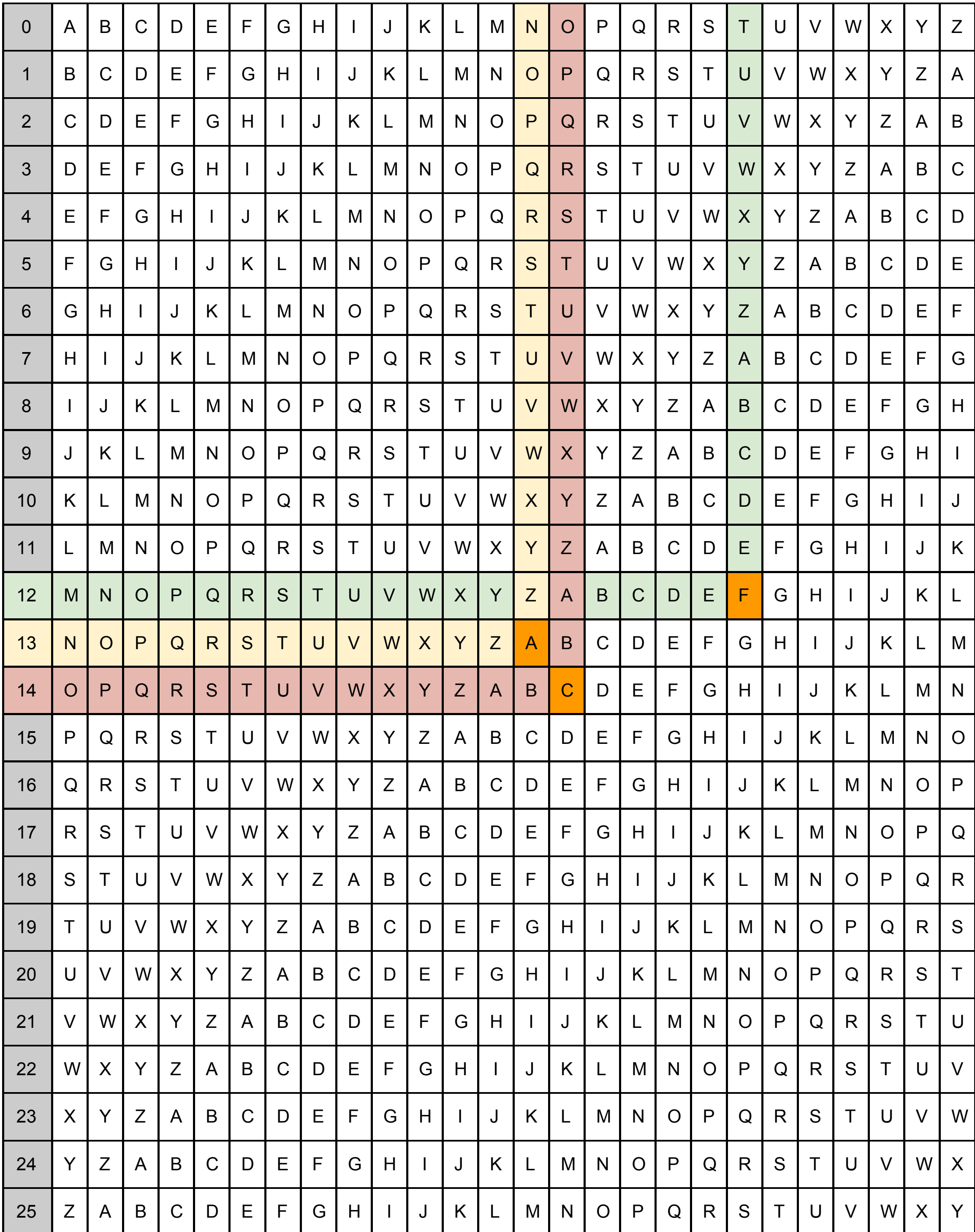
$$k = notice$$

2. Se escribe la palabra clave debajo del mensaje en claro tantas veces como sea necesario hasta cubrir todas las letras del mensaje.

$M =$	N	O	M	A	S	C	O	M	U	N	I	C	A	C	I	O	N	E	S
$K =$	n	o	t	i	c	e	n	o	t	i	c	e	n	o	t	i	c	e	n

3. Cada letra del texto a cifrar se codifica en base al alfabeto de la tabla con los alfabetos desplazados. Tomando la primera letra del mensaje en claro, la N, se observa que la que le corresponde en la palabra clave es la ‘n’. A continuación se busca en el “Tablero de Vigenère” el alfabeto que comienza con la letra N y la letra que le corresponde a la ‘n’, es decir, la letra A.

$M =$	N	O	M	A	S	C	O	M	U	N	I	C	A	C	I	O	N	E	S
$K =$	n	o	t	i	c	e	n	o	t	i	c	e	n	o	t	i	c	e	n
$E_k(M) =$	A	C	F	I	U	G	B	A	N	V	K	G	N	Q	B	W	P	I	F



0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Máquina de rotores

La más popular de las máquinas de rotores fue la creada por Arthur Scherbius (1878 - 1929) y Richard Ritter en 1918 llamada Enigma.

Una máquina de rotores es un dispositivo electromecánico compuesta por un teclado para introducir los mensajes en claro, la unidad modificadora que cifraba el mensaje en claro y un tablero de salida desde donde se mostraba el mensaje encriptado.

Cada letra que se tecleaba en el teclado de entrada, era transformada en otra letra por la unidad modificadora. Esta unidad era un rotor con un perímetro de 26 contactos eléctricos, uno por cada letra del alfabeto, tanto en la cara delantera como en la posterior del rotor. Cada vez

que se introducía una letra por el teclado, la unidad modificadora giraba un veintiseisavo de vuelta.

La debilidad que presentaba la primera versión de Enigma era que su sistema de rotor no era más que un cifrado de Vigenère y éste no era muy complicado de descifrar. La segunda versión de Enigma estaba compuesta por una segunda unidad modificadora enlazada con la primera, de forma que cuando el primer rotor completaba una vuelta entera, la segunda unidad modificadora giraba una posición. Con el segundo rotor se pasaba de tener una máquina de 26 posiciones a tener $26^2 = 676$ posiciones. Posteriormente hubo una modificación más en Enigma con la versión tres, era la inclusión de un tercer rotor más que giraba una posición cuando el segundo rotor completaba una vuelta, obteniendo de esta manera una máquina de 17576 posiciones.

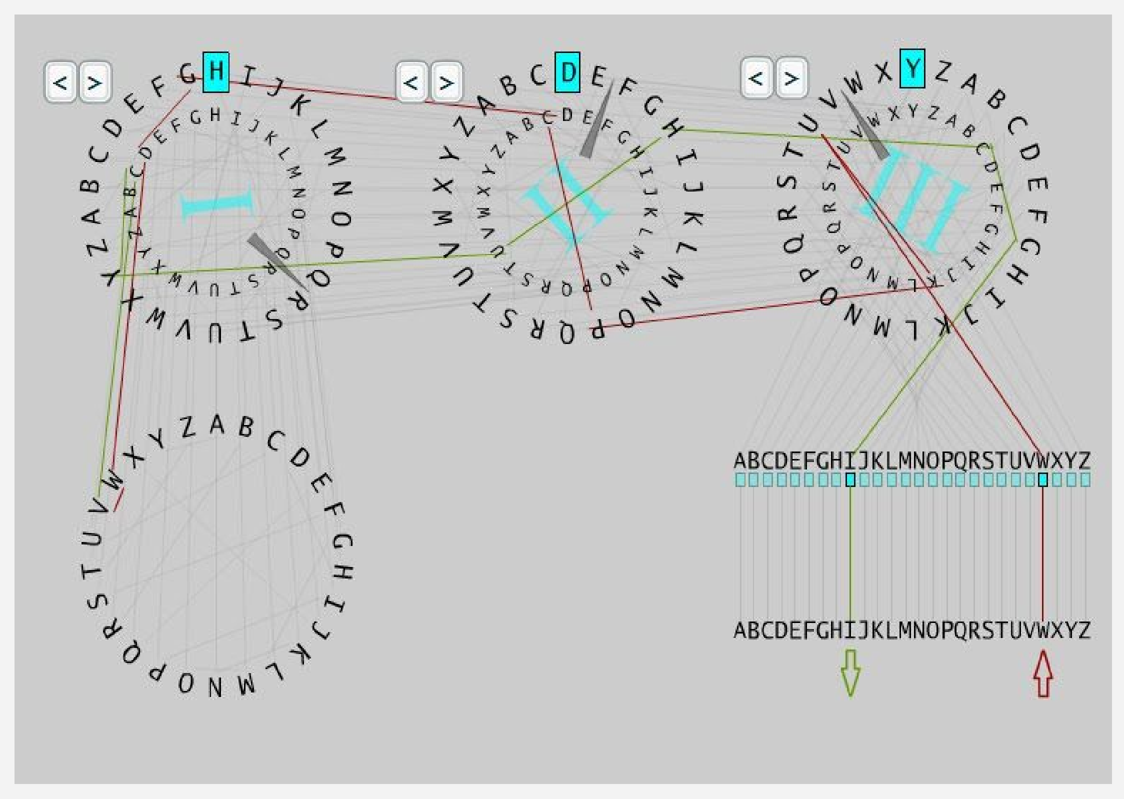


Imagen 6. Funcionamiento máquina Enigma

En agosto de 1940, en la Escuela Gubernamental de Códigos y Cifra de Bletchley Park, Alan Turing (1912 - 1954) y Gordon Welchman (1906 - 1985) consiguieron romper los cifrados de Enigma mediante un dispositivo electromecánico rotativo denominado la “Bombe”. La “Bombe” de Turing buscaba los ajustes de Enigma para un fragmento determinado de texto, del cual disponían de sus versiones en claro y en clave. Cuando se interceptaba un mensaje de Enigma, los criptoanalistas buscaban esos pequeños fragmento de texto encriptado cuyo significado en claro conocían.

Criptología moderna

La criptografía moderna, a diferencia de la clásica, sigue un enfoque científico y basa sus algoritmos criptográficos en suposiciones computacionales difíciles de romper por usuarios no autorizados. La criptografía moderna basa sus principios a partir de tres hechos significativos:

- La publicación de la teoría de la información de 1949 de Claude Elwood Shannon (30 de abril de 1916-24 de febrero de 2001).
- La aparición del estándar del sistema de cifrado DES (Data Encryption Standard) en 1974.
- La aplicación de funciones matemáticas de un solo sentido, denominado cifrado de llave pública en 1976 basada en el estudio de Whitfield Diffie y Martin Hellman.

Cifrado simétrico (Clave secreta)

Cifrado de flujo

Los cifrados de flujo, también llamados criptosistemas de clave compartida, son aquellos en los que el emisor y el receptor comparten una misma clave para cifrar y descifrar mensajes. Tanto el emisor como el receptor disponen de una misma clave (K), denominada semilla del generador, y de un mismo algoritmo determinístico, conocido como generador pseudoaleatorio. Al proporcionar la clave al generador pseudoaleatorio, éste genera una salida llamada secuencia cifrante (S).

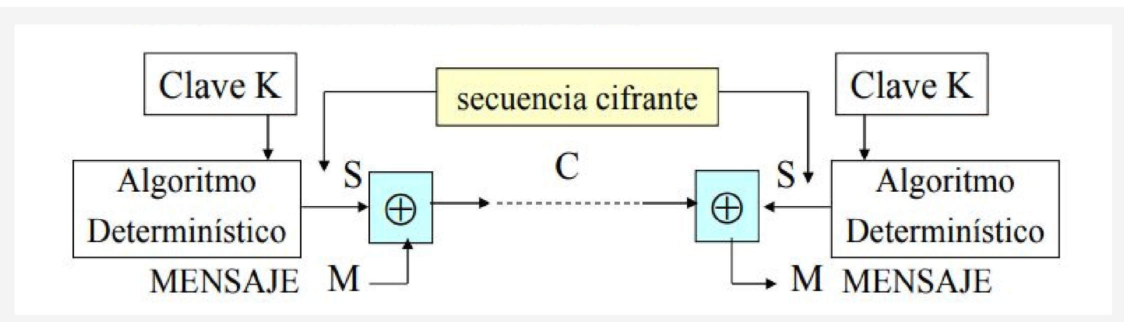


Imagen 7. Diagrama de un criptosistema de flujo

Para cifrar el mensaje, el emisor va sumando cada bit del mensaje en claro (M), con cada bit de la secuencia cifrante (S) mediante la operación xor. Cuando el receptor recibe el mensaje cifrado (C), suma bit a bit (xor) el mensaje que le llega con la secuencia cifrante (S) que resulta del algoritmo al proporcionarle la clave (K), de manera que el receptor obtiene el texto en claro (M) enviado por el emisor. De la imagen anterior se deduce que la velocidad de transmisión de datos entre el emisor y el receptor estará supeditada por el mínimo valor de velocidad de generación del mensaje y el mínimo valor de velocidad de generación de la secuencia cifrante.

Dado que el emisor y el receptor pueden cifrar y descifrar los mensajes con la misma semilla del generador (K), el generador pseudoaleatorio, es decir el algoritmo, debe ser determinístico; lo que impide garantizar la seguridad incondicional a partir de la secuencia cifrante. Por lo tanto, la secuencia cifrante, sin llegar a ser aleatoria, tendrá propiedades muy cercanas a las que tendría una secuencia cifrante totalmente aleatoria.

Si la secuencia no es aleatoria, significa que a partir de cierto tiempo se repite y es lo que se conoce como periodo. Podemos definir el periodo como:

Sea $\{s_i\}_{i \geq 0}$ una secuencia periódica, el periodo (p) es el entero más pequeño tal que $(S_{i+p} = S_i)\ \forall\ \ i \geq 0$.

Si se tiene en cuenta que el periodo se repite, al cabo de cierto tiempo es posible determinar toda la secuencia cifrante y por ende romper el criptosistema. Ergo es necesario que el periodo sea indistinguible de una secuencia totalmente aleatoria de la misma longitud. Por lo que la clave común a los agentes de la comunicación deberá ser tanto o más larga que el mensaje a cifrar. En la práctica se suele utilizar una semilla de entre 120 a 250 bits para generar períodos superiores a 10^{35}.

Cabe destacar que los criptosistemas de clave compartida basan la seguridad en el hecho de que la clave utilizada para cifrar y descifrar sólo es conocida por el emisor y el receptor. Aunque en el cifrado de flujo la clave no se utiliza directamente para cifrar, es importante que ésta no se haga pública, ya que el algoritmo determinístico es conocido y se podría obtener la secuencia cifrante a partir de éste y de la clave.

No sólo es importante que el periodo sea lo suficientemente largo sino que además la distribución de ceros y unos que forman la secuencia, estén distribuidos con cierta uniformidad. Es decir, que la secuencia cifrante tenga una grado elevado de impredecibilidad, con probabilidades cercanas al 0.5 tanto para los unos como para los ceros. La complejidad lineal es el concepto que mide el grado de impredecibilidad y mediante el algoritmo de Berlekamp-Massey podemos calcular dicha complejidad lineal.

Teniendo en cuenta todo lo anterior, para que una secuencia cifrante pueda ser considerada pseudoaleatoria, ésta, además debe cumplir los postulados de Golomb.

Postulados de Golomb

1. Dentro del periodo de una secuencia pseudoaleatoria, la cantidad de ceros y unos debe ser el mismo o debe diferir como máximo en una unidad. Es decir, debe ser:

    - $p/2$ si es par
    - $(p \pm 1)/2$ si es impar

2. El número total de ráfagas de longitud k en un período debe valer como mínimo $n/2^k$ siendo n el número total de ráfagas del periodo. Se define ráfaga como un conjunto de bits consecutivos iguales entre dos bits distintos.

3. La función de autocorrelación $AC(k)$ sólo puede tomar dos valores; 1 si k es múltiplo de p y otro valor constante si p no divide a k. La función de autocorrelación $AC(k)$ de una secuencia periódica se define como:

    $$AC(k) = \frac{A-D}{p}$$

    Siendo A y D, el número de coincidencias y de no coincidencias respectivamente de todo el periodo entre las sucesiones $\{S_i\}_{i \geq 0}$ y la misma sucesión desplazada k posiciones $\{S_{i+k}\}_{i \geq 0}$.

    - $A = |\{0 \leq i < p\ :\ S_i = S_{i+k}\}|$
    - $D = |\{0 \leq i < p\ :\ S_i \neq S_{i+k}\}|$

Aunque el análisis del algoritmo determinístico (generador pseudoaleatorio) queda fuera del alcance del presente trabajo, como referencia destacar que éstos pueden ser:

1. Generadores lineales que sólo ejecutan operaciones lineales sobre los datos de entrada.

2. Generadores no lineales, además de realizar operaciones lineales también realizan operaciones no lineales como por ejemplo permutaciones sobre los datos de entrada.

Uno de los algoritmos de cifrado de flujo más populares es el RC4 (Rivest Cipher 4), desarrollado por Ronald Rivest de la RSA Security. Se trata de un algoritmo de cifrado de flujo de clave compartida, hoy en día ya no se considera seguro y se debe considerar cuidadosamente su uso. El algoritmo de clave simétrica se utiliza de forma idéntica para el cifrado y el descifrado, de modo que la secuencia de datos de entrada simplemente se opera mediante el operador xor con la secuencia de clave generada. El algoritmo de cifrado RC4 es utilizado por estándares como IEEE 802.11 dentro de WEP (Wireless Encryption Protocol) usando claves de 40 y 128 bits.

Cifrado de bloque

Los cifrados de bloque pertenecen al grupo de criptosistemas de clave compartida, al igual que los cifrados de flujo, dado que la clave empleada por emisor y receptor es la misma y sirve tanto para cifrar como para descifrar los mensajes.

Los cifradores de bloque funcionan sin memoria y el mensaje cifrado sólo depende del texto en claro (M) y de la clave (K), de forma que dos textos en claro iguales serán cifrados del mismo modo cuando se utilice la misma clave. Esto último revela la posible facilidad a la hora de llevar a cabo un criptoanálisis de tipo estadístico.

Un cifrador de bloque ejecuta determinadas operaciones sobre un texto en claro (M) de longitud fija, para obtener un mensaje cifrado (C) a partir del uso de una clave (K) de manera que:

$$C = E_K(M)$$

Para el proceso inverso, descifrar un mensaje encriptado, se debe cumplir que:

$$M = D_K(C)$$

Los criptosistemas de bloque suelen realizar dos tipos de transformaciones, en la entrada y en la salida de datos. En cada una de estas transformaciones se lleva a cabo un número determinado de iteraciones en las que se ejecuta cierta función (f) no lineal para combinar los elementos que forman los bloques en que se divide el texto en claro y los elementos que forman la clave. De hecho, la clave se emplea para generar subclaves (K_i) que se emplean en cada una de las iteraciones anteriores.

Existen diversos modos de cifrado de bloque para gestionar cada uno de los bloques de datos en los que se dividen los textos en claro. A continuación repasamos los más relevantes.

Electronic Code Book (ECB)

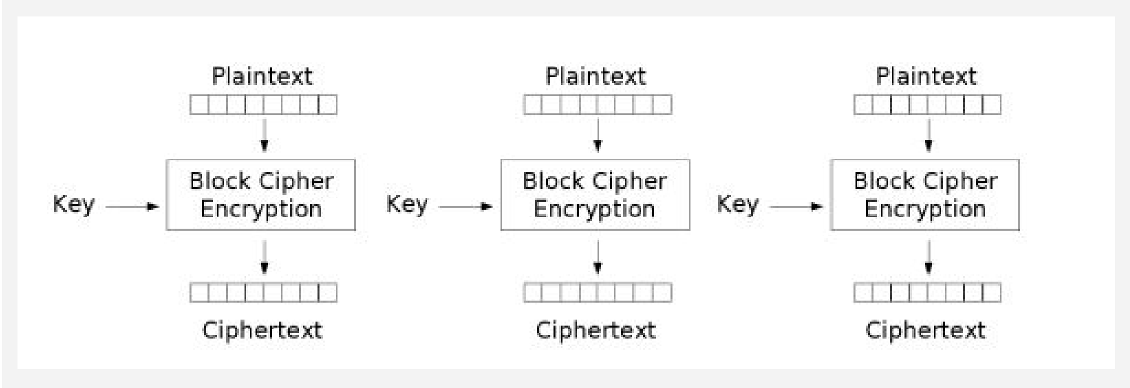


Imagen 8. Modo ECB de cifrado

Este modo, divide el texto en claro (M) en bloques de la misma longitud $M = \{M_1, M_2, ..., M_n\}$ y se cifra cada uno de ellos utilizando la misma clave (K). La longitud de cada bloque del texto en claro coincide con la longitud de cada bloque cifrado, en cierta medida a cada bloque de texto en claro le corresponde un bloque de texto cifrado que se vincula mediante la clave (K).

Este modo de cifrado presenta ciertas debilidades ya que:

a. Dado que cada bloque es cifrado de forma individual sin dependencia del anterior, un atacante podría interceptar la transmisión y borrar un bloque del mensaje cifrado. El receptor aún y así podría descifrar el resto de bloques sin llegar a percatarse de la eliminación de alguno de los bloques.

b. Un atacante podría insertar bloques de texto cifrado en la transmisión y el receptor no sería consciente de ello.

c. Es vulnerable al criptoanálisis estadístico dado que los bloques que contienen los mismos datos son cifrados de igual modo.

Cipher Block Chaining (CBC)

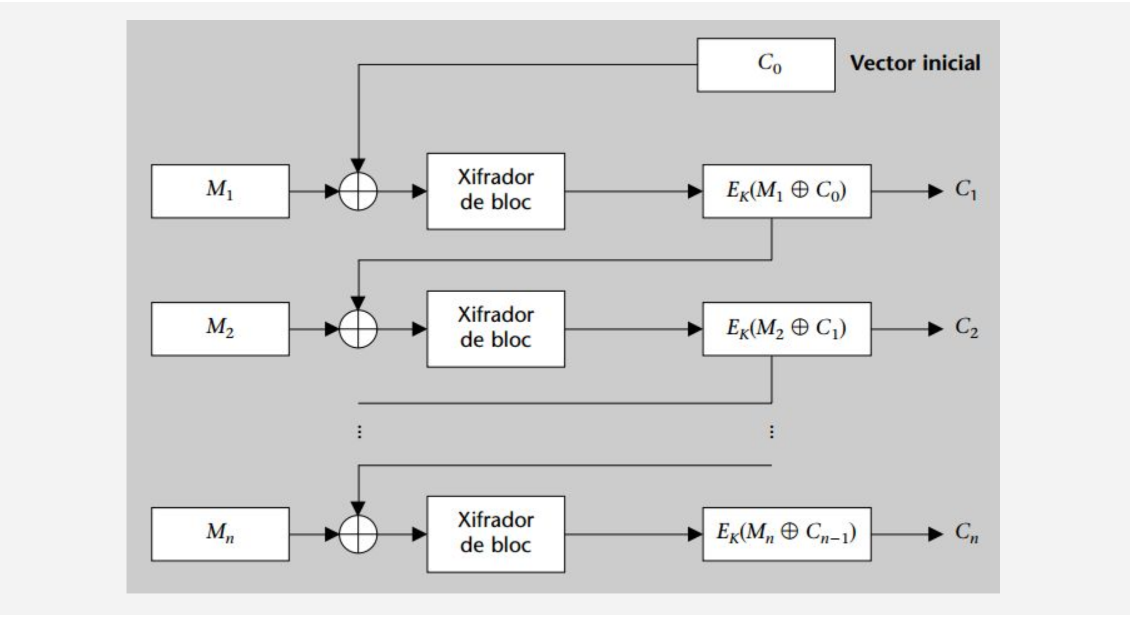


Imagen 9. Modo CBC de cifrado

En el modo de cifrado CBC existe una dependencia entre bloques mediante el encadenamiento de éstos. Cada bloque del texto en claro es combinado con el bloque de texto cifrado anterior, de manera que se solucionan muchas de las vulnerabilidades del modo ECB.

Sea (K) la clave, E la función de cifrado de bloque y M_i uno de los elementos del conjunto de bloques del texto en claro. Matemáticamente podemos expresar el bloque de texto cifrado como:

$$C_i = E_K(M_i \oplus C_{i-1})$$

Para cifrar el primero de los bloques es necesario disponer de un vector de inicialización (VI) que no tiene porque ser secreto pero sí aleatorio y no predecible.

La operación inversa de cifrado se llevaría a cabo de manera:

$$D_K(C_i) \oplus C_{i-1} = D_K(E_K(M_i \oplus C_{i-1})) \oplus C_{i-1}$$

siendo D la función para descifrar.

Cipher Feedback (CFB)

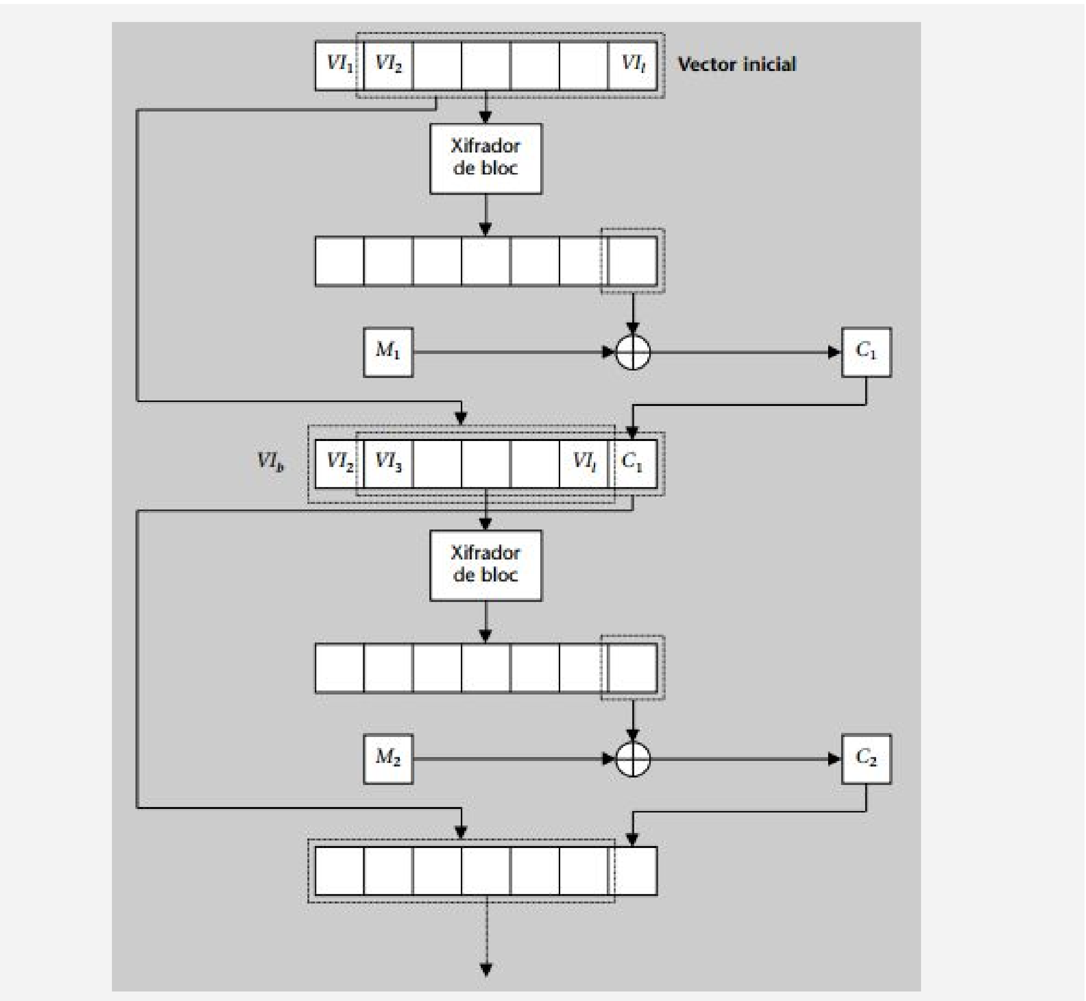


Imagen10. Modo CFB de cifrado

En el modo CFB se hace uso del cifrador de bloque no de manera directa para cifrar el texto en claro. El vector inicial se descompone en elementos donde cada elemento tiene una longitud de n bits, de manera que éstos se cifran con el cifrador de bloque y posteriormente se operan con el texto en claro mediante la operación xor.

Es decir, tenemos:

$$VI = VI_1,\ VI_2,\ ....,\ VI_i$$

que representa cada uno de los elementos del vector inicial. Al cifrar cada elemento del vector inicial nos queda:

$$E(VI) = E(VI_1),\ E(VI_2),\ ....,\ E(VI_i)$$

De manera que el primer bloque de texto en claro se cifrara con el último elemento del vector inicial cifrado de la siguiente manera:

$$C_1 = M_1 \oplus E(VI_i)$$

Una vez codificado el primero de los bloques de texto en claro, se genera un nuevo vector inicial desplazado bit a bit hacia la izquierda para dejar como último elemento el primero de los bloques codificados, de forma que nos quede:

$$VI_b = VI_2,\ VI_3,\ .....,\ VI_i,\ C_1$$

Para cifrar el segundo de los bloques de texto en claro se procede de la misma manera que con el primero, quedando de la siguiente manera:

$$C_2 = M_2 \oplus E(VI_b)_i$$

Donde el bloque C_2 pasará a formar parte de un nuevo vector inicial siendo su posición la última del vector y así sucesivamente hasta completar el cifrado del texto en claro.

Output Feedback (OFB)

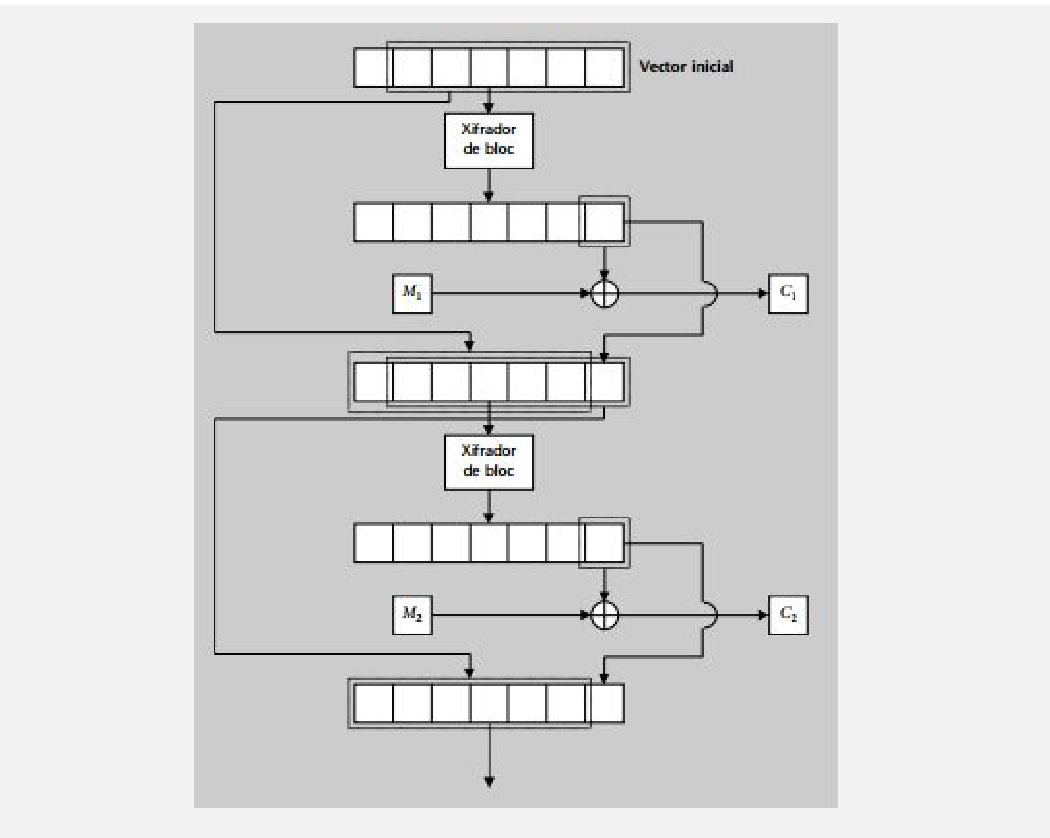


Imagen 11. Modo OFB de cifrado

El modo OFB es muy parecido al modo CFB pero con la particularidad de que el cifrador de bloque es utilizado a modo de generador pseudoaleatorio. En el modo OFB el vector inicial se realimenta directamente con el resultado del cifrado en bloque antes de realizar la operación bit a bit con el bloque de texto en claro.

Cifrado asimétrico (Clave pública)

El cifrado asimétrico o como también se le denomina, de clave pública, es un criptosistema que consta de dos claves, una pública, K_{pub} y la segunda privada K_{priv}, lo que garantiza la confidencialidad de los mensajes. La clave pública es de conocimiento público ya sea porque está publicada en algún directorio o porque el mismo propietario la distribuya, mientras que la segunda clave si debe ser absolutamente secreta y no debe ser compartida con nadie.

Ambas claves se generan al mismo tiempo mediante algoritmos de generación de claves, por lo que existe una relación matemática entre ambas claves lo que permite que lo que se cifra con una clave se pueda descifrar con la otra. No obstante, es imposible obtener la clave privada aún conociendo la clave pública o/y el texto cifrado.

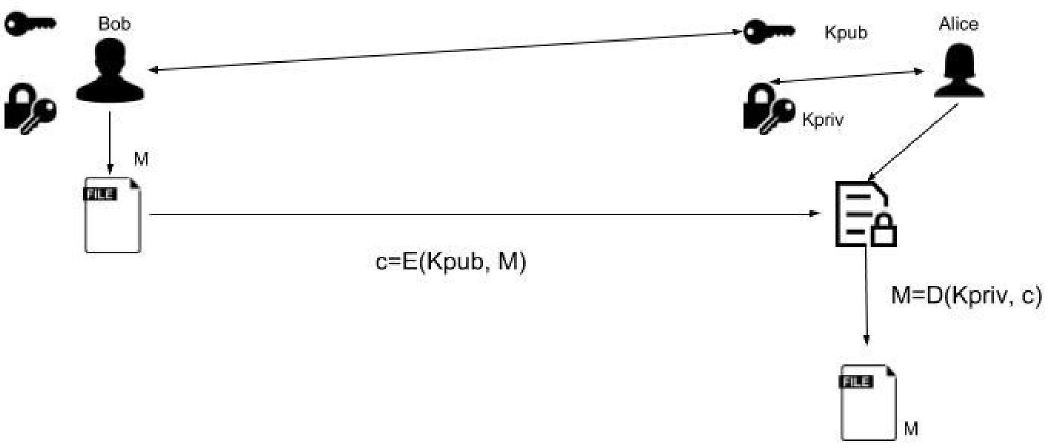


Imagen 12. Cifrado asimétrico

El funcionamiento del criptosistema asimétrico es el siguiente:

a. Bob obtiene la clave pública de Alice, K_{pub} (en ningún momento hace uso de sus propias claves).

b. A continuación Bob encripta mediante un algoritmo público de cifrado el mensaje en claro (M) que desea enviar con la llave pública de Alice, $c = E(K_{pub}, M)$.

c. Bob envía el mensaje cifrado, c, a Alice.

d. Una vez Alice recibe el mensaje encriptado, c, hace uso de su clave privada, K_{priv} y mediante un algoritmo de descifrado público lo desencripta obteniendo de nuevo el mensaje en claro, $M = D(K_{priv}, c)$.

Una de las debilidades potenciales de todos los algoritmos asimétricos es que la clave pública está disponible para los atacantes, lo que les permite generar mensajes cifrados. No hay que olvidarse que cualquiera puede cifrar un mensaje en claro con la clave pública del atacado, pero sólo el propietario de dicha clave podrá descifrar el mensaje con su clave privada. Por lo tanto el atacante, puede intentar descifrar un mensaje desconocido mediante el cifrado exhaustivo de secuencias de bits arbitrarios hasta que se logre una coincidencia con el mensaje objetivo. Este tipo de ataque conocido como un ataque de texto plano elegido, se ve

frustrado cuando se garantiza que todos los mensajes son más largos que la longitud de la clave, por lo que este tipo de ataque de fuerza bruta es menos factible que un ataque directo a la clave.

Algoritmo RSA

En 1978, Ron Rivest, Adi Shamir y Leonard Adleman del Massachusetts Institute of Technology (MIT) introdujeron un algoritmo criptográfico que reemplazaba el algoritmo de la Oficina Nacional de Estándares (NBS). El algoritmo RSA implementa un criptosistema de clave pública, así como firmas digitales.

El algoritmo RSA (Rivest, Shamir y Adleman) hace un uso intenso de operaciones aritméticas mediante la operación módulo. La operación módulo ($x \bmod y$) no es más que la operación que obtiene el resto de x al dividirlo por y.

En aritmética modular, al realizar las operaciones usuales de suma, multiplicación y exponenciación, el resultado de cada operación se reemplaza por el resto entero que queda cuando el resultado se divide por y. Tanto la suma como la multiplicación en aritmética modular se realiza de la siguiente manera:

- $[(x \bmod y) + (z \bmod y)] \bmod y = (x + z) \bmod y$
- $[(x \bmod y) - (z \bmod y)] \bmod y = (x - z) \bmod y$
- $[(x \bmod y) \cdot (z \bmod y)] \bmod y = (x \cdot z) \bmod y$

Supongamos que Bob quiere enviar un mensaje cifrado en RSA a Alice, para ello Alice debe realizar el siguiente procedimiento:

1. Generar un par de claves RSA:

   a. Debe elegir dos números primos grandes, p y q .Estos dos números deben ser lo suficientemente grandes como para que resulte complejo poder romper el cifrado RSA, se recomienda que el producto de p y q sea del orden de 1024 bits.

   > Ejemplo (Nota: los números elegidos son demasiado pequeños para garantizar la seguridad del cifrado y se eligen con el fin de simplificar lo cálculos):
   > $p = 5$
   > $q = 7$

   b. Se calcula $n = p \cdot q$ y $v = (p-1) \cdot (q-1)$.

   > $n = 35$
   > $v = 24$

   c. Se elige un número, e, menor que y que no tengan factores comunes, distintos del uno, con v.

   > $e = 5$ ya que el $m.c.d\ (Máximo\ común\ divisor) : (5,\ 24) = 1$

   d. Se busca un número, d tal que $(e \cdot d - 1)$ sea divisible, sin resto, por v, es decir:
   $$(e \cdot d) \bmod v = 1$$

   > $d = 29 \rightarrow \frac{(5 \cdot 29) - 1}{24} = 6$

e. Finalmente la clave pública de Alice es el par de números, $K_{pub} = (y,\ e)$ y la clave privada será, $K_{priv} = (y,\ d)$.

$K_{pub} = (35,\ 5)$
$K_{priv} = (35,\ 29)$

Una vez Alice ha generado su par de claves y ha distribuido de manera pública su clave pública a Bob el proceso de cifrado ocurre de la siguiente manera:

2. Cifrado de un mensaje en claro con el algoritmo RSA:

   a. Bob realiza una exponenciación sobre los bits que representan su mensaje en claro, m^e, para posteriormente calcular el resto entero de m^e dividido entre y, y así poder enviar a Alice el resultado en bits que obtiene Bob. Hay que recordar que en este escenario Bob emplea la clave pública de Alice compuesta por el par de números $K_{pub} = (y,\ e)$. Es decir:

$c = (m^e)\ mod\ n$

Texto en claro	m: representación numérica en el alfabeto	m^e	$c = (m^e)\ mod\ n$
h	8	32768	8
o	16	1048576	11
l	12	248832	17
a	1	1	1

   b. Cuando Alice recibe el mensaje cifrado, c, de Bob, Alice utilizando su propia clave privada, $K_{priv} = (y,\ d)$, calcula:

$$m = (c^d)\ mod\ n$$

c	c^d	$m = (c^d)\ mod$	Texto en claro
8	154742504910672534362390528	8	h
11	1586309297171491574414436704891	16	o
17	481968572106750915091411825223071697	12	l
1	1	1	a

El hecho que no exista algoritmo conocido que permita factorizar rápidamente un número, en este caso el valor público n - hay que recordar que se recomienda que el valor de n sea de al menos 1024 bits, lo que en decimal representa una cifra con más de 300 dígitos -, en los primos p y q, es la base de la seguridad en el algoritmo RSA. Si se supiera el valor de p y q, conociendo el valor público e, se podría calcular la clave secreta d.

Función Hash

Las firmas digitales son un requisito esencial para que un sistema digital sea seguro. Se necesitan para certificar la integridad de la información así como garantizar la autenticidad de la persona o entidad que envía dicha información.

Hasta ahora se ha visto como el cifrado asimétrico solucionaba el principal problema del cifrado simétrico, que es la necesidad de compartir una clave única para el cifrado y descifrado, lo que supone algo paradójico ya que si el canal por el que se comparte la clave es lo suficientemente seguro porqué no mandar los mensajes por el mismo canal.

El cifrado de clave pública garantiza la confidencialidad de los mensajes pero, cómo puede Alice estar segura que el mensaje que le ha enviado Bob no ha sido modificado por el camino o lo que es más importante, cómo sabe Alice que el mensaje realmente es de Bob y no de un atacante que se hace pasar por Bob. En el cifrado asimétrico se puede cifrar los mensajes con cualquiera de las dos llaves y sólo se podría garantizar la autenticidad de quién envía el mensaje si éste cifra el mensaje con su clave privada, pero dado que se podría descifrar con la clave pública, cualquiera podría interceptar el mensaje y descifrarlo por lo que perderíamos la confidencialidad del mensaje.

Para solventar los problemas planteados anteriormente, autenticidad e integridad, se dispone de un mecanismo conocido como firma digital. Este mecanismo emula el rol de una firma convencional, verificando que un mensaje o documento es una copia inalterada del producido por el que lo firma digitalmente. Pero para poder explicar el procedimiento de la firma digital antes hay que introducir el concepto de función resumen o también llamado función hash.

Una función hash, $h = H(M)$, es una función matemática que convierte el valor numérico de la entrada a la función en un valor numérico de salida comprimido, llamado hash o resumen . Mientras que la entrada a la función hash es de longitud arbitraria, la salida siempre es de longitud fija, actualmente en el orden de entre 160 y 512 bits.

Para que una función hash sea segura debe cumplir con las siguientes propiedades:

- Unidireccionalidad:

  Debe ser computacionalmente imposible invertir una función hash. Es decir, dado el resumen, $H(M)$ debería ser imposible hallar M. Esta propiedad protege contra el atacante que sólo tiene el valor hash e intenta encontrar el valor de la entrada.

- Resistente a colisión simple:

  Conociendo M debe ser computacionalmente imposible encontrar otro M' tal que $H(M) = H(M')$. Esto se conoce como resistencia débil a las colisiones, primera pre-imagen.

- Resistente a colisión fuerte:

  Esta propiedad significa que dada una entrada y su hash, debería ser prácticamente imposible encontrar una entrada distinta que produzca el mismo hash, $h(z) = h(y)$. Esta propiedad protege contra atacantes que aún disponiendo del valor de entrada a la función hash y su hash intenten sustituir el valor de entrada a la función hash para obtener el mismo hash.

Algoritmos Hash

Message-Digest Algorithm (MD)

La familia de algoritmos MD incluye las funciones hash MD2, MD4, MD5 y MD6. El MD5 de Ron Rivest, este algoritmo fue adoptado como el estándar de Internet [RFC 1321] y se trata de una función hash de 128 bits. El algoritmo calcula el hash de 128 bits en cuatro pasos:

1. Adición de bits - El mensaje es extendido de forma que se agrega un 1 seguido de tantos 0 como sean necesarios para que la longitud del mensaje satisfaga ciertas condiciones.

2. Longitud del mensaje - Un entero de 64 bits que representa la longitud, l, del mensaje antes de la adición de bits, se concatena al resultado del paso anterior. Si l es mayor que 2^{64}, entonces sólo los 64 bits de menor peso de l se utilizarán.

3. Inicialización del acumulador - Un búfer de cuatro palabras (A, B, C, D) se usa para calcular el resumen del mensaje. Cada una de las letras A, B, C, D representa un registro de 32 bits.

4. Procesado de bloques - los bloques de 16 palabras del mensaje se procesan (mutilados) en cuatro rondas.

Los hash MD5 han sido ampliamente utilizados en el mundo del software para proporcionar seguridad sobre la integridad del archivo transferido. Por ejemplo, los servidores de archivos suelen proporcionar una suma de comprobación MD5 pre calculada para los archivos, para que el usuario pueda comparar la suma de comprobación del archivo descargado. No obstante en 2004 el algoritmo fue comprometido y es por ello que no se recomienda su uso.

Secure Hash Algorithm (SHA)

El SHA-1 (Secure Hash Algorithm), basado en el MD4 de Rivest, es otro de los algoritmos populares de hoy en día, sobre todo para procesos de validación de identidad, es decir, para certificar que un documento, una conexión o un recurso es genuino. Produce un hash de 160 bits y aunque es sustancialmente más lento que MD5, su resumen de 160 bits ofrece una mayor seguridad contra ataques de fuerza bruta y ataques de cumpleaños.

Desde el año 2005 se considera, teóricamente, un algoritmo vulnerable y así lo ha demostrado recientemente (Febrero 2017) Google junto a la universidad CWI (Centrum Wiskunde & Informatica) de Amsterdam al conseguir colisionar dos hash de SHA-1, presentando dos archivos pdf (Portable Document Format) con distinto contenido y obteniendo el mismo resumen para ambos.

Las versiones posteriores al SHA-1 como el SHA-2 (2001) incluye seis funciones hash con diferentes tamaños de resumen: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 y SHA-512/256. La distintas longitudes de los hash, mayores que las de SHA-1, ofrecen más seguridad pero implican costos adicionales para la generación, almacenamiento y comunicación de firmas digitales y MAC (*Message Authentication code*).

El SHA-3 fue publicado por el National Institute of Standards and Technology (NIST) en 2015, se trata de una alternativa al SHA-0, SHA-1 y MD5.

3. *Blockchain* 1.0

Génesis del Bitcoin

El 15 de septiembre de 2008 cuando la empresa de servicios financieros Lehman and Brothers se declaró en bancarrota, Estados Unidos (EE.UU) entró financieramente en colapso al igual que el resto de economías mundiales, una vez más el viejo dicho se cumplía; "*Cuando EE.UU estornuda, el resto del mundo se resfría*".

La crisis financiera del 2008 se empezó a gestar en el 2002 con la burbuja inmobiliaria. El crecimiento en el poder adquisitivo de las familias americanas y las declaraciones del presidente de Estados Unidos (EE.UU) George W. Bush en las que solicitaba colaboración del sector financiero privado para que las familias americanas pudieran realizar el sueño americano de adquirir una vivienda, provocaron que la reserva federal (FED) disminuyera la tasa de interés del 6% al 1% en el transcurso de pocos meses, abaratando el dinero hasta tal punto que los miles de bancos de Estados Unidos empezaron a iniciar campañas hipotecarias muy agresivas para captar clientes potenciales.

El dinero resultaba tan barato que los bancos no tardaron en crear nuevos productos hipotecarios para llegar a aquellos ciudadanos que por regla general les hubiera sido imposible adquirir una hipoteca por su poca capacidad de endeudamiento. Nacían los créditos *subprime*, una modalidad de crédito cuya principal característica es que el riesgo de que se produzca un impago es superior a la media del resto de modalidades de préstamos hipotecarios. Es decir, los requisitos para la obtención de estos créditos son menos exigentes con la solvencia o la capacidad de endeudamiento de las personas a las que van dirigidos.

A pesar de las advertencias de muchos economistas, la facilidad para obtener un crédito hipotecario provocó que muchas de las familias americanas que accedían a los préstamos hipotecarios *subprime* empezaran a especular con los precios de los inmuebles, con la consecuente inflación en los precios de la vivienda. Las hipotecas estaban avaladas por el valor de las propiedades hipotecadas por lo que no era extraño que con el aumento del precio de la viviendas, debido a la especulación, aumentara de manera considerable el número de hipotecas *subprime*.

Ante tal demanda de préstamos hipotecarios, los bancos necesitaban más capital para poder seguir alimentando sus beneficios y para ello acudieron a los mercados internacionales, es justo en este momento cuando la burbuja inmobiliaria americana empieza a contaminar al resto de las economías mundiales.

Las Normas de Basilea limitan la cantidad de capital respecto al activo de un banco que puede tener, lo que deriva en que aquellas entidades financieras que piden dinero y ofrecen créditos baje su porcentaje de capital sobre su activo y por ende acaban no cumpliendo dichas normas. Así que no tardaron en buscar nuevas estrategias para cumplir las Normas de Basilea y éstas pasaban por emitir nuevas titulizaciones hipotecarias, las llamadas obligaciones hipotecarias garantizadas (CMO), es decir, se realizaban paquetes financieros que contenían hipotecas *subprime* para venderlos en los mercados financieros ofreciendo una alta rentabilidad. Para vender dichas titulizaciones hipotecarias y de esta manera limpiar el balance de las entidades financieras se crearon unas entidades filiales - en realidad seguían siendo el mismo banco - llamados *conduits*. Estas nuevas entidades se financiaban con créditos de otros bancos y para vender sus participaciones diversificaban el riesgo de sus titulaciones con el objetivo de mejorar

su rating, de manera que mezclaban hipotecas *subprime* con otros préstamos hipotecarios no tóxicos. Las hipotecas *subprime* se ofrecían a las familias americanas a un bajo interés durante los primeros años para luego elevarse con cada revisión anual y así incrementar el tipo de interés de manera contundente, esta era la justificación para vender estos nuevos paquetes financieros en los mercados internacionales, presumían de ser muy rentables a los pocos años. Así es como los bancos de inversión americanos empezaron a repartir sus paquetes de activos por todo el mundo.

La especulación inmobiliaria y el alza de precios no tardó en empezar a tambalear los cimientos de la economía americana, así que en el 2003 la FED empezó a subir la tasa de interés, del 1% ese mismo año al 3% en el 2005 y hasta 5.5% en el 2006. La subida de la tasa de interés del 2005 en las hipotecas *subprime* provocó un desplome en el valor de las propiedades del mercado inmobiliario. Muchos clientes se encontraron con propiedades que ya no valían lo que habían pagado por ellas y que además no podían hacer frente a las obligaciones de pago de sus hipotecas porque se habían encarecido con la subida de la tasa de interés. Así que para muchas familias americanas resultaba más fácil dejar que les embargaran la propiedad inmobiliaria que hacer frente al pago de una hipoteca por un valor mucho mayor al de la propiedad. Esto provocó grandes pérdidas a los bancos generando un colapso en los portafolios de los bancos de inversión a nivel mundial, debido a que habían estado comprando paquetes de las hipotecas tóxicas *subprime*.

El resultado a nivel internacional fue que las cajas y bancos de otros países empezaron a sufrir las consecuencias del mercado hipotecario americano. Habían invertido en los *conduits*, habían comprado participaciones de las titulizaciones '*subprime*' debido a su alta rentabilidad cuando el sistema funcionaba y ahora esos paquetes financieros se habían desplomado en valor porque la gente no hacía frente al pago de las hipotecas.

Con la caída del banco Lehman and Brothers el gobierno americano salió al rescate de otros bancos para evitar un desastre financiero mayor y lo mismo ocurrió en el resto del mundo. Como consecuencia de estos rescates, las economías se ralentizaron, la gente consumía menos, se gastaba menos, las empresas y particulares no encontraban financiación ante la desconfianza de los bancos y por lo tanto se consumía menos a todos los niveles.

Es entonces cuando los gobiernos actúan para poder estimular las economías de sus respectivos países. Para ello los gobiernos piden a los bancos centrales de cada país o zona que impriman más dinero. De esta manera, se intenta estimular la economía haciendo que más dinero esté disponible para el público, ya sean empresas o particulares, es decir para financiación. Esto conlleva a que la moneda de cada país pierda valor cada vez que se imprime más dinero, por lo que la riqueza del país disminuye.

Por ejemplo, supongamos que un país tiene 100 unidades de moneda en su economía, si se es poseedor de una unidad de moneda, representa que se tiene el 1% de la economía del país. Pero si el gobierno del país decide imprimir 100 unidades más, el 1% ahora pasa a valer el 0,5% ya que en el mercado hay 200 unidades de moneda. La riqueza personal se ha visto devaluada por la decisión de una autoridad central.

La crisis financiera provocó el cierre de muchos bancos y muchos de sus clientes que en un principio habían confiado en ese banco vieron cómo habían perdido sus ahorros. Así que de nuevo, los gobiernos imprimían más dinero para garantizar los ahorros de esos clientes, lo que a su vez reducía el valor del dinero que ya circulaba en el país.

Pocos después de la caída de Lehman and Brothers, el 31 de octubre del 2008, en un foro de criptografía, un desconocido Satoshi Nakamoto publicaba un artículo titulado "*Bitcoin – A Peer to Peer Electronic Cash System*". En ese documento, Nakamoto explicaba el funcionamiento de un nuevo sistema de dinero electrónico completamente descentralizado que no dependía de una autoridad central para la emisión o liquidación de la moneda ni para la validación de las transacciones. El artículo empezaba con toda una declaración de intenciones:

"*Una forma de dinero en efectivo electrónico puramente peer-to-peer debería permitir enviar pagos online directamente entre las partes y sin pasar a través de una institución financiera. Las firmas digitales son parte de la solución, pero los beneficios principales desaparecen si un tercero de confianza sigue siendo imprescindible para prevenir el doble gasto. Proponemos una solución para el problema del doble gasto usando una red peer-to-peer. La red sella las transacciones en el tiempo en una cadena continua de Proof-of-Work basada en hash, estableciendo un registro que no se puede modificar sin rehacer la Proof-of-Work. La cadena más larga no solo sirve de prueba efectiva de la secuencia de eventos, sino que también demuestra que procede del conjunto de CPU más potente. Mientras la mayoría de la potencia CPU esté controlada por nodos que no cooperen para atacar la propia red, se generará la cadena más larga y se aventajará a los atacantes. La red en sí misma precisa de una estructura mínima. Los mensajes se transmiten en base a "mejor esfuerzo", y los nodos pueden abandonar la red y regresar a ella a voluntad, aceptando la cadena Proof-of-Work más larga como prueba de lo que ha sucedido durante su ausencia*".

El 12 de enero de 2009, Satoshi Nakamoto realizó la primera transacción de bitcoins por una cantidad total de 10 bitcoins a Hal Finney, primer desarrollador para Bitcoin después de Nakamoto. A día de hoy, Satoshi Nakamoto sigue siendo un alias y la identidad de la persona o personas detrás de dicho alias sigue siendo desconocida. Lo que sí se sabe es que ni Nakamoto ni nadie más ejerce control sobre el sistema Bitcoin y que opera sobre la base de principios matemáticos totalmente transparentes.

¿Qué es el Bitcoin?

Bitcoin es el término que engloba la tecnología que forma la base de un ecosistema de dinero virtual en una red descentralizada *peer-to-peer*. Las unidades de intercambio de dicho sistema se llaman bitcoins y son utilizados para almacenar y transferir valor económico entre los usuarios de la red Bitcoin, el bitcoin es lo que se conoce como criptomoneda.

El protocolo de Bitcoin es un procedimiento de código abierto que los usuarios utilizan a través de Internet por medio de ordenadores, *tablets*, *smartphones*, etc para poder realizar transacciones con bitcoins. Los usuarios de Bitcoin pueden transferir bitcoins a otros usuarios o entidades para adquirir bienes, enviar dinero o simplemente para adquirir otras criptomonedas. Los bitcoins pueden comprarse, venderse o cambiarse por otras criptomonedas en los exchanges -un exchange es un sitio web que permite operar entre distintas criptomonedas y dinero fiduciario-, del mismo modo que se hace en el mercado de valores con las acciones.

Los bitcoins no existen físicamente, son virtuales y son intrínsecos a cada transacción que se realiza entre dos usuarios. Para llevar a cabo una transacción, cada usuario de bitcoin debe poseer unas claves que le permiten identificarse como el propietario de sus bitcoins y así poder realizar transacciones con ellos. Dichas claves se suelen guardar en lo que se denomina *wallet* (monedero).

El protocolo de Bitcoin implica la construcción del *Blockchain* de Bitcoin, éste puede considerarse como un libro mayor de contabilidad digital que realiza un seguimiento y registro de todas las transacciones de los usuarios de Bitcoin. El *Blockchain* de Bitcoin es una base de datos que registra cada uno de los movimientos de bitcoins que se llevan a cabo entre usuarios. Cada transacción registrada en el *Blockchain* de Bitcoin debe ser criptográficamente verificada para garantizar que los usuarios que intentan enviar bitcoins son realmente los propietarios de los bitcoins a transferir. Las transacciones de los usuarios de Bitcoin se agrupan cada cierto número para crear un bloque. Cada uno de estos bloques de transacciones se encadena al bloque previo para formar una cadena de bloques dependientes (*Blockchain*) mediante funciones hash. Para agregar cada bloque a la cadena, se hace uso de la criptografía para que los ordenadores que construyen el *Blockchain* colaboren en un sistema matemático de confianza automatizado, es lo que se conoce como *Proof-of-Work* (PoW).

La red Bitcoin es una red descentralizada, no existe un servidor central que gestione el valor de cada bitcoin así como tampoco una autoridad central que regule el bitcoin o las transacciones que se efectúan. De hecho, cualquier usuario puede comprobar cada una de las transacciones que se llevan a cabo en un momento determinado. Incluso, puede formar parte de usuarios verificadores de transacciones para agregar bloques al *Blockchain*. La red de ordenadores de Bitcoin que pueden verificar las transacciones y colaborar en la construcción del *Blockchain,* hace que la base de datos Bitcoin sea totalmente inmutable, dado que es imposible eliminar cualquier dato de un bloque ya verificado y agregado al *Blockchain.*

Los bitcoins se crean mediante un proceso llamado ‘minería’ el cual implica buscar una solución matemática al PoW y cuando éste se resuelve, el minero tiene el privilegio de agregar el bloque al *Blockchain* y ser recompensado con bitcoins. Cualquier usuario de Bitcoin puede ser minero, para ello deberá instalar el software adecuado y hacer uso de la potencia de procesamiento de su ordenador con el objetivo de intentar encontrar la solución matemática que plantea el PoW. La figura del minero descentraliza las funciones de emisión de divisas de un banco central convencional y reemplaza la necesidad de la existencia de cualquier tipo de banco central.

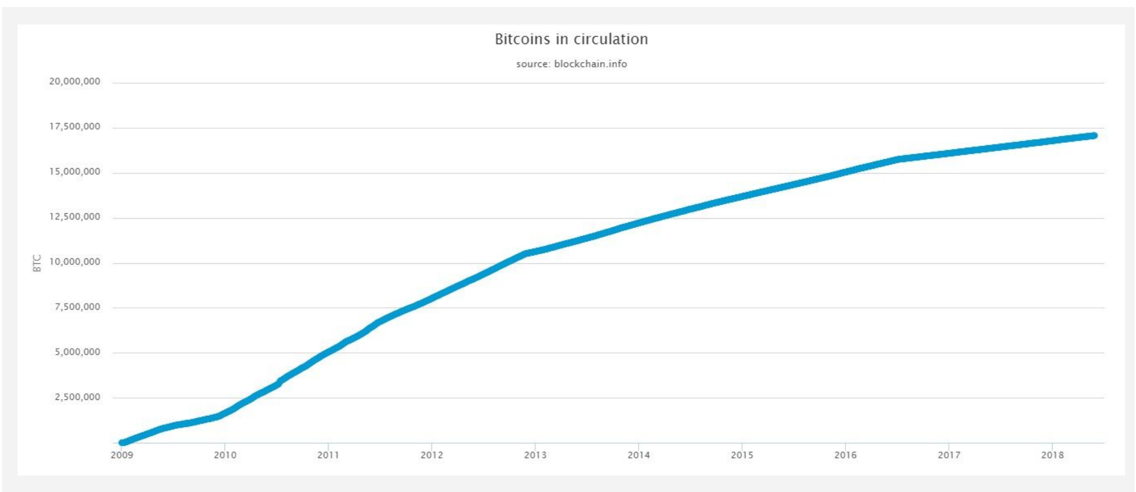


Imagen 13. Circulación de bitcoins

El protocolo de Bitcoin incluye algoritmos que regulan la dificultad del problema que se plantea a los mineros para resolver, el PoW se ajusta de manera dinámica de forma que algún minero pueda encontrar la solución independientemente del número de mineros o del número de ordenadores procesando una solución, con el fin de que se pueda resolver cada 10 minutos. El sistema también reduce a la mitad el número de bitcoins que se entregan a cada minero cada 4 años y limita el número total de bitcoins en la red a un total de 21 millones de bitcoins (BTC).

Esta automatización sigue una curva de crecimiento totalmente predecible poniendo fecha límite en el 2140, año en el que se habrán minado el total de monedas disponibles del sistema. Dado que el número de monedas está fijado por el propio sistema, bitcoin es una moneda deflacionaria debido a la tasa decreciente de emisión.

La red Bitcoin

El sistema Bitcoin, a diferencia de los sistemas bancarios, se basa en la confianza descentralizada. En lugar de una autoridad reguladora, en Bitcoin, la confianza se logra a partir de las interacciones entre los participantes de la red Bitcoin.

Bitcoin funciona sobre una arquitectura de red distribuida *Peer-to-Peer* (P2P) descentralizada, compuesta por miles de ordenadores ejecutando el protocolo Bitcoin, donde cada ordenador actúa de nodo en la red. Todos los nodos de la red comparten la carga de trabajo para proporcionar servicios a la red, es decir, cada nodo contiene la misma información que el resto de nodos sin la necesidad de responder a un nodo central. Por lo tanto, todos los nodos son iguales, no hay una jerarquía de nodos aunque sí existen distintos roles.

Los nodos que componen la red Bitcoin se interconectan en una red plana sin topología de forma que todas las transacciones se transmiten a todos los nodos. Cada nodo añade las transacciones recibidas en un bloque para poder publicarlo en el *Blockchain* en caso de resolver el *Proof-of-Work.* Todos los nodos aceptarán el bloque que se desea publicar sólo si todas las transacciones en él son válidas y no existe doble gasto.

Doble gasto son aquellas transacciones fraudulentas que intentan gastar la misma cantidad de bitcoins más de una vez. Cuando el bloque es aceptado por al menos 6 nodos, éste es encadenado al *Blockchain* y de nuevo los nodos vuelven a competir por publicar el siguiente bloque.

Tipos de nodos

Los nodos que forman la red Bitcoin aún siendo todos iguales, adoptan distintos roles en base a la funcionalidad que desarrollen: enrutamiento, *Blockchain*, *wallet,* minería. Todos los nodos comparten algunas de las funcionalidades, por ejemplo, todos los nodos tienen capacidades de enrutamiento, dado que Bitcoin se trata de una red P2P, todos los nodos de la red deben ser capaces de comunicarse con sus pares para poder propagar las transacciones que se llevan a cabo. Además, todos los nodos pueden validar las transacciones y bloques del *Blockchain*.

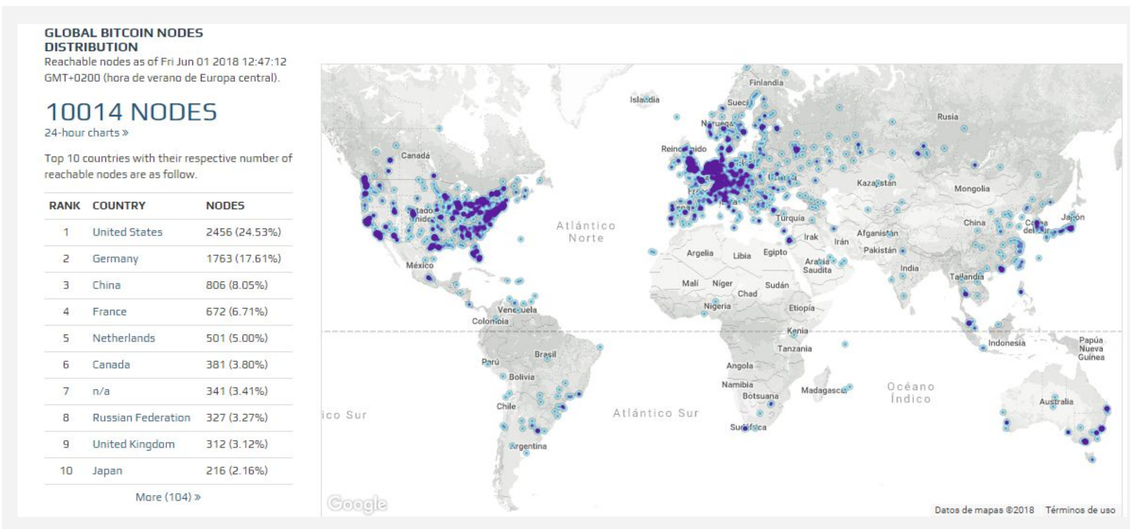


Imagen 14. Distribución de los nodos de Bitcoin en el mundo

Existen lo que se denomina *full nodes,* estos pueden realizar todas las funcionalidades mencionadas anteriormente, además mantienen una copia completa y actualizada del *Blockchain* (162 GB a 17 de abril de 2018) para verificar cualquier transacción de manera autónoma. Todos los nodos pueden validar transacciones y bloques del *Blockchain,* pero no necesariamente todos los nodos guardan una copia completa del *Blockchain.* Los que no guardan una copia completa verifican las transacciones por medio de la verificación de pagos simplificada (SPV).

Normalmente aquellos nodos que utilizan el método SPV, suelen ser nodos clientes, es decir, dispositivos distintos a un ordenador convencional con limitaciones en procesamiento o energía. Un ejemplo común son las *wallets,* muchos de los usuarios de Bitcoin realizan sus transacciones desde un teléfono móvil donde tienen instalado un programa cliente que hace la función de monedero y donde guardan sus claves para poder operar con los bitcoins. A este tipo de nodo también se le conoce como nodo SPV. Los nodos SPV sólo descargan los encabezados de los bloques en el *Blockchain* sin llegar a descargar las transacciones de cada bloque. La manera en que los nodos SPV verifican las transacciones es por demanda a otros *full nodes* para que les muestren las transacciones relevantes necesarias de un bloque concreto.

Los nodos que desarrollan funcionalidades de minería son aquellos que ejecutan el software adecuado para resolver el problema matemático *Proof-of-Work* y así poder publicar bloques en el *Blockchain*. Este tipo de nodo puede contener una copia completa del *Blockchain* o no necesariamente. Los nodos que no guardan una copia completa de la cadena de bloques pertenecen a un grupo de nodos que participan en la minería de bitcoins y dependen de un servidor para formar un *full node* que sí contiene una copia completa del *Blockchain*.

Las transacciones

Las transacciones (Tx) son la esencia de la red Bitcoin y el motivo por el que se creó. Bitcoin está diseñado para gestionar las transacciones de bitcoins entre usuarios, para poder generarlas, propagarlas por la red y finalmente añadirlas a un bloque del *Blockchain*. Una transacción es la acción que le dice a la red Bitcoin que el propietario de una serie de bitcoins ha autorizado la transferencia de algunos de esos bitcoins a otro propietario, de manera que el nuevo propietario podrá gastar esos bitcoins creando otra transacción que autorice la transferencia a otro propietario, y así sucesivamente, en una cadena de propiedad.

Las transacciones son como los asientos contables de un libro diario de contabilidad. En términos simples, cada transacción contiene una o más entradas que representan los créditos agregados a una cuenta de bitcoins. En el otro extremo de la transacción, hay una o más salidas que son los débitos de la cuenta de bitcoins. Cuando se genera una transacción de bitcoins entre dos usuarios, en realidad lo que se genera es una entrada y una salida en el libro contable de Bitcoin, el *Blockchain*. Por parte del usuario que envía bitcoins se genera una salida, un débito en el *Blockchain* y por parte del usuario que recibe los bitcoins se genera una entrada, un crédito.

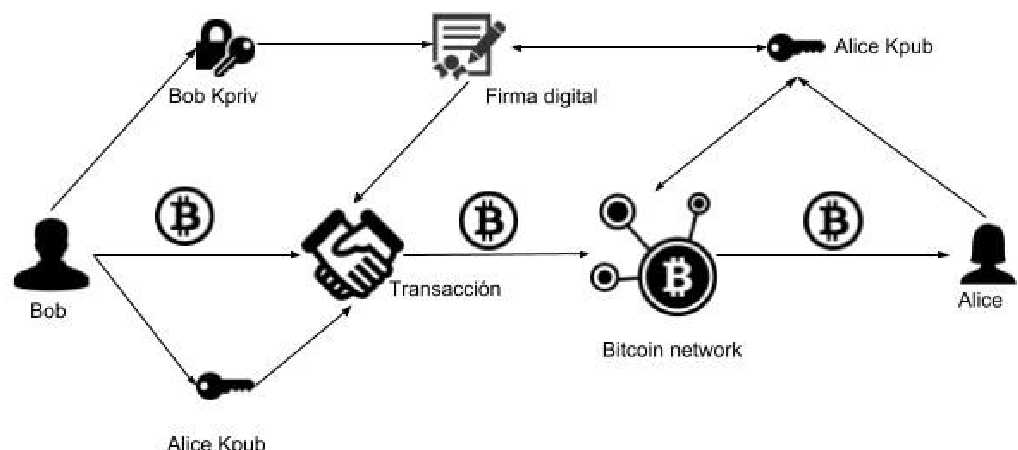


Imagen 15. Anatomía de una transacción de bitcoins

Las transacciones, implícitamente, llevan asociada una salida y éstas crean el elemento fundamental de las transacciones, las salidas de transacciones no gastadas (UTXO - *unspent transaction output*). Se trata de fragmentos de bitcoins bloqueados para su propietario para que pueda gastarlos en el futuro. Los UTXO están registrados en el *Blockchain* y cada vez que un usuario recibe bitcoins, la cantidad recibida se registra como UTXO. Por lo tanto no existe un saldo de usuario en donde se almacena las unidades de bitcoins a favor, lo que existe son UTXO bloqueados para cada uno de sus propietarios. El monedero de cada usuario calcula el saldo escaneando el *Blockchain* y agregando todos los UTXO que pertenecen a ese usuario.

Supongamos que Bob tiene 20 BTC y quiere enviar 5 BTC a su amiga Alice y 5 BTC a su hermana Mallory. Para llevar a cabo las transacciones, Bob dispone de su saldo UTXO de 20 BTC las cantidades de 5 BTC para Alice y 5 BTC para Mallory. Bob ya conoce las dirección de los monederos de Alice y Mallory, así que después de especificar las cantidades a transferir a cada una y de que se le haya descontado las cantidades de las comisiones por transacción para los mineros, Bob firma con su clave privada para finalizar las transacciones (Tx 0) y que ésta se retransmita por toda la red Bitcoin.

Dado que Bob no quiere gastar todo su saldo en Alice y Mallory, de manera implícita con las transacciones, Bob se gasta los 10 BTC restantes en su propia dirección de cambio. De no hacer esto la cantidad restante se destinaría a las comisiones de transacción pagadas a los mineros. La dirección de cambio es como el cambio que se recibe en un comercio al pagar un producto con una moneda de mayor valor que el propio producto y se recibe la diferencia del valor a modo de cambio.

En cuanto Mallory recibe la transferencia de bitcoins de Bob, recuerda que le debe 3 BTC a Alice, así que del mismo modo que Bob, realiza una transacción que firma mediante su clave privada para transferir dicha cantidad a Alice. De nuevo, Mallory se gastará el valor restante de su saldo en su propia cuenta de cambio para que éste se convierta en UTXO.
Finalmente, Alice recibe las transferencias de bitcoins de Bob y Mallory y su saldo total es de 8 BTC UTXO.

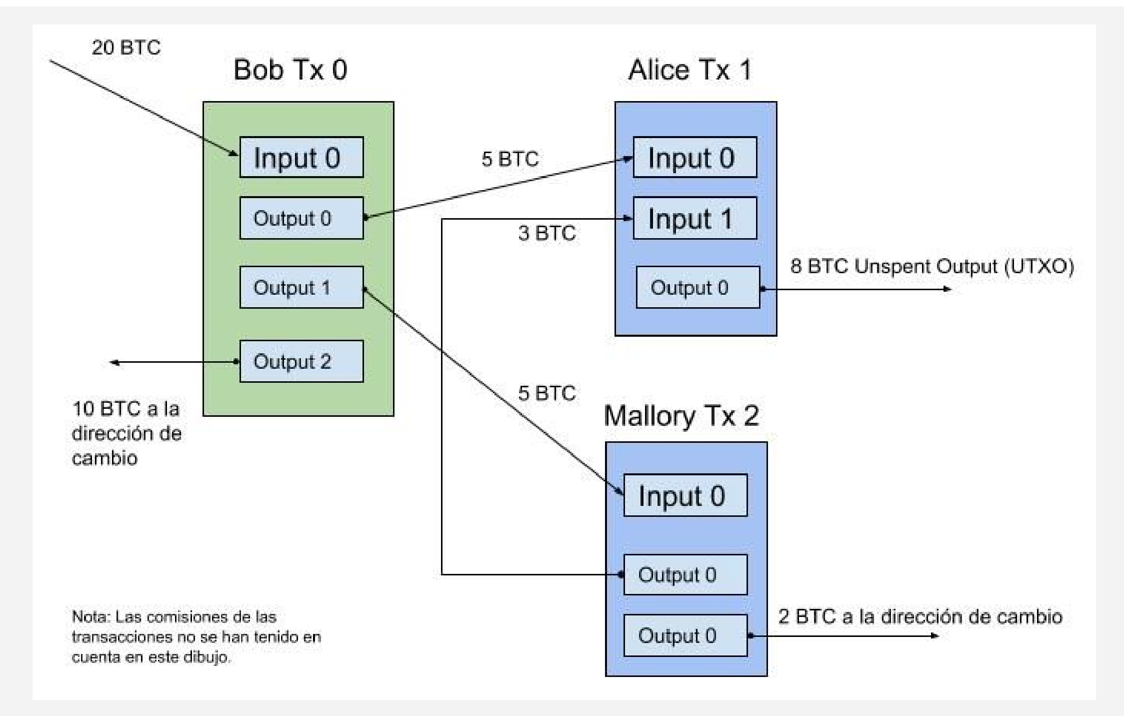


Imagen 16. Proceso de una transacción de bitcoins

Una vez que la transacción de Bob se ha transmitido a la red de Bitcoin, ésta queda almacenada en lo que se conoce como *mempool* (grupo de memoria) hasta que algún minero añada la transacción a un bloque para ser minado y añadido al *Blockchain*. *Mempool* es la memoria del sistema Bitcoin la cual contiene una lista de todas las transacciones no confirmadas que necesitan ser procesadas por los mineros.

Todas las transacciones son de carácter público en el *Blockchain* y pueden ser consultadas por cualquier persona, no obstante lo que no es público son los datos personales de cada una de las transacciones. Cada transacción además contiene información de la propiedad de los bitcoins transferidos, es decir, de las direcciones de monedero de los dos usuarios implicados en la transacción. Para que una transacción se lleve a cabo y los fondos pasen de un usuario a otro es necesario que quien transfiere los bitcoins valide la transacción por medio de una firma digital. Una vez la transacción es registrada en un bloque del *Blockchain*, ésta pasa a formar parte de manera permanente en el libro contable de Bitcoin, el *Blockchain* y su eliminación no será posible.

Transacción Ver información de una transacción de Bitcoin

65274b4f682654988173bb5e28ce9b01558f5b1087b68c233a2d2b05cd7f2b07

12NPTbLsJMWCBJ4odfrSvSviFGub33Yf2n → 1CkCQjcRYdo8AVL6pvDNb63R5SnBZCh1MK 0.51273557 BTC
191xwiNGxjR91jLPLet68J7oeWogKxYQdt 0.10744369 BTC

4 Confirmaciones 0.62017926 BTC

Resumen	
tamaño	226 (Bytes)
Peso	904
Hora de Recepción	2018-04-18 07:44:08
Tiempo de bloqueo	Bloquear: 518746
Incluídas en el Bloque	518747 (2018-04-18 07:45:44 + 2 minutos)
Confirmaciones	4 Confirmaciones
Visualizar	Ver Gráfico de Árbol

Entradas y Salidas	
Entrada total	0.62187426 BTC
Salida Total	0.62017926 BTC
Comisiones	0.001695 BTC
Tarifa por byte	750 sat/B
Tarifa por unidad de peso	187.5 sat/WU
Estimado de BTCs transaccionados	0.10744369 BTC
Scripts	Mostrar scripts y Coinbase

Imagen 17. Información de una transacción en Bitcoin

Mineros

Un minero es un individuo o una empresa que compite contra otros mineros resolviendo problemas matemáticos extremadamente complejos, lo que se denomina *Proof-of-Work*, para conseguir añadir bloques de transacciones al *Blockchain* a cambio de ser recompensado con bitcoins. A este proceso se le conoce como minería y requiere de muchos recursos de procesamiento además de económicos por el coste de los ordenadores óptimos específicos a utilizar así como el consumo de electricidad de dichas máquinas.

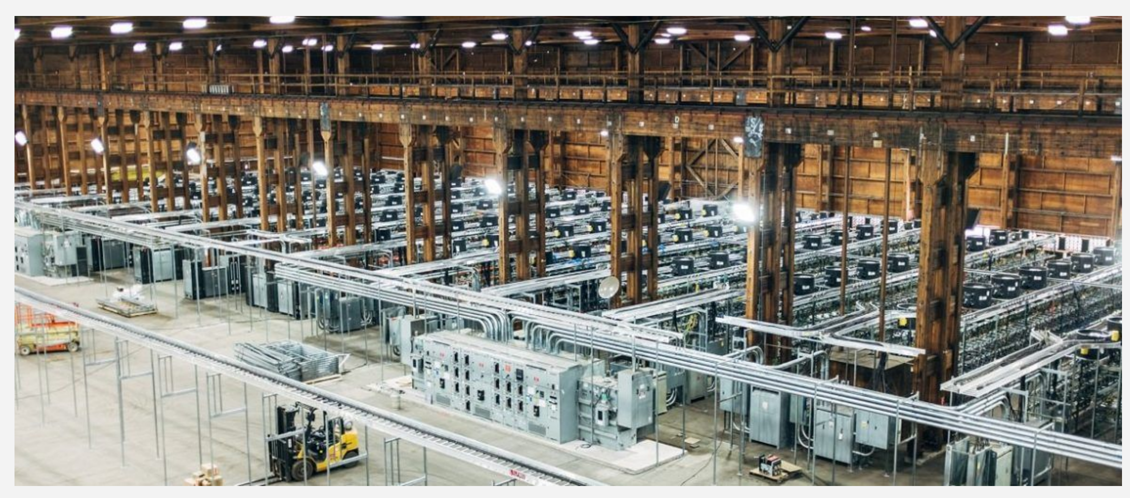


Imagen 18. Empresa china de minado de bloques de Bitcoin

Tal y como se ha podido leer anteriormente, después de que cada transacción se haya llevado a cabo, ésta no se convierte en parte del *Blockchain* hasta que esté verificada y se incluya en un bloque de transacciones para que finalmente se pueda añadir a la cadena de bloques de Bitcoin.

Bitcoin es un sistema que se basa en la confianza de los nodos que la componen y para ello se requiere de cálculos matemáticos muy complejos. Las transacciones se agrupan en bloques que requieren una enorme cantidad de procesamiento para ser verificados, pero muy poco procesamiento para comprobarlos como verificados. Este proceso tiene dos propósitos en Bitcoin:

1. La minería crea nuevos bitcoins para el minero en cada bloque de transacciones añadido al *Blockchain*, es como cuando un banco central imprime dinero nuevo. La cantidad de bitcoins que se crean por bloque añadido al *Blockchain* se fija y disminuye con el tiempo por el propio sistema Bitcoin (actualmente son 12.5 BTC)

2. La minería es la base del sistema de confianza de Bitcoin al garantizar que las transacciones sólo se confirman o verifican si se dedicó la suficiente potencia de procesamiento al bloque que las contiene. Cuantos más bloques más potencia de procesamiento y coste económico se requiere.

Cada bloque nuevo que contiene transacciones que ocurrieron desde el último bloque añadido a la cadena de bloques de Bitcoin, es añadido al *Blockchain* cada 10 minutos. Las transacciones que se convierten en parte de un bloque y se agregan a la cadena de bloques de Bitcoin se consideran verificadas, lo que permite a los nuevos propietarios de bitcoins gastar los bitcoins recibidos en esas transacciones.

Los mineros reciben dos tipos de beneficio por el proceso de la minería; bitcoins con cada nuevo bloque que consiguen añadir al *Blockchain* y las comisiones de las transacciones incluidas en el bloque. En la actualidad, las comisiones representan el 0,5% o menos de los ingresos de un minero, la gran mayoría provienen de los bitcoins acuñados con cada bloque que consiguen añadir al *Blockchain*. Sin embargo como se ha mencionado anteriormente, a medida que la recompensa de bitcoins disminuye con el tiempo por el propio sistema de Bitcoin y aumenta la cantidad de transacciones por bloque, los ingresos de los mineros serán cada vez mayor por comisiones. Hay que recordar que después de 2140, todas las ganancias del minero vendrán de las comisiones por transacciones ya que se habrán minado todas las unidades de bitcoin que el propio sistema proporciona.

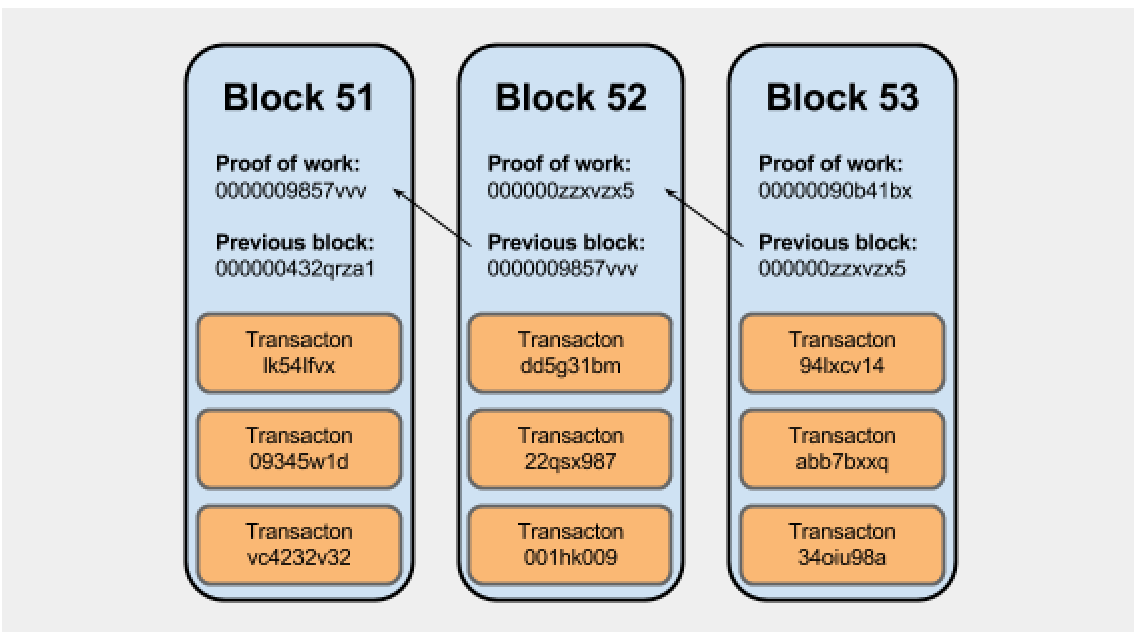


Imagen 63. Bloques de Bitcoin

Para que los mineros puedan beneficiarse, éstos compiten para resolver un problema matemático complejo basado en un algoritmo de criptografía *hash*. La solución al problema matemático, conocido como *Proof-of-Work* (PoW), se incluye en el nuevo bloque y sirve como prueba de que el minero realizó un gasto de procesamiento computacional.

Proof-of-Work

Los sistemas de pago tradicionales dependen de un modelo de confianza avalada por una autoridad central. Los bancos centrales de cada país son los que proporcionan un servicio de garantía y respaldan las transacciones que se efectúan en las distintas entidades bancarias. Sin embargo, Bitcoin no cuenta con ninguna autoridad central que garantice las transacciones. Como alternativa, Bitcoin cuenta con la participación de todos los nodos de su red, donde muchos de ellos, guardan constantemente una copia entera del *Blockchain* en el que poder confiar como registro autorizado. Es importante destacar que el *Blockchain* no está generado por ninguna autoridad central, sino que es ensamblado bloque a bloque por cada uno de los nodos de la red que consiguen superar el PoW. De hecho, hay que recordar que todas las transacciones que se llevan a cabo en Bitcoin son retransmitidas a todos los nodos de la red y todos los nodos tienen capacidad de verificar las transacciones y ensamblar bloques con ellas. Por ello, cada nodo de la red Bitcoin, en base a la información que va recibiendo, puede llegar a ensamblar el mismo bloque y por ende pueden generar el mismo *Blockchain*.

El consenso descentralizado de Bitcoin se basa en la interacción de cuatro procesos que ocurren independientemente en los nodos de la red:

1. La verificación independiente de cada transacción por parte de cada *full node*.

2. El procesamiento computacional independiente de cada transacción al ser agregada a nuevos bloques mediante la minería.

3. La verificación independiente de los nuevos bloques por cada nodo y el ensamblaje en cadena, es decir, la encadenación de los bloques al *Blockchain*.

4. La selección independiente por parte de cada nodo de los bloques con el mayor esfuerzo computacional acumulado, demostrado a través del *Proof-of-Work*.

Un algoritmo hash toma una entrada de datos de longitud arbitraria y produce un resultado determinista de longitud fija, en cierta manera se trata de una huella digital de la entrada. Para cualquier entrada específica, el hash resultante siempre será el mismo, no obstante, es imposible (en la actualidad) encontrar dos entradas distintas que produzcan el mismo hash y eso es la característica clave de todo algoritmo hash criptográfico. Otra característica esencial de los algoritmos hash es que es virtualmente imposible seleccionar una entrada que genere un hash deseado, para ello habría que probar aleatoriamente hasta encontrar el hash esperado.

Con SHA-256, la salida siempre tiene 256 bits de largo, independientemente del tamaño de la entrada. Por ejemplo, si utilizamos el siguiente código escrito en Python, podemos generar un SHA-256 para una entrada cualquiera:

```
import hashlib
# Obtenemos el hash en hex de un string
hash = hashlib.sha256(b'Pasado, Presente y Futuro del
Blockchain').hexdigest()
print(hash)
```

Entrada: “`Pasado, Presente y Futuro del` *`Blockchain`*”
Hash: `64139614e242da1c5390a36e201e5f90373a7dac35edb78bea905c374da06491`

Este hash en formato hex (hexadecimal) de 256 bits depende de cada letra de la frase de entrada. Agregar o borrar un sólo carácter o signo de puntuación producirá un hash completamente diferente.

Entrada: “`Pasado, Presente y Futuro del` *`Blockchain`*`.`”
Hash: `d1e2ba626b1aeb8a1a3b688231858548109b778501f4cba7e1be976087af1b20`

Nótese cómo al añadir el punto al final de la frase, se ha generado un hash completamente distinto.

Supongamos que deseamos obtener un hash donde el primer carácter del hash sea un 0, sin importar el resultado del resto de caracteres del hash. Como se ha comentado anteriormente, la única forma de conseguir esto es mediante la alteración de la frase de entrada e ir probando qué modificación produce el hash que buscamos. El siguiente código Python nos permitirá añadir un dígito a nuestra frase de entrada y obtener un hash.

```
import hashlib
frase = "Pasado, Presente y Futuro del Blockchain."
# Iteracion de 0 a 10
for nonce in xrange(10):
    # Agregar digito al final de la frase de entrada
    input = frase + str(nonce)
    # calcula el SHA-256 hash de la frase entrada + digito
    hash = hashlib.sha256(input).hexdigest()
    print input, '=>', hash
```

Al ejecutar el programa obtenemos los siguientes hash:

Pasado, Presente y Futuro del *Blockchain*.0 =>
b2bc6ec2f19485832d126a80c2045b357d4c6f9b34b85648c8316a91912c1bbd

Pasado, Presente y Futuro del *Blockchain*.1 =>
e715bb9560d4d598121af17ddacdd647c188b805063ad73687596efa00282809

Pasado, Presente y Futuro del *Blockchain*.2 =>
b4bdf0e06e4966acc93a92468b1f6e80ea5b4898aa930e90d732b756a0150bd4

Pasado, Presente y Futuro del *Blockchain*.3 =>
cff4ccc1b84e87d06c86d64d201aa6f72d7ba42780a5b2b940b9b1654dc0b1a8

Pasado, Presente y Futuro del *Blockchain*.4 =>
188998f1c90335755763358f84dda2b2c29025514e92631d294e5b07883c0df5

Pasado, Presente y Futuro del *Blockchain*.5 =>
04d98450cefad6d7a6db6f2d10223428e7685897850d80d333ae9c9b1efd3fa7

Pasado, Presente y Futuro del *Blockchain*.6 =>
2cbbf855950d62fd47b6dcd1901b075b3b21e96abdf50d2b5ec039e03d36e04a

Pasado, Presente y Futuro del *Blockchain*.7 =>
006a891115cbc03aa027270fac67a0e46b32c75ed5c372eff2deb91fd47ea5da

Pasado, Presente y Futuro del *Blockchain*.8 =>
938f3daf8f71732556d15fd8697a885e16717f3f86282348090465b4205292e8

Pasado, Presente y Futuro del *Blockchain*.9 =>
88dbf4a01b299323d7e8a9a0b0717a70984e2e234e780c6a323d929dd6231021

Tal y como se puede observar, hemos tenido que iterar 6 veces hasta obtener el primer hash con su primer carácter cero, ha sido al añadir el dígito 5 al final de la frase de entrada. También vemos que en la iteración 8, al añadir el dígito 7 también obtenemos un hash deseado. Tanto el 5 como el 7, son lo que se conoce como palabras sueltas (*nonce*). El *nonce* se usa para variar la salida de una función criptográfica, en este caso para variar la huella digital SHA-256 de la frase de entrada. En el caso de Bitcoin, el *nonce* es un número entero entre 0 y 4.294.967.296.

En términos probabilísticos si la salida de la función hash está distribuida uniformemente, esperaríamos encontrar un hash con prefijo 0 una vez cada 16 hashes ya que en hexadecimal los valores empiezan en cero y finalizan en 16 (F), es decir, habría que encontrar un número menor al valor de $1 \cdot 10^{63}$ (los hash en hexadecimal están compuestos por 64 caracteres). Esto es lo que se conoce como el umbral del objetivo y la meta es encontrar un hash que numéricamente sea menor que dicho objetivo. A medida que reducimos el valor del objetivo, hallar el hash deseado se vuelve más complejo.

Imaginemos que tenemos un dado de 6 caras y en una apuesta nos piden sacar un valor objetivo menor al máximo de los valores del dado, es decir, a 6. La probabilidad de sacar el resultado deseado es de ⅚ dado que todas las caras de un dado a excepción del 6 son menores. En la siguiente ronda nos piden sacar un valor objetivo menor a 4, ahora la probabilidad de conseguir el resultado es de 3/6, las probabilidades se han reducido a la mitad. En el momento en que nos pidan sacar un valor menor a 2, la probabilidad de obtener un 1 será sólo de ⅙.

Volviendo al ejemplo de hallar el primer carácter del hash con valor a cero, comprobar que aplicando el *nonce* 5 al final de la frase de entrada “Pasado, Presente y Futuro del *Blockchain*.” se obtiene el hash deseado no requiere un gran esfuerzo computacional, el primer código Python que hemos puesto como ejemplo lo realizaría en una única iteración, no obstante, para hallar por primera vez el hash deseado el programa (segundo código Python) ha necesitado iterar 5 veces hasta obtener el resultado y por lo tanto en cuestión de esfuerzo computacional ha sido mayor. Es decir, generar el hash ha costado más que simplemente comprobar que efectivamente el *nonce* era el deseado.

El funcionamiento del PoW de Bitcoin es muy similar a lo expuesto, se trata de obtener un resultado que cumple con un nivel de dificultad predeterminado, principalmente el número de ceros con el que empiezan los hash de cada nuevo bloque encadenado al *Blockchain*. Es decir,

Bitcoin impone a los mineros que el hash del bloque a encadenar en el *Blockchain*, empiece con 18 ceros (a fecha en la que se ha realizado este trabajo). El minero construye un bloque a partir de las transacciones recibidas desde el último bloque encadenado al *Blockchain*. Una vez completado el bloque con transacciones (1MB es el tamaño máximo de los bloques en Bitcoin), el minero encripta el encabezado del bloque con una función hash.

El encabezado del bloque se compone de diversos campos como podemos ver a continuación:

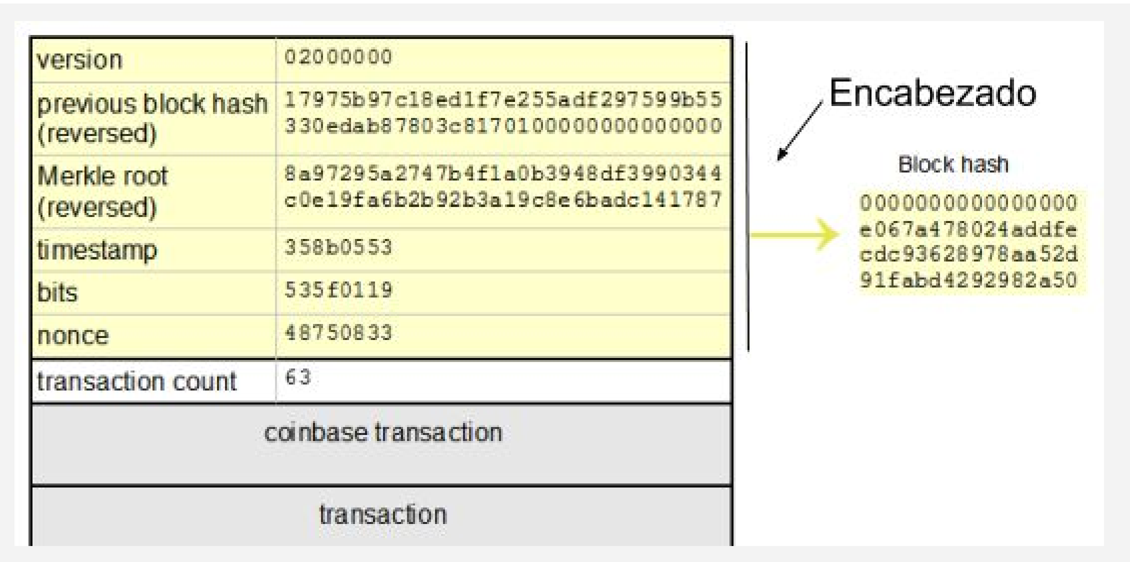


Imagen 19. Encabezado de un bloque de Bitcoin

Los campos que se muestran en la imagen anterior son los que hacen que el *Blockchain* de Bitcoin no pueda ser alterado. Cada nuevo bloque del *Blockchain* contiene el hash del bloque anterior más el sello de tiempo, por lo que hace que cada bloque dependa del anterior. El intento de cualquier modificación en el *Merkle root* de un sólo bloque haría cambiar el hash y no podría relacionarse con el bloque posterior al alterado, por lo que sería imposible añadir el bloque alterado al *Blockchain*.

Los *Merkle root* (árbol Merkle) son utilizados en Bitcoin para resumir las transacciones que contiene un bloque. Sirven como estructura para resumir de manera eficiente grandes cantidades de datos. Cada transacción tiene un hash asociado. En un bloque, de cada una de las transacciones se obtiene su hash (incluso varias veces) y posteriormente se vuelve a obtener un hash de todas ellas juntas en el bloque obteniendo de esta manera un *Merkle root*.

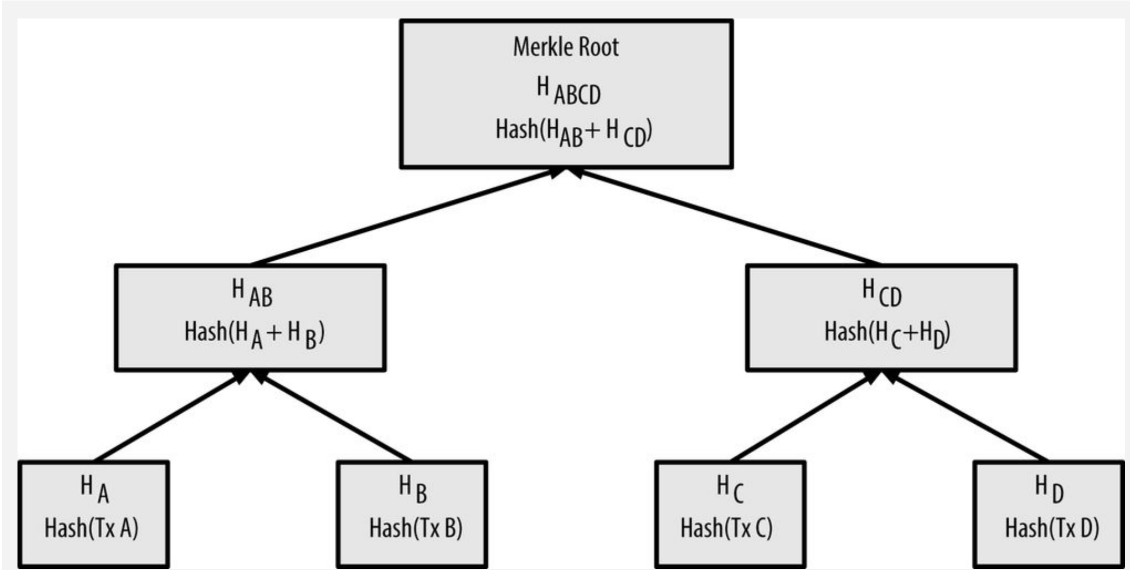


Imagen 20. Merkle root

En otras palabras, el árbol de Merkle es el hash de todos los hash de todas las transacciones en el bloque. La *Merkle root* está incluida en el encabezado del bloque. Con este esquema, es posible verificar de forma segura que una transacción ha sido aceptada por la red descargando sólo los encabezados de los bloques y el árbol de Merkle.

Si el hash obtenido no es menor que el umbral objetivo, el minero modifica el *nonce* para volver a intentarlo de nuevo. Actualmente la dificultad para los mineros de Bitcoin es la necesidad de iterar cuatrillones de veces antes de encontrar un *nonce* que resulte en un hash de bloque suficientemente bajo. De hecho, cada vez que se restringe un bit más a cero, se reduce el espacio de búsqueda a la mitad lo que provoca un aumento exponencial en el tiempo a la hora de hallar la solución.

Actualmente, se necesita más de 150 cuatrillones de cálculos de hash por segundo para que la red descubra el siguiente bloque, si tenemos en cuenta que los nodos de la red ofrecen 100 Peta hashes por segundo de potencia, esto proporciona un promedio de 10 minutos para que se pueda encadenar un bloque al *Blockchain*. Este tiempo promedio es la cadencia de Bitcoin que garantiza la frecuencia en la emisión de su criptomoneda y la velocidad en la liquidación de las transacciones. Es necesario que este tiempo se mantenga a lo largo del tiempo ya que si tenemos en cuenta la ley de Moore y que la potencia de procesamiento de los equipos aumenta a un ritmo veloz, sumado al crecimiento de nuevos mineros, haría que el tiempo en encontrar el *nonce* se redujera. Por ello Bitcoin ajusta la dificultad para tener en cuenta estos cambios a lo largo del tiempo.

La dificultad es un parámetro dinámico que se ajusta periódicamente para obligar a cumplir los 10 minutos de promedio para la generación de un nuevo bloque. Cada 2016 bloques todos los nodos se ajustan a la dificultad del PoW impuesta por Bitcoin. Para ajustar el nivel de dificultad, Bitcoin mide el tiempo que se tardó en encontrar los últimos 2016 bloques y los compara con el tiempo esperado, 20160 minutos. Si se están generando nuevos bloques más rápido que el promedio de 10 minutos, la dificultad aumenta, por el contrario si el proceso es más lento de lo esperado, la dificultad disminuye.

4. *Blockchain* 2.0

DApps (Decentralized applications)

¿Qué son las DApps?

Una DApp (*Decentralized application*) es una aplicación cuyo *backend* se ejecuta en una red distribuida y descentralizada *Peer-to-Peer* (P2P). No existe ningún nodo en la red que tenga el control sobre la DApp. En base a la funcionalidad de la DApp, se utilizarán diferentes estructuras para almacenar los datos relacionados con la DApp. Un ejemplo de DApp es el Bitcoin, se trata de una DApp que gestiona las transacciones de su propia criptomoneda, el bitcoin.

No hay que confundir las aplicaciones descentralizadas con las aplicaciones distribuidas. Las aplicaciones distribuidas son aquellas aplicaciones, cuyos datos e incluso la misma aplicación, se encuentran replicados en varios servidores en lugar de sólo uno pero gestionados por la misma entidad o persona. Este tipo de aplicaciones se emplean cuando los datos que gestiona la aplicación son de un tamaño considerable y la posibilidad de un fallo en el sistema de la aplicación no es una opción.

Los pares que forman la red P2P de una DApp pueden ser cualquier ordenador conectado a Internet. Esto hace que la detección de cambios intencionados o no sobre los datos de la DApp sean difíciles de gestionar. Es por ello que se necesita un consenso entre los participantes de la red, los nodos, para confirmar si los datos generados por la aplicación son o no correctos. De nuevo, hay que recordar que en una red descentralizada no existe un servidor central que valide los datos. En Bitcoin se ha expuesto como protocolo de consenso que utiliza para garantizar la validez de las transacciones de sus usuarios el *Proof-of-Work*.

Muchas DApps requieren que sus usuarios cuenten con funcionalidades que permitan modificar sus datos de usuario. Este requisito que es muy sencillo en las aplicaciones centralizadas de implementar, en las DApps se convierten en un reto. Las DApps no pueden basar esta funcionalidad en un simple nombre de usuario y contraseña ya que las DApps no cuentan con procesos de verificación de usuario. Para ello se emplea el uso de claves públicas y privadas. Para llevar a cabo un cambio en los datos de la cuenta, el usuario debe firmar la modificación con su clave privada. Es por esto que es muy importante que los usuarios almacenen en lugar seguro sus claves privadas.

Las criptomonedas

Dado que las DApps no tienen un propietario sino que son los nodos de la red P2P quienes la ejecutan, éstos necesitan recursos para ejecutar la DApp y mantenerla en funcionamiento. Por lo tanto, los nodos de una DApp necesitan algún tipo de beneficio a cambio de mantener la DApp en sus máquinas. Aquí es donde entra en juego el concepto de criptomoneda. La gran mayoría de DApps cuentan con su propia moneda (o con la moneda de la plataforma *Blockchain* en la que se ejecutan) que no sólo sirve para adquirir posibles servicios que pueda ofrecer la DApp sino que es lo que alimenta a los nodos a seguir ejecutándola. El protocolo de consenso es lo que determina la cantidad de dinero que recibe un nodo. En Bitcoin se ha visto cómo los nodos que desarrollan tareas de minero son recompensados con bitcoins cada vez que consiguen encadenar un nuevo bloque al *Blockchain* de Bitcoin.

La incógnita para muchos usuarios y no usuarios es de qué manera se le proporciona valor a una criptomoneda. Siguiendo las reglas de todo sistema económico, siempre que exista demanda y si las unidades de la criptomoneda interna de la DApp son inferiores a la demanda, la criptomoneda adquirirá valor. Es por este motivo que los usuarios de una DApp deben pagar con la criptomoneda interna por utilizarla, esto resuelve la cuestión de la demanda. Si el número de usuarios de una DApp incrementa, también lo hace el valor de su criptomoneda. Al mismo tiempo, si se fija el número de unidades de criptomoneda que se pueden generar, ésta adquiere más valor. Las criptomonedas se acuñan a lo largo del tiempo en vez de generarlas todas en una única vez al ejecutar la DApp por primera vez, esto permite que nuevos nodos que se unan a la red para ejecutar la DApp puedan ser recompensados económicamente.

Ventajas y desventajas de una DApp

Las DApps ofrecen una serie de ventajas frente a las aplicaciones centralizadas sean o no distribuidas.

- Las DApps son resistentes a fallos de la red ya que están distribuidas por cada nodo que compone la red.

- Son inmunes a la censura dado que no existe una autoridad central que las regule, ergo no pueden eliminar contenido de las DApps. Ni siquiera se puede bloquear la IP (*Internet Protocol Address*) de la aplicación porque las DApps carecen de IP. Resultaría imposible bloquear las IPs de cada nodo de una red si ésta es muy grande, como por ejemplo la de Bitcoin.

- La confianza por parte de los usuarios en las DApps es mayor que en las aplicaciones centralizadas, ya que al estar descentralizada no existe una autoridad que pueda engañar a los usuarios con fines de lucro.

También cuentan con desventajas frente a las aplicaciones centralizadas:

- La corrección de errores de las DApps son más lentas de llevar a cabo ya que todos los pares de la red deben actualizar su software de nodo.

- Algunas aplicaciones requieren que el usuario se identifique y el proceso de verificación se convierte en un problema al desarrollar dichas aplicaciones ya que no existe una autoridad central que verifique la identidad del usuario mediante el proceso *Know your customer* (KYC) (algunas DApps requieren de sus usuarios copias de los documentos de identidad), este es uno de los motivos de que las DApps proporcionen anonimato a sus usuarios.

- Son complejas de implementar por los protocolos empleados para lograr el consenso, y éstos deben programarse desde el principio de la aplicación, por lo que añadir nuevas funcionalidades o escalar la aplicación resulta complejo.

Ethereum

¿Qué es Ethereum?

Ethereum es una plataforma de código abierto (*open source*) basada en tecnología de *Blockchain* pública que permite a desarrolladores implementar aplicaciones descentralizadas en una red P2P, de cualquier tipo, para ser ejecutadas bajo la plataforma Ethereum. Creada por el programador ruso Vitalik Buterin y lanzada por primera vez en el año 2015, Ethereum se diferencia de Bitcoin tanto en propósito como en capacidad. Bitcoin es una DApp concreta de la tecnología *Blockchain* de Bitcoin que proporciona un sistema efectivo P2P para realizar pagos en línea (*online*) con su propia criptomoneda. Bitcoin emplea el *Blockchain* para validar las transacciones realizadas y garantizar la propiedad de los bitcoins de sus usuarios. Por el contrario, el *Blockchain* de Ethereum permite que cierto código de una DApp se ejecute en función de unas condiciones preestablecidas y que ello quede registrado de manera inalterable en el *Blockchain* de Ethereum.

Ethereum es un conjunto de protocolos que definen la propia plataforma y en el núcleo de ésta, está la Máquina Virtual (EVM) que permite ejecutar cualquier tipo de código arbitrario. En cierto modo, Ethereum es “Turing Completo”, es decir, Ethereum puede ejecutar cualquier tipo de operación si se dispone de los recursos necesarios. El concepto “Turing Completo” proviene del matemático Alan Turing y su máquina Turing Universal por el que afirmó que podría realizar cualquier tipo de cálculo matemático y lógico si se disponía de recursos físicos ilimitados.

Ethereum entre su pila de protocolos incluye el de la red P2P para que su base de datos, *el Blockchain* pueda ser mantenida y actualizada por los nodos mineros. En Ethereum existen dos tipos de nodos, los normales y los mineros. Los primero tan sólo guardan una copia del *Blockchain* mientras que los segundos lo construyen minando bloques nuevos. Al igual que en Bitcoin, Ethereum cuenta con su propia criptomoneda, el Ether. El Ether es por lo que los mineros trabajan, pero además, en Ethereum los desarrolladores y usuarios de DApps también pagan Ethers para interactuar con las DApps. Cada uno de los nodos que forman la red Ethereum, ejecuta la EVM para ejecutar las mismas instrucciones, es por esta razón que a Ethereum se le denomina *The World Computer* (el ordenador mundial). El principal motivo por el que cada uno de los nodos ejecuta la EVM es para establecer el consenso a través del *Blockchain*, permitiendo a Ethereum un alto grado de resistencia a fallas. Además es la base para que las aplicaciones se ejecuten exactamente como se programaron sin censura, fraude o interferencia de terceros.

Ethereum permite el desarrollo de cualquier tipo de DApp, no obstante aquellas aplicaciones que automaticen la interacción entre pares o grupos, sin necesidad de intermediarios, a través de una red se verán más beneficiadas como por ejemplo aquellas aplicaciones destinadas a la automatización de contratos financieros o aplicaciones que requieran de un entorno de confianza, seguridad y permanencia (votaciones, IoT, registros de activos, subastas, apuestas, etc).

Smart Contracts

Los *smart contracts* (contratos inteligentes) son los programas implementados por desarrolladores para ser ejecutados en el EVM y que residen en una dirección específica en el *Blockchain* de Ethereum. Las DApps de Ethereum se escriben con uno o más *smart contracts*.

En Ethereum, los *smart contract*s se pueden escribir en distintos lenguajes, de alto nivel, de programación como por ejemplo Solidity, Serpent, Lisp Like Language (LLL), etc y compilados en *bytecode* para ser subidos al *Blockchain*.

Los contratos inteligentes pueden:

- Funcionar como cuentas de validación con firma múltiple. De modo que los fondos que se incluyan en el *smart contract* sólo puedan ser gastados cuando un porcentaje requerido de las personas involucradas esté de acuerdo.
- Administrar los acuerdos entre usuarios. Ejecuta las condiciones pactadas entre dos usuarios cuando se cumple cierta condición.
- Almacenar datos de otro o para otro contrato inteligente.

Para subir los contratos inteligentes al *Blockchain* o llamar a sus métodos hay que pagar lo que se denomina *Gas*. El *Gas* es la unidad de medida, para tarificar los pasos computacionales necesarios, para llevar a cabo cualquier interacción con un *smart contract* en el *Blockchain* de Ethereum. Cuando se invoca a un método en un *smart contract* o un método invoca a otro método de otro *smart contract,* el *Gas* se deduce de la cuenta del usuario que invocó el método.

Los mineros son los que deciden el precio del *Gas*. Si una transacción tiene un precio de *Gas* menor que el precio decidido por el minero, éste puede negarse a procesar la transacción hasta que la acepte algún otro minero. Cuando se realiza una transacción en Ethereum -transacción es toda aquella interacción con un contrato inteligente-, es requisito indispensable que la misma incluya un campo donde especificar el límite de *Gas* y la tarifa dispuesta a pagar por *Gas*. De esta manera, los mineros tienen la opción de incluir la transacción y cobrar la comisión. El límite de gas es decidido por votación por cada minero y cada uno de éstos determina qué precio de gas está dispuesto a aceptar. En caso de que el Gas utilizado por la transacción sea menor o igual al límite establecido, la transacción se procesa. Por el contrario, si el *Gas* total necesario para la transacción excede el límite de gas indicado, todos los cambios se revierten a excepción de que la transacción sigue siendo válida y el minero puede cobrar la tarifa. Es por este motivo que como desarrollador hay que estar muy seguro de que el contrato inteligente que se ha implementado esté libre de errores.

El coste del *Gas* en las transacciones afecta directamente al tiempo máximo para que la transacción sea ejecutada. Por ejemplo, no se puede transferir el saldo total de una cuenta a otra ya que no quedaría saldo suficiente para pagar el *Gas* de la transacción por lo que la transferencia no ocurriría nunca. Por el contrario, si se establece una tarifa de *Gas* alta, dispuesta a pagar, el tiempo de transacción será más veloz.

Dado que Ethereum tiene un límite de gas, por este motivo los bloques no tienen un tamaño de bloque máximo como ocurre en Bitcoin.

Cuentas de usuario

Las cuentas de usuario en Ethereum tan sólo necesitan de un par de claves de usuario asimétricas. Ethereum hace uso de curvas elípticas para ajustar la velocidad y la seguridad y utiliza cifrados de 256 bits. Dado que los procesadores no pueden representar números tan grandes, éstos están codificado en formato hexadecimal con una longitud de 64 caracteres, igual que ocurre en Bitcoin.

Todas las cuentas de usuario están representadas por una dirección. Esta dirección se genera a partir de las claves de usuario. El proceso es el siguiente:

1. Se genera el hash de la clave pública, a partir del algoritmo hash keccak-256, lo que produce un número de 256 bits.

2. Se descartan los primero 96 bits del hash obtenido, con lo que quedan 160 bits.

3. Se codifican los 160 bits en formato hexadecimal y el resultado es la dirección de cuenta de usuario.

Transacciones

Las transacciones en Ethereum son paquetes de datos firmados por el usuario que pueden contener transferencias de Ether entre cuentas, o invocaciones a un método de un contrato inteligente o implementan un nuevo *smart contract*. Las transacciones se firman mediante un algoritmo de firma digital de curva elíptica (ECDSA -*Elliptic Curve Digital Signature Algorithm*).

El paquete de datos, es decir, el bloque de las transacciones de Ethereum contiene una serie de campos, entre los más relevantes:

- La dirección del destinatario de la transacción.

- La firma digital que identifica al remitente y valida el contenido de la transacción.

- Un campo [VALUE] que especifica la cantidad de wei a transferir a la cuenta de usuario destino si se trata de una transferencia o si la transacción requiere de algún tipo de pago. Wei es la unidad indivisible de base del Ether. 1 Ether equivale a $1 \cdot 10^{18}$ wei.

- Un campo [STARTGAS] que representa el número máximo de pasos computacionales que la transacción necesita.

- Un campo [GASPRICE] que especifica la tarifa dispuesta a pagar por el emisor de la transacción.

- Un campo *nonce* de 64 bits, de valor sin sentido que se ajusta para encontrar la solución al PoW.

- Un campo con una marca de tiempo (*timestamp*) que indica el tiempo en el que se inició el bloque.

- Un campo -opcional- con el código del *smat contract* si se trata de subirlo al *Blockchain*.

- Un campo -opcional- [DATA] en caso de invocar a algún método de un *smart contract*.

- Un campo de dificultad, se trata de un valor correspondiente al nivel de dificultad de este bloque. Esto se puede calcular a partir del nivel de dificultad del bloque anterior y la fecha y hora

.

Proof-of-Work

En Ethereum, al contrario que en Bitcoin, todos los nodos de la red contienen una copia entera del *Blockchain*. Para garantizar que ningún nodo pueda alterar alguno de los bloques del *Blockchain* o para llevar a cabo la verificación de los nuevos bloques o si más no, para decidir qué dos bloques válidos distintos agregar al *Blockchain*, Ethereum utiliza el protocolo de consenso PoW (*Proof-of-Work*).

No todas las DApps que se suben al *Blockchain* implementan el mismo conjunto de algoritmos, por lo que a la hora de hallar la solución al PoW, ésta puede diferir en tiempo y coste computacional. Cualquier usuario de la red Ethereum puede convertirse en minero y ser recompensado con cinco ethers por cada nuevo bloque agregado al *Blockchain*, además de todas las comisiones de las transacciones incluidas en el bloque.

La potencia de procesamiento no está directamente relacionada con la capacidad de minado ya que los parámetros empleados para hallar la solución al PoW no son los mismos para todos los mineros. Esto se debe a que el hash del bloque que los mineros extraen, es diferente para cada minero, dado que el hash depende de parámetros como la marca de tiempo, la dirección del minero, etc., y es poco probable que sea igual para todos. Por lo tanto, no se trata de una carrera por ser el más rápido en resolver el rompecabezas sino que es más una cuestión de lotería. Disponer de más potencia de procesamiento significa tener más papeletas de lotería.

Al igual que en Bitcoin, no existe un número máximo de bloques que pueda contener el *Blockchain* y una característica importante respecto al Bitcoin es que en Ethereum no hay límite de las unidades de ether que se pueden acuñar.

Cada vez que un minero genera un bloque válido, lo transmite al resto de nodos de la red. Los bloques contienen un encabezado y un conjunto de transacciones. Cada bloque del *Blockchain*, contiene el hash del bloque anterior, creando así una cadena conectada.

Una vez el minero procesa las transacciones no minadas, las filtra para descartar las que no son válidas. Se considera que una transacción es válida cuando está debidamente firmada digitalmente por el usuario y dispone de saldo suficiente en su cuenta para realizar la transacción.

La solución al PoW es muy parecido al de Bitcoin, se trata de hallar un valor de *nonce* que añadido al hash del bloque, genere un hash de valor menor a un objetivo, para ello Ethereum utiliza el algoritmo de hash ethash. La única manera de encontrar el *nonce* es mediante iteraciones hasta dar con el valor deseado, un resultado de 64 bits que se calcula en función de varios factores como ya se ha hecho referencia.

La forma en que Ethereum calcula el nivel de dificultad para hallar el *nonce* es el siguiente:

1. Se calcula la diferencia entre el tiempo de formación del bloque padre (bloque anterior) y el bloque actual.

2. El resultado anterior se divide por 10 y se almacena sólo el entero. Esto permite la creación de rangos, de forma que si el resultado anterior está entre 1 y 9 se asigna el valor 0, si el resultado anterior está entre 10 y 19 se asigna el valor 1 y así sucesivamente.

3. Para crear tres rangos, se resta 1 a los valores asignados anteriormente. Los tres rangos serán $[-ve,\ 0,\ +ve]$. Así pues, cuando la salida del paso 1 esté entre 0 y 9, el rango será $-ve$, cuando la salida esté entre 10 y 19 será 0 y cuando la salida esté entre 20 y 29 será $+ve$.

4. A continuación se compara el valor del rango anterior con -99, si es menor se adopta el valor -99 de lo contrario se deja tal cual.

5. Se divide la dificultad del bloque padre por el divisor 2048.

6. El valor obtenido en el paso 4 se multiplica por el del paso 5. Con esto se obtiene la dificultad del nuevo bloque con el antiguo bloque padre.

7. Se agrega la salida del paso 6 a la dificultad principal y el resultado será la dificultad del nuevo bloque.

8. Se verifica que la dificultad calculada sea al menos mayor que el valor umbral mínimo de 131072.

9. Antes de asignar la dificultad se verifica si el número del bloque es superior a 200.000, si lo es, entonces se aplica la lógica de la “Bomba” para aumentar la dificultad de forma exponencial.

10. Para incrementar la dificultad de manera exponencialmente, el nuevo número de bloque se calcula sumando uno al número de bloque padre.

11. El nuevo número de bloque se divide por 100.000.

12. Si el nuevo número de bloque sigue siendo superior a 200.000, a la salida del paso 11 se le resta 2.

13. La dificultad exponencialmente delta se calcula mediante 2 ^ (salida del paso 12).

14. Finalmente la nueva dificultad se calcula sumando la salida del paso anterior a la dificultad calculada en el paso 7.

Así es como Ethereum mantiene una diferencia de tiempo de minería entre bloques de 10 - 19 segundos, bastante más rápido que los 10 minutos de Bitcoin. Si observamos el paso 2, se deduce que la división por 10 ayuda a crear los tres rangos, de forma que si el valor cae en el primer rango, la dificultad aumenta, si cae en el segundo rango la dificultad se mantiene y si cae en el tercero la dificultad se reduce. Si se desea cambiar los rangos de dificultad basta con cambiar el valor a dividir del paso 2.

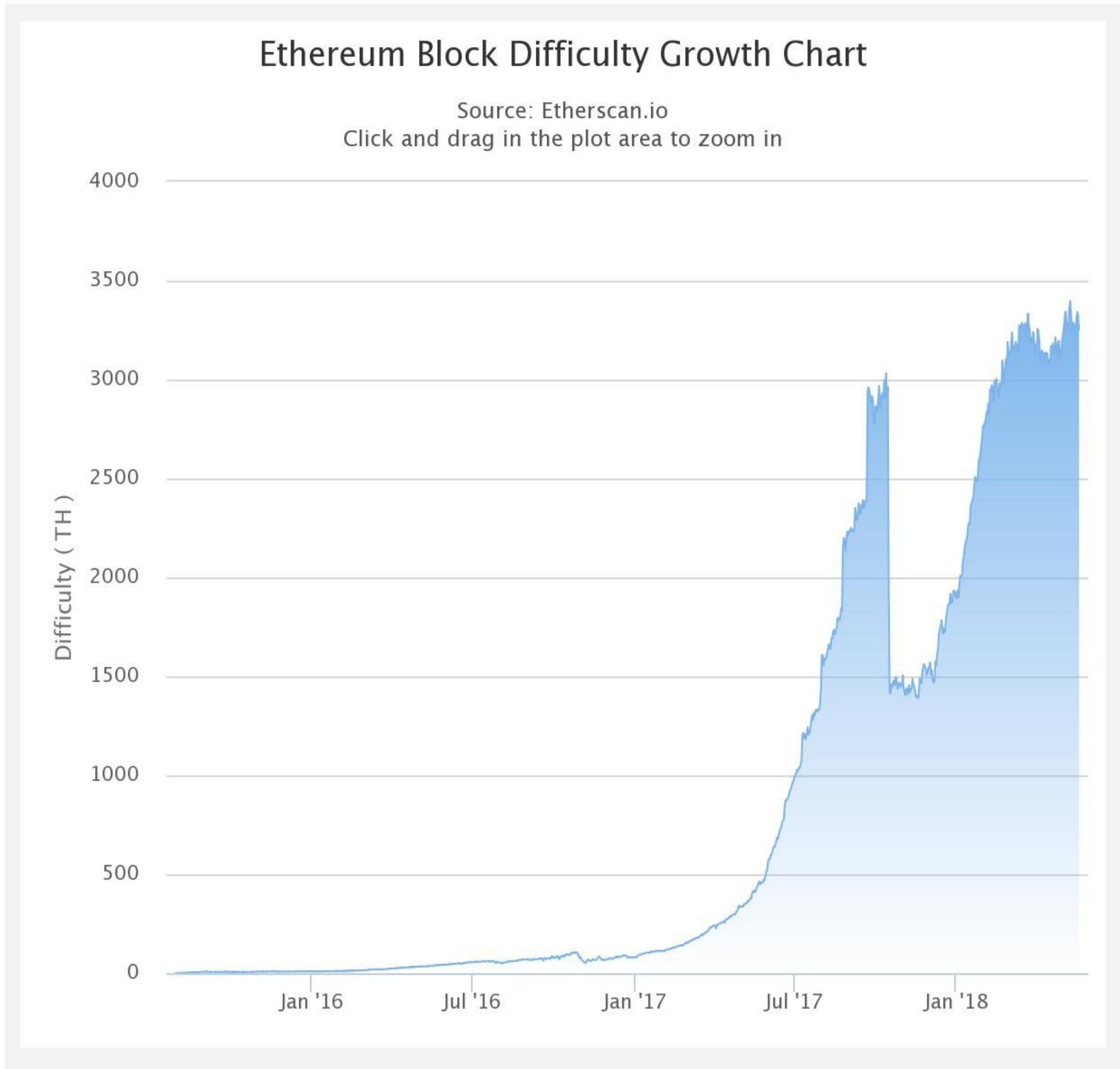


Imagen 21. Histórico de la dificultad en los bloques de Ethereum

Forking

Cuando existe un conflicto entre quienes gestionan los nodos de la red con respecto a la validez del *Blockchain*, provoca que se generen dos *Blockchains* en la misma red y que los mineros se divididan respecto a qué *Blockchain* validar, es lo que se llama un *fork*. Normalmente cuando un nodo de Ethereum recibe dos *Blockchains* válidos distintos, da por válido aquel que acumule un nivel de dificultad combinada mayor de todos los bloques.

Existen tres tipos de *fork*, *regulars*, *soft* y *hard*. Un *regular fork* ocurre cuando el conflicto es temporal, dos mineros consiguen validar un nuevo bloque prácticamente al mismo tiempo. La resolución del conflicto se realiza midiendo la dificultad de cada bloque y validando aquel que tenga un valor de dificultad mayor.

Cuando se produce un *soft fork* en numerosas ocasiones suele nacer una nueva plataforma *Blockchain* con su propia criptomoneda. Este tipo de *forks* sucede cuando la actualización del protocolo implica que el nuevo sistema crea una bifurcación del *Blockchain* y obligando a aquellos mineros que sumen más del 50% del *hash power* a tener que actualizarse. Aquellos que no quieran actualizarse quedan fuera de la red. *Hash power* es una unidad de medida que representa cuánta energía está consumiendo la red para estar en continuo funcionamiento.

En caso de que todos los mineros deban actualizarse, entonces estamos ante un *hard fork*. Un "*hard fork*" es un cambio de reglas en el software que valida los bloques lo que hace que el viejo sistema invalide los nuevos bloques por agregar al *Blockchain*.

Tanto Bitcoin como Ethereum han sufrido diversos *forks*.

Logo	Fork Name	Fork Symbol	Blockchain	Fork Date	Fork Block	Coin Distribution	Status
	Ether Inc	ETI	Ethereum	Monday, March 12, 2018	5078585	1 ETH = 1 ETI	Forked
	EtherZero	ETZ	Ethereum	Friday, January 19, 2018	4936270	1 ETH = 1 ETZ	Forked
	EthereumFog	ETF	Ethereum	Monday, January 01, 2018	4730999	1 ETH = 1 ETF	Forked
	Ethereum Modification	EMO	Ethereum	Friday, December 15, 2017	4730666	1 ETH = 1 EMO	Forked
	EtherGold	ETG	Ethereum	Thursday, December 14, 2017	4730666	1 ETH = 1 ETG	Forked

Imagen 22. Forks de Ethereum

Logo	Fork Name	Fork Symbol	Blockchain	Fork Date	Fork Block	Coin Distribution	Status
	ClassicBitcoin	CBTC	Bitcoin	Sunday, April 01, 2018	516095	1 BTC = 10000 CBTC	Forked
No Logo	Bitcoin Lite	BTCL	Bitcoin	Tuesday, January 30, 2018	0	1 BTC = 1 BTCL	Forked
	Bitcoin Atom	BCA	Bitcoin	Wednesday, January 24, 2018	505888	1 BTC = 1 BCA	Forked
	Bitcoin Interest	BCI	Bitcoin	Monday, January 22, 2018	505083	1 BTC = 1 BCI	Forked

Imagen 23. Forks de Bitcoin

5. Desarrollo DApp

Arquitectura DApp en Ethereum

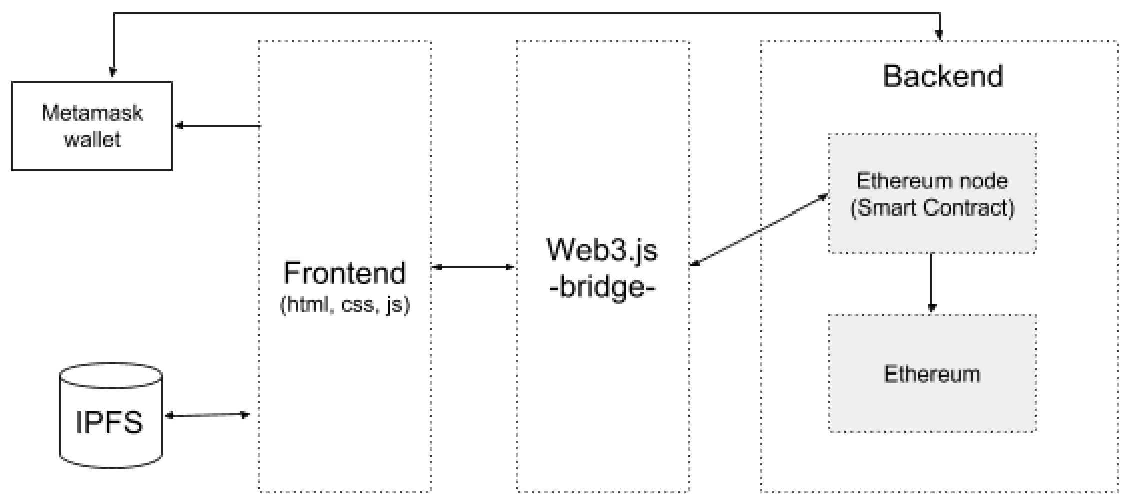


Imagen 24. Arquitectura de una DApp en Ethereum

Capa de aplicación - Backend

Los contratos inteligentes de Ethereum residen en el *Blockchain* en formato específico de Ethereum, en *bytecode* que es ejecutado por la *Ethereum Virtual Machine* (EVM). Éstos pueden ser escritos en distintos lenguajes de programación de alto nivel como Solidity, Serpent o Lisp Like Languaje (LLL). Los *smart contract*s están replicados en cada uno de los nodos del *Blockchain* de la red P2P y representan la capa de aplicación en la arquitectura multicapa de las DApps en Ethereum.

Capa de presentación - Frontend

El *frontend* es el *User Interface* (UI) de arquitectura cliente-servidor es la capa de presentación en la arquitectura multicapa de las DApps de Ethereum. Se comunica con el *backend a través* de la API (*Application Programming Interface*) Web3.js de JavaScript que implementa la especificación genérica JSON RPC. El UI puede estar implementado en multitud de web *frameworks* (HTML, Angular, PHP, etc…).

Capa de acceso - Almacén de datos

Los datos que se envían a los *smart contract*s son almacenados en las propias transacciones de Ethereum, es decir, en el *Blockchain*. No obstante, existen alternativas para el almacenamiento de datos fuera de la cadena mediante *InterPlanetary File System* (IPFS). Se trata de un protocolo diseñado dirigido a crear un método descentralizado para el almacenamiento y la compartición de archivos.

Entorno de trabajo

Para la implementación de la siguiente DApp se hará uso de:

Interface Development Environment (IDE)

- Remix [https://remix.ethereum.org]: Se trata del IDE creado por el equipo de desarrollo de Ethereum. Se puede instalar localmente o usarlo *online*. Permite escribir, compilar y subir *smart contracts* al *Blockchain* de Ethereum. Además, proporciona herramientas de depuración del código y para interactuar con el código una vez está subido al *Blockchain*.

- Sublime text [https://www.sublimetext.com/]: Se trata de un editor de código local. Para el desarrollo de *smart contracts* es necesario instalar el Package control [https://packagecontrol.io/] que permitirá la instalación del paquete Ethereum [https://packagecontrol.io/packages/Ethereum]. Sublime, además nos permitirá editar ficheros html, css, js, etc...

Cliente Ethereum

- Ganache-cli [https://github.com/trufflesuite/ganache-cli]: Ganache CLI forma parte del paquete de herramientas de desarrollo de Ethereum Truffle [http://truffleframework.com/], se trata de la versión de la consola de comandos de Ganache. Ganache CLI utiliza ethereumjs [https://github.com/ethereumjs]para simular el comportamiento completo del cliente.

- Node.js [https://nodejs.org/es/]: es una librería y entorno de ejecución de E/S dirigida por eventos y por lo tanto asíncrona que se ejecuta sobre el intérprete de JavaScript.

- npm [https://www.npmjs.com/] : *Node Package Manager* (npm) es un gestor de paquetes que facilita el trabajo a la hora de trabajar con Node. Proporciona un gran número de librerías disponibles de manera sencilla con sólo un comando. Ayuda a administrar módulos, distribuir paquetes y agregar dependencias de una manera sencilla.

- Web3.js [https://github.com/ethereum/web3.js/] : Se trata de la API de JavaScript compatible con Ethereum que implementa la especificación Generic JSON RPC.

- Metamask [https://metamask.io/] : Metamask es una extensión para Google Chrome, Firefox y Brave que además de ofrecer un *wallet* para Ethereum, proporciona acceso a las redes de prueba de Ethereum para el desarrollo de DApps. Además, en cada página del UI, inyecta la librería web3.

Servidor UI

- Lite-Server [https://github.com/johnpapa/lite-server]: Se trata de un servidor de nodo para aplicaciones web, abre la aplicación directamente en el buscador, actualiza la aplicación cuando cambia el html o el javascript e inyecta cambios de CSS usando *sockets*.

Diseño del sistema

Introducción de la DApp

El objetivo de la DApp a implementar pretende mostrar las etapas que debe seguirse a la hora de desarrollar una solución completa en Ethereum y mostrar el funcionamiento de ésta dentro de la plataforma Ethereum.

En concreto se ha implementado tanto el *frontend* como el *backend* de una DApp. La DApp es una solución sencilla para el cálculo y pago automático, en ethers, del salario del trabajador, en función de las horas que haya invertido. El trabajador podría verificar sus horas mediante cualquier sistema biométrico (huella dactilar, reconocimiento facial, etc) o una *card* ID para alimentar los datos del *frontend*. Sin embargo, en el presente trabajo y para que sirva a modo ilustrativo, se alimentarán los distintos campos del *frontend* a través de una UI manual que también podría ser válida en un entorno real si el trabajador desempeña el trabajo físicamente en la empresa. Obviamente la DApp puede completarse con muchas más prestaciones como por ejemplo la gestión de los impuestos a deducir del salario por parte de la empresa, el cálculo del nivel de productividad del trabajador, estadísticas de rendimiento entre los distintos trabajadores, funcionalidades para el cobro por parte de la empresa a sus clientes y así depositar dinero en el *smart contract* para el pago de los trabajadores, etc. Dichas prestaciones quedan fuera del alcance del presente trabajo y sólo se pretende mostrar las fases necesarias para el objetivo propuesto.

La DApp que se presenta podría ser implantada, por ejemplo, en aquellas empresas que contratan distintos perfiles profesionales autónomos y que trabajan a distancia o de manera local, para completar las distintas fases de un proyecto. En función de la tarea que desempeñe el trabajador y acorde a un precio establecido de la hora trabajada, la DApp ingresará de manera automática el salario en la cuenta de usuario de Ethereum del trabajador. Así mismo la DApp, permitirá a la empresa depositar fondos en el contrato inteligente y así garantizar un balance suficiente para realizar los pagos a los distintos trabajadores.

Arquitectura del sistema

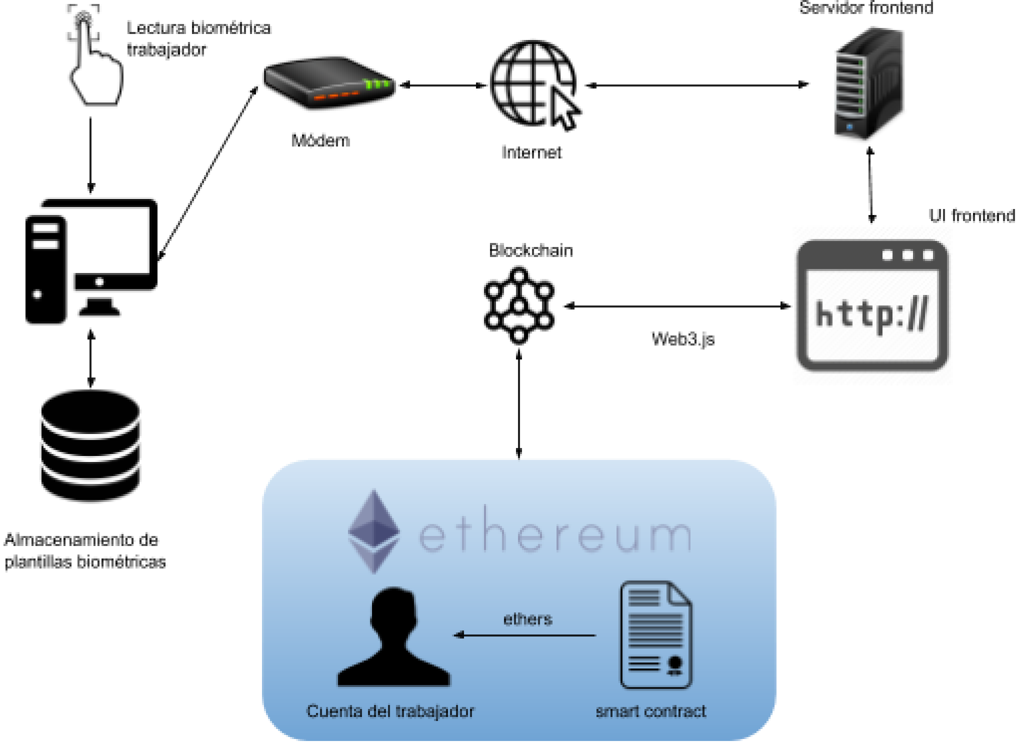


Imagen 25. Arquitectura del sistema

Diseño del programa

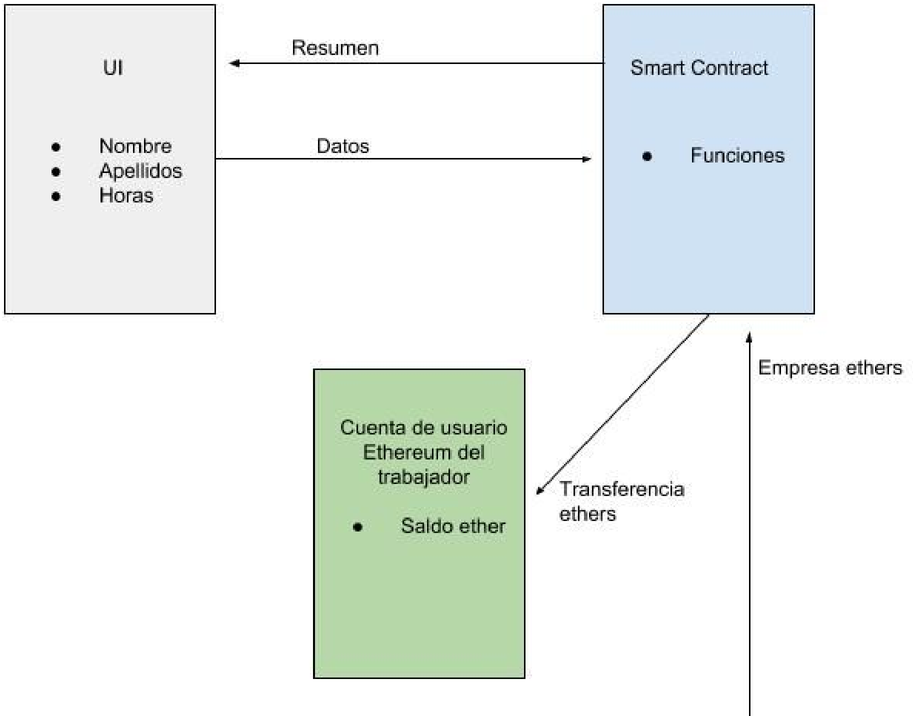


Imagen 26. Estructura de datos de la DApp

La DApp consta de:

Frontend

Esta es la UI que recoge los datos del trabajador. Como se ha mencionado antes, los datos pueden ser proporcionados por cualquier otro sistema de reconocimiento biométrico. Básicamente los datos de interés son:

- Nombre del trabajador
- Apellidos de trabajador
- Horas trabajadas

Backend

Esta es la parte del *smart contract*. Se implementa en lenguaje Solidity y debe contemplar:

- Estructura de datos que represente al perfil del trabajador.
- Una función para guardar los datos de los distintos trabajadores.
- Una función para depositar dinero en el contrato inteligente.

- Un función para calcular el sueldo a pagar y descontarlo del balance total de dinero que el contrato inteligente pueda tener.
- Una función para transferir el sueldo del trabajador a su cuenta de usuario de Etherum.
- Una función para saber qué balance queda en el *smart contract.*
- Una función para contabilizar el número de trabajadores guardados.
- Una función para obtener las direcciones de las cuentas de usuario de los trabajadores.
- Una función que devuelva al UI la confirmación de los datos introducidos por el trabajador.

Codificación

Backend

Todos los *smart contracts* escritos en Solidity deben empezar con la línea que define la versión del propio lenguaje para a continuación definir el nombre del contrato

```
pragma solidity ^0.4.24;

contract sueldo {
```

A continuación definimos la estructura de datos que contendrá la información del empleado.

```
//Estrucutra de datos de los empleados
struct perfilEmpleado {
    address direccion;//dirección de cuenta de usuario de Ethereum
    string nombre;//nombre del empleado
    string apellidos;//apellidos del empleado
    uint horas; //horas trabajadas que registra el empleado
    uint  aPagar;//sueldo que recibe por las horas trabajadas
    uint totalPagado;//total que le han pagado del total de veces contratado
}
```

Las variables nombre y apellidos, para las pruebas locales se definen de tipo string pero para las pruebas online en la red Ropsten se deben pasar a `bytes16` ya que las variables de tipo `string` tienen un coste de gas más alto al ser procesadas. Las variables de tipo dirección tienen un valor de 20 bytes y representan una dirección de la cuenta de usuario de Ethereum.

Como se ha mencionado anteriormente, para escalar la aplicación y ofrecer otras prestaciones, cualquier otro dato relevante relativo al trabajador iría en esta estructura de datos. Solidity admite distintos tipos de variables de tipo: `int/uint, bool, address, bytes, string, mapping, struct`.

A continuación declaramos una variable de tipo `uint` con visibilidad interna que nos permita establecer un precio en wei de la hora trabajada. En el ejemplo que mostramos, la variable será una constante pero si la DApp fuera además de lo presentado, una plataforma de subastas de proyectos, donde la empresa oferta un determinado trabajo y los trabajadores potenciales se inscriben a la oferta, esta variable pasaría a se implementada en una función setter para que cada trabajador potencial la pudiese establecer.

```
//Precio hora de trabajo
uint internal precioHora = 1000000000000000000;

//@dev returns el sueldo total del trabajador en función de las horas trabajadas
//@ arg horas trabajadas
function sueldoEmpleado(uint _horas) internal view returns (uint aPagar){
    uint totalEther = (_horas*precioHora)/1000000000000000000;
    return totalEther;
}
```

La función `sueldoEmpleado()` de visibilidad interna devuelve el salario del trabajador en ethers. En los contratos, por defecto las unidades de la criptomoneda de Ethereum son en wei.

A continuación declaramos una variable de tipo mapping para poder almacenar los datos de los empleados que hemos definidos en la estructura de datos anterior. Los mappings en Solidity son parecidas a las tablas hash de java. Se declaran como un mapeo `(_KeyType => _ValueType)` que se inicializan virtualmente, de manera que cada clave posible, existe y se le asigna un valor por defecto cuya representación en bytes son todo ceros. En Solidity, los datos clave no se almacenan realmente y sólo es el hash que se usa para buscar el valor.

```
//Variable para guardar la estruc. de datos de los empleados
mapping (address => perfilEmpleado) internal listaEmpleados;
//Variable de tipo array para guardar direcciones de Ethereum de los empleados
address[] cuentaEmpleados;
```

La variable de tipo mapping, `listaEmpleados` acepta primero el valor clave que será de tipo dirección, la del usuario de Ethereum del trabajador y luego el tipo valor que será la estructura de datos del trabajador. Con el valor clave del mapping, permite buscar a un trabajador específico y recuperar sus datos.

La siguiente línea define un *array* de direcciones para almacenar todas las direcciones de las cuentas de usuario de Ethereum de los trabajadores. Esto es necesario ya que actualmente la máquina virtual de Ethereum no permite devolver directamente sólo las direcciones del mapping.

```
//Evento paa disparar eventos en el UI
event fichar(
    string nombre,
    string apellidos,
    uint horas,
    uint aPagar,
    uint totalPagado
);
```

En Solidity, existe los `event()`, se tratan de señales enviadas por los contratos inteligentes para disparar eventos. Las DApps conectadas a Ethereum pueden escuchar estos eventos y actuar en consecuencia. Es decir, la finalidad de los `event()` es permitir la devolución de llamadas javascript en un UI, lo que permite ejecutar cierto código en función de si el evento tuvo o no éxito y poder pasar desde el *smartcontract*, valores que pueden ser utilizados en javascript.

La primera de las funciones a implementar es un setter para cuando se pasen los datos desde el UI al *smartcontract*. En cuestión de trata de la función set de los empleados.

```
//@dev Establece los datos de un empleado
//@arg _direccion, _nombre, _apellidos, _horas
/*****Para el ejemplo del TFG los datos se rellenan manualmente como si el propio empleado
/*****fuese quien rellenase los campos. No obstante esta función podría estar alimentada automáticamente
/*****desde una base de datos del sistema de reconocimiento biométrico****/
function setEmpleado(address _direccion, string _nombre, string _apellidos, uint _horas) public {

    var empleado = listaEmpleados[_direccion]; //Variable de tipo mapping

    //Establecemos el valor de cada variable de la estructura de datos de empleado
    empleado.direccion = _direccion;
    empleado.nombre = _nombre;
    empleado.apellidos = _apellidos;
    empleado.horas = _horas;
    empleado.aPagar = sueldoEmpleado(_horas);
    //Acumula el total de lo que el empleado ha recibido de la empresa
    empleado.totalPagado = totalPagadoAcumulado(empleado.direccion, empleado.aPagar, empleado.totalPagado);
    //Variable que no se pasa desde el UI por lo que la declaramos dentro del setter
    uint aPagar = empleado.aPagar;
    //Llamamos a la función para actualizar el balance total del contrato y pagar al empleado
    //Si lanza una excepción se revierte toda la transacción del empleado
    updateBalanceAndPay(aPagar, empleado.direccion);
    //guardamos la dirección de usuario en el array
    cuentaEmpleados.push(_direccion) -1;
    //declaramos la variable para poder pasarla al evento
    uint totalPagado = empleado.totalPagado;
    //Llamamos al evento
    emit fichar(_nombre, _apellidos, _horas, aPagar, totalPagado);
}
```

Los argumentos que se pasan a la función, son menos que las variables definidas en la estructura de datos del empleado. Esto se debe a que los argumentos que acepta la función coinciden con los campos a rellenar en el UI.

Solidity proporciona la palabra clave `var` para declarar variables. El tipo de la variable en este caso se decide dinámicamente según el primer valor que se le asigne. Una vez que se ha asignado un valor, el tipo es fijo, por lo que en caso de asignarle otro tipo, se generará una conversión de tipo en la variable.

Declaramos una variable `empleado` de tipo mapping a la que se le pasa la dirección del trabajador como la clave, la clave es lo que servirá para recuperar los datos del empleado que se busque. Seguidamente establecemos las siguientes variables de la estructura de datos del empleado.

```
//Función que calcula y guarda el total de los saldos que ha cobrado un empleado concreto
function totalPagadoAcumulado (address _direccion, uint aPagar, uint _totalPagado) internal returns (uint acumulado){
    uint x = listaEmpleados[_direccion].totalPagado;
    uint y = aPagar;
    return acumulado = x+y;
}
```

La variable `totalPagado` se establece mediante la función `totalPagadoAcumulado()` a la que se le pasan como argumentos las variables previamente establecidas: `direccion`, `aPagar`, y `totalPagado`. La función nos permite recuperar la variable `totalPagado` de un empleado concreto a partir de su dirección de usuario de Ethereum y actualizar el valor en ella de forma que permite acumular el total de los sueldos que el empleado ha cobrado.
La variable `aPagar` se establece mediante la función `sueldoEmpleado()`.

Una vez se han establecido todas las variables del empleado comprobamos mediante la función `updateBalanceAndPay()`, a la que se le pasan dos argumentos: `aPagar` y `direccion` del empleado, si hay suficiente saldo en el contrato y si es así se ordena pagar el sueldo a la cuenta de usuario de Ethereum del empleado. Si no hubiera saldo suficiente, la función devolvería una excepción y se revertiría el estado de todas las variables del empleado cancelando la transacción. Lo único que hay que tener en cuenta es que Ethereum no devuelve el gas del coste de las transacciones, por lo que aunque se revierta el estado del empleado, la transacción se sigue cobrando.

```
//Esta función comprueba el balance total del contrato, si hay menos saldo de lo que hay que
//pagar, lanza una excepción y revierte la transacción del empleado.
//El gas de la transacción no se recupera
function updateBalanceAndPay(uint _aPagar, address _direccion) internal {
    //Descontamos el sueldo del empleado del balance total del contrato
    uint balance  = address(this).balance;
    require (balance > _aPagar);//Si hay menos balance se revierte la transacción a partir de aquí
    uint subTotal = address(this).balance - _aPagar;
    uint totalPagar = (balance - subTotal)*1000000000000000000;
    _direccion.transfer(totalPagar);
    //transferimos en ether el sueldo la cuenta de usuario de Ethereum del empleado
}
```

Finalmente guardamos la dirección del empleado en el array de direcciones `cuentaEmpleados` que nos permitirá contabilizar el número de empleados o simplemente listas las direcciones de los empleados.

Cuando el empleado interactúa con la UI y genera una transacción con el *smart contract*, internamente, el contrato inteligente obtiene la dirección de usuario a partir de la transacción. Es por ello que en la UI no veremos ningún campo donde el empleado deba introducir su número de cuenta de Ethereum sino que deberá estar logeado en su cliente Ethereum mediante Metamask.

La siguiente línea es la llamada al evento creado anteriormente en la que se le pasan cinco variables como argumentos, las definidas en la estructura de datos del empleado. Cuando en el UI se haga click sobre el botón ‘fichar’, se creará un evento a través de javascript que nos servirá para esconder o mostrar un gráfico de carga de datos.

La siguiente función es un getter que permite recuperar la información de cada empleado. En la UI, cuando el empleado complete los distintos campos y envíe los datos al *smart contract*, si la transacción se ha realizado con éxito verá en la pantalla los datos que ha enviado y el sueldo que ha ganado.

```
//Función que recupera lso datos de un empleado
function getEmpleado(address _direccion)  view public returns (string nombre, string apellidos, uint horas, uint aPagar, uint totalPagado)
    return(listaEmpleados[_direccion].nombre,
           listaEmpleados[_direccion].apellidos,
           listaEmpleados[_direccion].horas,
           listaEmpleados[_direccion].aPagar,
           listaEmpleados[_direccion].totalPagado);
```

La función `deposit()` permite depositar fondos en el contrato inteligente, en la DApp implementada se depositan los fondos de forma manual por el dueño del contrato, es decir, por la empresa. Pero se podría implementar una función para que de manera automática se pudiera cobrar a los clientes de la empresa a través de otro contrato inteligente y depositar si no toda la cantidad cobrada, parte de ella en el *smart contract* para garantizar fondos suficientes para pagar a los empleados.

```
    //Función que deposita fondos en el contrato
    function deposit() payable public {
        // nothing to do!
     }
    //Función para recuperar el balance de fondos del contrato
     function getBalance() public view returns (uint256) {
         return address(this).balance;
     }
     //Función para recuperar las direcciones de los empleados guardadas
    function getDireccionesEmpleados() view  public returns (address[]) {
        return cuentaEmpleados;
    }
    //Función para saber el número total de empleados guardados
     function numeroEmpleados() view public returns (uint) {
        return cuentaEmpleados.length;
     }
}
```

La siguiente función, `getBalance()` que se observa es la que permite depositar fondos en el contrato mientras que la segunda función nos devuelve un balance de los fondos que hay en el contrato.

Las últimas dos funciones implementadas nos permiten recuperar datos concretos como por ejemplo; la primera de ellas, devuelve las direcciones de usuario de Ethereum de los empleados guardados en el array `cuentaEmpleados` y la segunda función, devuelve el número total de empleados que hay guardados en el *smart contract*.

En Solidity es posible determinar el nivel de visibilidad de las variables o funciones frente a otras funciones dentro o fuera del contrato inteligente, existen cuatro tipos de visibilidades tanto para funciones como para variables:

Público: Permite definir funciones o variables que pueden llamarse internamente o mediante mensajes.

Privado: Las variables y funciones privadas sólo están disponibles en el contrato actual y no para los contratos derivados, es decir otro contrato distinto que el contrato actual puede heredar.

Interna: Las funciones y variables sólo son accesibles de forma interna en el contrato actual o por otro contrato.

Externo: Las funciones se pueden invocar desde otros contratos y transacciones. No se pueden llamar internamente, sino es mediante la llamada "this.functionName ()".

El código final (sueldo.sol) es el siguiente:

```
pragma solidity ^0.4.24;

contract sueldo {

    //Estrucutra de datos de los empleados
    struct perfilEmpleado {
        address direccion;//dirección de cuenta de usuario de Ethereum
        bytes16 nombre;//nombre del empleado
        bytes16 apellidos;//apellidos del empleado
        uint horas; //horas trabajadas que registra el empleado
        uint  aPagar;//sueldo que recibe por las horas trabajadas
        uint totalPagado;//total que le han pagado del total de veces
contratado
    }

    //Precio hora de trabajo
    uint internal precioHora = 1000000000000000000;

    //@dev returns el sueldo total del trabajador en función de las horas
trabajadas
    //@ arg horas trabajadas
    function sueldoEmpleado(uint _horas) internal view returns (uint
aPagar){
        uint totalEther = (_horas*precioHora)/1000000000000000000;
        return totalEther;
    }
    //Evento paa disparar eventos en el UI
    event fichar(
        bytes16 nombre,
        bytes16 apellidos,
        uint horas,
        uint aPagar,
        uint totalPagado,
        address direccion
    );

    //Variable para guardar la estruc. de datos de los empleados
    mapping (address => perfilEmpleado) internal listaEmpleados;
    //Variable de tipo array para guardar direcciones de Ethereum de los
empleados
    address[] cuentaEmpleados;

    //@dev Establece los datos de un empleado
    //@arg _direccion, _nombre, _apellidos, _horas
    /*****Para el ejemplo del TFG los datos se rellenan manualmente como si
el propio empleado
    /*****fuese quien rellenase los campos. No obstante esta función podría
```

```
estar alimentada automáticamente
    /*****desde una base de datos del sistema de reconocimiento
biométrico****/
    function setEmpleado(address _direccion, bytes16 _nombre, bytes16
_apellidos, uint _horas) public {

        var empleado = listaEmpleados[_direccion]; //Variable de tipo
mapping

        //Establecemos el valor de cada variable de la estructura de datos
de empleado
        empleado.direccion = _direccion;
        empleado.nombre = _nombre;
        empleado.apellidos = _apellidos;
        empleado.horas = _horas;
        empleado.aPagar = sueldoEmpleado(_horas);
        //Acumula el total de lo que el empleado ha recibido de la empresa
        empleado.totalPagado = totalPagadoAcumulado(empleado.direccion,
empleado.aPagar, empleado.totalPagado);
        //Variable que no se pasa desde el UI por lo que la declaramos
dentro del setter
        uint aPagar = empleado.aPagar;
        //Llamamos a la función para actualizar el balance total del
contrato y pagar al empleado
        //Si lanza una excepción se revierte toda la transacción del
empleado
        updateBalanceAndPay(aPagar, empleado.direccion);
        //guardamos la dirección de usuario en el array
        cuentaEmpleados.push(_direccion) -1;
        //declaramos la variable para poder pasarla al evento
        uint totalPagado = empleado.totalPagado;
        //Llamamos al evento
        emit fichar(_nombre, _apellidos, _horas, aPagar, totalPagado,
_direccion);
   }

    //Esta función comprueba el balance total del contrato, si hay menos
saldo de lo que hay que
    //pagar, lanza una excepción y revierte la transacción del empleado.
    //El gas de la transacción no se recupera
    function updateBalanceAndPay(uint _aPagar, address _direccion) internal
{
        //Descontamos el sueldo del empleado del balance total del contrato
        uint balance  = address(this).balance;
        require (balance > _aPagar);//Si hay menos balance se revierte la
transacción a partir de aquí
        uint subTotal = address(this).balance - _aPagar;
        uint totalPagar = (balance - subTotal)*1000000000000000000;
        _direccion.transfer(totalPagar);
        //transferimos en ether el sueldo la cuenta de usuario de Ethereum
del empleado
    }
    //Función que calcula y guarda el total de los saldos que ha cobrado un
```

```
empleado concreto
    function totalPagadoAcumulado (address _direccion, uint aPagar, uint
_totalPagado) internal returns (uint acumulado){
        uint x = listaEmpleados[_direccion].totalPagado;
        uint y = aPagar;
        return acumulado = x+y;
    }

   //Función que recupera lso datos de un empleado
   function getEmpleado(address _direccion)  view public returns (bytes16
nombre, bytes16 apellidos, uint horas, uint aPagar, uint totalPagado,
address direccion) {
        return(listaEmpleados[_direccion].nombre,
                listaEmpleados[_direccion].apellidos,
                listaEmpleados[_direccion].horas,
                listaEmpleados[_direccion].aPagar,
                listaEmpleados[_direccion].totalPagado,
                listaEmpleados[_direccion].direccion);
   }
   //Función que deposita fondos en el contrato
   function deposit() payable public {
        // nothing to do!
    }
   //Función para recuperar el balance de fondos del contrato
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
    //Función para recuperar las direcciones de los empleados guardadas
   function getDireccionesEmpleados() view  public returns (address[]) {
       return cuentaEmpleados;
   }
   //Función para saber el número total de empleados guardados
    function numeroEmpleados() view public returns (uint) {
       return cuentaEmpleados.length;
    }
}
```

Frontend UI (User Interface)

El código html del UI es bastante explicativo por sí solo, no obstante comentaremos la parte referente a los eventos en la que se ha utilizado jQuery. En Solidity se ha creado un evento para que cuando se establezcan las variables de un empleado ocurra cierta acción, en cuestión la aparición de un gráfico que muestra el proceso de carga de los datos al *smart contract*.

El siguiente código es el encargado de pasar a la función `setEmpleado()` del *smart contract* los datos del empleado, además se observa que si no hay error alguno que devuelva el contrato, se muestra un 'loader', el gráfico de progreso de carga. Si hay algún error, el 'loader' se esconde y se muestra por consola del explorador un mensaje.

```
$("#button").click(function() {
            $("#loader").show();
            sueldo.setEmpleado(web3.eth.defaultAccount, $("#nombre").val(),
$("#apellidos").val(), $("#horas").val(), (err, res) =>{
                if(err){
                    $("#loader").hide();
                    console.log('Ups!! algo no funciona como es debido');
                }
            });
        });
```

Creamos una variable para referenciar el evento, `ficharEvent` a la que le pasamos un objeto vacío para el primer parámetros y 'latest' como segundo. 'Latest' hace referencia a la última dirección de bloque. A continuación se hace uso del método `watch()` sobre la variable `ficharEvent` para pasar la función de llamada con los parámetros de error y resultado. En caso de que la llamada no devuelva error y que la dirección del bloque donde se ha añadido la transacción sea nueva, escondemos el gráfico loader.

Lo siguiente es mostrar un resumen de los datos del empleado que ha enviado al *smart contract*, dado que las variables de nombre y apellidos en Solidity las habíamos declarado de tipo bytes16, aquí casteamos las variables a AScii.

```
var ficharEvent = sueldo.fichar({}, 'latest');
        //Activamos el gráfico del loader para mostrar el tiempo de
transacción
        ficharEvent.watch(function(error, result){
            if (!error)
                {   //Si la dirección del bloque es nueva, escondemos el
loader
                    if(result.blockHash != $("#hashBlockAddress").html())
                        $("#loader").hide();

                    $("#hashBlockAddress").html('Dirección Block Hash: '+
result.blockHash);
                    //Imprimimos un resumen de los datos enviados a la vez
que casteamos de byte16 a Ascii
                    $("#resumen").html(web3.toAscii(result.args.nombre)+'
'+web3.toAscii(result.args.apellidos) +' / '+(result.args.horas)+' horas /
'+(result.args.aPagar)+' ether / Se te ha pagado un total de ' +
(result.args.totalPagado)+ ' ethers');
                } else {
                    $("#loader").hide();
                    console.log(error);
                }
        });
```

index.html

```
<!-- Cabecera HTML -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <!-- Referencia al archivo de estilos css -->
    <link rel="stylesheet" type="text/css" href="main.css">
    <!-- Referencia al módulo web3 a utilizar guardado en la carpeta de
módulos -->
    <script src="./node_modules/web3/dist/web3.min.js"></script>
</head>

<!-- Cuerpo de la UI -->
<body>
    <div class="container">

        <h1>Sueldo empleado</h1>
        <!-- Imprime el número de empleados guardados en el smart contract
(sc) -->
        <span id="contadorEmpleados"></span>
        <!-- Imprime un resumen de los datos enviados al sc -->
        <h2 id="resumen"></h2>
        <!-- Imprime la dirección hash del bloque del Blockchain en el que
se ha guardado la transacción -->
        <span id="hashBlockAddress"></span>
        <hr>

        <!-- carga un gráfico de progreso de la transacción -->
        <img id="loader"
src="https://loading.io/spinners/double-ring/lg.double-ring-spinner.gif">
        <!-- Campo para el nombre del trabajador -->
        <label for="name" class="col-lg-2 control-label">Nombre
empleado</label>
        <input id="nombre" type="text">
        <!-- Campo para el apellido del trabajador -->
        <label for="name" class="col-lg-2 control-label">Apellidos
empleado</label>
        <input id="apellidos" type="text">
        <!-- Campo para el número de horas que el trabajador ha invertido
-->
        <label for="name" class="col-lg-2 control-label">Horas de
jornada</label>
        <input id="horas" type="uint256">
        <!-- Botón para iniciar la transacción y enviar los datos al sc -->
        <button id="button">Fichar</button>

    </div>
    <!-- Uso de jQuery  -->
    <script
```

```
src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>

    <script>
        /* Al clicar el botón fichar se ejecuta el siguiente código*/
       if (typeof web3 !== 'undefined') {//Si se usa un cliente Ethereum
como Metamask
            web3 = new Web3(web3.currentProvider);
        } else {
            // Proveedor de Web3 local
            web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
        }

        //Cuenta de usuario por defecto a utilizar en entorno de pruebas
local
        web3.eth.defaultAccount = web3.eth.accounts[0];

        //Se almacena el ABI del smart contract una vez compilado
        var sueldoContract = web3.eth.contract('ABI aquí');

        // Se almacena la direcci'n del smart contract que se ha subido al
Blockchain
        var sueldo = sueldoContract.at(' dirección del contrato aquí');
        console.log(sueldo);

        var ficharEvent = sueldo.fichar({}, 'latest');
        //Activamos el gráfico del loader para mostrar el tiempo de
transacción
        ficharEvent.watch(function(error, result){
            if (!error)
                {   //Si la dirección del bloque es nueva, escondemos el
loader
                    if(result.blockHash != $("#hashBlockAddress").html())
                        $("#loader").hide();

                    $("#hashBlockAddress").html('Dirección Block Hash: '+
result.blockHash);
                    //Imprimimos un resumen de los datos enviados a la vez
que casteamos de byte16 a Ascii
                    $("#resumen").html(web3.toAscii(result.args.nombre)+'
'+web3.toAscii(result.args.apellidos) +' / '+(result.args.horas)+' horas /
'+(result.args.aPagar)+' ether / Se te ha pagado un total de ' +
(result.args.totalPagado)+ ' ethers');
                } else {
                    $("#loader").hide();
                    console.log(error);
                }
        });
        //Obtenemos el número de empleados
        sueldo.numeroEmpleados((err, res) => {
            if (res)
                $("#contadorEmpleados").html(res.c + ' Empleados');
        });
```

```
        //Al clicar el botón fichar pasamos los datos al smart contract
        $("#button").click(function() {
            $("#loader").show();
            sueldo.setEmpleado(web3.eth.defaultAccount, $("#nombre").val(),
$("#apellidos").val(), $("#horas").val(), (err, res) =>{
                if(err){
                    $("#loader").hide();
                    console.log('Ups!! algo no funciona como es debido');
                }
            });
        });
    </script>
</body>
</html>
```

Implementamos una hoja de estilos sencilla para que el UI tenga una aspecto más agradable.

main.css

```
body {
    background-color:#AEB404;
    padding: 2em;
    font-family: 'Raleway','Source Sans Pro', 'Arial';}
.container {
    width: 50%;
    margin: 0 auto;}
label {
    display:block;
    margin-bottom:10px;}
input {
    padding:10px;
    width: 50%;
    margin-bottom: 1em;}
button {
    margin: 2em 0;
    padding: 1em 4em;
    display:block;}
#resumen {
    padding:1em;
    background-color:#E1F5A9;
    margin: 2em 0;}
#loader {
    width: 100px;
    display:none;}
```

Pruebas

Red virtual

La primera de las pruebas locales que realizaremos será desde el mismo IDE Solidity. Este IDE no sólo nos permite escribir en lenguaje Solidity y mostrarnos posible errores de código sino que además puede compilar. También tiene una opción de máquina virtual en javascript que simula a la red Ethereum facilitando hasta cinco cuentas de usuario distintas con 100 ether de balance en cada una de ellas.

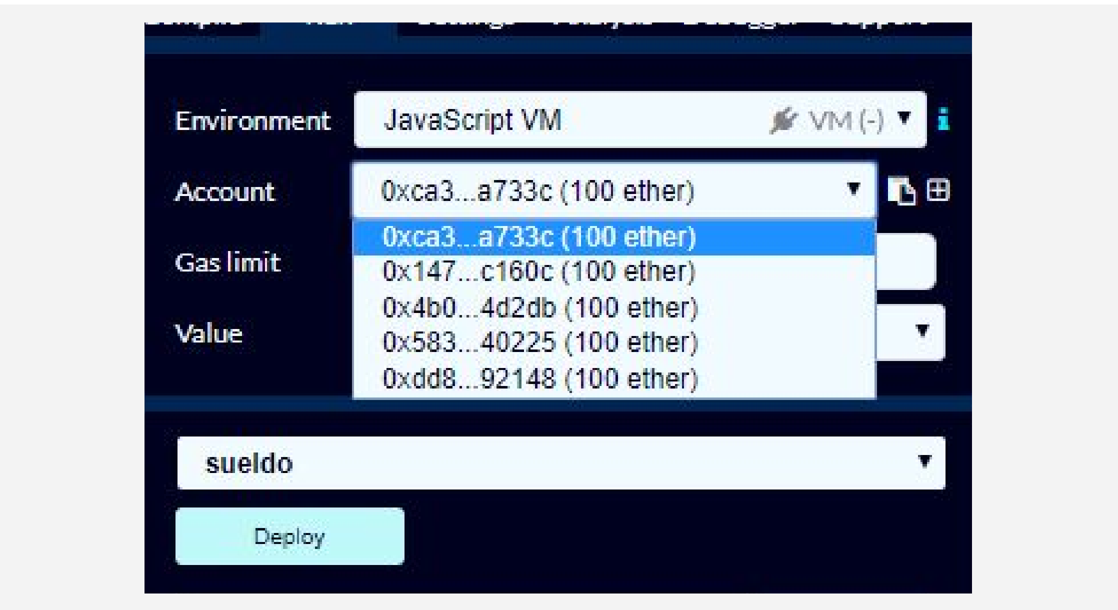


Imagen 27. Cuenta de usuarios de en Solidity Remix

Para compilar el contrato se elige una cuenta de usuario de Ethereum para que sea la cuenta propietaria del contrato, es decir, en este caso sería la empresa la propietaria del contrato. Elegimos la primera de las cuentas con la dirección:

[0xca35b7d915458ef540ade6068dfe2f44e8fa733c]

Una vez el contrato se ha compilado y subido al *Blockchain*, se observa en la consola del IDE como, si no han habido errores, la transacción ha sido minada y añadida a un bloque.

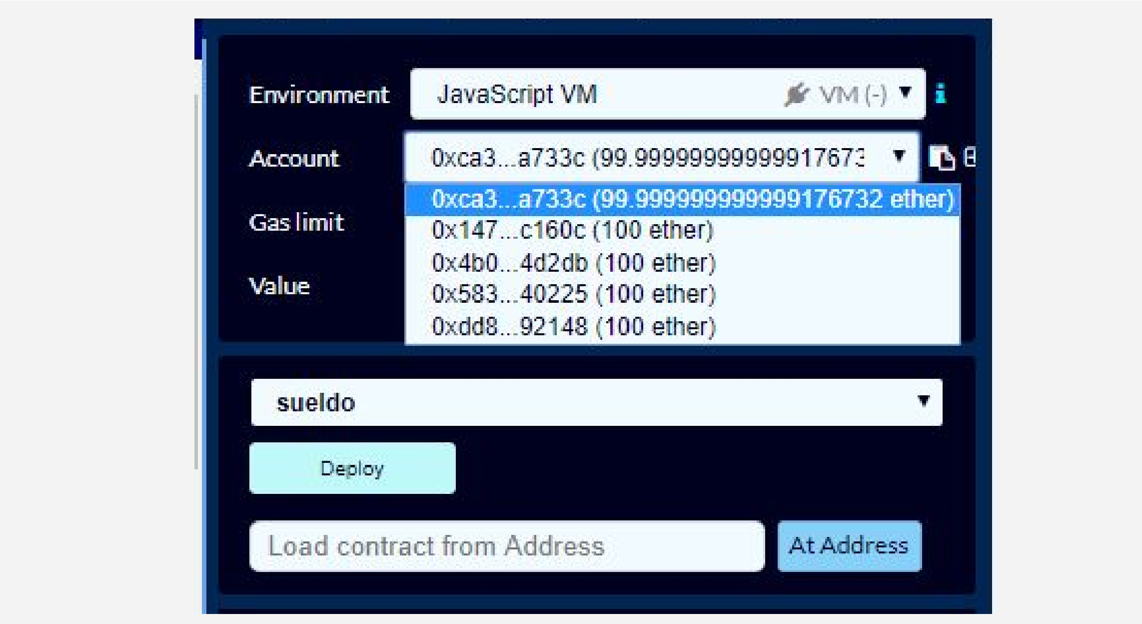


Imagen 28. Cuenta de usuarios en Solidity Remix

Se observa como el saldo de la cuenta a disminuido, esto se debe al pago del gas por realizar la transacción de subir el contrato al *Blockchain*.

```
creation of sueldo pending...

[vm] from:0xca3...a733c to:sueldo.(constructor) value:0 wei data:0x608...00029 logs:0 ha
```

status	0x1 Transaction mined and execution succeed
contractAddress	0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	sueldo.(constructor)
gas	3000000 gas
transaction cost	823268 gas
execution cost	583600 gas
input	0x6080604052670de0b6b3a764000060005534801561001c57600080fd5b50 260043610610083576000357c010000000000000000000000000000000000000 806312065fe014610088578063 5ce8e051146100b35780636380b2461461010

Imagen 29. Transacción compilación contrato en Solidity Remix

En el mensaje de confirmación de la consola se puede ver que la transacción ha sido minada de forma exitosa. Una vez compilada y subida al *Blockchain* el contrato adquiere una dirección **[contractAddress: 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a]** que nos servirá más adelante en las pruebas online con la red Ropsten para vincular la UI con el contrato.
Lo siguiente que se especifica es la dirección de quién ha generado la transacción, como se puede observar coincide con la cuenta de usuario desde la que hemos compilado el contrato. Se especifica el coste de la transacción en gas y el coste de ejecución.

El IDE de Solidity nos facilita botones y campos de acceso a las funciones que se han implementado.

Imagen 30. Funciones del contrato en Solidity Remix

Ahora se debe depositar saldo en el contrato para que se pueda pagar a los empleados. Esta acción se puede llevar a cabo especificando la cantidad de ether que queremos transferir al contrato y validar la acción mediante la función que se ha implementado de `deposit()`.

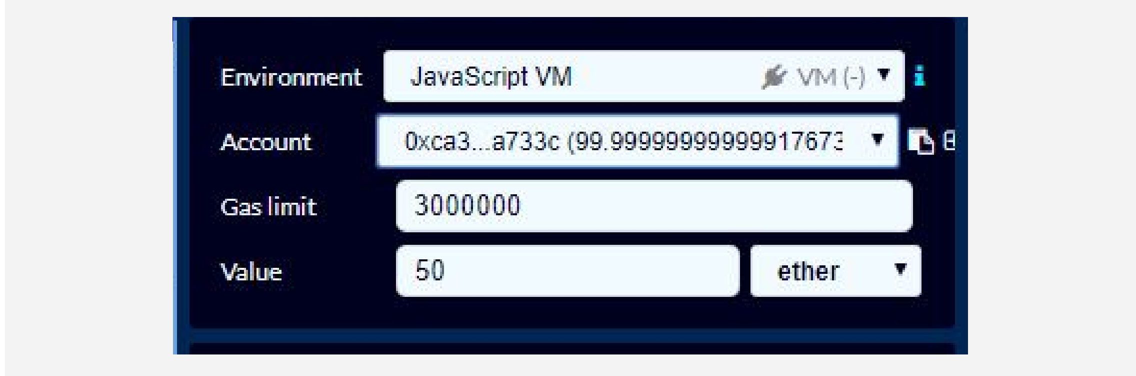


Imagen 31. Transferencia de ethers al contrato

Para el ejemplo de pruebas, haremos que la cuenta propietaria del contrato transfiera 50 ethers al contrato.

Imagen 32. Cuenta de usuario propietario

Fíjese como una vez llevada la acción a cabo, el balance de la cuenta de usuario ha disminuido en 50 ethers, cabe recordar que las cuentas se habían iniciado con un saldo de 100 ether. Teniendo en cuenta que subir el contrato y depositar fondos en el contrato tienen un coste de gas, el balance final de la cuenta propietaria del contrato queda casi a la mitad.

En la consola podemos ver de nuevo como la transacción se ha llevado a cabo con éxito.

[vm] from:0xca3...a733c to:sueldo.deposit() 0x692...77b3a value:50000000000000000000 wei data:0xd
hash:0x8b0...93135

status	0x1 Transaction mined and execution succeed
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	sueldo.deposit() 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	21491 gas
execution cost	219 gas
input	0xd0e30db0
decoded input	{}
decoded output	{}
logs	[]
value	50000000000000000000 wei

Imagen 33. Transacción de transferencia de ethers al contrato

Se puede observar desde qué dirección se ha realizado la transacción (dirección del usuario propietario del contrato) y hacia dónde se ha realizado (dirección del contrato). Nos especifican el coste de la transacción y de la ejecución y en el campo [value] se especifica la cantidad en wei que se han transferido al contrato.

Una vez el *smart contract* cuenta con saldo, es el momento de que un empleado que ha sido contratado por la empresa para realizar tareas de programación, detalle sus datos y las horas que ha trabajado.

Para ello elegiremos la segunda de las cuentas que nos facilitan con la dirección: [0x14723a09acff6d2a60dcdf7aa4aff308fddc160c].

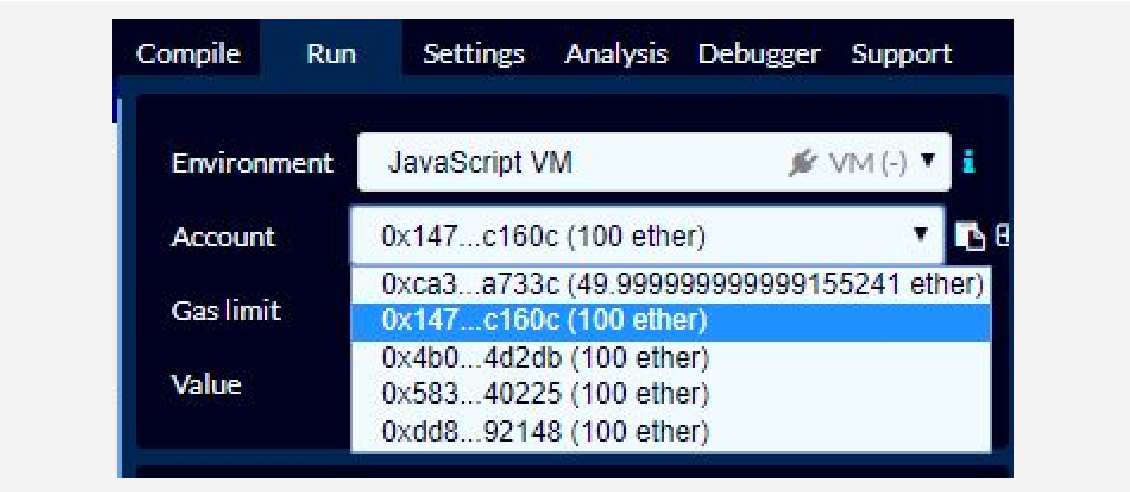


Imagen 34. Cuenta de usuario empleado

A partir de este momento es como si el trabajador en cuestión entrase en las instalaciones de la empresa y al acceder al edificio se identificara con su huella dactilar. Una vez hubiese finalizado su trabajo, volvería a fichar con su huella dactilar para comunicar a la DApp que su trabajo ya está hecho y que se le puede transferir el salario de su trabajo que previamente haya sido revisado por un supervisor de la empresa. Al poner su dedo para la lectura de su huella el sistema biométrico pasaría la siguiente información al UI:

Imagen 35. Horas trabajadas por el empleado

Una vez los datos son validados por el sistema biométrico, éstos se pasarían a la UI de la DApp y se generaría una nueva transacción para el empleado desde su cuenta de usuario de Ethereum.

En la consola del IDE podemos ver la transacción:

```
[vm] from:0x147...c160c to:sueldo.setEmpleado(address,string,string,uint256) 0x692...77b3a valu
hash:0x1d7...0902e
```

status	0x1 Transaction mined and execution succeed
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	sueldo.setEmpleado(address,string,string,uint256) 0x692a70d2e424a56d2c6
gas	3000000 gas
transaction cost	201707 gas
execution cost	177171 gas
input	0x917b65ad00000000000000000000000014723a09acff6d2a60dcdf7aa4aff308fddc1 000000000000000000000000000000000080000000000000000000000000000000000000 00c000000000000000000000000000000000000000000000000000000000000000000050 0000000000000000000000000000000064361726c6f73000000000000000000000000000 00000000000000000000000000000000000000000000000000000000000045275697a0 000000000000000000000000000
decoded input	{ "address _direccion": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC16", "string _nombre": "Carlos", "string _apellidos": "Ruiz", "uint256 _horas": "5" }
decoded output	{}
logs	[{ "from": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a", "topic": "0x02cb6123bfcf0c1f7c5ee464446fb83b653f98a8ab8", "event": "fichar", "args": { "0": "Carlos", "1": "Ruiz", "2": "5", "3": "5", "4": "5", "nombre": "Carlos", "apellidos": "Ruiz", "horas": "5", "aPagar": "5", "totalPagado": "5", "length": 5 } }]
value	0 wei

Imagen 36. Transacción del empleado al fichar

Desde la dirección de la cuenta de usuario de Ethereum [0x14723a09acff6d2a60dcdf7aa4aff308fddc160c] del empleado, se ha generado la transacción que ha llamado a la función `setEmpleado()` del contrato con dirección [0x692a70d2e424a56d2c6c27aa97d1a86395877b3a]. También se puede observar la llamada al `evento()`.

La transferencia de ethers desde el contrato a la cuenta de Ethereum del empleado es una transacción interna dentro del propio *Blockchain* de Ethereum. Comprobamos que el saldo de la cuenta del empleado ha aumentado en 5 ethers que justamente son las horas que había introducido. En el *smart contract* se había especificado un precio de hora trabajada de 1 ether.

Imagen 37. Saldo en la cuenta del empleado

Comprobamos el balance total que queda en el contrato inteligente con la función que se ha implementado `getBalance()`.

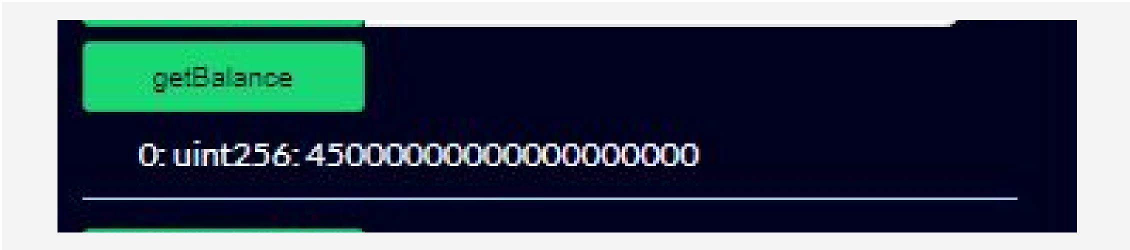


Imagen 38. Saldo en el contrato

Si queremos saber la dirección de cuenta de usuario de un empleado para poder recuperar los datos del empleado, sólo tenemos que llamar a la función `getDireccionesEmpleados()` y una vez copiada la dirección podemos recuperar los datos de ese empleado a través de la función `getEmpleado()`.

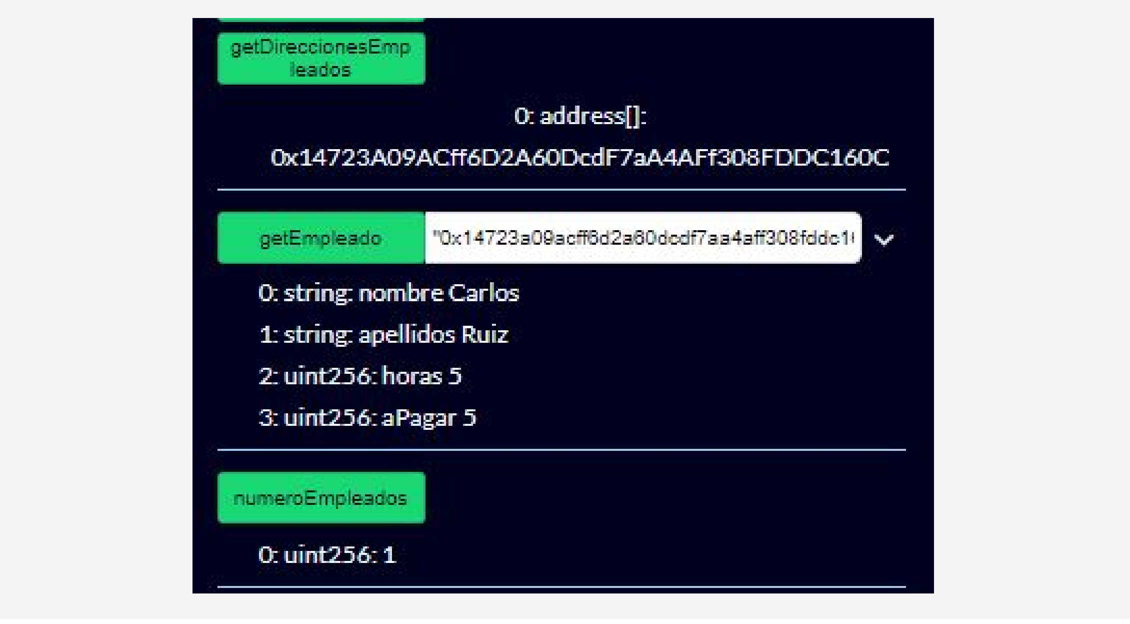


Imagen 39. Llamada a la función getEmpleado()

La función `numeroEmpleados()` no indica que sólo hay un empleado guardado en el contrato.

Verificación

Online en red Ropsten

Para llevar a cabo las siguiente prueba instalamos el cliente de Ethereum Metamask como una extensión del navegador chrome. Metamask ofrece una *wallet* en Ethereum y acceso a las redes de prueba para desarrolladores. Otra de las funciones importantes de Metamask es que en cada página inyecta la librería web3 permitiendo que cada aplicación DApp pueda integrar Metamask para que el usuario pueda usar la aplicación de una manera fácil e intuitiva. Una vez instalado, generamos tres cuentas distintas, una para la empresa y dos para empleados distintos. También nos aseguramos de conectarnos a la red Ropsten de Ethereum. Se trata de la red Ethereum pero con un *Blockchain* mucho más pequeño y permite realizar transacciones con ethers virtuales, sin coste real.

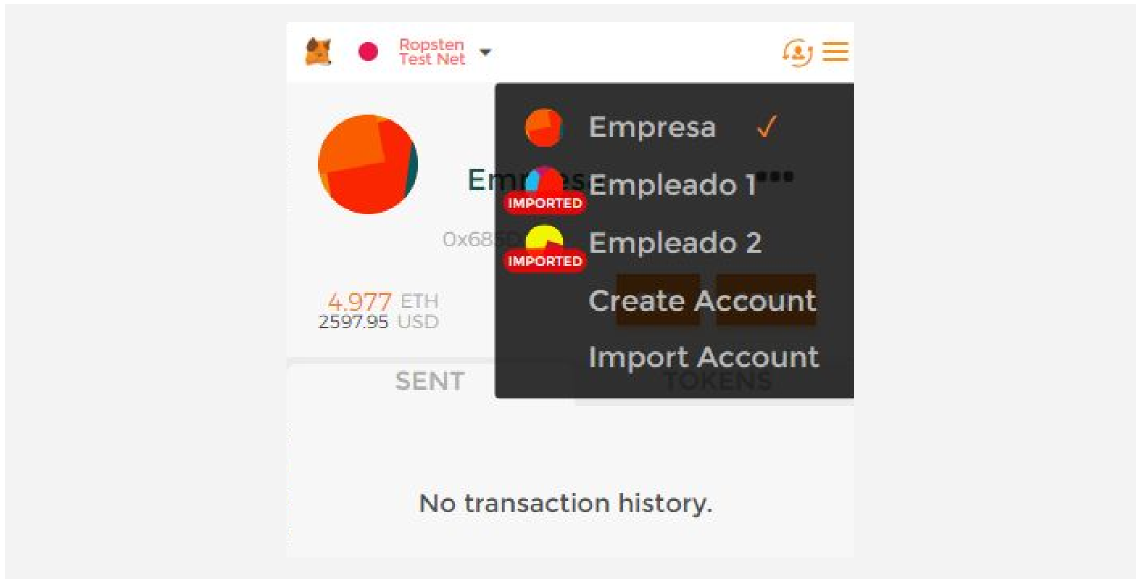


Imagen 40. Cuentas de usuarios en Metamask

CUENTA	DIRECCION	SALDO
empresa	0x685Dbb78631601D22D58bD5530Ba40D8dA6Fe864	25.9 ETH
empleado 1	0xF65C0253398307e2F7578bC07B71c8c2dF3A1297	19.1 ETH
empleado 2	0xF8BF94eEDDb12Ea1094309dA351ba43772Af8871	86.2 ETH

Lo siguiente es instalar lite-server para permitir montar un servidor virtual que aloje el UI de la DApp. Una vez instalado lo iniciamos en la carpeta del proyecto con el comando `[npm run dev]`, en nuestro caso en la consola de windows.

```
Microsoft Windows [Versión 10.0.17134.48]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\carlo\Documents\UOC\TFG\code\final>npm run dev

> tfg_dapp@1.0.0 dev C:\Users\carlo\Documents\UOC\TFG\code\final
> lite-server

Did not detect a `bs-config.json` or `bs-config.js` override file. Using lite-server defaults...
** browser-sync config **
{ injectChanges: false,
  files: [ './**/*.{html,htm,css,js}' ],
  watchOptions: { ignored: 'node_modules' },
  server: { baseDir: './', middleware: [ [Function], [Function] ] } }
[Browsersync] Access URLs:
 --------------------------------------
       Local: http://localhost:3000
    External: http://169.254.175.255:3000
 --------------------------------------
          UI: http://localhost:3001
 UI External: http://169.254.175.255:3001
 --------------------------------------
[Browsersync] Serving files from: ./
[Browsersync] Watching files...
18.05.29 12:42:37 304 GET /index.html
18.05.29 12:42:39 304 GET /node_modules/web3/dist/web3.min.js
18.05.29 12:42:39 200 GET /main.css
```

Imagen 41. Puesta en marcha de lite-server

En cuanto se inicie el servidor, arrancará el UI en el explorador que tengamos por defecto.

Imagen 42. Interfaz de usuario (UI) - Frontend

La UI se compone de un contador del número de empleados que hay almacenados en el contrato inteligente. Seguido, un campo resumen donde aparecerá un resumen de los datos introducidos más el salario que se le ha transferido a su la cuenta de usuario del empleado.
Los siguientes campos sirven para pasar los datos personales y las horas trabajadas del empleado al *smart contract*. FInalmente el botón ‘Fichar’ es el encargado de llamar a la función `setEmpleado().`

Lo primero que realizaremos como usuario empresa es compilar el contrato inteligente y subirlo al *Blockchain* para que esté disponible en la red Ropsten. Una vez compilado y subido al *Blockchain*, depositamos fondos suficientes para pagar la los empleados. Este paso de la empresa no se hace a través de la UI, aunque se podría generar otra UI sólo para el responsable de la empresa sobre el contrato inteligente para que pudiera gestionar algunos de los valores de las constantes fijas del contrato o para poder disponer de acceso directo a las funciones que devolvieran información de los empleados.

Para compilar y subir el contrato, se realizará de nuevo desde el mismo IDE Solidity. Un aspecto importante a tener en cuenta para llevar a cabo esta prueba, el entorno de Solidity debe estar seleccionado en "Injected Web".

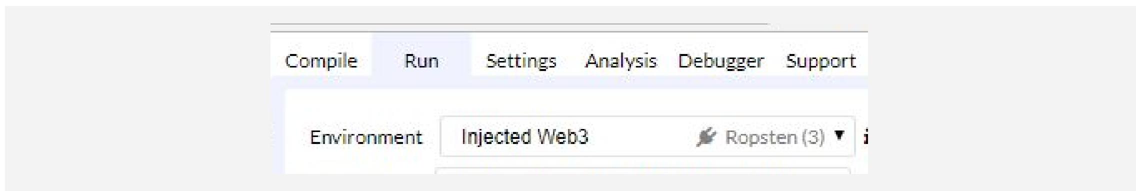


Imagen 43. Entorno de Solidity Remix

Seleccionamos la cuenta de empresa de Metamask para compilar y subir el contrato inteligente al *Blockchain* de Ethereum de la red Ropsten. Al seleccionar la cuenta, automáticamente Solidity se autoselecciona para sincronizar las cuentas con Metamask.

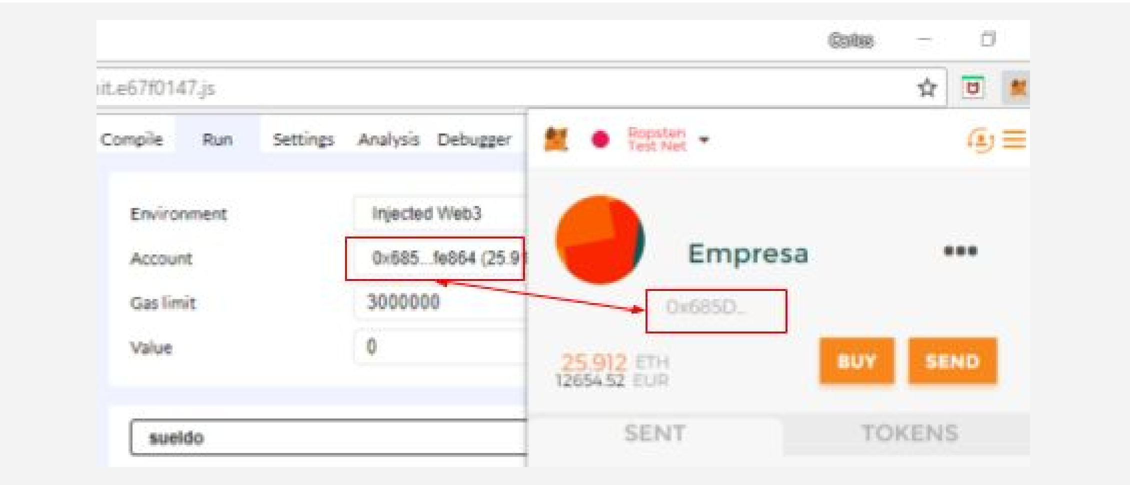


Imagen 44. Sincronización de cuentas entre Metamask y Solidity Remix

En cuanto accionemos el botón 'deploy' iniciaremos la transacción, no sin antes especificar el límite de gas dispuesto a pagar y la tarifa de gas que queremos pagar. Para ello Metamask lanzará una ventana desde donde podremos seleccionar los parámetros de coste.

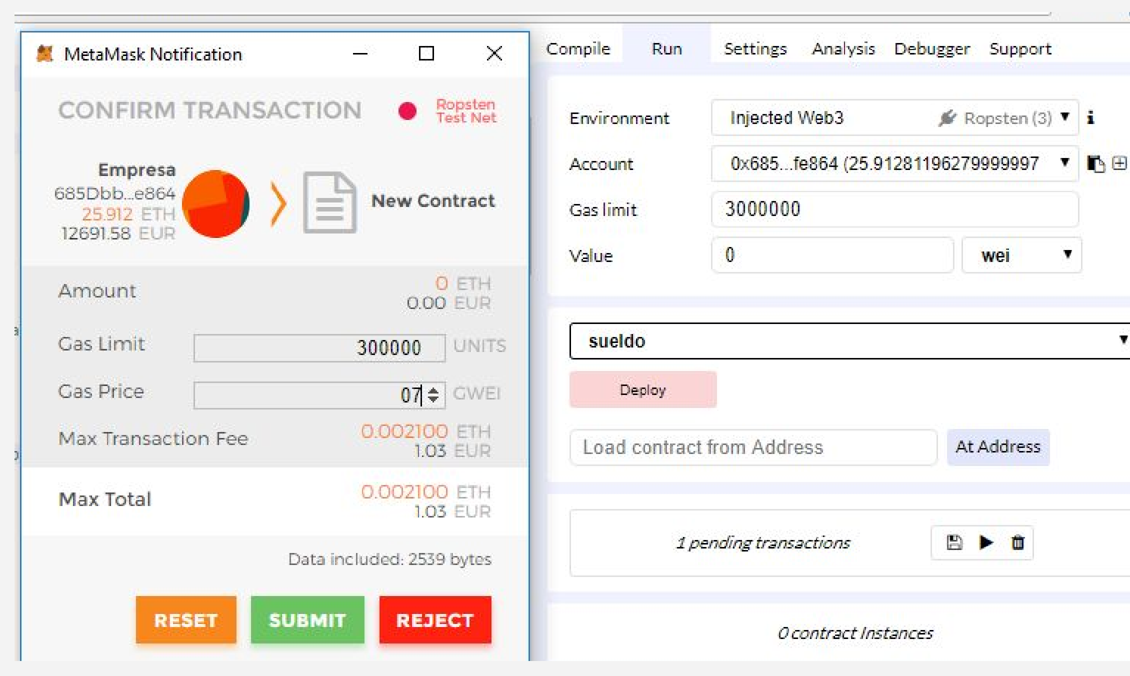


Imagen 45. Precio de gas

En esta ocasión el responsable del contrato de la empresa decide especificar un límite de gas de 300000 wei y quiere pagar 0.002100ETH. A partir de este momento la transacción está pendiente de ser validada y añadida a un bloque.

```
creation of sueldo pending...
https://ropsten.etherscan.io/tx/0x2ce5280456133a3931bf0c32791ed75ff938ef7a55f198be8e22d6b874c013c9
```

Imagen 46. Enlace a la transacción en la red Ropsten

Podemos clicar sobre el enlace de la consola de Solidity para acceder a la transacción directamente y ver el estado.

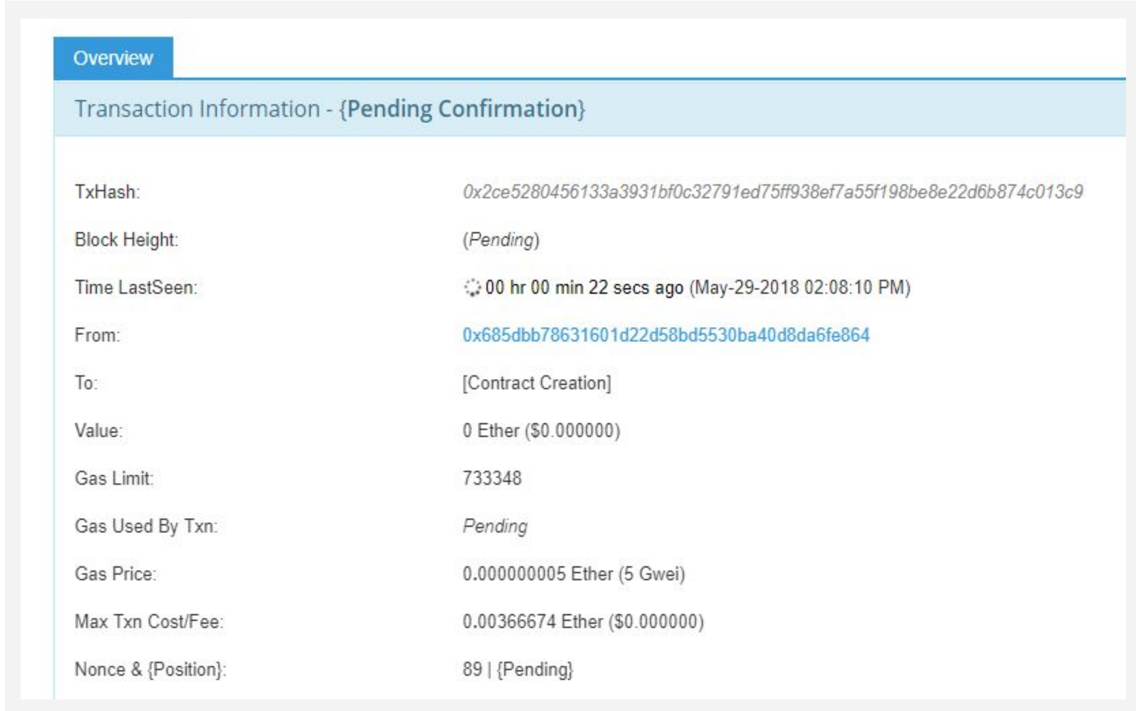


Imagen 47. Transacción de subir el contrato al *Blockchain*

Mientras la transacción está pendiente de ser validada y agregada a un bloque, podemos ver la dirección de Ethereum de la transacción (TxHash). El Block Height es la cantidad de bloques que preceden a este bloque en particular en la cadena de bloques. Por ejemplo, el bloque génesis tiene una altura de cero porque no tiene ningún bloque que le preceda. Este valor se actualizará en cuanto se valide la transacción y se añada a un bloque. También podemos ver el timestamp, quién inicia la transacción y hacia dónde se inicia la transacción que una vez validada, será la dirección que la red Ethereum asigne al contrato. Los costes de gas que está teniendo la transacción y finalmente el *nonce* que tiene la transacción para ser validada.

Una vez la transacción se ha validado obtenemos la siguiente lectura del bloque:

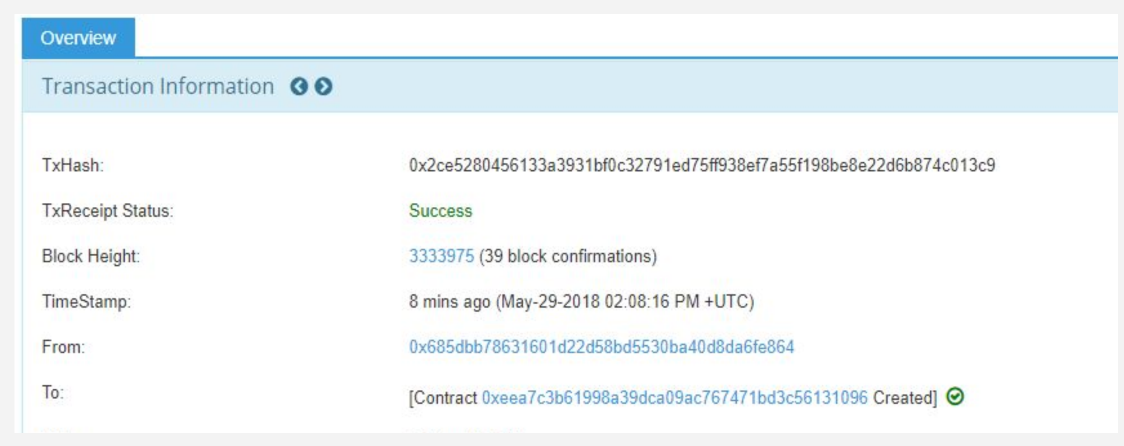


Imagen 48. Confirmación de la transacción

Si clicamos sobre la dirección del contrato inteligente podemos ver todas las transacciones que se lleven a cabo con el contrato y por parte de quién. Ahora que el contrato ya está en el *Blockchain*, la empresa debe depositar fondos. En este caso particular la empresa decide depositar 15 ETH y configura lo costes de gas a pagar.

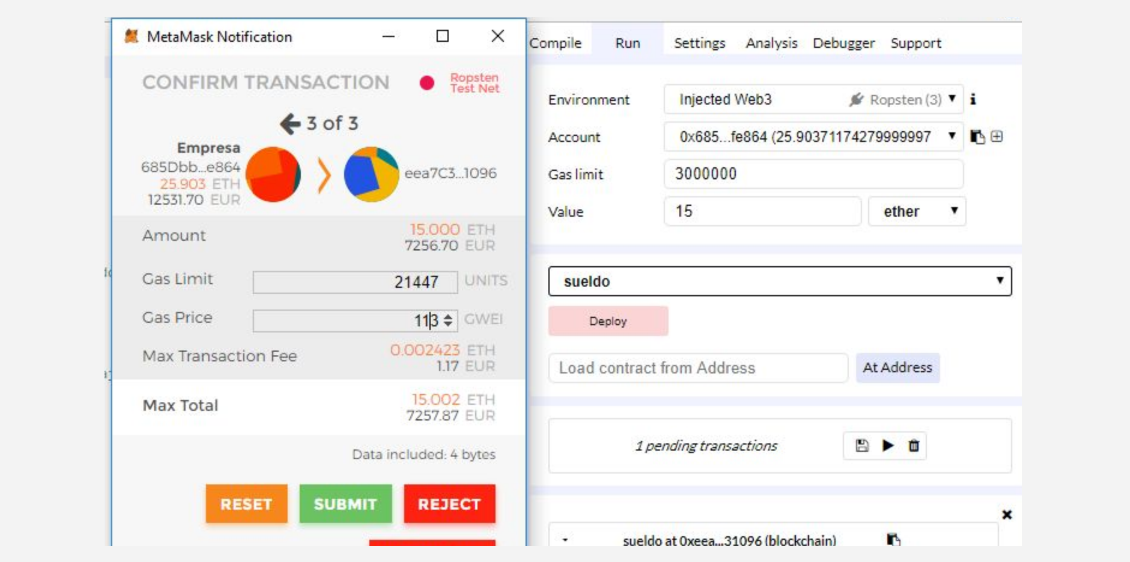


Imagen 49. Transferencia de fondos al contrato y pago de gas

Accedemos a la dirección del contrato directamente y podemos ver que la transacción de depositar ETH en el contrato se ha llevado a cabo con éxito, también podemos apreciar cuando se creó y por quién el contrato inteligente.

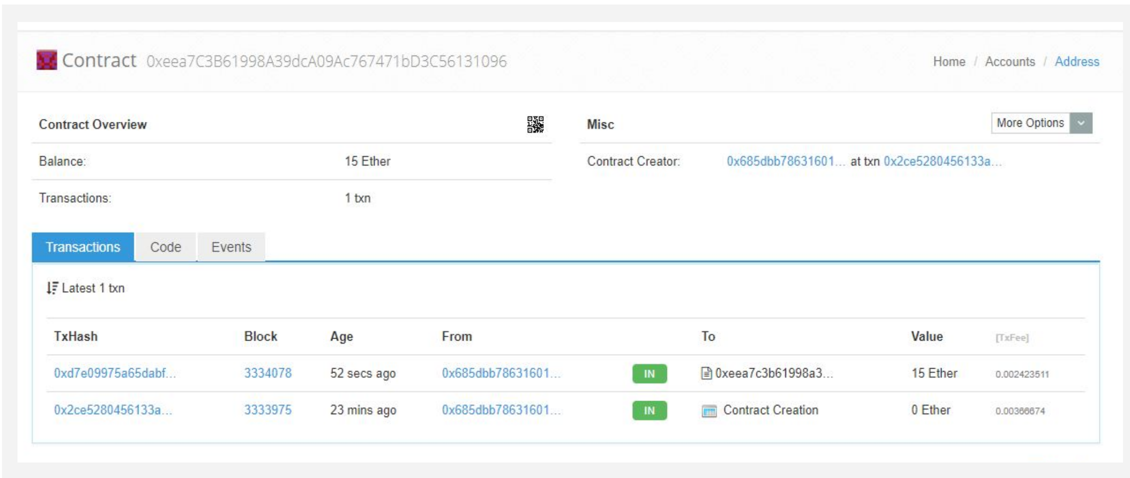


Imagen 50. transacciones del contrato

Una vez la empresa ha subido el contrato a la red Ethereum y ha depositado fondos en el contrato, lo siguiente que debe realizar el responsable de la empresa es vincular el de la DApp con el *backend*. Para ello se debe copiar la dirección del contrato inteligente y el ABI del contrato en el archivo index.html, en las variables definidas para ello.

```
//Se almacena el ABI del smart contract una vez compilado
var sueldoContract = web3.eth.contract('ABI aquí');

// Se almacena la dirección del smart contract que se ha subido al blockchain
var sueldo = sueldoContract.at(' DIrección del contrato aquí');
console.log(sueldo);
```

El ABI del contrato se puede encontrar bajo la pestaña del IDE de Solidity, `run/details`.

Imagen 51. ABI del contrato

El empleado 1 ha sido contratado para realizar un trabajo de programación así que cuando se dispone a trabajar en la tarea, se registra en el sistema telemático de fichar de la empresa mediante un lector biométrico de lectura de huellas dactilares desde su casa. El empleado 1 ha necesitado 5 horas para realizar el trabajo que le han encargado y una vez lo ha supervisado el responsable de la empresa, ha autorizado al empleado 1 a fichar en el sistema para que le paguen por el trabajo realizado. Así que para el caso nuestro, los distintos campos del UI serán rellenados manualmente, pero éstos podrían ser autocompletados por el sistema de la empresa a la hora de fichar.

Para llevar a cabo la parte del empleado 1, hay que asegurarse de cambiar de cuentas en Metamask.

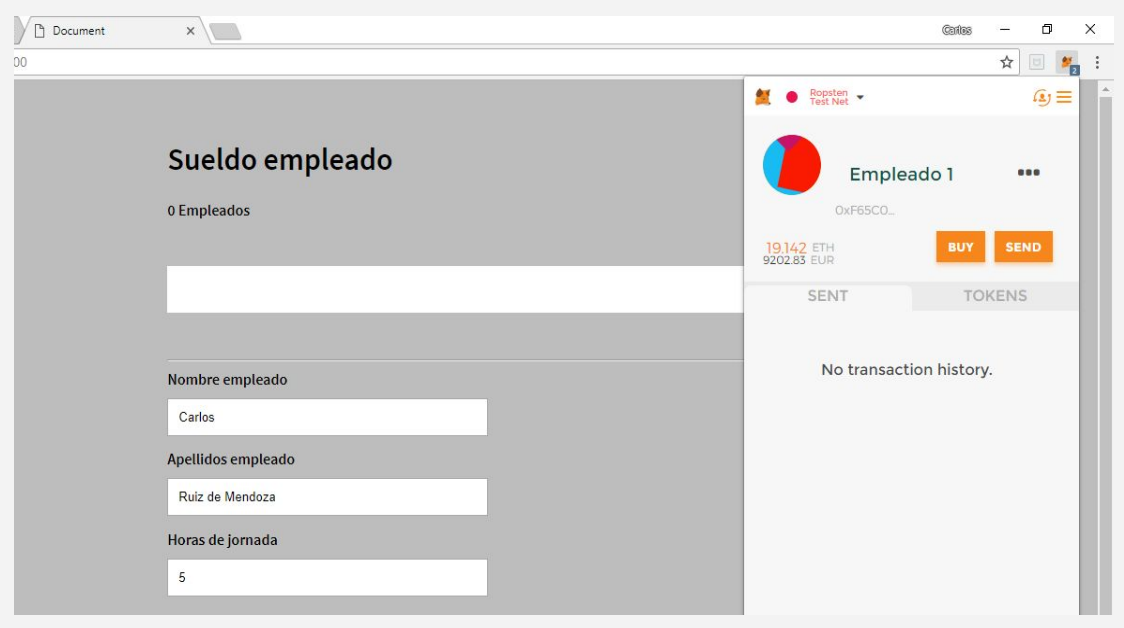


Imagen 52. Ficha el empleado 1

En cuanto el empleado 1 acciona el botón fichar, la ventana de Metamask salta al empleado para que configure los gastos de gas que está dispuesto a pagar para llevar a cabo la transacción de fichar y así que le transfieran a su cuenta el sueldo.

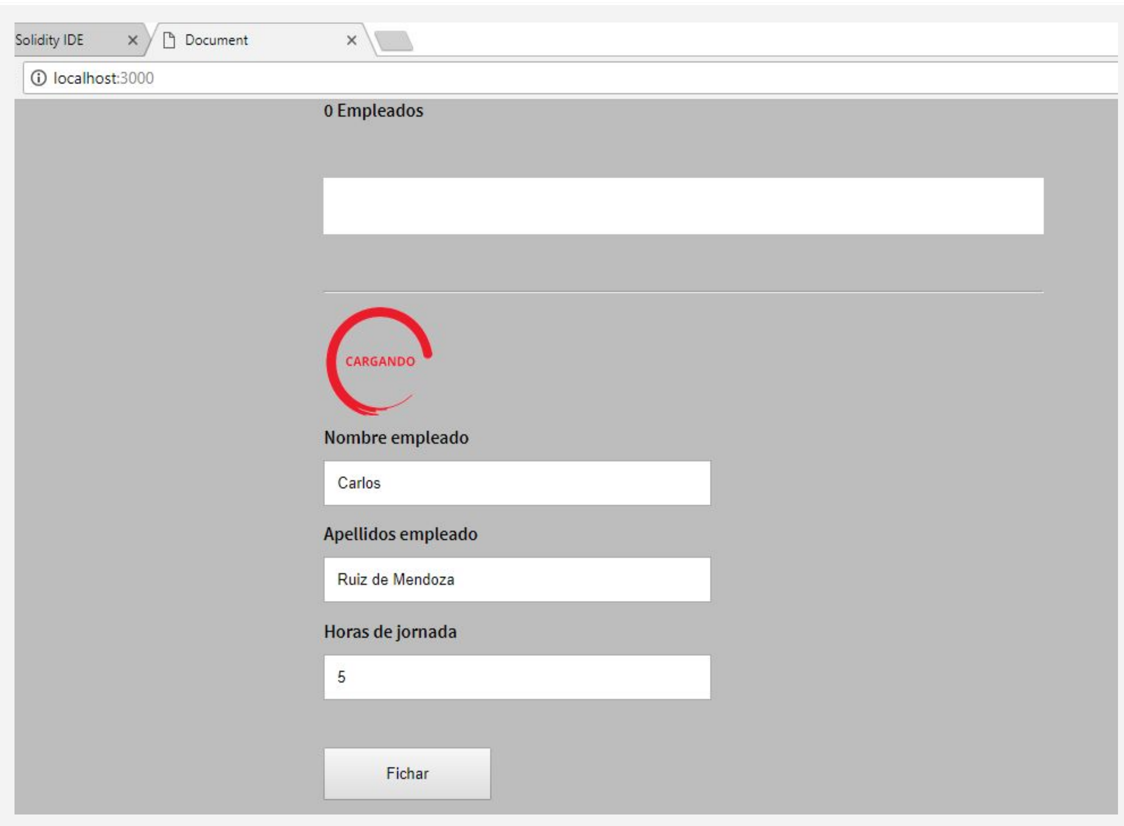


Imagen 53. La UI en proceso de carga de datos al contrato

Al accionar el botón fichar, se ha disparado el evento que se ha implementado en Solidity y que hemos implementado también en jQuery en el archivo index.html del UI. El evento no ha dado ningún error por lo que ha saltado el gráfico de proceso de carga. Desde de la cuenta de usuario de Metamask podemos ver el estado de la transacción clicando sobre el link de la transacción.

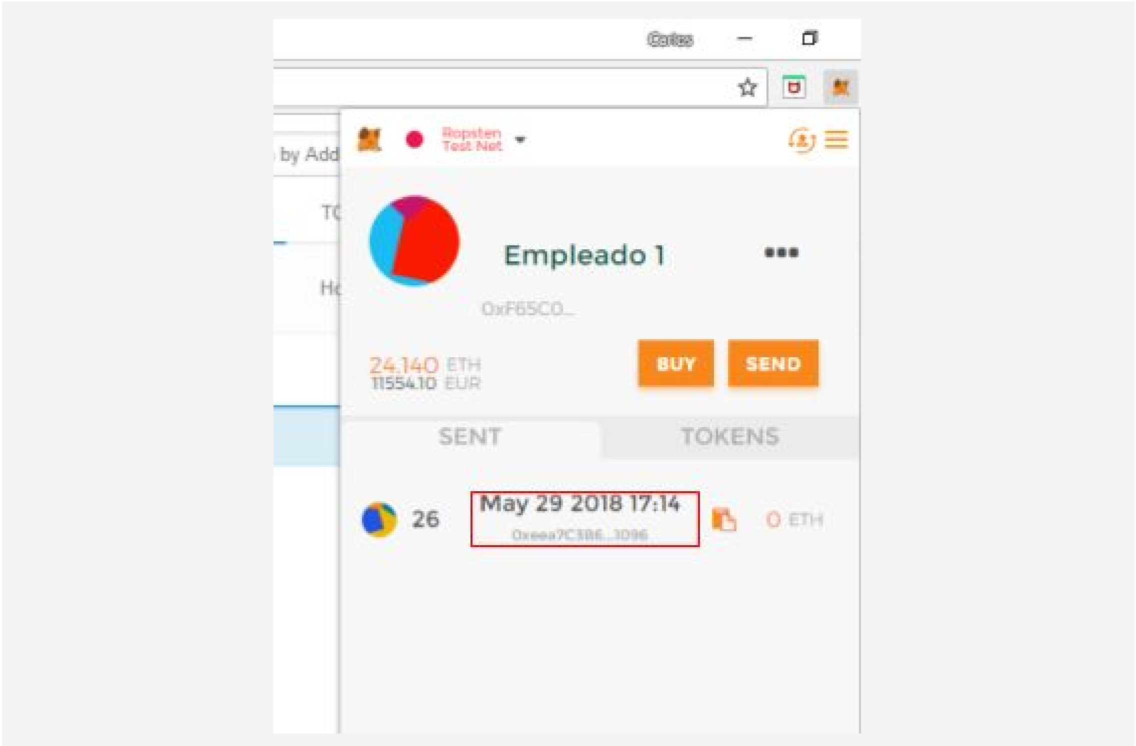


Imagen 54. Enlace a la transacción desde la cuenta de Metamask

La información que obtenemos de la transacción es la siguiente una vez se ha validado.

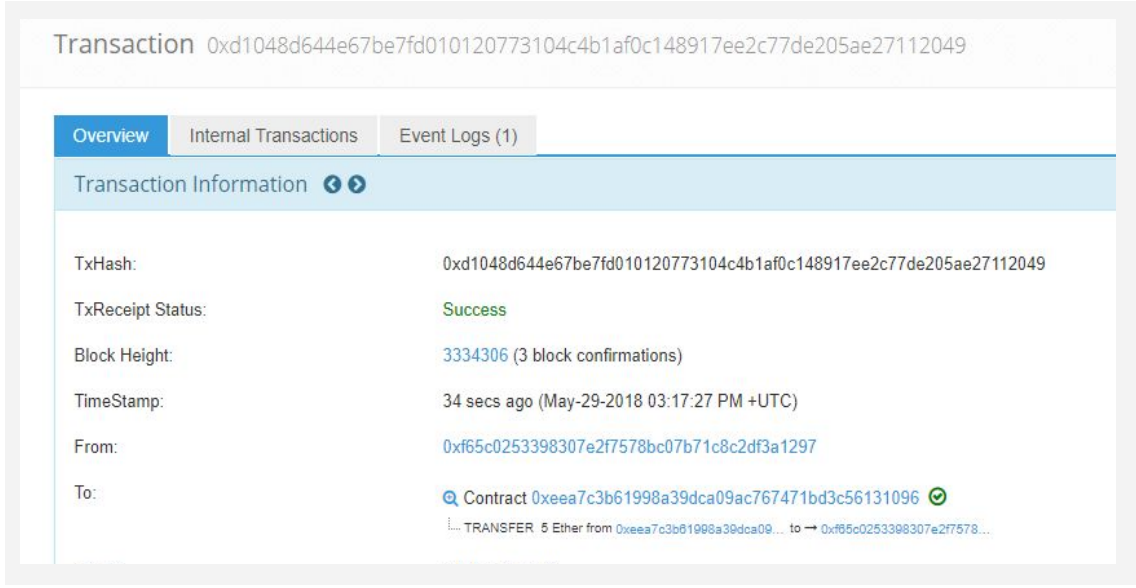


Imagen 55. Información de la transacción en Ropsten

En el campo hacia quién se ha originado la transacción, es decir, hacia el contrato inteligente, vemos cómo se ha generado la transacción interna desde el *smart contract* hacia la cuenta de usuario con una transferencia de 5 ETH a la cuenta de usuario de Ethereum del empleado 1. Podemos comprobarlo en la pestaña de '*Internal Transaction*'.

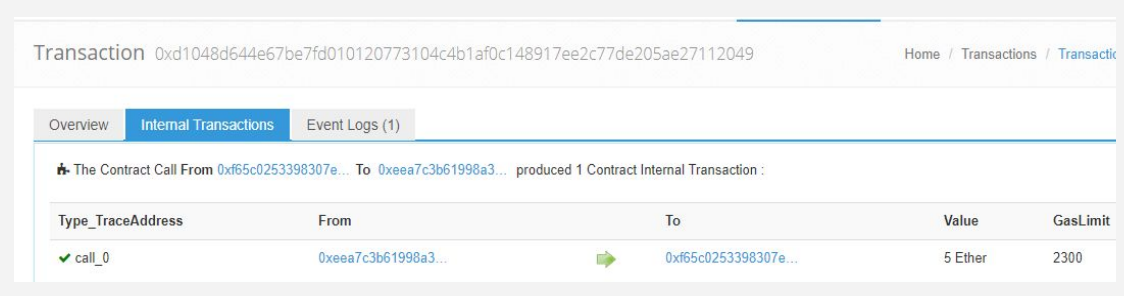


Imagen 56. Transacción interna del contrato

El Block Height nos lleva directamente al bloque en el que se ha añadido la transacción, al acceder al bloque podemos ver la dirección del bloque, entre otros parámetros, que coincide con el la dirección de bloque que aparece en el UI.

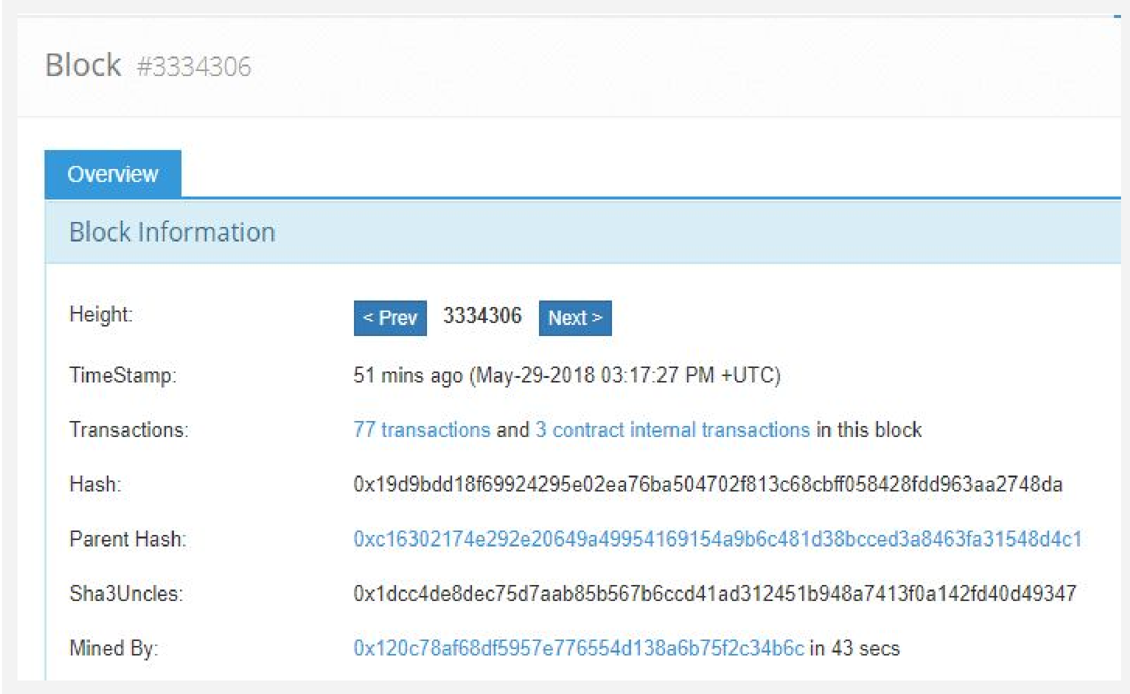


Imagen 57. Información del bloque que agrega la transacción

Mientras, en la UI obtenemos el resumen de la transacción que se ha llevado con éxito.

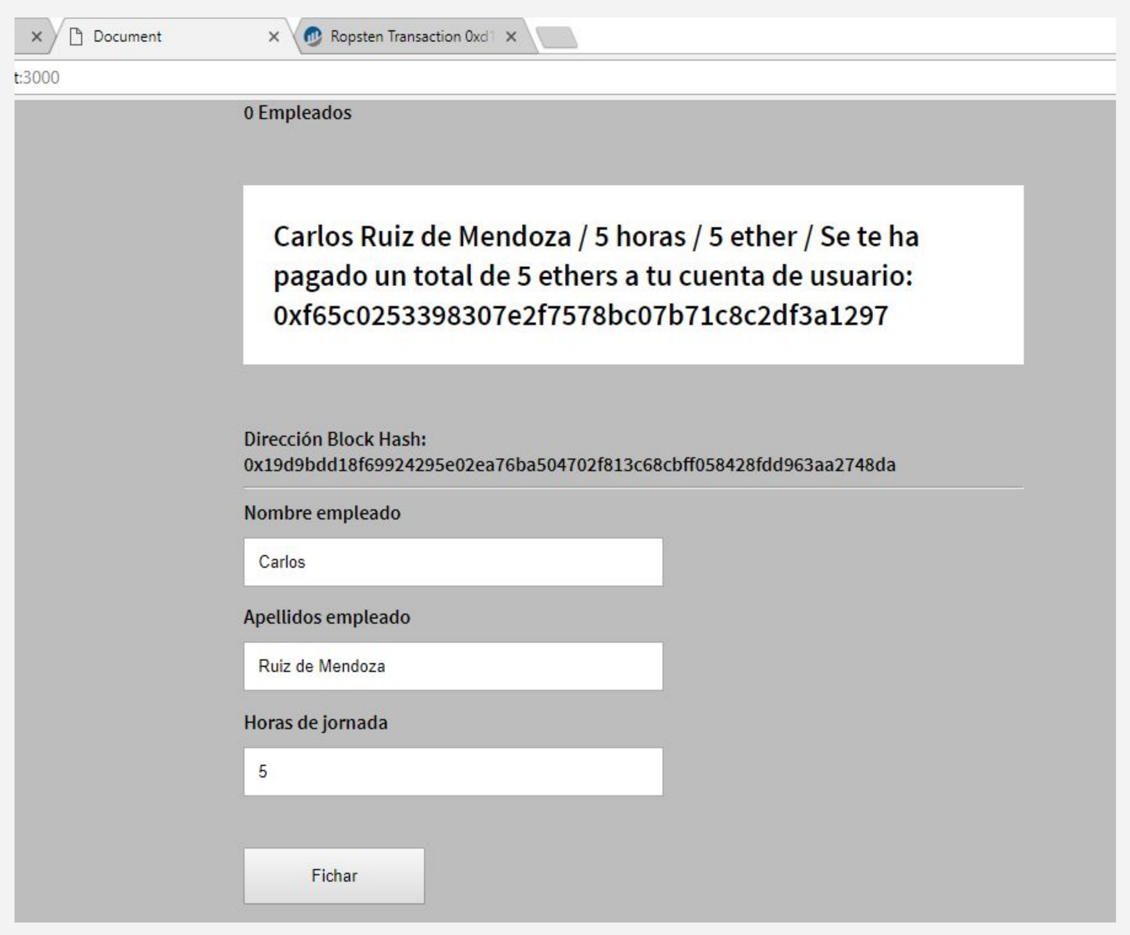


Imagen 58. Resumen de datos en la UI

Efectivamente vemos que que la transacción ha sido añadida a un nuevo bloque del *Blockchain* con la misma dirección que se podía ver en la página https://ropsten.etherscan.io/ que es a la web que Metamask dirige desde la cuenta de usuario para ver el estado de las transacciones..

Si comprobamos el saldo de la cuenta de usuario de Ethereum del empleado 1 en Metamask, vemos que se ha incrementado su saldo desde los 19 ETH hasta los 24 ETH, es decir en 5 ETH.

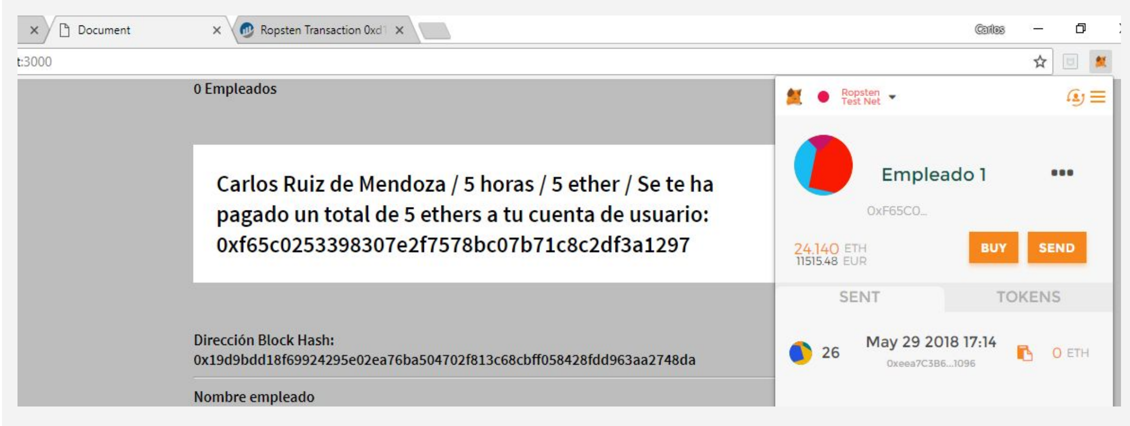


Imagen 58. Saldo resultante en la cuenta de usuario en Metamask

El balance del contrato es de algo más de 10 ETH tal y como se puede comprobar al llamar a la función `getBalance()`.

Imagen 59. Llamada a la función getBalance() del contrato

De nuevo, la empresa contrata los servicios de un segundo programador, pero esta vez el empleado 2 vive en la misma ciudad que la empresa por lo que se dirige a las instalaciones de la empresa y ficha mediante una tarjeta identificativa que le han facilitado previamente. El empleado 2 ha necesitado un total de 11 horas para finalizar el trabajo. En cuanto acción el botón fichar de la UI, salta la ventana de Metamask de su cuenta de usuario y le advierte que un error de excepción ha saltado en el contrat.

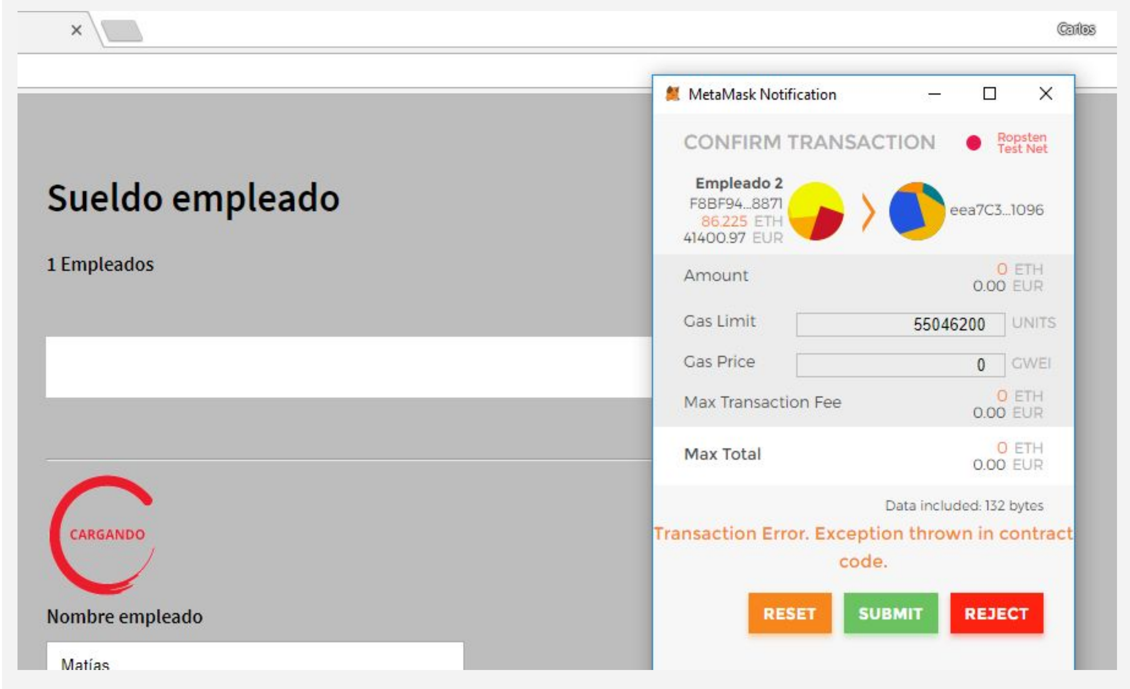


Imagen 60. Ficha el empleado 2

Aún y así el empleado decide seguir con la transacción y establece los costes de gas y acepta el envío de la transacción. Cuando clica en el link de la transacción desde Metamask se encuentra con la siguiente información de la transacción.

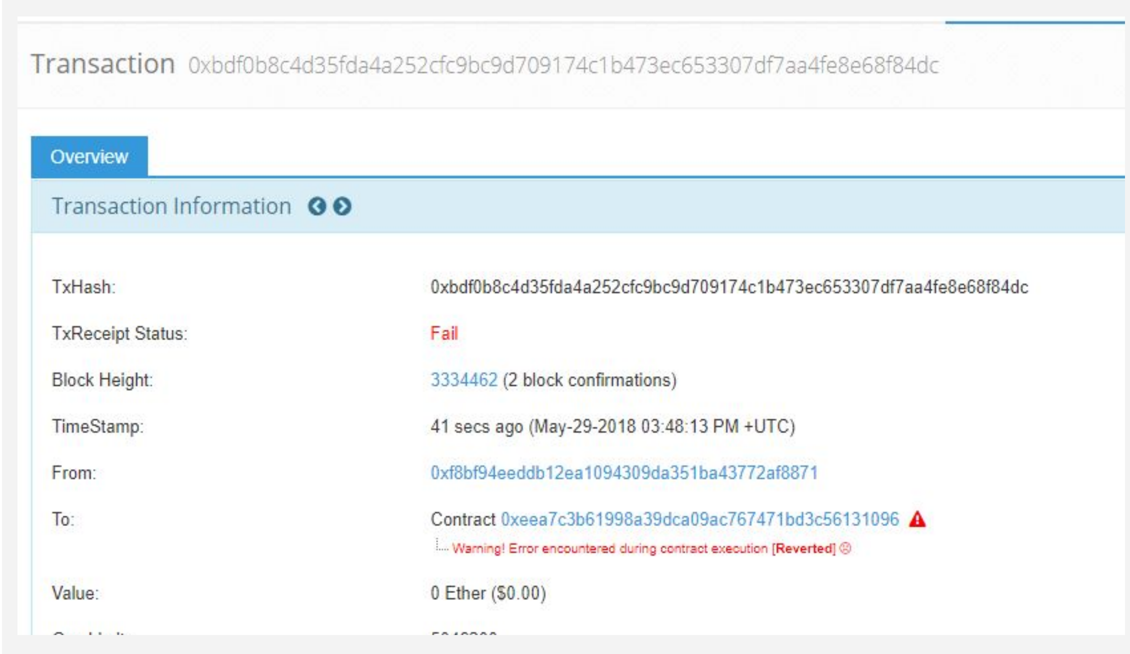


Imagen 61. Fallo en la transacción del empleado 2

Observa que se ha encontrado un error y que todos los datos que ha introducido para fichar han sido revertidos por lo que su fichaje no ha podido llevarse a cabo y por ende no le han transferido el sueldo.

En el *smart contract* se ha implementado la función:

```
function updateBalanceAndPay(uint _aPagar, address _direccion) internal {
    //Descontamos el sueldo del empleado del balance total del contrato
    uint balance  = address(this).balance;
    require (balance > _aPagar);//Si hay menos balance se revierte la transacción a partir de aquí
    uint subTotal = address(this).balance - _aPagar;
    uint totalPagar = (balance - subTotal)*1000000000000000000;
    _direccion.transfer(totalPagar);
    //transferimos en ether el sueldo la cuenta de usuario de Ethereum del empleado
}
```

Esta función antes de transferir los sueldos de los empleados comprueba si existe saldo suficiente, si el balance total del contrato es menor al sueldo que hay que pagar, salta una excepción y revierte los datos de la transacción, la línea responsable de esto es `require (balance > _aPagar)`.

El empleado 1 es vuelto a contratar y en esta ocasión factura 4 horas más. El responsable de la empresa en un momento dado necesitar saber cuánto se le ha pagado en total al empleado 1, por lo que llamando a la función `getEmpleado()` comprueba el total pagado al empleado 1.

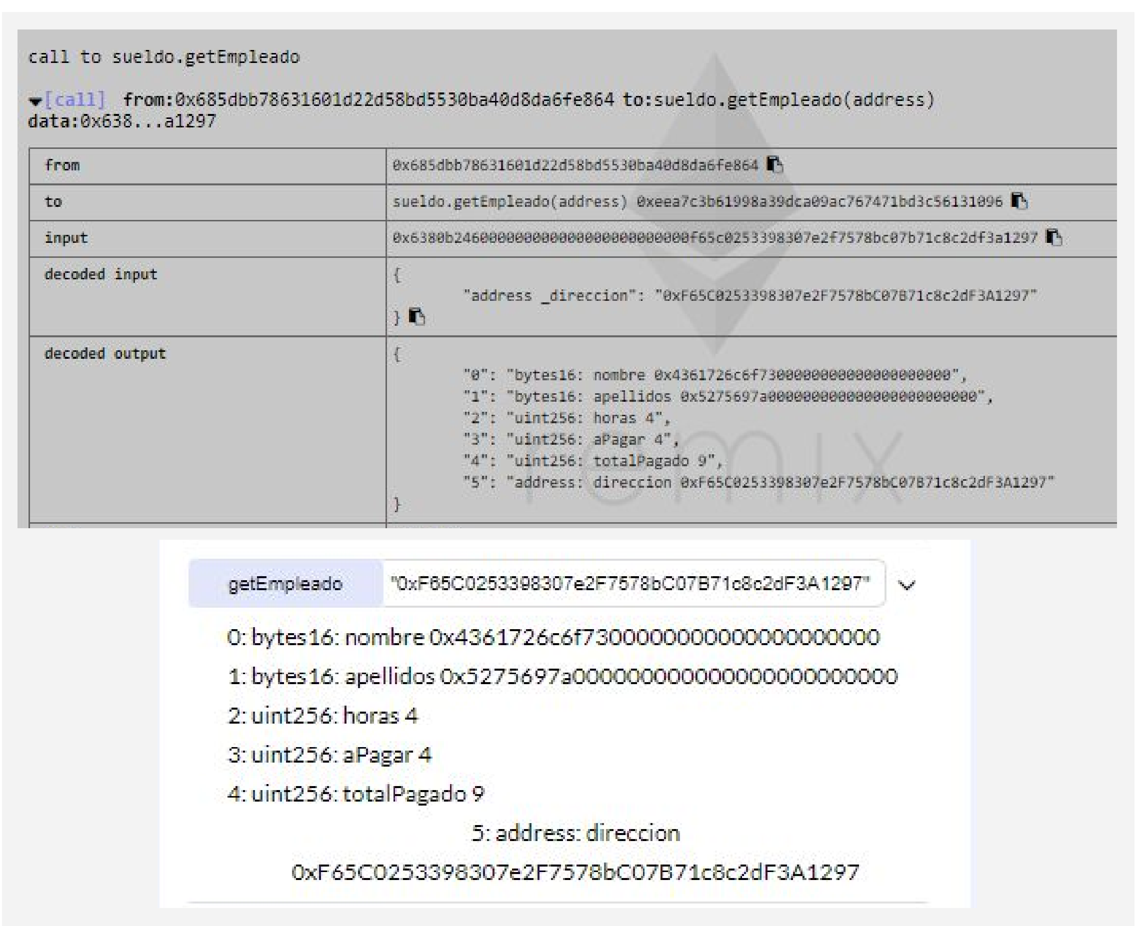


Imagen 62. Llamada a la función getEmpleado()

6. Conclusiones

Una vez finalizados los bloques que componen el presente trabajo final de grado, las conclusiones que se extraen son:

1. El *Blockchain* permite que los datos se compartan sin la necesidad de un administrador central.
2. Las transacciones tienen sus propias pruebas de verificación lo que permite que todos los nodos de la red permanezcan sincronizados.
3. Los usuarios tienen el control de toda su información y de las transacciones, gracias al cifrado asimétrico.
4. Los datos en el *Blockchain* son consistentes, precisos y totalmente disponibles, creando transparencia e inmutabilidad gracias a las funciones de criptografía hash.
5. Debido a las redes descentralizadas P2P, el *Blockchain* no tiene un punto central de falla y tiene mayor capacidad de soportar ataques malignos, como los de denegación de servicio (DoS).
6. Los usuarios pueden confiar en que la transacción se ejecutará exactamente como se defina en los contratos inteligentes, eliminando la necesidad de un tercero de confianza.

Por otro lado, las desventajas que se concluyen son:

1. La tecnología *Blockchain* es más lenta que las bases de datos centralizadas y esto se debe a los procesos de verificación de la firma en las transacciones, a la redundancia de los procesos por parte de los nodos de la red y al tiempo invertido en hallar la solución al *Proof-of-Work.*
2. Se trata de una tecnología poco madura y realmente, aún no se conoce el su potencial.
3. El *Blockchain* y las criptomonedas se enfrentan a un obstáculo generalizado para ser adoptadas por las instituciones financieras de los gobiernos. De nuevo, la tecnología va un paso por delante de la regulación legal.
4. El proceso de minado consume grandes cantidades de energía.
5. Aún existen inquietudes sociales relacionadas con los ataques cibernéticos que deben abordarse antes de que el público en general confíe sus datos personales a la tecnología *Blockchain*.

Por lo tanto, es innegable que se trata de una tecnología innovadora y con un gran poder para descentralizar registros de valor que puede perjudicar a los sectores profesionales que regulan cualquier tipo de transacción, ya que amenaza directamente la figura del intermediario.

Las proyecciones de futuro para la tecnología *Blockchain* estarán supeditadas al marco regulatorio legal de sus criptomonedas y a la confianza en su uso por parte de la sociedad. Se abre una puerta muy interesante para el control y la comunicación entre dispositivos IoT, ya que debido al gran volumen de éstos, el *Blockchain* permite un sistema para llevar un registro inmutable de ellos.

En el desarrollo de la aplicación descentralizada se ha alcanzado con éxito los objetivos propuestos. La programación en Solidity resulta sencilla si se tiene experiencia con lenguajes de programación de alto nivel como javascript o java.

Bibliografía

Libros

1. Kurose Ross, Computer Networking *A top-Down Approach*, 7ª edición, Global Edition, Inglaterra, 2017.
2. George Coulouris, Jean Dollimore, Tim Kindberg y Gordon Blair, Distributed Systems *Concepts and Design,* 5ª edición, Pearson, Inglaterra 2012.
3. Charles Jensen, *Blockchain Science and Career guide*, 2017.
4. Don Tapscott y Alex Tapscott, *Blockchain* Revolution, 1ª edición, Penguin, Nueva York EEUU, 2016.
5. Chris Dannen, Ethereum and Solidity *Foundations of Cryptocurrency and Blockchain Programming for Beginners*, 1ª edición, Apress, Nueva York, EE UU, 2017.
6. Chris Burniske y Jack Tatar, Cryptoassets *The innovative Investor's guide to Bitcoin and Beyond*, 1ª edición, Mc Graw Hill Education, EEUU, 2018.
7. Jordi Herrera Joancomartí, Xifres de clau compartida: xifres de bloc, 4ª edición, Oberta UOC Publishing, SL, Barcelona, España, 2014.
8. Josep Domingo Ferrer, Signatures digitals, 4ª edición, Oberta UOC Publishing, SL, Barcelona, España, 2014.
9. Jordi Herrera Joancomartí, Xifres de clau compartida: xifres de flux,, 4ª edición, Oberta UOC Publishing, SL, Barcelona, España, 2014.

Web

1. http://solidity.readthedocs.io/en/v0.4.24/, 05/2018.
2. http://www.ethdocs.org/en/latest/, 04/2018.
3. https://bitcoin.org/en/developer-documentation, 04/2018.
4. https://medium.com/@robbertvermeulen/learn-solidity-the-ethereum-smart-contract-programming-language-7f106fc26d6, 05/2018.
5. https://bitcoin.org/en/faq, 03/2018.
6. https://www.oroyfinanzas.com/2015/01/como-ajusta-dificultad-minado-bitcoin/, 04/2018.
7. http://dlerch.blogspot.com/2007/07/modos-de-cifrado-ecb-cbc-ctr-ofb-y-cfb.html, 03/2018.
8. https://www.youtube.com/channel/UCaWes1eWQ9TbzA695gl_PtA, 04/2018.
9. https://eklitzke.org/an-overview-of-bitcoin-utxos, 04/2018.
10. https://hackernoon.com/ethereum-development-walkthrough-part-1-smart-contracts-b3979e6e573e, 05/2018.
11. https://unamcriptografia.wordpress.com/2011/10/06/breve-historia-de-la-criptografia-2/, 03/2018
12. http://www.egov.ufsc.br/portal/sites/default/files/la_criptografia_desde_la_antigua_grecia_hasta_la_maquina_enigma1.pdf, 03/2018
13. http://usuaris.tinet.cat/acl/html_web/seguridad/cripto/cripto_2.html, 03/2018
14. https://es.slideshare.net/econtinua/criptografa-y-su-importancia-en-nuestra-vida-diaria, 03/2018.
15. http://www.dma.fi.upm.es/recursos/aplicaciones/matematica_discreta/web/aritmetica_modular/polialfabeto.html, 03/2018.
16. https://bitcoin.org/bitcoin.pdf, Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 04/2018
17. https://github.com/ethereum/, 05/2018