



Facultad de Ciencias



TRABAJO DE FIN DE GRADO

Implementación de protocolos de conocimiento cero con fines docentes

Autor: Pedro Ramos Suárez
Doble Grado en Ingeniería Informática y Matemáticas

Tutor: Rafael Alejandro Rodríguez Gómez
Departamento de Teoría de la Señal, Telemática y Comunicaciones

CURSO 2022/2023

GRANADA, A 24 DE JUNIO DE 2023

Agradecimientos

A mis padres, Fernando y María José, por motivarme a elegir y continuar con la carrera en todo momento. A mis hermanos y a toda mi familia por siempre confiar en mí, por apoyarme y hacerme disfrutar de cada momento que pasamos juntos.

A mis amigos y compañeros de clase, en especial a David y Jesús, por estar siempre a mi lado y apoyándome, y por todos los momentos compartidos y el esfuerzo realizado para llegar hasta donde estamos.

A los profesores que me han dado clase durante la carrera y en particular, a mi tutor durante este trabajo, Rafael Alejandro Rodríguez Gómez, por toda su disponibilidad durante todos estos meses, eficacia contestando dudas y correcciones.

Resumen

En los últimos años, debido en parte a los ataques de ciberseguridad, la seguridad ha adquirido una gran importancia, especialmente en cualquier proceso online. Esto dificulta compartir información, ya que esto puede requerir revelar información confidencial.

Para solucionarlo, surgen los protocolos de conocimiento cero (ZKP), que pretenden demostrar que algo es verdadero o falso sin revelar ninguna información al respecto.

Implementar algoritmos de ZKP es y será fundamental en el próximo futuro así que se espera un aumento de la necesidad de profesionales en este ámbito. Por este motivo, el presente trabajo tiene como objetivo implementar una herramienta que facilite el aprendizaje de estos algoritmos.

Palabras clave: Protocolos de conocimiento cero, ZKP, Descomposición Cuadrada, Bulletproofs, Criptografía, Ciberseguridad, Herramienta Docente.

Abstract

In recent years, due in part to cybersecurity attacks, security has become very important, especially in any online process. This makes it difficult to share information, as this may require revealing confidential information.

To solve this, zero-knowledge protocols (ZKP) emerge, which aim to prove that something is true or false without revealing any information about it.

Implementing ZKP algorithms is and will be fundamental in the near future, so the need for professionals in this field is expected to increase. For this reason, the present work aims to implement a tool that facilitates the learning of these algorithms.

Keywords: Zero-knowledge protocols, ZKP, Square Decomposition, Bulletproofs, Cryptography, Cybersecurity, Teaching Tool.

Índice de contenidos

1. Introducción	1
1.1. Motivación	3
1.2. Solución Propuesta	5
1.3. Estructura de la memoria	6
2. Contexto de Zero Knowledge Proofs	8
2.1. Aplicaciones	8
2.2. Marco histórico	10
2.3. Protocolos de conocimiento cero en el contexto docente	11
3. Marco teórico	13
3.1. Clases de protocolos de conocimiento cero	14
3.2. Algoritmos basados en ZKP	16
3.3. Algoritmos ZKRP	18
3.3.1. Descomposición cuadrada (<i>Square decomposition</i>)	20
3.3.2. Basado en firma (<i>Signature-based</i>)	42
3.3.3. Bulletproofs	45
3.4. Selección de algoritmo	53
4. Objetivos y Planificación	57
4.1. Objetivos	57
4.2. Planificación	58
5. Desarrollo de la propuesta	61
5.1. Implementación de <i>Square Decomposition</i>	61
5.1.1. Bibliotecas	62
5.1.2. Implementación	62
5.2. Interfaz de la herramienta	76

5.3. Verificación del funcionamiento del algoritmo	79
6. Conclusiones y Líneas de Trabajo Futuro	85
6.1. Conclusiones	85
6.2. Líneas de Trabajo Futuro	86
Bibliografía	90
Anexo A. Manual de usuario para la herramienta docente	91
Ejecución del servidor	92
Elementos en común	92
Inicio	94
Pruebas	96
Verificaciones	98

Índice de tablas

1.1.	Ejemplo de prueba del color de las bolas [6]	2
1.2.	Pérdidas en redes sociales [8]	3
1.3.	Aumento de usos y valor de Zcash y Monero respectivamente	4
3.1.	Esquema de ZKP [13]	14
3.2.	ZK-STARK vs ZK-SNARK [14]	18
3.3.	Algoritmo extendido de Euclídes	24
3.4.	Gráfica de la curva <i>secp256k1</i>	46
3.5.	Comparación entre los distintos algoritmos [2]	54
3.6.	Comparación entre Bulletproofs, ZK-STARKS y ZK-SNARKS [15]	55
3.7.	Comparación entre Bulletproofs, ZK-STARKS y ZK-SNARKS [15]	55
3.8.	Comparación en milisegundos entre SHARP y Bulletproofs [3]	56
4.1.	Tiempo del desarrollo de la propuesta	60
5.1.	Esquema de la prueba	65
5.2.	Esquema de la verificación	66
5.3.	Esquema de las páginas web	80
5.4.	Matriz de confusión de los resultados	82
5.5.	Número de errores	83
5.6.	Tiempos de ejecución	84
1.	Página de la prueba de la Descomposición Cuadrada	93
2.	Página de la prueba de la Descomposición Cuadrada	94
3.	Página de inicio	95
4.	Ejemplo de campo de entrada	96
5.	Página de la prueba de la Descomposición Cuadrada	97
6.	Página de la verificación de la Descomposición Cuadrada	99

Capítulo 1

Introducción

Supongamos que tiene un amigo daltónico que no puede distinguir entre una bola verde y una roja (no sabe si las bolas son de diferentes colores). Quieres probar que los colores de las bolas son diferentes pero tu amigo necesita algo más que tus palabras para estar convencido. ¿Hay alguna forma de probar que son de colores distintos sin revelar información sobre las bolas?

Una prueba de conocimiento cero (*ZKP* o *Zero-knowledge proof*) es una forma de probar la validez de una declaración sin revelar la declaración en sí. Un protocolo de conocimiento cero es un método mediante el cual una parte (el probador) puede demostrar a otra parte (el verificador) que algo es cierto, sin revelar ninguna información aparte del hecho de que la afirmación específica es cierta.

Un método ZKP para el problema inicial podría dividirse en los siguientes pasos:

1. Tu amigo toma las bolas y te deja ver qué bola está en qué mano.
2. Luego, decide si cambiar las bolas entre sus manos detrás de su espalda o mantenerlas en el orden original.
3. A continuación, te presenta las bolas y te pregunta si las cambió o no. Como puedes distinguir la bola verde de la roja, puedes dar fácilmente la respuesta correcta.
4. Tu amigo no está convencido, ya que tienes un 50% de posibilidades de adivinar correctamente si cambió las bolas o no y las bolas aún pueden ser del mismo color.
5. Sin embargo, si se repite este experimento varias veces, eventualmente, la

probabilidad de que adivines correctamente si cambió las bolas o no cada vez sería muy baja. Esto le permite a su amigo verificar que las bolas son de diferentes colores sin saber los colores reales de las bolas.



Figura 1.1: Ejemplo de prueba del color de las bolas [6]

Las pruebas de conocimiento cero representaron un gran avance en la criptografía aplicada, permitiendo mejorar la seguridad de la información. Por ejemplo, podemos considerar como declaración “Ser ciudadano de X país”, que queremos probar a un proveedor de servicios. Para probar esta declaración podemos utilizar “pruebas”, como un pasaporte o un carnet de conducir. Pero hay problemas con este enfoque, principalmente la falta de privacidad. La información de identificación personal (*Personal Identifiable Information*) compartida con servicios de terceros se almacena en bases de datos centrales, que en caso de que sufran algún ataque podría ser revelada a personas no deseadas. Dado que el robo de identidad se está convirtiendo en un problema crítico, como podemos ver en la figura [Figura 1.2](#), se piden más medios de protección de la privacidad para compartir información confidencial.

Las pruebas de conocimiento cero resuelven este problema al eliminar la necesidad de revelar información para probar la validez de las afirmaciones. El protocolo de conocimiento cero utiliza la declaración (llamada “testigo”) como entrada para generar una prueba sucinta de su validez. Esta prueba proporciona fuertes garantías de que una declaración es verdadera sin exponer la información utilizada para crearla.

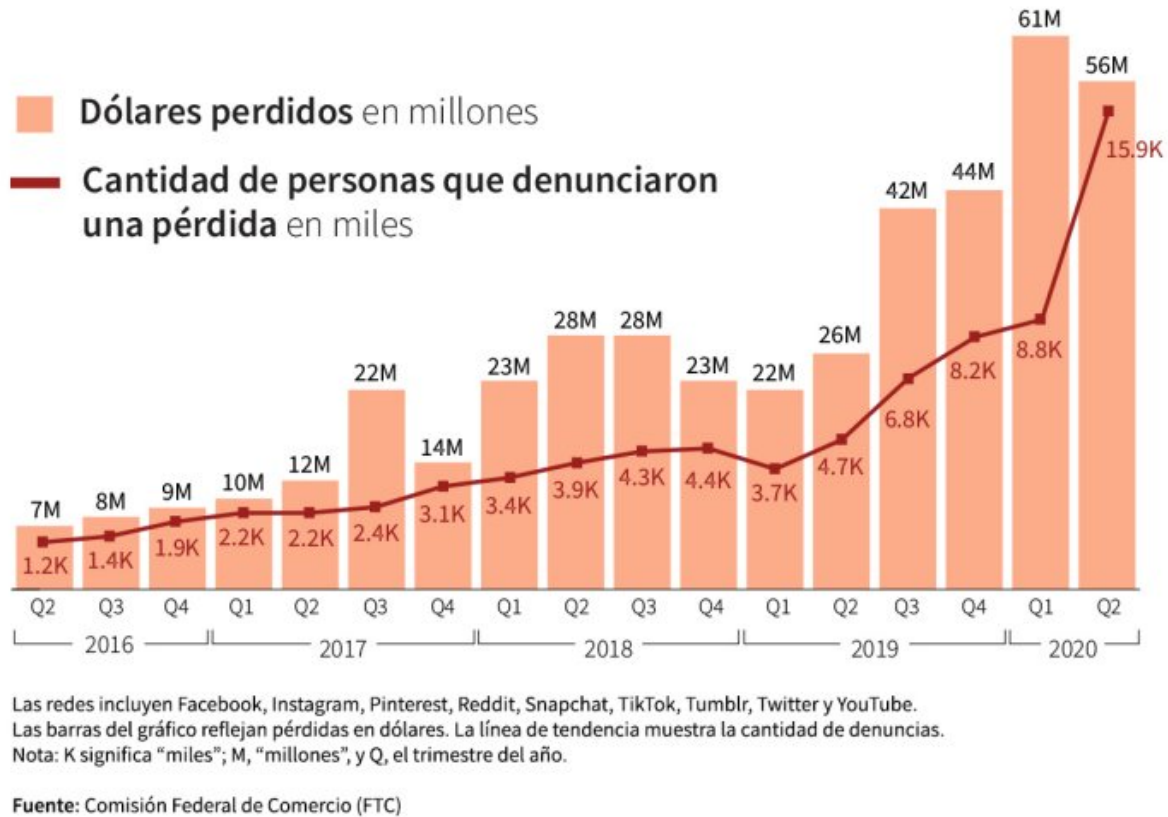


Figura 1.2: Pérdidas en redes sociales [8]

Volviendo a nuestro ejemplo anterior sobre la ciudadanía, la única evidencia que se necesita para probar la declaración de ciudadanía es una prueba de conocimiento cero. El verificador sólo tiene que verificar si ciertas propiedades de la prueba son verdaderas para estar convencido de que la declaración subyacente también lo es. Por ejemplo, más adelante veremos que podemos usar los protocolos de conocimiento cero para probar que un valor pertenece a un intervalo, sin dar ninguna información acerca de este valor. Esto nos permitiría probar que el número del pasaporte pertenece al intervalo correspondiente a un determinado país y así demostrar que es ciudadano de dicho país. O, si asociamos un número a cada país, podemos demostrar que el valor de su país pertenece a un intervalo que representa una región (como la Unión Europea) o a un continente sin revelar el país.

1.1. Motivación

Debido a la naturaleza de ZKP, puede utilizarse en muchos casos en los que la privacidad es deseable. Y lo que es más importante, también puede utilizarse

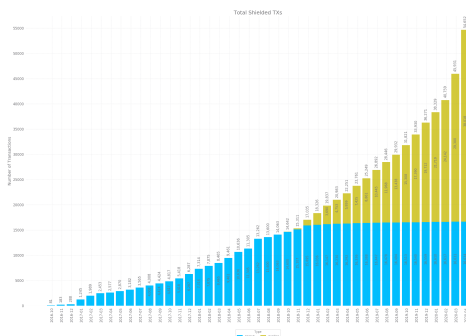
como base para construir protocolos más sofisticados.

Gracias a la enorme capacidad de anonimato, privacidad y seguridad de este tipo de protocolos, sus principales casos de uso apuntan a sistema de comunicación seguro. Por ejemplo, los militares y las organizaciones de espionaje emplean este tipo de tecnología para asegurar comunicaciones. Esto con el fin de permitir el despliegue en campo de sistemas de comunicación muy seguros. También son muy utilizados en sistemas de autenticación, incluso vía web [11].

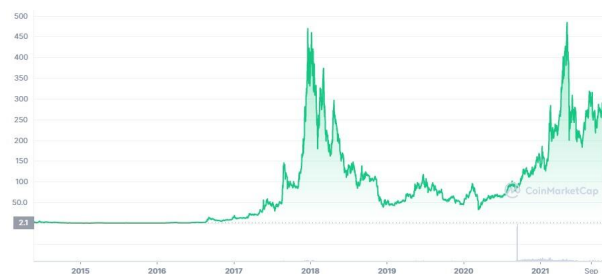
La tecnología además tiene amplios usos dentro de sistemas de votación seguros. Con las ZKP es posible que el votante pueda realizar su voto, demostrar que votó, pero de ninguna manera nadie podrá saber por qué opción ha votado. De esta forma, ZKP puede ayudar a los sistemas de votación a mantener el secreto del voto y otorgar transparencias a estos sistemas.

Otro caso de uso muy visto en la actualidad se da en las criptomonedas, como el caso de Zcash y Monero. Ambas criptomonedas implementan el uso de Zero Knowledge Protocol. Como es de esperar, la finalidad es poder garantizar la privacidad y anonimato de sus usuarios.

En el caso de Zcash, su sistema de pruebas zk-SNARKs está basado en el funcionamiento de ZKP. De estas, existe una evolución bajo el nombre zk-STARK que presentan mejores características en cuanto a seguridad y rendimiento, especialmente resistencia a computación cuántica. Por su lado, Monero y sus Bulletproof son también una adaptación de ZKP y Transacciones Confidenciales, lo que también le confiere un alto nivel de seguridad.



(a) Aumento de usos de Zcash [9]



(b) Evolución del valor de Monero [10]

Figura 1.3: Aumento de usos y valor de Zcash y Monero respectivamente

Como se puede ver en la [Figura 1.3](#), Zcash y Monero han estado aumentando en valor y en uso en los últimos años, por lo que cabe esperar que sean fundamentales

en el próximo futuro y un aumento de la necesidad de profesionales en este ámbito. Además, como vimos en la [Figura 1.2](#), los robos de identidad y las pérdidas que estos conllevan es un tema importante en la actualidad, y los ZKP se pueden utilizar para aumentar la seguridad frente a estos robos permitiendo ocultar la información. Todo esto nos motiva en un estudio de estos protocolos.

Sin embargo, ya existen diversos artículos que definen estos protocolos y plantean nuevos algoritmos con diversas capacidades y ventajas. Por lo que, decidimos centrarnos en la docencia y, tal y como recomienda el artículo *Transformation of the mathematics classroom with the internet* de 2020 [4], en lugar de crear un extenso documento acerca de su funcionamiento y de los distintos algoritmos relacionados (aunque si incluiremos un breve resumen para familiarizarse con su funcionamiento), decidimos crear una herramienta digital que permita interactuar con estos protocolos y así facilitar el aprendizaje sobre estas tecnologías.

1.2. Solución Propuesta

Para que pueda conocerse el funcionamiento de los protocolos de conocimiento cero, este trabajo intenta detallar un algoritmo de la forma más simple posible, implementando el algoritmo y ofreciendo al usuario una herramienta interactiva que le permite ver y modificar los valores, los distintos pasos y las salidas. De esta forma, siendo capaz de seguir paso a paso el algoritmo y modificando sus valores el usuario podrá comprender en mayor detalle el funcionamiento de un algoritmo de ZKP.

A continuación se detallan las contribuciones principales del presente trabajo fin de grado:

- Explicación teórica de los protocolos de conocimientos cero, así como algunos posibles ejemplos de sus aplicaciones.
- Explicación y comparación entre distintos algoritmos de conocimiento cero. Además, se indica porqué decidimos centrarnos en uno de ellos en particular.
- Implementación del algoritmo seleccionado.
- Herramienta interactiva con la que se puede interactuar con dicho algoritmo, indicándole los valores que tomará como entrada y pudiendo modificar los resultados que se obtienen, para poder entender su funcionamiento.
- Comprobación del correcto funcionamiento del algoritmo implementado, rea-

lizando un número elevado de pruebas funcionales con distintos valores aleatorios.

1.3. Estructura de la memoria

Este trabajo se divide en los siguientes capítulos:

- 1. Introducción:** El objetivo de este capítulo es presentar un breve resumen de los protocolos de conocimiento cero que veremos en este trabajo, así como motivar su importancia y el porqué de este trabajo. Finalmente, se presenta un breve resumen de la estructura de la memoria y el contenido de cada capítulo.
- 2. Contexto de Zero Knowledge Proofs:** Esta sección sirve para motivar el estudio de los protocolos de conocimiento cero, demostrando su utilidad hoy en día con algunos ejemplos de posibles aplicaciones. Además, se estudian qué otras aportaciones hay en la implementación de los protocolos de conocimiento cero con fines docentes.
- 3. Marco teórico:** En esta sección se definen los protocolos de conocimiento cero, y se explica el funcionamiento de algunos en detalle. Además, se ofrecen ejemplos numéricos de uno de ellos, el que será el centro de este trabajo. Finalmente, se realiza una comparación entre los distintos algoritmos, justificando porque nos centramos en el algoritmo seleccionado.
- 4. Objetivos y Planificación:** Una vez que se comprende qué son los algoritmos de conocimiento cero y se hace una idea de cuál es su funcionamiento, se presentan los objetivos de este TFG. También se añade una planificación temporal con el objetivo de presentar el esfuerzo dedicado a cada una de las tareas y subtareas del presente trabajo.
- 5. Desarrollo de la propuesta:** En este capítulo se detalla la implementación que se ha realizado, tanto del algoritmo seleccionado como de la herramienta docente, con el objetivo de facilitar la comprensión de dicho algoritmo. Además, se explicarán las distintas herramientas utilizadas.
- 6. Conclusiones y líneas de trabajo futuro:** Finalmente, se ofrece un breve resumen de todo el trabajo y los resultados obtenidos, para analizar si este trabajo consigue completar los objetivos que planteados. También se indican las principales líneas de trabajo futuro que surgen tras la finalización del presente trabajo fin de grado.

- **Anexos:** Su objetivo es servir como manual, para comprender como utilizar las distintas implementaciones de los algoritmos que se estudian en este trabajo, para que así se puedan utilizar y comprobar los resultados obtenidos con mayor facilidad.

Capítulo 2

Contexto de Zero Knowledge Proofs

2.1. Aplicaciones

Con el fin de motivar al lector a investigar más sobre las pruebas de conocimiento cero, este capítulo contiene algunas aplicaciones interesantes.

- Pagos anónimos: Los pagos con tarjeta de crédito a menudo son visibles para varias partes, incluido el proveedor de pagos, los bancos y otras partes interesadas (por ejemplo, autoridades gubernamentales). Si bien la vigilancia financiera tiene beneficios para identificar actividades ilegales, también socava la privacidad de los ciudadanos comunes.

Las criptomonedas estaban destinadas a proporcionar un medio para que los usuarios realizaran transacciones privadas entre pares. Pero la mayoría de las transacciones de criptomonedas son visibles abiertamente en cadenas de bloques públicas. Las identidades de los usuarios a menudo son seudónimas y están vinculadas intencionalmente a identidades del mundo real (por ejemplo, al incluir direcciones ETH en perfiles de Twitter o GitHub) o pueden asociarse con identidades del mundo real utilizando análisis de datos básicos dentro y fuera de la cadena.

Al incorporar tecnología de conocimiento cero en el protocolo, las redes de cadena de bloques centradas en la privacidad permiten que los nodos validen las transacciones sin necesidad de acceder a los datos de la transacción.

- **Autenticación:** El uso de servicios en línea requiere que demuestre su identidad y derecho a acceder a esas plataformas. Esto a menudo requiere proporcionar información personal, como nombres, direcciones de correo electrónico, fechas de nacimiento, etc. También es posible que deba memorizar contraseñas largas o correr el riesgo de perder el acceso.

Sin embargo, las pruebas de conocimiento cero pueden simplificar la autenticación tanto para las plataformas como para los usuarios. Una vez que se ha generado una prueba ZK utilizando entradas públicas (por ejemplo, datos que acrediten la membresía del usuario en la plataforma) y entradas privadas (por ejemplo, los detalles del usuario), el usuario puede simplemente presentarla para autenticar su identidad cuando necesite acceder al servicio. Esto mejora la experiencia de los usuarios y libera a las organizaciones de la necesidad de almacenar grandes cantidades de información de los usuarios.

- **Más de 18 ZKRP:** Se puede usar para demostrar que alguien tiene más de 18 años sin revelar su edad exacta. Así es posible permitir que la persona consuma algún servicio sin exigirle que muestre documentos en papel, que contienen más información de la necesaria para la validación de la edad.

En esta situación es importante contar con una parte de confianza para generar un compromiso, que acredite que la información contenida en el mismo es correcta. La persona no puede generar el compromiso por sí misma porque un usuario malintencionado podría utilizar datos falsos para probar la declaración deseada, aunque los datos reales no respeten esa propiedad.

- **Know Your Customer (KYC):** ZKRP permite validar que una determinada información privada pertenece a un intervalo numérico. Esta propiedad se puede utilizar para garantizar el cumplimiento y, al mismo tiempo, preservar la privacidad del cliente. Por ejemplo, un caso de uso interesante son las llamadas credenciales anónimas, donde una parte de confianza puede atestiguar que una credencial de usuario contiene atributos cuyos valores son correctos, lo que permite probar ciertas propiedades en forma de conocimiento cero.
- **Evaluación del riesgo hipotecario.** Es posible probar que el salario de una persona está por encima de cierto umbral para que se apruebe una hipoteca. En general, la validación del umbral es una verificación clave que debe realizarse en la evaluación del riesgo financiero. Por lo tanto, ZKRP resulta ser muy importante para las instituciones financieras.
- **Calificación y grado de inversión** El problema de calificar empresas según

su nivel de productividad o salud financiera se puede modelar determinando una partición de un intervalo numérico, dado por una secuencia de números crecientes A_0, A_1, \dots, A_k , tales que la puntuación más alta se atribuye a las empresas calificadas por encima de A_k (o, a veces, por debajo de A_0). La salud de la empresa se mide para obtener un valor x , y la nota resultante depende del sub-intervalo al que pertenezca x . Por lo tanto, es necesario verificar si $x \in [A_i, A_{i+1})$ para cada $0 \leq i \leq k$. Hasta donde sabemos, no hay investigaciones que apliquen ZKRP a este problema específico, donde tal vez podrían existir construcciones más eficientes, en comparación con la solución sencilla de usar ZKRP $k + 1$ veces.

- Voto electrónico: Este es un tema importante de investigación, que atrajo la atención de muchos investigadores en los últimos años. Se propusieron diferentes soluciones para diferentes tipos de elecciones. Algunas soluciones se basan en pruebas de conocimiento cero, como ZKRP, prueba de barajado, prueba de descifrado y otras técnicas relacionadas, mientras que otros utilizan diferentes primitivas criptográficas, como el cifrado de umbral homomórfico y la computación multipartita (MPC).
- Subastas y adquisiciones electrónicas: Las subastas electrónicas seguras es un tema que ha sido foco de investigación durante mucho tiempo, y es una motivación importante en el estudio de ZKRPs, ya que es una de las principales técnicas criptográficas que se pueden utilizar para construir protocolos seguros.

2.2. Marco histórico

Las pruebas de conocimiento cero fueron concebidas por primera vez en 1985 por Shafi Goldwasser, Silvio Micali y Charles Rackoff en su artículo “The Knowledge Complexity of Interactive Proof-Systems” [12]. Este documento introdujo la jerarquía IP de los sistemas de prueba interactivos y concibió el concepto de complejidad del conocimiento, una medida de la cantidad de conocimiento sobre la prueba transferida del probador al verificador. También dieron la primera prueba de conocimiento cero para un problema concreto, el de decidir los no residuos cuadráticos mod m . Junto con un artículo de László Babai y Shlomo Moran, este artículo histórico inventó sistemas de prueba interactivos, por los cuales los cinco autores ganaron el primer Premio Gödel en 1993.

En sus propias palabras, Goldwasser, Micali y Rackoff dicen:

“De particular interés es el caso donde este conocimiento adicional es esencialmente 0 y mostramos que es posible probar interactivamente que un número es cuadrático sin residuo mod m liberando 0 conocimiento adicional. Esto es sorprendente ya que no se conoce ningún algoritmo eficiente para decidir el mod m de los residuos cuadráticos cuando no se proporciona la factorización de m . Además, todas las pruebas NP conocidas para este problema exhiben la descomposición en factores primos de m . Esto indica que agregar interacción al proceso de prueba puede disminuir la cantidad de conocimiento que se debe comunicar para probar un teorema.”

Oded Goldreich, Silvio Micali y Avi Wigderson llevaron esto un paso más allá y demostraron que, suponiendo la existencia de un cifrado irrompible, se puede crear un sistema de prueba de conocimiento cero para el problema de coloreado de gráficos NP completos con tres colores. Dado que todos los problemas en NP (conjunto de problemas que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista) se pueden reducir eficientemente a este problema, esto significa que, bajo esta suposición, todos los problemas en NP tienen pruebas de conocimiento cero. El motivo de la suposición es que, como en el ejemplo anterior, sus protocolos requieren cifrado. Una condición suficiente comúnmente citada para la existencia de un cifrado irrompible es la existencia de funciones unidireccionales, pero es concebible que algunos medios físicos también puedan lograrlo.

Además de esto, también demostraron que el problema de no isomorfismo de gráficos, el complemento del problema de isomorfismo de gráficos, tiene una prueba de conocimiento cero. Este problema está en co-NP, pero actualmente no se sabe que esté en NP ni en ninguna clase práctica. De manera más general, Russell Impagliazzo y Moti Yung, así como Ben-Or et al. continuaría mostrando que, también asumiendo funciones unidireccionales o encriptación indescifrable, que hay pruebas de conocimiento cero para todos los problemas en $IP = PSPACE$, o en otras palabras, cualquier cosa que pueda probarse mediante un sistema de prueba interactivo puede probarse con cero conocimiento.

2.3. Protocolos de conocimiento cero en el contexto docente

Debido a la importancia de los protocolos de conocimiento cero en los últimos años, han surgido un gran número de artículos que se centran en estudiar sus posibles aplicaciones y los resultados que se pueden conseguir con ellos, como [17]

ó [18]. También han surgido artículos que se centran en el estudio de un nuevo algoritmo de conocimiento cero y comparar su rendimiento y resultados con otros ya existentes previamente, como [3]. E incluso existen algunos que recompilan varios de ellos para indicar las ventajas de unos frente a los otros, como [2].

Sin embargo, no hemos podido encontrar ninguno que se centre en el marco académico, intentando explicar el funcionamiento general de estos protocolos y mostrando ejemplos o proporcionando herramientas para facilitar el entendimiento de su funcionamiento.

Por ejemplo, podemos destacar artículos que han motivado este trabajo, como [1], que ofrece una explicación en detalle del algoritmo *Square Decomposition*, pero no ofrece pruebas ni herramientas que faciliten su comprensión ni demuestran su funcionamiento; o [2], que proporciona una visión general de los protocolos de conocimiento cero, centrándose en las *Bulletproofs* y sus posibles optimizaciones, pero igualmente sólo ofrece una descripción de su funcionamiento, detallando su algoritmo, pero que puede ser bastante difícil de comprender debido a la complejidad de dicho algoritmo.

Por esta ausencia de herramientas para facilitar la comprensión de los protocolos de conocimiento cero y por la importancia de dichos protocolos especialmente en el marco de la ciberseguridad, surge la motivación de este TFG de proporcionar una explicación sobre su funcionamiento general acompañado por distintas herramientas que intentan ofrecer una manera sencilla de comprender cómo funcionan y cuáles pueden ser sus posibles aplicaciones.

Es por eso que en este trabajo, en lugar de centrarnos en algoritmos que pueden resultar más potentes, decidimos centrarnos en otros más simples y fáciles de comprender, proporcionando una herramienta que podría facilitar su estudio y comprensión.

Capítulo 3

Marco teórico

Una prueba de conocimiento cero le permite probar la verdad de una declaración sin compartir el contenido de la declaración o revelar cómo descubrió la verdad. Para que esto sea posible, los protocolos de conocimiento cero se basan en algoritmos que toman algunos datos como entrada y devuelven “verdadero” o “falso” como salida.

Se utiliza una serie de algoritmos criptográficos en las aplicaciones del mundo real de los ZKP para permitir la verificación de una declaración computacional. Por ejemplo, utilizando métodos ZKP, un receptor de pago puede verificar que el pagador tiene suficiente saldo en su cuenta bancaria sin obtener ninguna otra información sobre el saldo del pagador.

Un protocolo de conocimiento cero debe satisfacer los siguientes criterios:

- Completitud (*Completeness*): Si la entrada es válida, el protocolo de conocimiento cero siempre devuelve “verdadero”. Por lo tanto, si la declaración subyacente es verdadera y el probador y el verificador actúan honestamente, la prueba puede ser aceptada.
- Solidez (*Soundness*): Si la entrada no es válida, es teóricamente imposible engañar al protocolo de conocimiento cero para que devuelva “verdadero”. Por lo tanto, un probador mentiroso no puede engañar a un verificador honesto para que crea que una declaración inválida es válida (excepto con un pequeño margen de probabilidad).
- Conocimiento cero (*Zero-knowledge*): El verificador no aprende nada sobre una declaración más allá de su validez o falsedad (tiene “conocimiento ce-

ro” de la declaración). Este requisito también evita que el verificador derive la entrada original (el contenido de la declaración) de la prueba.

El esquema de un protocolo de conocimiento cero queda resumido en la [Figura 3.1](#):



Figura 3.1: Esquema de ZKP [13]

3.1. Clases de protocolos de conocimiento cero

Lo anterior describe la estructura de una **prueba interactiva de conocimiento cero**. Los primeros protocolos de conocimiento cero usaban pruebas interactivas, donde verificar la validez de una declaración requería una comunicación de ida y vuelta entre probadores y verificadores.

Esta prueba interactiva tiene una utilidad limitada, ya que requiere que las dos partes estén disponibles e interactúen repetidamente. Incluso si un verificador está convencido de la honestidad de un probador, la prueba no está disponible para una verificación independiente (el cálculo de una nueva prueba requiere un nuevo conjunto de mensajes entre el probador y el verificador).

Para resolver este problema, surgen las **pruebas de conocimiento cero no interactivas** donde el probador y el verificador tienen una clave compartida. Esto permite que el probador demuestre su conocimiento de cierta información (es decir, testigo) sin proporcionar la información en sí.

A diferencia de las pruebas interactivas, las pruebas no interactivas requieren sólo una ronda de comunicación entre los participantes (proveedor y verificador). El probador pasa la información secreta a un algoritmo especial para calcular una prueba de conocimiento cero. Esta prueba se envía al verificador, quien comprueba que el probador conoce la información secreta utilizando otro algoritmo.

La prueba no interactiva reduce la comunicación entre el probador y el verificador, lo que hace que las pruebas de conocimiento cero sean más eficientes. Además, una vez que se genera una prueba, está disponible para que cualquier otra persona (con acceso al algoritmo de verificación) la verifique.

Las pruebas de conocimiento cero pueden expresarse de la siguiente forma: Dado un elemento x de un lenguaje $\mathcal{L} \in NP$, una entidad llamada probador es capaz de convencer a un verificador de que x pertenece efectivamente a \mathcal{L} , es decir, existe un testigo w para x .

Un esquema de prueba de conocimiento cero no interactivo (NIZK) se define mediante los algoritmos de configuración, prueba y verificación de la siguiente manera:

- El algoritmo de configuración (*Setup*): Es responsable de la generación de parámetros. Concretamente, tenemos que $params = Setup(\lambda)$, donde la entrada es el parámetro de seguridad λ y la salida son los parámetros del sistema de algoritmos ZKP.
- La sintaxis de prueba (*Prove*): Viene dada por $proof = Prove(x, w)$. El algoritmo recibe como entrada una instancia x de algún lenguaje NP \mathcal{L} , y el testigo w , y genera la prueba de conocimiento cero.
- El algoritmo de verificación (*Verify*): Recibe la prueba $proof$ como entrada y genera un bit b , que es igual a 1 si el verificador acepta la prueba, o 0 si la rechaza.

Con estos términos, los criterios vistos previamente se pueden expresar de la siguiente forma:

- Completitud (*Completeness*): Acepta si la entrada es verdadera, es decir, dado un testigo w que satisface la instancia x , tenemos que

$$Verify(Prove(x, w)) = 1$$

- Solidez (*Soundness*): Si la entrada no es válida, rechaza excepto con un pequeño margen de error; es decir, si el testigo w no satisface x , entonces la

probabilidad

$$P[\text{Verify}(\text{Prove}(x, w)) = 1]$$

es suficientemente baja.

- Conocimiento cero (*Zero-knowledge*): El verificador no aprende nada sobre una declaración más allá de su validez o falsedad. Para ello, dada la interacción entre el probador y el verificador, llamamos a esta interacción una vista, y para capturar la propiedad de conocimiento cero, usamos un simulador de tiempo polinomial, que tiene acceso a la misma entrada proporcionada al verificador (incluida su aleatoriedad), pero no acceso a la entrada del probador, para generar una vista simulada. Decimos que el esquema ZKP tiene conocimiento cero perfecto si la vista simulada, bajo el supuesto de que $x \in \mathcal{L}$, tiene la misma distribución que la vista original. Decimos que el esquema ZKP tiene conocimiento estadístico cero si esas distribuciones son estadísticamente cercanas. Decimos que el esquema ZKP tiene conocimiento computacional cero si no hay un distintivo de tiempo polinomial para esas distribuciones. Intuitivamente, la existencia de un simulador de este tipo significa que cualquier cosa que el verificador pueda calcular a partir de la interacción con el probador, ya era posible calcularla antes de tal interacción, por lo que el verificador no aprendió nada de ella. Además, decimos que es una prueba de conocimiento si podemos encontrar un extractor, que tiene acceso de caja negra rebobinable al probador, que puede calcular el testigo w con una probabilidad no despreciable.

3.2. Algoritmos basados en ZKP

ZK-SNARK es un acrónimo de *Zero-Knowledge Succinct Non-Interactive Argument of Knowledge*. El protocolo ZK-SNARK tiene las siguientes cualidades:

- Conocimiento cero (*Zero-knowledge*): Un verificador puede validar la integridad de una declaración sin saber nada más sobre la declaración. El único conocimiento que tiene el verificador de la declaración es si es verdadera o falsa.
- Sucinto (*Succinct*): La prueba de conocimiento cero es más pequeña que el testigo y se puede verificar rápidamente.
- No interactivo (*Non-interactive*): La prueba es “no interactiva” porque el probador y el verificador solo interactúan una vez, a diferencia de las pruebas

interactivas que requieren múltiples rondas de comunicación.

- Argumento (*Argument*): La prueba satisface el requisito de “solidez”, por lo que es extremadamente improbable hacer trampa.
- De conocimiento (*Of Knowledge*): La prueba de conocimiento cero no puede construirse sin acceso a la información secreta (testigo). Es difícil, si no imposible, para un probador que no tiene el testigo calcular una prueba válida de conocimiento cero.

ZK-STARK es un acrónimo de Zero-Knowledge Scalable Transparent Argument of Knowledge. Los ZK-STARK son similares a los ZK-SNARK. Sin embargo añade las siguientes cualidades:

- Escalable (*Scalable*): ZK-STARK es más rápido que ZK-SNARK en la generación y verificación de pruebas cuando el tamaño del testigo es mayor. Con las pruebas STARK, los tiempos de prueba y verificación solo aumentan ligeramente a medida que crece el testigo (los tiempos de prueba y verificación de SNARK aumentan linealmente con el tamaño del testigo).
- Transparente (*Transparent*): ZK-STARK se basa en la aleatoriedad verificable públicamente para generar parámetros públicos para probar y verificar en lugar de una configuración confiable. Por lo tanto, son más transparentes en comparación con los ZK-SNARK.

Los ZK-STARK producen pruebas más grandes que los ZK-SNARK, lo que significa que generalmente tienen gastos generales de verificación más altos. Sin embargo, hay casos (como probar grandes conjuntos de datos) en los que los ZK-STARK pueden ser más rentables que los ZK-SNARK.

Finalmente, como caso particular de los algoritmos de cero conocimiento tenemos la prueba de rango de conocimiento cero (*Zero Knowledge Range Proof*), que permiten probar que un valor entero secreto pertenece a un intervalo. Por ejemplo, si definimos el intervalo como todos los enteros entre 18 y 200, una persona puede usar ZKRP para probar que es mayor de edad.

En este trabajo, nos centraremos en algoritmos de este último tipo, en el que el probador quiere demostrar que conoce un valor secreto que pertenece a un intervalo.

Comparison between zk-STARKs and zk-SNARKS

	zk-STARKs	zk-SNARKs
Trusted set up requirement	To build trustworthy verifiable computing systems, zk-STARKs leverage publicly verifiable randomness	zk-SNARKs require a trusted setup phase
Scalability	More scalable than zk-SNARKs	Less scalable than zk-STARKs
Resistance to attacks from quantum computers	zk-STARKs are quantum attacks resistant	Quantum computers can attack zk-SNARKs because snarks use public-private key pairs

 cointelegraph.com

Figura 3.2: ZK-STARK vs ZK-SNARK [14]

3.3. Algoritmos ZKRP

Este trabajo se va a centrar en el estudio de una prueba de rango de conocimiento cero que será implementada para la herramienta docente. Esta prueba va a ser la Descomposición cuadrada (*Square Decomposition*), elección que se justificará más adelante en la sección [Selección de algoritmo](#). Debido a esto, aunque se comenten las distintas pruebas que existen, se entrará más en detalle y se añadirán ejemplos numéricos exclusivamente de este algoritmo.

Las ZKRP pueden clasificarse en:

- Propuestas de representación entera:
 - Descomposición cuadrada (*Square decomposition*): Una de las ideas que se pueden utilizar para obtener pruebas de rango de conocimiento cero es la descomposición del elemento secreto en una suma de cuadrados. Para probar que $x \in [a, b]$ es equivalente a probar que $x - a \geq 0$ y que $b - x \geq 0$.

Entonces, si podemos probar que $x - a$ y $b - x$ se pueden expresar como cuadrados con una cierta tolerancia θ , estaremos probando que son positivos y, por lo tanto, que el secreto x pertenece al intervalo $[a, b]$.

El problema de este procedimiento es que es de tiempo $\mathcal{O}(k^4)$, donde k es el tamaño del elemento secreto, por lo que suele provocar un mal rendimiento con pruebas de tamaño elevado.

- Basado en firma (*Signature-based*): Inicialmente, todos los elementos en el intervalo están firmados, y la prueba de que el probador conoce la firma significa que este número entero pertenece al intervalo esperado, que puede ser cualquier conjunto finito posible.
- Propuestas de representación binaria:
 - Decomposición multi-base (*Multi-base decomposition*): Consiste en descomponer el elemento secreto en la representación binaria, lo que permite probar que pertenece al intervalo usando aritmética booleana. El probador tiene que proveer una prueba *zero knowledge* de que cada bit del elemento secreto es de hecho un bit, y que la representación es válida. En lugar de una descomposición en bits, podemos usar una representación u-aria, obteniendo una mayor eficiencia.
 - Compromisos homomórficos de dos niveles (*Two-tiered homomorphic commitments*): Se basa en un argumento para multiplicación en *batches* de elementos de \mathbb{Z}_p , que puede ser usado para probar que $u_i v_i = w_i$, con $u_i, v_i, w_i \in \mathbb{Z}_p$ para $0 < i < N$, siendo N la longitud en bits del secreto. Para construir ZKRP, si los bits del secreto están dados por w_i , entonces el argumento puede usarse para mostrar que $w_i w_i = w_i$, lo que demuestra al verificador de que, de hecho, $w_i \in \{0, 1\}$. Además, mostró cómo probar que $w = \sum_{i=0}^N w_i 2^i$, demostrando así $w \in [0, 2^N)$. El argumento se puede adaptar fácilmente a un intervalo general $[A, B)$. La idea clave de la construcción de Groth es utilizar emparejamientos bilineales para comprometerse con un vector de compromisos de Pedersen. Por ejemplo, dado el emparejamiento $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ y los elementos $v, u_1, \dots, u_N \in \mathbb{G}_2$, podemos comprometernos con el vector $[c_1, \dots, c_N] \in \mathbb{G}_1^n$ eligiendo un número aleatorio $t \in \mathbb{G}$ y calculando $C = e(t, v) \prod_{i=0}^N e(c_i, u_i)$.
 - Bulletproofs: Todos los algoritmos mencionados anteriormente dependen de una configuración confiable, lo que es un problema en el contexto de las criptomonedas, ya que si un adversario puede evitar esta configuración confiable, podría crear dinero de la nada. Recientemente, en 2017, *Stanford Applied Cryptography Group* propuso una nueva idea

para construir ZKRP, llamada Bulletproofs. Consiste en utilizar una prueba interna del producto para lograr ZKRP con tamaños de prueba muy pequeños.

En resumen, utilizando este protocolo ZKP, el probador convence a un verificador de que conoce vectores cuyo producto interno es igual a un valor público determinado.

A continuación se desarrollan, explicando su funcionamiento, los algoritmos de representación entera (Descomposición Cuadrada y Basados en firmas) y el más relevante de representación binaria, *Bulletproofs*. Como se comentó previamente, este trabajo se centrará en la Descomposición Cuadrada, por lo que será el que se desarrollará en más detalle. Los *Bulletproofs*, al ser el único que se detalla de representación binaria, también estará algo más extendido. Finalmente, los Basados en firmas se comentarán para tener un algoritmo de representación entera con el que poder comparar el funcionamiento, por lo que sólo se hará un breve resumen al respecto.

3.3.1. Descomposición cuadrada (*Square decomposition*)

Vamos a estudiar la Descomposición cuadrada en más detalle, desarrollando en más detalle su funcionamiento. Para ello, añadiremos los distintos algoritmos que utiliza, así como ejemplos numéricos que faciliten su entendimiento. Además, desarrollaremos algunos de los conceptos matemáticos en los que se basa dicho algoritmo.

Sea un entero positivo x , y sea $E = g^x h^r$, y supongamos que Alice quiere probar a Bob que $x \in [a, b]$.

Inicialmente, Alice escribe el entero positivo $x - a$ como la suma de x_1^2 , el cuadrado mayor menor que x , y de ρ , un número positivo menor que $2\sqrt{x - a}$. Luego, selecciona aleatoriamente r_1, r_2 en $[0, 2^s n - 1]$ de modo que $r_1 + r_2 = r$ y calcula $E_1 = g^{x_1^2} h^{r_1}$ y $E_2 = g^\rho h^{r_2}$. Lee demuestra a Bob que E_1 oculta un cuadrado y que E_2 oculta un número cuyo valor absoluto es menor que $2^{t+\ell+1}\sqrt{b - a}$ mediante una prueba CFT. Finalmente, aplica el mismo método a $b - x$. Esto conduce a una demostración de que $x \in [a - 2^{t+\ell+1}\sqrt{b - a}, b + 2^{t+\ell+1}\sqrt{b - a}]$. La tasa de expansión de esta prueba es igual a $1 + (2^{t+\ell+1}/\sqrt{b - a})$, que se acerca a 1 cuando $b - a$ es grande.

Luego, aumentamos artificialmente el tamaño de x estableciendo $x' = 2^T x$. Usando el esquema anterior, demostramos que $x' \in [2^T a - 2^{t+\ell+T/2+1}\sqrt{b - a}, 2^T b +$

$2^{t+\ell+1}\sqrt{b-a}$, y si T es suficientemente grande (es decir, T es tal que $2^{t+\ell+T/2+1}\sqrt{b-a} < 2^T$), Bob está convencido de que $x' \in [2^T a - 2^T + 1, 2^T b + 2^T - 1]$, y entonces $x \in [a - \epsilon, b + \epsilon]$ donde $0 \leq \epsilon < 1$. Entonces, como x es un número entero, Bob está convencido de que $x \in [a, b]$.

Esta construcción requiere algunos bloques de construcción, como la prueba de conocimiento cero de que dos compromisos ocultan el mismo secreto y la prueba de conocimiento cero de que el secreto es un cuadrado.

Notación

Denotamos por $x \in_R [a, b]$ a la elección de forma aleatoria de un elemento x en el intervalo $[a, b]$.

Denotamos por $\lfloor x \rfloor$ al entero más cercano a x , por $\lfloor x \rfloor$ al entero menor que x más cercano a x , y por $\lceil x \rceil$ al entero mayor que x más cercano a x . Es decir, si $x \in [a, a + 1]$, con $a \in \mathbb{Z}$, entonces $\lfloor x \rfloor = a + 1$ y $\lceil x \rceil = a$. Nótese que si $x \in \mathbb{Z}$, entonces $\lfloor x \rfloor = \lfloor x \rfloor = \lceil x \rceil = x$.

Finalmente, denotamos por $a||b$ a la concatenación de los *strings* a y b .

Propiedades

En los siguientes algoritmos utilizaremos los siguientes elementos:

Definición 1 (Grupo) *Un grupo G es un conjunto no vacío junto con una operación interna $+$: $G \times G \rightarrow G$ satisfaciendo:*

1. *Propiedad asociativa:*

$$(a + b) + c = a + (b + c) \quad \forall a, b, c \in G$$

2. *Existencia del elemento neutro 0:*

$$0 + a = a + 0 = a \quad \forall a \in G$$

3. *Existencia de opuestos:*

$$\forall a \in G \quad \exists -a \in G \text{ tal que } a - a = 0$$

Si además verifica la propiedad conmutativa ($ab = ba \forall a, b \in G$), entonces el grupo es abeliano o conmutativo.

Definición 2 (Anillo) *Un anillo unitario es una terna $(A, +, \cdot)$ tal que $(A, +)$ es un grupo abeliano y con la operación \cdot al producto se verifica:*

1. *Asociatividad del producto:*

$$(ab)c = a(bc) \quad \forall a, b, c \in A$$

2. *Existencia del elemento neutro 1:*

$$\exists 1 \in A \text{ tal que } a1 = a = 1a$$

3. *Propiedad distributiva:*

$$a(b + c) = ab + ac \quad \forall a, b, c \in A$$

donde $ab = a \cdot b \quad \forall a, b \in A$.

Definición 3 (Inversos multiplicativos) *En un anillo unitario A , se definen los inversos multiplicativos como:*

- *El elemento $b \in A$ es inverso multiplicativo por la izquierda o inverso por la izquierda de $a \in A$ si:*

$$b \cdot a = 1$$

- *El elemento $c \in A$ es inverso multiplicativo por la derecha o inverso por la derecha de $a \in A$ si:*

$$a \cdot c = 1$$

Un elemento no tiene que tener inverso, o puede ser que tenga inverso por la izquierda pero no por la derecha, o viceversa.

Si un elemento $a \in A$ tiene un inverso por la izquierda y por la derecha, entonces ambos son iguales y se denota como elemento inverso a^{-1} .

En un anillo $\mathbb{Z}_n = \{0, 1, 2, \dots, n-2, n-1\}$, el elemento $a \in \mathbb{Z}_n$ tiene inverso si y sólo si $\text{mcd}(a, n) = 1$, es decir, si a y n son primos relativos.

Definición 4 (Congruencia) *Dado un anillo A , una congruencia (de anillos) en A es una relación de equivalencia \equiv en A compatible con la estructura de anillo, lo que significa:*

- *Compatible con la suma:*

$$\begin{cases} x \equiv y \\ z \equiv t \end{cases} \Rightarrow x + z \equiv y + t$$

- *Compatible con el producto:*

$$\begin{cases} x \equiv y \\ z \equiv t \end{cases} \Rightarrow xz \equiv yt$$

- *Compatible con los opuestos:*

$$x \equiv y \Rightarrow -x \equiv -y$$

Definición 5 (Algoritmo extendido de Euclides) *Para el cálculo de inversos utilizaremos el algoritmo extendido de Euclides que consiste en, a partir de dos elementos a y b , con $a < b$, calculamos su máximo común divisor $\text{mcd}(a, b)$ de la siguiente forma:*

Si uno de los elementos involucrados es nulo, la cuestión es fácil:

$$\text{mcd}(a, 0) = a = 1 \cdot a + 0 \cdot 0$$

Supongamos que $a, b \neq 0$, y el algoritmo consiste en reconstruir, recursivamente, una sucesión de elementos del anillo r_1, r_2, \dots, r_n partiendo de $r_1 = a$ y $r_2 = b$, por el siguiente procedimiento:

Si $r_i \neq 0$ entonces r_{i+1} es un resto de dividir r_{i-1} entre r_i

Así, r_3 es el resto de dividir $r_1 = a$ entre $r_2 = b$. Si $r_3 \neq 0$, entonces r_4 es el resto de dividir r_2 entre r_3 , etc.

La sucesión de números $r_1, r_2, \dots, r_i, \dots$ es estrictamente decreciente y no puede continuar indefinidamente, es decir, habrá un $n \geq 1$ tal que $r_{n+1} = 0$. Entonces tenemos que:

$$\text{mcd}(a, b) = r_n$$

En efecto, probamos por inducción que para todo $i = 1, 2, \dots, n$ se verifica que $(a, b) = (r_i, r_{i+1})$. Para $i = 1$, esto es obvio, pues $r_1 = a$ y $r_2 = b$. Supongamos

que $i > 1$ y que $(a, b) = (r_{i-1}, r_i)$. Si q_i es el cociente de dividir r_{i-1} entre r_i , será $r_{i+1} = r_{i-1} - r_i q_i$. Entonces:

$$(a, b) = (r_{i-1}, r_i) = (r_i, r_{i-1} - r_i q_i) = (r_i, r_{i+1})$$

Finalmente, $(a, b) = (r_{n-1}, r_n) = r_n$ ya que $r_n \mid r_{n-1}$ al ser $r_{n+1} = 0$.

Este algoritmo queda expresado en la siguiente tabla:

	a	1	0
	b	0	1
q_2	r_3	$u_3 = 1 - 0 \cdot q_2$	$v_3 = 0 - q_2 \cdot 1$
\dots	\dots	\dots	\dots
q_{i-2}	r_{i-1}	u_{i-1}	v_{i-1}
q_{i-1}	r_i	u_i	v_i
q_i	r_{i+1}	$u_{i+1} = u_{i-1} - u_i \cdot q_i$	$v_{i+1} = v_{i-1} - q_i \cdot v_i$
\dots	\dots	\dots	\dots

Tabla 3.3: Algoritmo extendido de Euclídes

Proposición 1 (Cálculo de inversos) Para el cálculo del inverso de a en \mathbb{Z}_n con $(a, n) = 1$ utilizaremos el algoritmo extendido de Euclídes definido previamente. Supongamos que queremos encontrar b tal que:

$$a^{-1} \equiv b \pmod{n}$$

O, equivalentemente, encontrar b tal que:

$$a \cdot b \equiv 1 \pmod{n}$$

Esto es:

$$a \cdot b + n \cdot x = 1$$

Utilizamos el algoritmo extendido de Euclídes para calcular

$$1 = (a, n) = u \cdot a + v \cdot n$$

Por lo que:

$$1 \equiv u \cdot a + v \cdot n \equiv u \cdot a \pmod{n}$$

Tenemos que u es el inverso de a .

Prueba de que dos compromisos esconden el mismo secreto

Denotamos la prueba de conocimiento cero de que dos compromisos ocultan el mismo secreto por $PK_{SS} = \{x, r_1, r_2 : E = g_1^x h_1^{r_1} \pmod{n} \wedge F = g_2^x h_2^{r_2} \pmod{n}\}$. Los parámetros para el esquema PK_{SS} están dados por $param_{SS} = (t, \ell, s_1, s_2)$, que deben configurarse para lograr el nivel de seguridad deseado. Es decir, tenemos que la solidez viene dada por 2^{t-1} , mientras que la propiedad de conocimiento cero está garantizada dado que $1/\ell$ es despreciable.

Sea n un número con el mínimo número de divisores posible (preferiblemente primo) cuya factorización es desconocida por el probador y verificador, g_1 un elemento de orden grande en \mathbb{Z}_n^* y g_2, h_1, h_2 elementos del grupo generado por g_1 tales que el logaritmo discreto de g_1 en base h_1 , el logaritmo discreto de h_1 en base g_1 , el logaritmo discreto de g_2 en base h_2 y el logaritmo discreto de h_2 en base g_2 son desconocidos para el probador. La función hash es tal que genera cadenas de $2t$ bits. Finalmente, tenemos que s_1 y s_2 deben elegirse para tener compromisos seguros, es decir, 2^{s_i} debe ser despreciable para $i \in \{1, 2\}$.

Sea $x \in_R [0, b]$. Entonces r_1 y r_2 son tal que $r_1 \in_R [-2^{s_1}n + 1, 2^{s_1}n - 1]$ y $r_2 \in_R [-2^{s_2}n + 1, 2^{s_2}n - 1]$.

Supongamos que Alice conoce secretamente que $x \in [0, b]$, y sean $E = g_1^x h_1^{r_1} \pmod{n}$ y $F = g_2^x h_2^{r_2} \pmod{n}$. Quiere probar a Bob que conoce x, r_1, r_2 tal que $E = g_1^x h_1^{r_1} \pmod{n}$ y $F = g_2^x h_2^{r_2} \pmod{n}$, es decir, que E y F esconden el mismo secreto. Entonces el protocolo es el siguiente:

Algorithm 1: Prueba del mismo secreto: $Prove_{SS}$

Input: $x, r_1, r_2, g_1, g_2, h_1, h_2, F, param_{SS}$

Output: $proof_{SS}$

$w \in_R [1, 2^{\ell+t}b - 1]$

$\eta_1 \in_R [1, 2^{\ell+t+s_1}n - 1]$

$\eta_2 \in_R [1, 2^{\ell+t+s_2}n - 1]$

$\Omega_1 = g_1^w h_1^{\eta_1} \pmod{n}$

$\Omega_2 = g_2^w h_2^{\eta_2} \pmod{n}$

$c = \text{Hash}(\Omega_1 || \Omega_2)$

$D = w + cx$

$D_1 = \eta_1 + cr_1$

$D_2 = \eta_2 + cr_2$

return $proof_{SS} = (c, D, D_1, D_2)$

Bob puede realizar la comprobación de la siguiente forma:

Algorithm 2: Prueba del mismo secreto: $\text{Verify}_{\text{SS}}$

Input: $E, F, n, g_1, g_2, h_1, h_2, \text{proof}_{\text{SS}}$
Output: True o False

if $c == \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$ **then**

 | **return** *True*
else

 | **return** *False*
end

Fácilmente podemos comprobar que esta verificación es correcta si la prueba se ha generado con el algoritmo anterior, ya que:

$$\begin{aligned}
 & \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n)) = \\
 & = \text{Hash}(g_1^{w+cx} h_1^{\eta_1+cr_1} (g_1^x h_1^{r_1})^{-c}(\text{mod } n) || g_2^{w+cx} h_2^{\eta_2+cr_2} (g_2^x h_2^{r_2})^{-c}(\text{mod } n)) = \\
 & = \text{Hash}(g_1^w g_1^{cx} g_1^{-cx} h_1^{\eta_1} h_1^{cr_1} h_1^{-cr_1}(\text{mod } n) || g_2^w g_2^{cx} g_2^{-cx} h_2^{\eta_2} h_2^{cr_2} h_2^{-cr_2}(\text{mod } n)) = \\
 & = \text{Hash}(g_1^w h_1^{\eta_1}(\text{mod } n) || g_2^w h_2^{\eta_2}(\text{mod } n)) = \text{Hash}(\Omega_1 || \Omega_2) = c
 \end{aligned}$$

Ejemplo

Sean los parámetros de seguridad $t = 5, l = 3, s_1 = 4, s_2 = 6$.

Sean $x = 13, n = 13 * 17 = 221, g_1 = 7, g_2 = 14, h_1 = 21, h_2 = 28, b = 30$.

Entonces tenemos

$$r_1 \in_R [-2^{s_1}n+1, 2^{s_1}n-1] = [-2^4*221+1, 2^4*221-1] = [-3535, 3535] \Rightarrow r_1 = -101$$

$$r_2 \in_R [-2^{s_2}n+1, 2^{s_2}n-1] = [-2^6*221+1, 2^6*221-1] = [-14143, 14143] \Rightarrow r_2 = 1115$$

Nótese que tenemos:

$$21^{-101}(\text{mod } 221) = 200$$

ya que, usando el algoritmo extendido de Euclídes, tenemos:

221	1	0
21	0	1
10	11	$1-0*10 = 1$ $0-1*10 = -10$
1	10	$0-1*1=-1$ $1-(-10)*1 = 11$
1	1	$1-(-1)*1 = 2$ $-10-11*1 = -21$

Es decir, $221 * 2 + 21 * (-21) = 1$, por lo que $-21 = 221 - 21 = 200$ es el inverso de 21 y, por tanto:

$$21^{-101}(\text{mod } 221) = (21^{-1})^{101}(\text{mod } 221) = 200^{101}(\text{mod } 221)$$

Y como:

$$200^4(\text{mod } 221) = 1600000000(\text{mod } 221) = 1$$

Tenemos:

$$21^{-101}(\text{mod } 221) = 200^{101}(\text{mod } 221) = 200^{101 \% 4}(\text{mod } 221) = 200^1(\text{mod } 221) = 200$$

Donde $\%$ es el resto de la división, y hemos usado las propiedades vistas previamente.

Para el resto de cálculos similares de ahora en adelante indicaré directamente el resultado sin realizar todos estos pasos.

Finalmente

$$E = g_1^x h_1^{r_1}(\text{mod } n) = 7^{13} 21^{-101}(\text{mod } 221) = 7^{13} 200(\text{mod } 221) = 61$$

$$F = g_2^x h_2^{r_2}(\text{mod } n) = 14^{13} 28^{1115}(\text{mod } 221) = 111$$

Con todos estos valores, podemos comenzar el algoritmo de la [Prueba de que dos compromisos esconden el mismo secreto](#), que tiene como entrada:

$$(x, r_1, r_2, E, F, t, \ell, s_1, s_2) = (13, -101, 1115, 61, 111, 5, 3, 4, 6)$$

Y el algoritmo sería:

$$w \in_R [1, 2^{\ell+t} b - 1] = [1, 2^{3+5} 30 - 1] = [1, 7679] \Rightarrow w = 5247$$

$$\eta_1 \in_R [1, 2^{\ell+t+s_1} n - 1] = [1, 2^{3+5+4} 36 - 1] = [1, 147455] \Rightarrow \eta_1 = 96487$$

$$\eta_2 \in_R [1, 2^{\ell+t+s_2} n - 1] = [1, 2^{3+5+6} 36 - 1] = [1, 589823] \Rightarrow \eta_2 = 274978$$

$$\Omega_1 = g_1^w h_1^{\eta_1}(\text{mod } n) = 7^{5247} 21^{96487}(\text{mod } 221) = 116$$

$$\Omega_2 = g_2^w h_2^{\eta_2}(\text{mod } n) = 14^{5247} 28^{274978}(\text{mod } 221) = 192$$

Como función *Hash* utilizaremos la identidad, es decir, $\text{Hash}(x) = x$.

$$c = \text{Hash}(\Omega_1 || \Omega_2) = 116 || 192 = 116192$$

$$D = w + cx = 5247 + 116192 * 13 = 1515743$$

$$D_1 = \eta_1 + cr_1 = 96487 + 116192 * (-101) = -11638905$$

$$D_2 = \eta_2 + cr_2 = 274978 + 116192 * 1115 = 129829058$$

Por lo que el algoritmo devuelve $\text{proof}_{\text{ss}} = (c, D, D_1, D_2) = (116192, 1515743, -11638905, 129829058)$.

Veamos ahora el algoritmo de [Prueba de que dos compromisos esconden el mismo secreto](#), que realiza el verificador. Este algoritmo tiene de entrada:

$$(E, F, c, D, D_1, D_2) = (61, 111, 116192, 1515743, -11638905, 129829058)$$

Y aceptará cuando:

$$c == \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

Es decir, cuando ocurra:

$$\begin{aligned} & 116192 = \\ & = \text{Hash}(7^{1515743} 21^{-11638905} 61^{-116192}(\text{mod } 221) || 14^{1515743} 28^{129829058} 111^{-116192}(\text{mod } 221)) = \end{aligned}$$

Comenzamos calculando los inversos:

$$\begin{aligned} & 21^{-1}(\text{mod } 221) = 200(\text{mod } 221) \Rightarrow \\ & \Rightarrow 21^{-11638905}(\text{mod } 221) = 200^{11638905}(\text{mod } 221) = 200 \end{aligned}$$

$$\begin{aligned} & 61^{-1}(\text{mod } 221) = 29(\text{mod } 221) \Rightarrow \\ & \Rightarrow 61^{-116192}(\text{mod } 221) = 29^{116192}(\text{mod } 221) = 35 \end{aligned}$$

$$\begin{aligned} & 111^{-1}(\text{mod } 221) 2(\text{mod } 221) \Rightarrow \\ & \Rightarrow 111^{-116192}(\text{mod } 221) = 2^{116192}(\text{mod } 221) = 35 \end{aligned}$$

Y por lo tanto la ecuación anterior queda:

$$\begin{aligned} 116192 &= \text{Hash}(158 * 200 * 35(\text{mod } 221) || 79 * 121 * 35(\text{mod } 221)) = \\ &= \text{Hash}(116 || 192) = 116192 \end{aligned}$$

Por lo que, como esperábamos, acepta.

Nótese que, como queríamos, el verificador no tiene acceso en ningún momento al secreto x .

Prueba de que un número comprometido es un cuadrado

Denotamos la prueba de conocimiento cero de que un secreto es un cuadrado por $PK_S = \{x, r_1 : E = g^{x^2} h^r\}$. Tenemos que $params_S = (t, \ell, s)$ representa los parámetros para el esquema PK_S , por lo que la solidez está dada por 2^{t-1} y la propiedad de conocimiento cero está garantizada si $1/\ell$ es despreciable, como antes. Los siguientes algoritmos corresponden a $Prove_S$ y $Verify_S$, respectivamente. Además, el logaritmo discreto de g con respecto a h , o su inversa, debe ser desconocido, de lo contrario el compromiso no es seguro.

Supongamos que Alice tiene en secreto $x \in [0, b]$. Sea $E = g^{x^2} h^r$ un compromiso con el cuadrado de x . Quiere demostrarle a Bob que conoce x y r_1 tal que $E = g^{x^2} h^r$, es decir, que E oculta el cuadrado x^2 . Entonces:

Algorithm 3: Prueba de Cuadrado: $Prove_S$

Input: $x, n, E, r_1, g, h, b, params_S$

Output: $proof_S$

$r_2 \in_R [-2^s n + 1, 2^s n - 1]$

$F = g^x h^{r_2} \pmod{n}$

$r_3 = r_1 - r_2 x$

$proof_{ss} = Prove_{ss}(x, r_2, r_3, E, F)$

return $proof_S = (E, F, proof_{ss})$

Nótese que $E = F^x h^{r_3} \pmod{n}$

Y Bob lo comprueba:

Algorithm 4: Prueba de Cuadrado: $Verify_S$

Input: $n, g, h, proof_S$

Output: True o False

return $Verify_{ss}(E, F, n, F, g, h, h, proof_{ss})$

Ejemplo

Tomamos los mismos valores que en el ejemplo anterior: $n = 221, x = 13, g = 7, h = 21, r = -101, t = 5, \ell = 3, s = 4$. En este caso tenemos:

$$E = g^{x^2} h^r \pmod{n} = 7^{13^2} 21^{-101} \pmod{221} = 7^{169} 200 \pmod{221} = 113$$

Con esto el algoritmo de [Prueba de que un número comprometido es un cuadrado](#) queda:

$$r_2 \in_R [-2^s n + 1, 2^s n - 1] = [-2^4 221 + 1, 2^4 221 - 1] = [-3535, 3535] \Rightarrow r_2 = 2483$$

$$F = g^x h^{r_2} \pmod{n} = 7^{13} 21^{2483} \pmod{221} = 61$$

$$r_3 = r_1 - r_2 x = -101 - 2483 * 13 = -32380$$

$$\text{proof}_{\text{ss}} = \text{Prove}_{\text{ss}}(x, r_3, r_2, E, F) = \text{Prove}_{\text{ss}}(13, -32380, 2483, 113, 61)$$

Utilizando el algoritmo de [Prueba de que dos compromisos esconden el mismo secreto](#), tenemos:

$$E = F^x h^{r_3} \pmod{n} \Rightarrow g_1 = F = 61, h_1 = h = 21, r_1 = r_3 = -32380$$

$$F = g^x h^{r_2} \pmod{n} \Rightarrow g_2 = g = 7, h_2 = h = 21, r_2 = r_2 = 2483$$

$$g_1 = 61, g_2 = 7, h_1 = 21, h_2 = 21$$

$$(x, r_1, r_2, E, F) = (13, -32380, 2483, 113, 61)$$

$$w \in_R [1, 2^{\ell+t} b - 1] = [1, 7679] \Rightarrow w = 3610$$

$$\eta_1 \in_R [1, 2^{t+\ell+s_1} n - 1] = [1, 2^{5+3+4} * 221 - 1] = [1, 905215] \Rightarrow \eta_1 = 857159$$

$$\eta_2 \in_R [1, 2^{t+\ell+s_2} n - 1] = [1, 2^{5+3+4} * 221 - 1] = [1, 905215] \Rightarrow \eta_2 = 617720$$

$$\Omega_1 = g_1^w h_1^{\eta_1} \pmod{n} = 61^{3610} 21^{857159} \pmod{221} = 162$$

$$\Omega_2 = g_2^w h_2^{\eta_2} \pmod{n} = 7^{3610} 21^{617720} \pmod{221} = 121$$

$$c = \text{Hash}(\Omega_1 || \Omega_2) = \text{Hash}(162 || 121) = 162121$$

$$D = w + cx = 3610 + 162121 * 13 = 2111183$$

$$D_1 = \eta_1 + cr_1 = 857159 + 162121 * -32380 = -5248620821$$

$$D_2 = \eta_2 + cr_2 = 617720 + 162121 * 2483 = 403164163$$

$$\text{proof}_{\text{ss}} = (c, D, D_1, D_2) = (162121, 2111183, -5248620821, 403164163)$$

Finalmente, la comprobación en el algoritmo de la [Prueba de que un número comprometido es un cuadrado](#) toma como entrada:

$$(E, F, \text{proof}_{\text{ss}}) = (E, F, (c, D, D_1, D_2)) =$$

$$= (113, 61, (162121, 2111183, -5248620821, 403164163))$$

Y opera llamando al algoritmo de la [Prueba de que dos compromisos esconden el mismo secreto](#) de la siguiente forma:

$$c = \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

En donde:

$$g_1^D h_1^{D_1} E^{-c}(\text{mod } n) = 61^{2111183} 21^{-5248620821} 113^{-162121}(\text{mod } 221) = 162$$

$$g_2^D h_2^{D_2} F^{-c}(\text{mod } n) = 7^{2111183} 21^{403164163} 45^{-162121}(\text{mod } 221) = 121$$

Y por tanto:

$$c = 162121 = \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

Por lo que devuelve verdadero.

Prueba de que un número comprometido pertenece a un intervalo (prueba CFT)

Denotamos la prueba de conocimiento cero de que un secreto pertenece a un intervalo mayor, usando la notación $\text{PK}_{\text{LI}} = x, r : E = g^x h^r \wedge x \in [-2^{t+\ell}b, 2^{t+\ell}b]$. Tenemos que $\text{params}_{\text{LI}} = (t, \ell, s)$ representa los parámetros para el esquema PK_{LI} , por lo que la completitud se logra con una probabilidad mayor que $1 - 2^\ell$; la solidez está dada por 2^{t-1} y la propiedad de conocimiento cero está garantizada si $1/\ell$ es insignificante. Los siguientes algoritmos corresponden a Prove_{LI} y $\text{Verify}_{\text{LI}}$, respectivamente. Además, el logaritmo discreto de g con respecto a h , o su inversa, debe ser desconocido.

Algorithm 5: Prueba de intervalo mayor: Prove_{LI}

Input: $x, n, g, h, r, b, \text{params}_{\text{LI}}$

Output: proof_{LI}

while $D_1 \notin [cb, 2^{t+\ell}b - 1]$ **do**
 $w \in_R [0, 2^{t+\ell}b - 1]$
 $\eta \in_R [-2^{t+\ell+s}n + 1, 2^{t+\ell+s}n - 1]$
 $\Omega = g^w h^\eta \pmod n$
 $C = \text{Hash}(\Omega)$
 $c = C \pmod{2^t}$
 $D_1 = w + xc$
 $D_2 = \eta + rc \in \mathbb{Z}$

end

return proof_{LI} = (C, D_1, D_2)

Bob comprueba que $x \in [-2^{t+\ell}b, 2^{t+\ell}b]$:

Algorithm 6: Prueba de intervalo mayor: Verify_{LI}

Input: $E, n, g, h, b, \text{proof}_{\text{LI}}$

Output: True o False

if $D_1 \in [cb, 2^{t+\ell}b - 1] \wedge C == \text{Hash}(g^{D_1} h^{D_2} E^{-c} \pmod n)$ **then**
 | **return** *True*
else
 | **return** *False*
end

Ejemplo

De nuevo, para el algoritmo (5) tomamos los valores de los ejemplos anteriores, es decir, $x = 13, n = 221, E = 45, t = 5, l = 3, s = 4, b = 30, g = 7, h = 21, r = -101$. Entonces tenemos:

$$w \in_R [0, 2^{t+\ell}b - 1] = [0, 2^{5+3}30 - 1] = [0, 7679] \Rightarrow w = 4621$$

$$\begin{aligned} \eta \in_R [-2^{t+\ell+s}n + 1, 2^{t+\ell+s}n - 1] &= [-2^{5+3+4}221 + 1, 2^{5+3+4}221 - 1] = \\ &= [-905215, 905215] = -96754 \end{aligned}$$

$$\Omega = g^w h^\eta \pmod n = 7^{4621} 21^{96754} \pmod{221} = 45$$

De nuevo, usamos como función *Hash* la función identidad:

$$C = \text{Hash}(\Omega) = \text{Hash}(45) = 45$$

$$c = C(\bmod 2^t) = 45(\bmod 2^5) = 45(\bmod 32) = 13$$

$$D_1 = w + xc = 4621 + 13 * 13 = 4790$$

$$D_2 = \eta + rc = -96754 + (-101) * 13 = -98067$$

Como $D_1 \in [cb, 2^{t+\ell}b - 1] \Rightarrow 4790 \in [13 * 30, 2^{5+3}30 - 1] = [390, 7679]$, hemos terminado y devolvemos: $\text{proof}_{\text{LI}} = (C, D_1, D_2) = (45, 4790, -98067)$

Para la demostración (6) tenemos como entrada $\text{proof}_{\text{LI}} = (C, D_1, D_2) = (45, 4790, -98067)$, y se cumplirá si:

$$D_1 \in [cb, 2^{t+\ell}b - 1] \wedge C == \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\bmod n))$$

es decir, si

$$D_1 \in [cb, 2^{t+\ell}b - 1] \Rightarrow 4790 \in [13 * 30, 2^{5+3}30 - 1] = [390, 7679]$$

y

$$\begin{aligned} C &== \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\bmod n)) \Rightarrow \\ &\Rightarrow 45 == \text{Hash}(7^{4790}21^{-98067}61^{-13}(\bmod 221)) = \text{Hash}(45) = 45 \end{aligned}$$

Square Decomposition

Antes de describir la construcción ZKRP de la descomposición cuadrada, primero necesitamos una demostración con tolerancia, denotada por $\text{PK}_{\text{WT}} = x, r : E = g^x h^r \wedge x \in [a - \theta, b + \theta]$, donde $\theta = 2^{t+\ell+1}\sqrt{b-a}$, como se muestra en los siguientes algoritmos.

Algorithm 7: Prueba con tolerancia: Prove_{WT}

Input: x, n, g, h, r, a, b, E

Output: proof_{WT}

$$E_a = E/g^a \pmod n$$

$$E_b = g^b/E \pmod n$$

$$x_a = x - a$$

$$x_b = b - x$$

$$x_{a_1} = \lfloor \sqrt{x - a} \rfloor$$

$$x_{a_2} = x_a - x_{a_1}^2$$

$$x_{b_1} = \lfloor \sqrt{b - x} \rfloor$$

$$x_{b_2} = x_b - x_{b_1}^2$$

while $r_{a_2} \notin [-2^s n + 1, 2^s n - 1]$ **do**

$r_{a_1} \in_R [-2^s n + 1, 2^s n - 1]$

$r_{a_2} = r - r_{a_1}$

end

Seleccionar r_{b_1} y r_{b_2} tales que $r_{b_1} + r_{b_2} = -r$.

$$E_{a_1} = g^{x_{a_1}^2} h^{r_{a_1}} \pmod n$$

$$E_{a_2} = g^{x_{a_2}} h^{r_{a_2}} \pmod n$$

$$E_{b_1} = g^{x_{b_1}^2} h^{r_{b_1}} \pmod n$$

$$E_{b_2} = g^{x_{b_2}} h^{r_{b_2}} \pmod n$$

$$\text{proof}_{S_a} = \text{Proves}(x_{a_1}, r_{a_1}, E_{a_1})$$

$$\text{proof}_{S_b} = \text{Proves}(x_{b_1}, r_{b_1}, E_{b_1})$$

$$\text{proof}_{LI_a} = \text{Prove}_{LI}(x_{a_2}, r_{a_2}, E_{a_2})$$

$$\text{proof}_{LI_b} = \text{Prove}_{LI}(x_{b_2}, r_{b_2}, E_{b_2})$$

return proof_{wt} = $(E_{a_1}, E_{a_2}, E_{b_1}, E_{b_2}, \text{proof}_{S_a}, \text{proof}_{S_b}, \text{proof}_{LI_a}, \text{proof}_{LI_b})$

Algorithm 8: Prueba con tolerancia: Verify_{WT}

Input: proof_{WT}

Output: True o False

if $E_{a_2} == E_a/E_{a_1} \pmod n \wedge E_{b_2} == E_b/E_{b_1} \pmod n$ **then**

$b_S = \text{Verify}_S(\text{proof}_{S_a}) \wedge \text{Verify}_S(\text{proof}_{S_b})$

$b_{LI} = \text{Verify}_{LI}(\text{proof}_{LI_a}) \wedge \text{Verify}_{LI}(\text{proof}_{LI_b})$

return $b_S \wedge b_{LI}$

end

return *False*

Sea x el número conocido por Alice y oculto por E . Bob está convencido de que $x - a$ es el valor oculto por E_a y $b - x$ es el valor oculto por E_b . Entonces, Bob

está convencido de que $x - a \geq -\theta$ y $b - x \geq -\theta$, es decir, que x pertenece a $[x - \theta, b + \theta]$, donde $\theta = 2^{t+\ell+1}\sqrt{b-a}$.

Ejemplo

- Tomamos los valores de los ejemplos anteriores: $x = 13, n = 221, g = 7, h = 21, a = 0, b = 30, t = 5, l = 3, s = 4$.

Seleccionamos r de forma aleatoria:

$$\begin{aligned} r \in_R [-2^s n - 1, 2^s n + 1] &\Rightarrow r \in_R [-2^4 221 - 1, 2^4 221 + 1] \Rightarrow \\ &\Rightarrow r \in [-3535, 3535] \Rightarrow r = 1027 \end{aligned}$$

Y tenemos:

$$E = g^x h^r (\bmod n) = 7^{13} 21^{1027} (\bmod 221) = 61$$

Entonces en el algoritmo (7) tenemos:

$$E_a = E/g^a (\bmod n) = 61/7^0 (\bmod 221) = 61$$

$$\begin{aligned} E_b = g^b/E (\bmod n) &= 7^{30}/61 (\bmod 221) = 7^{30} 61^{-1} (\bmod 221) = \\ &= 25 * 29 (\bmod 221) = 62 \end{aligned}$$

$$x_a = x - a = 13 - 0 = 13$$

$$x_b = b - x = 30 - 13 = 17$$

$$x_{a_1} = \lfloor \sqrt{x - a} \rfloor = \lfloor \sqrt{13 - 0} \rfloor = \lfloor 3,6056 \rfloor = 3$$

$$x_{a_2} = x_a - x_{a_1}^2 = 13 - 3^2 = 4$$

$$x_{b_1} = \lfloor \sqrt{b - x} \rfloor = \lfloor \sqrt{30 - 13} \rfloor = \lfloor 4,1231 \rfloor = 4$$

$$x_{b_2} = x_b - x_{b_1}^2 = 17 - 4^2 = 1$$

$$r_{a_1} \in_R [-2^s n + 1, 2^s n - 1] = [-2^4 221 + 1, 2^4 221 - 1] = [-3535, 3535] \Rightarrow r_{a_1} = 1824$$

$$r_{a_2} = r - r_{a_1} = 1027 - 1824 = -797$$

Como $r_{a_2} \in [-2^s n + 1, 2^s n - 1] = [-2^4 221 + 1, 2^4 221 - 1] = [-3535, 3535]$, continuamos, y seleccionamos $r_{b_1} = 539, r_{b_2} = -1566$, tal que $r_{b_1} + r_{b_2} = 539 - 1566 = -1027 = -r$

$$E_{a_1} = g^{x_{a_1}^2} h^{r_{a_1}} (\text{mod } n) = 7^{3^2} 21^{1824} (\text{mod } 221) = 112$$

$$E_{a_2} = g^{x_{a_2}} h^{r_{a_2}} (\text{mod } n) = 7^4 21^{-797} (\text{mod } 221) = 188$$

$$E_{b_1} = g^{x_{b_1}^2} h^{r_{b_1}} (\text{mod } n) = 7^{4^2} 21^{539} (\text{mod } 221) = 149$$

$$E_{b_2} = g^{x_{b_2}} h^{r_{b_2}} (\text{mod } n) = 7^1 21^{-1566} (\text{mod } 221) = 214$$

- A continuación llama al algoritmo de la [Prueba de que un número comprometido es un cuadrado](#) para proof_{s_a} :

$$\text{proof}_{s_a} = \text{Prove}_S(x_{a_1}, r_{a_1}, E_{a_1}) = \text{Prove}_S(3, 1824, 112)$$

Este algoritmo calcula:

$$g = 7, h = 21, x = x_{a_1} = 3, r_1 = r_{a_1} = 1824, E = E_{a_1} = 112$$

$$r_2 \in_R [-2^s n + 1, 2^s n - 1] = [-2^4 221 + 1, 2^4 221 - 1] = [-3535, 3535] \Rightarrow r_2 = -3218$$

$$F = g^x h^{r_2} (\text{mod } n) = 7^3 21^{-3218} (\text{mod } 221) = 99$$

$$r_3 = r_1 - r_2 x = 1824 - (-3218) * 3 = 11478$$

$$\text{proof}_{ss} = \text{Prove}_{ss}(x, r_2, r_3, E, F) = \text{Prove}_{ss}(3, -3218, 11478, 112, 99)$$

Este algoritmo llama a la [Prueba de que dos compromisos esconden el mismo secreto](#):

$$E = F^x h^{r_3} (\text{mod } n) \Rightarrow g_1 = F = 99, h_1 = h = 21, r_1 = r_3 = 11478$$

$$F = g^x h^{r_2} (\text{mod } n) \Rightarrow g_2 = g = 7, h_2 = h = 21, r_2 = r_2 = -3218$$

$$w \in_R [1, 2^{\ell+t} b - 1] = [1, 7679] \Rightarrow w = 5346$$

$$\begin{aligned} \eta_1 \in_R [1, 2^{l+t+s_1} n - 1] &= [1, 2^{5+3+4} * 221 - 1] = [1, 905215] \Rightarrow \\ &\Rightarrow \eta_1 = 330972 \end{aligned}$$

$$\begin{aligned} \eta_2 \in_R [1, 2^{l+t+s_2} n - 1] &= [1, 2^{5+3+4} * 221 - 1] = [1, 905215] \Rightarrow \\ &\Rightarrow \eta_2 = 452816 \end{aligned}$$

$$\Omega_1 = g_1^w h_1^{\eta_1} (\text{mod } n) = 99^{5346} 21^{330972} (\text{mod } 221) = 77$$

$$\Omega_2 = g_2^w h_2^{\eta_2} (\text{mod } n) = 7^{5346} 21^{452816} (\text{mod } 221) = 168$$

$$c = \text{Hash}(\Omega_1 || \Omega_2) = \text{Hash}(77 || 168) = 77168$$

$$D = w + cx = 5346 + 77168 * 3 = 236850$$

$$D_1 = \eta_1 + cr_1 = 330972 + 77168 * 11478 = 886065276$$

$$D_2 = \eta_2 + cr_2 = 452816 + 77168 * (-3218) = -247873808$$

$$\text{proof}_{ss} = (c, D, D_1, D_2) = (77168, 236850, 886065276, -247873808) = \text{proof}_{s_a}$$

- Y llama a [Prueba de que un número comprometido es un cuadrado](#) para proof_{s_b} :

$$\text{proof}_{s_b} = \text{Prove}_s(x_{b_1}, r_{b_1}, E_{b_1}) = \text{Prove}_s(4, 539, 149)$$

Este algoritmo calcula:

$$g = 7, h = 21, x = x_{b_1} = 4, r_1 = r_{b_1} = 539, E = E_{b_1} = 149$$

$$r_2 \in_R [-2^s n + 1, 2^s n - 1] = [-2^4 221 + 1, 2^4 221 - 1] = [-3535, 3535] \Rightarrow r_2 = 220$$

$$F = g^x h^{r_2} (\text{mod } n) = 7^4 21^{220} (\text{mod } 221) = 191$$

$$r_3 = r_1 - r_2 x = 539 - 220 * 4 = -341$$

$$\text{proof}_{ss} = \text{Prove}_{ss}(x, r_2, r_3, E, F) = \text{Prove}_{ss}(4, 220, -341, 149, 191)$$

Este algoritmo llama a la [Prueba de que dos compromisos esconden el mismo secreto](#):

$$E = F^x h^{r_3} (\text{mod } n) \Rightarrow g_1 = F = 191, h_1 = h = 21, r_1 = r_3 = -341$$

$$F = g^x h^{r_2} (\text{mod } n) \Rightarrow g_2 = g = 7, h_2 = h = 21, r_2 = r_2 = 220$$

$$w \in_R [1, 2^{\ell+t} b - 1] = [1, 7679] \Rightarrow w = 4018$$

$$\eta_1 \in_R [-2^{l+t+s_1} n + 1, 2^{s_1} n - 1] = [1, 2^{5+3+4} * 221 - 1] = [1, 905215] \Rightarrow$$

$$\Rightarrow \eta_1 = 415424$$

$$\eta_2 \in_R [-2^{l+t+s_2} n + 1, 2^{s_2} n - 1] = [1, 2^{5+3+4} * 221 - 1] = [1, 905215] \Rightarrow$$

$$\Rightarrow \eta_2 = 390798$$

$$\Omega_1 = g_1^w h_1^{\eta_1}(\text{mod } n) = 191^{4018} 21^{415424}(\text{mod } 221) = 152$$

$$\Omega_2 = g_2^w h_2^{\eta_2}(\text{mod } n) = 7^{4018} 21^{905215}(\text{mod } 221) = 87$$

$$c = \text{Hash}(\Omega_1 || \Omega_2) = \text{Hash}(152 || 87) = 15287$$

$$D = w + cx = 4018 + 15287 * 3 = 65166$$

$$D_1 = \eta_1 + cr_1 = 415424 + 15287 * (-341) = -4797443$$

$$D_2 = \eta_2 + cr_2 = 390798 + 15287 * 220 = 3753938$$

$$\text{proof}_{\text{ss}} = (c, D, D_1, D_2) = (15287, 65166, -4797443, 3753938) = \text{proof}_{\text{sb}}$$

- Llama al algoritmo de [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#) para $\text{proof}_{\text{LI}_a}$:

$$\text{proof}_{\text{LI}_a} = \text{Prove}_{\text{LI}}(x_{a_2}, r_{a_2}, E_{a_2}) = \text{Prove}_{\text{LI}}(4, -797, 188)$$

Y este algoritmo calcula:

$$w \in_R [0, 2^{t+\ell}b - 1] = [0, 2^{5+3}30 - 1] = [0, 7679] \Rightarrow w = 4051$$

$$\begin{aligned} \eta \in_R [-2^{t+\ell+s}n + 1, 2^{t+\ell+s}n - 1] &= [-2^{5+3+4}221 + 1, 2^{5+3+4}221 - 1] = \\ &= [-905215, 905215] = -378828 \end{aligned}$$

$$\Omega = g^w h^\eta(\text{mod } n) = 7^{4051} 21^{-378828}(\text{mod } 221) = 71$$

De nuevo, usamos como función *Hash* la función identidad:

$$C = \text{Hash}(\Omega) = \text{Hash}(71) = 71$$

$$c = C(\text{mod } 2^t) = 71(\text{mod } 2^5) = 71(\text{mod } 32) = 7$$

$$D_1 = w + xc = 4051 + 4 * 7 = 4079$$

$$D_2 = \eta + rc = -378828 + (-797) * 7 = -384407$$

Como $D_1 \in [cb, 2^{t+\ell}b - 1] \Rightarrow 4079 \in [7 * 30, 2^{5+3}30 - 1] = [210, 7679]$, hemos terminado y devolvemos: $\text{proof}_{\text{LI}} = (C, D_1, D_2, c) = (71, 4079, -384407, 7)$

- Y finalmente llama al algoritmo de la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#) para $\text{proof}_{\text{LI}_b}$:

$$\text{proof}_{\text{LI}_b} = \text{Prove}_{\text{LI}}(x_{b_2}, r_{b_2}, E_{b_2}) = \text{Prove}_{\text{LI}}(1, -1566, 214)$$

Y este algoritmo calcula:

$$w \in_R [0, 2^{t+\ell}b - 1] = [0, 2^{5+3}30 - 1] = [0, 7679] \Rightarrow w = 3213$$

$$\begin{aligned} \eta \in_R [-2^{t+\ell+s}n + 1, 2^{t+\ell+s}n - 1] &= [-2^{5+3+4}221 + 1, 2^{5+3+4}221 - 1] = \\ &= [-905215, 905215] = -244070 \end{aligned}$$

$$\Omega = g^w h^\eta \pmod{n} = 7^{3213} 21^{-244070} \pmod{221} = 96$$

De nuevo, usamos como función *Hash* la función identidad:

$$C = \text{Hash}(\Omega) = \text{Hash}(96) = 96$$

$$c = C \pmod{2^t} = 96 \pmod{2^5} = 96 \pmod{32} = 0$$

$$D_1 = w + xc = 3213 + 1 * 0 = 3213$$

$$D_2 = \eta + rc = -244070 + (-1566) * 0 = -244070$$

Como $D_1 \in [cb, 2^{t+\ell}b - 1] \Rightarrow 3213 \in [0 * 30, 2^{5+3}30 - 1] = [0, 7679]$, hemos terminado y devolvemos: $\text{proof}_{\text{LI}} = (C, D_1, D_2, c) = (96, 3213, -244070, 0)$

- Para la demostración de la [Square Decomposition](#) tenemos como entrada:

$$\begin{aligned} \text{proof}_{\text{wt}} &= (E_{a_1}, E_{a_2}, E_{b_1}, E_{b_2}, \text{proof}_{\text{S}_a}, \text{proof}_{\text{S}_b}, \text{proof}_{\text{LI}_a}, \text{proof}_{\text{LI}_b}) = \\ &= (112, 188, 149, 214, (77168, 236850, 886065276, -247873808), \\ &(15287, 65166, -4797443, 3753938), (71, 4079, -384407, 7), (96, 3213, -244070, 0)) \end{aligned}$$

Comienza calculando:

$$E_{a_2} == E_a / E_{a_1} \pmod{n} \Rightarrow 188 == 61 / 112 \pmod{221} = 188$$

$$E_{b_2} == E_b / E_{b_1} \pmod{n} \Rightarrow 214 == 62 / 149 \pmod{221} = 214$$

Y llama a los algoritmos de la [Prueba de que un número comprometido es un cuadrado](#) y de la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#):

- La comprobación de $\text{Verify}_S(\text{proof}_{s_a})$ utiliza el algoritmo de la [Prueba de que un número comprometido es un cuadrado](#) que toma como entrada:

$$\begin{aligned} (E, F, \text{proof}_{ss}) &= (E, F, (c, D, D_1, D_2)) = \\ &= (112, 99, (77168, 236850, 886065276, -247873808)) \end{aligned}$$

Y opera llamando al algoritmo de la [Prueba de que dos compromisos esconden el mismo secreto](#) de la siguiente forma:

$$c = \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

En donde:

$$g_1^D h_1^{D_1} E^{-c}(\text{mod } n) = 99^{236850} 21^{886065276} 79^{-77168}(\text{mod } 221) = 77$$

$$g_2^D h_2^{D_2} F^{-c}(\text{mod } n) = 7^{236850} 21^{-247873808} 99^{-77168}(\text{mod } 221) = 168$$

Y por tanto:

$$c = 77168 = \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

Por lo que devuelve verdadero.

- La comprobación de $\text{Verify}_S(\text{proof}_{s_b})$ utiliza el algoritmo de la [Prueba de que un número comprometido es un cuadrado](#) que toma como entrada:

$$\begin{aligned} (E, F, \text{proof}_{ss}) &= (E, F, (c, D, D_1, D_2)) = \\ &= (149, 191, (15287, 65166, -4797443, 3753938)) \end{aligned}$$

Y opera llamando al algoritmo de la [Prueba de que dos compromisos esconden el mismo secreto](#) de la siguiente forma:

$$c = \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

En donde:

$$g_1^D h_1^{D_1} E^{-c}(\text{mod } n) = 191^{65166} 21^{-4797443} 72^{-15287}(\text{mod } 221) = 152$$

$$g_2^D h_2^{D_2} F^{-c}(\text{mod } n) = 7^{65166} 21^{3753938} 33^{-15287}(\text{mod } 221) = 87$$

Y por tanto:

$$c = 15287 = \text{Hash}(g_1^D h_1^{D_1} E^{-c}(\text{mod } n) || g_2^D h_2^{D_2} F^{-c}(\text{mod } n))$$

Por lo que devuelve verdadero.

- Para la demostración de $\text{Verify}_{\text{LI}}(\text{proof}_{\text{LI}_a})$ utiliza el algoritmo de la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#) tenemos como entrada $\text{proof}_{\text{LI}} = (C, D_1, D_2, c) = (71, 4079, -384407, 7)$, y se cumplirá si:

$$D_1 \in [cb, 2^{t+\ell}b - 1] \wedge C == \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\text{mod } n))$$

es decir, si

$$D_1 \in [cb, 2^{t+\ell}b - 1] \Rightarrow 4079 \in [7 * 30, 2^{5+3}30 - 1] = [210, 7679]$$

y

$$\begin{aligned} C &== \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\text{mod } n)) \Rightarrow \\ &\Rightarrow 71 == \text{Hash}(7^{4079}21^{-384407}188^{-7}(\text{mod } 221)) = \text{Hash}(71) = 71 \end{aligned}$$

- Finalmente para la demostración de $\text{Verify}_{\text{LI}}(\text{proof}_{\text{LI}_b})$ utiliza el algoritmo de la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#) tenemos como entrada $\text{proof}_{\text{LI}} = (C, D_1, D_2, c) = (96, 3213, -244070, 0)$, y se cumplirá si:

$$D_1 \in [cb, 2^{t+\ell}b - 1] \wedge C == \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\text{mod } n))$$

es decir, si

$$D_1 \in [cb, 2^{t+\ell}b - 1] \Rightarrow 3213 \in [0 * 30, 2^{5+3}30 - 1] = [0]$$

y

$$\begin{aligned} C &== \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\text{mod } n)) \Rightarrow \\ &\Rightarrow 75 == \text{Hash}(7^{3213}21^{-244070}214^{-0}(\text{mod } 221)) = \text{Hash}(96) = 96 \end{aligned}$$

- Como todos estos algoritmos devuelven verdadero, el algoritmo [Square Decomposition](#) devuelve verdadero.

Prueba sin tolerancia

El protocolo anterior le permite a Alice demostrarle a Bob que el número comprometido x pertenece al intervalo deseado $[a, b]$. Para lograr una prueba de pertenencia sin tolerancia, aumentamos artificialmente el tamaño de x estableciendo $x' = 2^T x$, donde $T = 2(t + \ell + 1) + |b - a|$.

Por lo que, los siguientes algoritmos describen el esquema ZKRP de la descomposición cuadrada:

Algorithm 9: Prueba de rango de descomposición cuadrada: Prove_{SD}

Input: x, r, R

Output: proof_{SD}

$$x' = 2^T x$$

$$r' = 2^T r$$

$$T = 2(t + \ell + 1) + |b - a|$$

$$E' = E^{2^T}$$

$$\text{proof}_{\text{WT}} = \text{Prove}_{\text{WT}}(x', r', E')$$

$$\text{return } \text{proof}_{\text{SD}} = (E', \text{proof}_{\text{WT}})$$

Algorithm 10: Prueba de rango de descomposición cuadrada:

$\text{Verify}_{\text{SD}}$

Input: proof_{SD}

Output: True o False

if $E' == E^{2^T}$ **then**

return $\text{Verify}_{\text{WT}}(\text{proof}_{\text{WT}})$

end

return *False*

3.3.2. Basado en firma (*Signature-based*)

La idea del protocolo es que el verificador inicialmente calcula las firmas digitales para cada elemento en el conjunto objetivo S . El probador luego oculta esta firma digital elevándola a un exponente elegido al azar $v \in \mathbb{Z}_p$, de modo que es computacionalmente inviable determinar que se firmó el elemento. El probador usa el emparejamiento para calcular la prueba, y la bilinealidad del emparejamiento permite al verificador verificar que, de hecho, uno de los elementos de S fue elegido inicialmente. Los algoritmos siguientes muestran los detalles de este protocolo. El esquema depende de las firmas digitales Boneh-Boyen, resumidas a continuación:

Firmas Boneh-Boyen: En resumen, la clave privada del firmante viene dada por $x \in_R \mathbb{Z}_p$ y la clave pública es $y = g^x$. Dado el mensaje m , tenemos que la firma digital se calcula como $\sigma = g^{1/(x+m)}$, y la verificación se logra calculando $e(\sigma, yg^m) == e(g, g)$.

Algorithm 11: Establecer membresía: $\text{Setup}_{\text{ZKSM}}$

Input: g, h y un conjunto S

Output: $y \in \mathbb{G}$ y $A \in G^{|S|}$

$x \in_R \mathbb{Z}_p$

$y = g^x$

for $i \in S$ **do**

$A_i = g^{\frac{1}{x+i}}$

end

return $y, [A_i]$

Algorithm 12: Establecer membresía: $\text{Prove}_{\text{ZKSM}}$

Input: g, h , un compromiso C y un conjunto S

Output: δ, γ tal que $C = g^\delta h^\gamma$ y $\delta \in S$

$\tau \in_R \mathbb{Z}_p$

$V = A_\delta^T$

$s, t, m \in_R \mathbb{Z}_p$

$a = e(V, g)^{-s} \cdot e(g, g)^t$

$D = g^s h^m$

$c = \text{Hash}(V, a, D)$

$z_\delta = s - \delta c$

$z_\tau = t - \tau c$

$z_\gamma = m - \gamma c$

return $\text{proof}_{\text{ZKSM}} = (V, a, D, z_\delta, z_\tau, z_\gamma)$

Algorithm 13: Establecer membresía: $\text{Verify}_{\text{ZKSM}}$

Input: g, h , un compromiso C , $\text{proof}_{\text{ZKSM}}$

Output: True o False

return $D == C^C h^{z_\gamma} g^{z_\tau} \wedge a == e(V, y)^c \cdot e(V, g)^{-z_\delta} \cdot e(g, g)^{z_\tau}$

Prueba de rango: Para obtener ZKRP, podemos descomponer el secreto δ en base u , de la siguiente manera:

$$\delta = \sum_{0 \leq j \leq \ell} \delta_j u^j$$

Por tanto, si cada δ_j pertenece al intervalo $[0, u)$, entonces tenemos que $\delta \in [0, u^\ell)$. Los algoritmos ZKSM se pueden adaptar fácilmente para realizar este cálculo, como se muestra en los algoritmos siguientes.

Algorithm 14: Prueba de rango basada en firma: Setup_{ZKSM} para el intervalo $[0, u^\ell]$

Input: g, h, u, ℓ

Output: $y \in [0, u^\ell)$ y $A \in G^{|\mathbb{Z}_u|}$

$x \in_R \mathbb{Z}_p$

$y = g^x$

for $i \in \mathbb{Z}_u$ **do**

$A_i = g^{\frac{1}{x+i}}$

end

return $y, [A_i]$

Algorithm 15: Prueba de rango basada en firma: Prove_{ZKSM} para el intervalo $[0, u^\ell)$

Input: g, h, u, ℓ y un compromiso C

Output: δ, γ tal que $C = g^\delta h^\gamma$ y $\delta \in [0, u^\ell)$

Encuentra $[\delta_j]$ tal que $C = g^\delta h^\gamma$ y $\delta \in [0, u^\ell)$.

$\tau_j \in R\mathbb{Z}_p$

$D =$ elemento identidad en \mathbb{G}

for $j \in \mathbb{Z}_\ell$ **do**

$V_j = A_{\delta_j}^{\tau_j}$

$s_j, t_j, m_j \in_R \mathbb{Z}_p$

$a_j = e(V_j, g)^{-s_j} \cdot e(g, g)^{t_j}$

$D = D g^{u^j s_j} h^{m_j}$

end

$c = \text{Hash}([V_j], a, D)$

for $j \in \mathbb{Z}_\ell$ **do**

$z_{\delta_j} = s_j - \delta_j c$

$z_{\tau_j} = t_j \tau_j c$

end

$z_\gamma = m - \gamma c$ **return** proof_{ZKSM} = $(z_\gamma, [z_{\delta_j}], [z_{\tau_j}])$

Algorithm 16: Establecer membresía: $\text{Verify}_{\text{ZKSM}}$

Input: g, h , un compromiso C , $\text{proof}_{\text{ZKSM}}$

Output: True o False

for $j \in \mathbb{Z}_j$ **do**

$a = a \wedge (a_j == e(V_j, y)^c \cdot e(V_j, g)^{-z_{\delta_j}} \cdot e(g, g)^{z_{\tau_j}})$

end

return $D == C^C h^{z_\gamma} \prod_j (u^j z_{\delta_j}) \wedge a$

Para obtener Pruebas de Rango de Conocimiento Cero para rangos arbitrarios $[a, b)$ mostramos que $\delta \in [a, a + u^\ell)$ y $\delta \in [b - u^\ell, b)$, usando 2 veces el esquema ZKRP descrito en el Algoritmo $\text{Prove}_{\text{ZKSM}}$. Es decir, tenemos que demostrar que $\delta - b + u^\ell \in [0, u^\ell)$ y $\delta - a \in [0, u^\ell)$.

3.3.3. Bulletproofs

Notación

Sea \mathbb{G} un grupo cíclico de orden primo p , y sea \mathbb{Z}_p el anillo de enteros módulo p . Sean \mathbb{G}^n y \mathbb{Z}_p^n espacios de vectores de dimensión n sobre \mathbb{G} y \mathbb{Z}_p respectivamente. Denotamos por \mathbb{Z}_p^* a $\mathbb{Z}_p \setminus \{0\}$.

Utilizaremos negrita para denotar vectores, por ejemplo, $\mathbf{a} \in \mathbb{F}^n$ es un vector con elementos $a_1, a_2, \dots, a_n \in \mathbb{F}$.

Dado un array $\mathbf{a} \in \mathbb{G}^n$, usaremos la notación de Python para denotar porciones de matriz:

$$\mathbf{a}[:l] = [a_1, \dots, a_l] \in \mathbb{G}^l$$

$$\mathbf{a}[l:] = [a_{l+1}, \dots, a_n] \in \mathbb{G}^{n-l}$$

Dado un $k \in \mathbb{G}$, denotamos el vector que contiene las potencias de k por:

$$\mathbf{k}^n = [1, k, k^2, \dots, k^{n-1}]$$

Dado $\mathbf{g} = [g_1, \dots, g_n] \in \mathbb{G}^n$ y $\mathbf{a} \in \mathbb{Z}_p^n$, definimos $\mathbf{g}^{\mathbf{a}}$ como:

$$\mathbf{g}^{\mathbf{a}} = \prod_{i=1}^n g_i^{a_i}$$

Dado $c \in \mathbb{Z}_p$, denotamos $\mathbf{b} = c \cdot \mathbf{a}$ como el vector tal que $b_i = c \cdot a_i$.

Denotamos el producto Hadamard por $\mathbf{a} \circ \mathbf{b} = (a_1 b_1, \dots, a_n b_n)$; el vector polinomial $p(X) = \sum_{i=1}^n p_i^{X^i} \in \mathbb{Z}_p^n[X]$, donde cada coeficiente \mathbf{p}_i es un vector en \mathbb{Z}_p^n ; y el producto interno de dichos polinomios está definido como:

$$\langle \mathbf{l}(X), \mathbf{r}(X) \rangle = \sum_{i=0}^d \sum_{j=0}^i \langle \mathbf{l}_i, \mathbf{r}_j \rangle X^{i+j} \in \mathbb{Z}_p[X]$$

Configuración

Muchas construcciones de ZKRP dependen de una configuración confiable. En breve, los parámetros necesarios para generar y verificar las pruebas de conocimiento cero subyacentes deben ser calculados por una parte confiable, porque si tales parámetros se generan utilizando una trampa, entonces esta trampa podría usarse para subvertir el protocolo, permitiendo generar dinero de la nada.

Para evitar la configuración confiable, Bulletproofs usa la estrategia Nothing Up My Sleeve (NUMS), donde se utiliza una función hash para calcular los generadores que serán necesarios para los compromisos de Pedersen, y utilizando la curva elíptica *secp256k1*, que es la curva de la forma $y^2 \equiv x^3 + ax + b$ con $a = 0, b = 7$ y con

$$p = \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFC2F}$$

o, en decimal:

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$

cuya forma se puede ver en [Figura 3.4](#).

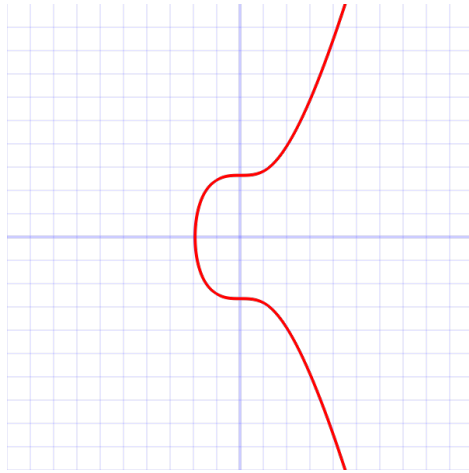


Figura 3.4: Gráfica de la curva *secp256k1*

Este algoritmo es el siguiente:

Algorithm 17: Nothing Up My Sleeve: MapToGroup

Input: La cadena de entrada m y el módulo primo de campo p .

Output: Un punto de curva elíptica si tiene éxito o algún error.

$i = 0$

while $i < 256$ **do**

$x = \text{Hash}(m||i)$

$rhs = x^3 + 7 \pmod{p}$

if rhs es un cuadrado \pmod{p} **then**

$y = \sqrt{rhs} \pmod{p}$

if (x, y) no es el punto en el infinito **then**

return (x, y)

end

end

$i = i + 1$

end

return “No se puede asignar al grupo”

Algorithm 18: Computa Generadores: ComputeGenerators

Input: La curva elíptica pública generadora $g \in \mathbb{G}$ y un entero n .

Output: El conjunto de generadores $(g, h, \mathbf{g}, \mathbf{h})$

Computa $h = \text{MapToGroup}(\text{'algún string público'}, p)$.

$i = 0$

while $i < n$ **do**

$c \in_R \mathbb{Z}_p$

$d \in_R \mathbb{Z}_p$

$\mathbf{g}[i] = c.G$

$\mathbf{h}[i] = d.G$

$i = i + 1$

end

return $(g, h, \mathbf{g}, \mathbf{h})$

Con esto, la configuración queda de la siguiente forma:

Algorithm 19: Setup_{IP}

Input: El conjunto de generadores $(g, h, \mathbf{g}, \mathbf{h})$.

Output: params_{IP}

$u = \text{MapToGroup}(\text{'algún string público'}, p)$.

return params_{IP} = $(g, h, \mathbf{g}, \mathbf{h}, u)$

Algorithm 20: Setup_{RP}

Input: La curva elíptica pública generadora $g \in \mathbb{G}$ y un entero n .

Output: params_{RP}

if b no es una potencia de 2 **then**

return “ b tiene que ser una potencia de 2”

else

$n = \log_2(b)$

$(g, h, \mathbf{g}, \mathbf{h}) = \text{ComputeGenerators}(g, n)$

 params_{IP} = SetUp_{IP} $(g, h, \mathbf{g}, \mathbf{h})$

 params_{RP} = (params_{IP}, n)

return params_{RP}

end

Argumento del producto interno

En esta sección presentamos el bloque de construcción principal de Bulletproofs, que es el argumento del producto interno. En resumen, utilizando este protocolo ZKP, el probador convence a un verificador de que conoce vectores cuyo producto interno es igual a un valor público determinado. Primero describimos el procedimiento de inicialización en el siguiente algoritmo:

Algorithm 21: Compromiso de vectores: Commit_{IP}

Input: (params_{IP}, \mathbf{a}, \mathbf{b})

Output: El compromiso P .

$P = g^{\mathbf{a}} h^{\mathbf{b}} \in \mathbb{G}$

return P

Algorithm 22: Prueba de Producto Interno: Prove_{IP}

Input: (params_{IP}, commit_{IP}, c , \mathbf{a} , \mathbf{b})

Output: proof_{IP}

$x = \text{Hash}(\mathbf{b}, \mathbf{h}, P, c) \in \mathbb{Z}_p^*$

$P' = u^{x \cdot c} P$

Asignar arrays $l, r \in \mathbb{G}^n$

ComputeProof($\mathbf{g}, \mathbf{h}, P', u^x, \mathbf{a}, \mathbf{b}, \mathbf{l}, \mathbf{r}$)

proof_{IP} = ($\mathbf{g}, \mathbf{h}, P', u^x, \mathbf{a}, \mathbf{b}, \mathbf{l}, \mathbf{r}$)

return proof_{IP}

Luego presentamos el protocolo principal, dado por el siguiente algoritmo:

Algorithm 23: Prueba de Producto Interno: ComputeProof

Input: ($\mathbf{g}, \mathbf{h}, P, u, a, b, \mathbf{l}, \mathbf{r}$)

Output: ($\mathbf{g}, \mathbf{h}, P, u, a, b, \mathbf{l}, \mathbf{r}$)

$x = \text{Hash}(\mathbf{g}, \mathbf{h}, P, c) \in \mathbb{Z}_p^*$

$P' = u^{x \cdot c} P$

if $n == 1$ **then**

return ($\mathbf{g}, \mathbf{h}, P, u, a, b, \mathbf{l}, \mathbf{r}$)

else

$n' = \frac{n}{2}$

$C_L = \langle \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]} \rangle \in \mathbb{Z}_p$

$C_R = \langle \mathbf{a}_{[n':]}, \mathbf{b}_{[:n']} \rangle \in \mathbb{Z}_p$

$L = \mathbf{g}_{[n':]}^{\mathbf{a}_{[:n']}} \mathbf{h}_{[n':]}^{\mathbf{b}_{[n':]}} u^{c_L} \in \mathbb{G}$

$R = \mathbf{g}_{[n':]}^{\mathbf{a}_{[n':]}} \mathbf{h}_{[n':]}^{\mathbf{b}_{[:n']}} u^{c_R} \in \mathbb{G}$

 Añadir L, R a \mathbf{l}, \mathbf{r} respectivamente.

$x = \text{Hash}(L, R)$

$\mathbf{g}' = \mathbf{g}_{[n':]}^{x^{-1}} \mathbf{g}_{[n':]}^x \in \mathbb{G}^{n'}$

$\mathbf{h}' = \mathbf{g}_{[n':]}^x \mathbf{g}_{[n':]}^{x^{-1}} \in \mathbb{G}^{n'}$

$P' = L^{x^2} P R^{x^{-2}} \in \mathbb{G}$

$\mathbf{a}' = \mathbf{a}_{[:n']} x + \mathbf{a}_{[n':]} x^{-1} \in \mathbb{Z}_p^{n'}$

$\mathbf{b}' = \mathbf{b}_{[n':]} x^{-1} + \mathbf{b}_{[:n']} x \in \mathbb{Z}_p^{n'}$

 Ejecutar recursivamente ComputeProof en ($\mathbf{g}', \mathbf{h}', P', u, \mathbf{a}', \mathbf{b}', \mathbf{l}, \mathbf{r}$)

end

Algorithm 24: Prueba de Producto Interno: $\text{Verify}_{\text{IP}}$

Input: $\text{params}_{\text{IP}}, \text{commit}_{\text{IP}}, \text{proof}_{\text{IP}}$

Output: True o False

$i = 0$

while $i < \log(n)$ **do**

$n' = \frac{n}{2}$

$x = \text{Hash}(\mathbf{l}[i], \mathbf{r}[i])$

$\mathbf{g}' = \mathbf{g}_{[:n']}^{x^{-1}} \mathbf{g}_{[n':]}^x \in \mathbb{G}^{n'}$

$\mathbf{h}' = \mathbf{g}_{[:n']}^x \mathbf{g}_{[n':]}^{x^{-1}} \in \mathbb{G}^{n'}$

$P' = L^{x^2} P R^{x^{-2}} \in \mathbb{G}$

$i = i + 1$

end

El verificador calcula $c = a.b$ y acepta si $P = g^a h^b u^c$

El hecho de que Bulletproofs permita reducir a la mitad el tamaño del problema en cada nivel de recursión en el algoritmo ComputeProof significa que es posible obtener un tamaño de prueba logarítmico.

Argumento de prueba de rango

Dado un valor secreto v , si queremos probar que pertenece al intervalo $[0, 2^n)$, tenemos que hacer lo siguiente:

- Probar que $\mathbf{a}_L \in 0, 1^n$ es la descomposición en bits de v . En otras palabras, tenemos que probar que:

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v$$

- Definir \mathbf{a}_R como el complemento por componentes de \mathbf{a}_L , lo que significa que, por cada $i \in [0, n]$, si el i -ésimo bit de \mathbf{a}_L es 0, entonces el i -ésimo bit de \mathbf{a}_R es 1; y si el i -ésimo bit de \mathbf{a}_L es 1, entonces el i -ésimo bit de \mathbf{a}_R es 0.

Equivalentemente, esta condición se puede expresar como:

$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}^n$$

$$\mathbf{a}_R = \mathbf{a}_L - 1^n \pmod{2}$$

Para probar que \mathbf{a}_L y \mathbf{a}_R satisfacen ambas relaciones podemos seleccionar de forma aleatoria un elemento $y \in \mathbb{Z}_p$ y computar:

$$\langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle = 0$$

$$\langle \mathbf{a}_L - \mathbf{1}^n - \mathbf{a}_R, \mathbf{y}^n \rangle = 0$$

Estas dos ecuaciones pueden ser combinadas en un único producto interno seleccionando un elemento aleatorio $z \in \mathbb{Z}_p$, y computando:

$$\langle \mathbf{a}_L - z.\mathbf{1}^n, \mathbf{y}^n \circ (\mathbf{a}_R + z.\mathbf{1}^n) + z^2.\mathbf{2}^n \rangle = z^2v + \delta(y, z)$$

donde $\delta(y, z) = (z - z^2) \langle \mathbf{1}^n, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}^n, \mathbf{2}^n \rangle \in \mathbb{Z}_p$.

Si el probador pudiese enviar los vectores en la ecuación anterior, entonces el verificador será capaz de comprobar el producto interno él mismo. Sin embargo, este vector revela informacion sobre \mathbf{a}_L , por lo tanto revelando bits del valor secreto v . Para solucionar este problema, el proveedor aleatoriamente elige vectores \mathbf{s}_L y \mathbf{s}_R para esconder \mathbf{a}_L y \mathbf{a}_R respectivamente. Considerando los siguientes polinimios:

$$l[X] = \mathbf{a}_L - z.\mathbf{1}^n + \mathbf{s}_L.X \in \mathbb{Z}_{p'}^n$$

$$r[X] = \mathbf{y}^n \circ (\mathbf{a}_R + z.\mathbf{1}^n + \mathbf{s}_R.X) + z^2.\mathbf{2}^n \in \mathbb{Z}_{p'}^n$$

$$t[X] = \langle l[X], r[X] \rangle = t_0 + t_1.X + t_2.X^2$$

donde el producto interno anterior es computado como:

$$\langle \mathbf{l}(X), \mathbf{r}(X) \rangle = \sum_{i=0}^d \sum_{j=0}^i \langle \mathbf{l}_i, \mathbf{r}_j \rangle X^{i+j} \in \mathbb{Z}_p[X]$$

Tenga en cuenta que los términos constantes de $l[X]$ y $r[X]$ corresponden a los vectores en la ecuación previa:

$$\langle \mathbf{a}_L - z.\mathbf{1}^n, \mathbf{y}^n \circ (\mathbf{a}_R + z.\mathbf{1}^n) + z^2.\mathbf{2}^n \rangle = z^2v + \delta(y, z)$$

Por lo tanto, si el probador publica $l[X]$ y $r[X]$ para un $x \in \mathbb{Z}_p$ específico, entonces tenemos que los términos \mathbf{s}_L y \mathbf{s}_R aseguran que no se revele información sobre \mathbf{a}_L y \mathbf{a}_R . Explicitamente tenemos que:

$$t_1 = \langle \mathbf{a}_L - z.\mathbf{1}^n, \mathbf{y}^n.\mathbf{s}_R \rangle + \langle \mathbf{s}_L, \mathbf{y}^n.(\mathbf{a}_R + z.\mathbf{1}^n) \rangle$$

$$t_2 = \langle \mathbf{s}_L, \mathbf{y}^n.\mathbf{s}_R \rangle$$

Finalmente, tenemos que el algoritmo es el siguiente:

Algorithm 25: Bulletproofs: Prove_{RP}

Input: params_{RP}, v **Output:** proof_{RP}

$$\gamma \in_R \mathbb{Z}_p$$

$$V = g^v h^\gamma \in \mathbb{G}$$

$$\mathbf{a}_L \in \{0, 1\}^n \text{ tal que } \langle \mathbf{a}_L, \mathbf{2}^n \rangle = v$$

$$\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n \in F_p^n$$

$$\alpha \in_R \mathbb{Z}_p$$

$$A = h^\alpha \mathbf{g}^{\mathbf{a}_L} \mathbf{h}^{\mathbf{a}_R} \in \mathbb{G}$$

$$s_L, s_R \in_R \mathbb{Z}_p^n$$

$$\rho \in_R \mathbb{Z}_p$$

$$S = h^\rho \mathbf{g}^{s_L} \mathbf{h}^{s_R} \in \mathbb{G}$$

$$y = \text{Hash}(A, S) \in \mathbb{Z}_p^*$$

$$z = \text{Hash}(A, S, y) \in \mathbb{Z}_p^*$$

$$\tau_1, \tau_2 \in_R \mathbb{Z}_p$$

$$T_1 = g^{\tau_1} h^{\tau_1} \in \mathbb{G}$$

$$T_2 = g^{\tau_2} h^{\tau_2} \in \mathbb{G}$$

$$x = \text{Hash}(T_1, T_2) \in \mathbb{Z}_p^*$$

$$\mathbf{l} = l(X) = \mathbf{a}_L - z\mathbf{1}^n + s_L X \in \mathbb{Z}_p^n$$

$$\mathbf{r} = r(X) = \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}^n + s_R X) + z^2 \mathbf{2}^n \in \mathbb{Z}_p^n$$

$$\hat{t} = \langle \mathbf{l}, \mathbf{r} \rangle \in \mathbb{Z}_p$$

$$\tau_x = \tau_2 x^2 + \tau_1 x + z^2 \gamma \in \mathbb{Z}_p$$

$$\mu = \alpha + \rho x \in \mathbb{Z}_p$$

$$\text{commit}_{\text{IP}} = \text{Commit}_{\text{IP}}(\text{params}_{\text{IP}}, \mathbf{l}, \mathbf{r})$$

$$\text{proof}_{\text{IP}} = \text{Prove}_{\text{IP}}(\text{params}_{\text{IP}}, \text{commit}_{\text{IP}}, \hat{t}, \mathbf{l}, \mathbf{r})$$

$$\text{proof}_{\text{RP}} = (\tau_x, \mu, \hat{t}, V, A, S, T_1, T_2, \text{commit}_{\text{IP}}, \text{proof}_{\text{IP}})$$

return proof_{RP}

Algorithm 26: Bulletproofs: $\text{Verify}_{\text{RP}}$ **Input:** $\text{params}_{\text{RP}}, \text{proof}_{\text{RP}}$ **Output:** True o False.

$$y = \text{Hash}(A, S) \in \mathbb{Z}_p^*$$

$$z = \text{Hash}(A, S, y) \in \mathbb{Z}_p^*$$

$$x = \text{Hash}(T_1, T_2) \in \mathbb{Z}_p^*$$

$$h_i = h_i^{y^{-i+1}} \in \mathbb{G}, \forall i \in [1, n]$$

$$P_l = P.h^\mu$$

$$P_r = A.S^x.g^z.(h')^{z.y^n+z^2.2^n} \in \mathbb{G}$$

$$\text{output}_1 = (P_l == P_r)$$

$$\text{output}_2 = (g^{\hat{t}h^{\tau x}} == V^{z^2}.g^{\delta(y,z)}.T_1^x.T_2^{x^2})$$

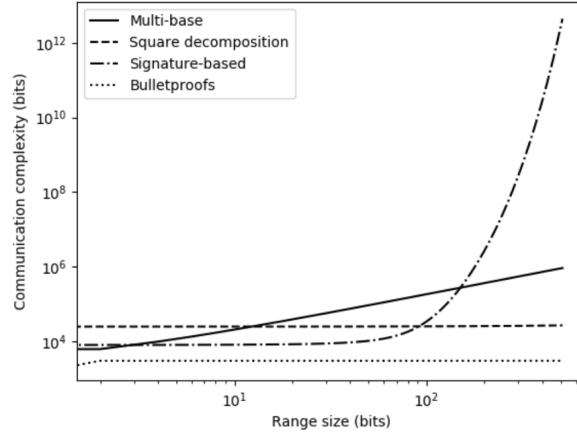
$$\text{output}_3 = \text{Verify}_{\text{IP}}(\text{proof}_{\text{IP}}$$

$$\text{return } \text{output}_1 \wedge \text{output}_2 \wedge \text{output}_3$$

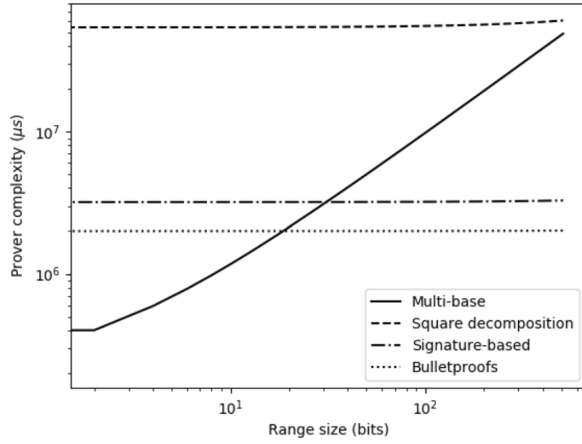
3.4. Selección de algoritmo

Para elegir el algoritmo en el que centraremos el estudio, comparemos los algoritmos *Multi-base*, *Descomposición Cuadrada* (*Square Decomposition*), *Basado en firma* (*Signature-based*) y *Bulletproofs* desarrollados previamente.

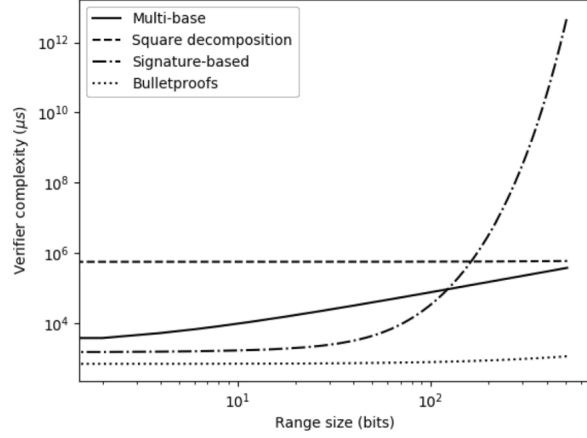
Lo primero que se considera a la hora de elegir el algoritmo que vamos a desarrollar es ver cuál de ellos es más eficiente. Para ello, se estudia cómo afecta el tamaño de las pruebas a la complejidad en bits, y cómo afecta el tamaño de las pruebas a la complejidad en tiempo para el probador y el verificador.



(a) Tamaño de las pruebas



(b) Complejidad del probador



(c) Complejidad del verificador

Figura 3.5: Comparación entre los distintos algoritmos [2]

Se puede ver que los dos algoritmos más interesantes son *Bulletproofs* y *Descomposición Cuadrada*, ya que al aumentar el tamaño de la prueba la complejidad en bits y la complejidad en tiempo del probador y el verificador se mantienen casi constante. Esto permite descartar los algoritmos *Signature-based* y *Multi-base*.

También es interesante comparar ahora *Bulletproofs* con los algoritmos *ZK-STARKS* y *ZK-SNARKS* estudiados brevemente en [Clases de protocolos de conocimiento cero](#), donde tenemos:

	Proof size	Prover time	Verification time
SNARKs (has trusted setup)	288 bytes	2.3s	10ms
SNARKs	45KB-200KB	1.6s	16ms
Bulletproofs	~1.3KB	30s	1100ms

Figura 3.6: Comparación entre Bulletproofs, ZK-STARKS y ZK-SNARKS [15]

Es decir, tenemos el siguiente esquema:

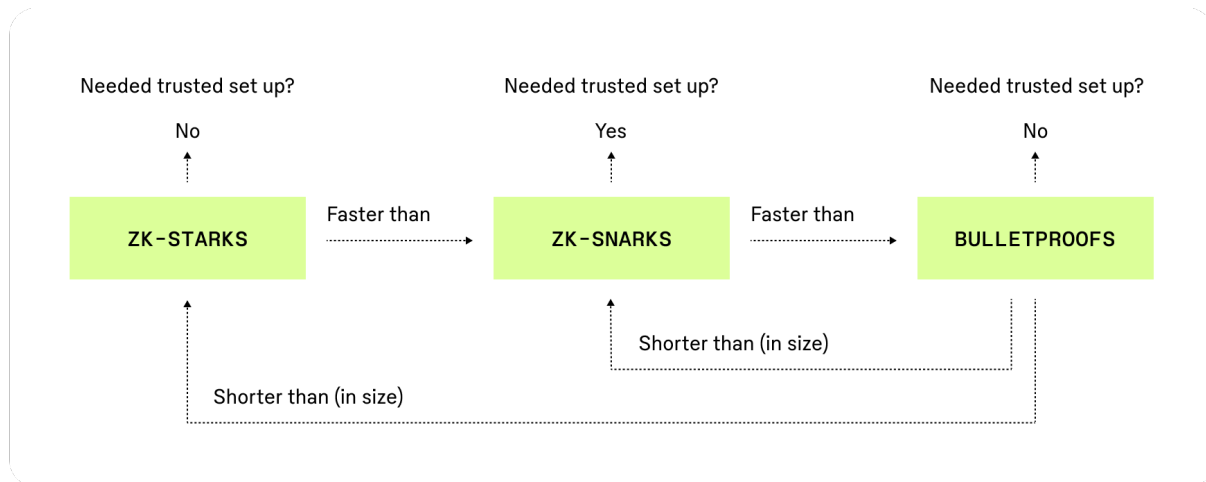


Figura 3.7: Comparación entre Bulletproofs, ZK-STARKS y ZK-SNARKS [15]

Y como se vio en la [Figura 3.5](#), el algoritmo *Descomposición Cuadrada* funciona similar a las *Bulletproofs*, pero más lento y con una complejidad de mayor tamaño.

Como los *Bulletproofs* no requieren establecimiento de confianza y son menores en tamaño, aunque sean algo más lentos, por lo que es uno de los algoritmos más interesantes hoy en día, y el principal que consideramos como objeto de este estudio. Sin embargo, debido a la complejidad de dicho algoritmo y que nuestro

objetivo es facilitar la comprensión de los protocolos de conocimiento cero, decidimos centrarnos en uno más simple, la *Descomposición Cuadrada*, ya que es más fácil comprender su funcionamiento utilizando la herramienta que desarrollaremos. Además, en septiembre del 2022 se publicó un artículo, *Sharp: Short Relaxed Range Proofs* por Geoffroy Couteau, Dahmun Goudarzi, Michael Klooß y Michael Reichle [3], que presenta un nuevo algoritmo, Sharp, basado en la *Descomposición cuadrada* y cuyas pruebas son casi un 50 % más cortas que las de *Bulletproof*, lo cual nos da incluso más motivos para seleccionar este algoritmo.

$(\lambda, \log B)$	N	Bulletproofs		Sharp _{SO} ^{Po}	
		Prover's work	Verifier's work	Prover's work	Verifier's work
128, 64	1	20.6	2.55	1.17	0.75
	8	157	12.1	7.47	3.88
128, 32	1	10.5	1.46	0.97	0.74
	8	80.0	6.93	6.74	3.39

Figura 3.8: Comparación en milisegundos entre SHARP y Bulletproofs [3]

Capítulo 4

Objetivos y Planificación

4.1. Objetivos

El objetivo general de este trabajo de fin de grado es implementar una herramienta que facilite el estudio de las pruebas de conocimiento cero. Este objetivo puede dividirse en los siguientes objetivos específicos:

1. Conocer el funcionamiento y algunas de las posibles aplicaciones de los protocolos de conocimiento cero.

Para ello, se estudia qué son estos protocolos, así como los distintos tipos que existen, y se desarrolla el funcionamiento de tres protocolos distintos: Descomposición cuadrada, Basados en firma y Bulletproof. El trabajo se centra en particular en el primero de ellos, viendo en detalle su implementación con distintos ejemplos.

2. Desarrollo de una herramienta que permita al usuario experimentar con los protocolos de conocimiento cero, siendo posible interactuar con el mismo para ver como distintos valores producen resultados distintos, y como los secretos que el probador quiere ocultar son inaccesibles para el verificador.
3. Verificar el funcionamiento correcto se espera de dichos algoritmos, realizando distintas pruebas con valores de distintas magnitudes.

Además, se intenta conseguir los siguientes objetivos:

1. La herramienta que permite al usuario experimentar con los protocolos de conocimiento cero pueda ser ejecutada en cualquier ordenador, sin importar

el sistema operativo ni las especificaciones de dicho ordenador. Para ello, es implementada en una aplicación web.

2. Esa misma herramienta debe ser de fácil comprensión para que permita entender el funcionamiento del algoritmo de una manera sencilla.
3. Que la verificación del algoritmo sea lo suficientemente extensa como para poder afirmar que proporciona los resultados esperados.

4.2. Planificación

Para representar gráficamente el esfuerzo dedicado a cada tarea del trabajo, se detalla el tiempo dedicado a cada una de ellas. Para ello, el siguiente esquema indica las fechas de comienzo y finalización de cada una de las partes:

- Planificación del trabajo. Este periodo incluye la búsqueda de artículos similares, que tratasen los protocolos de conocimiento cero aplicados a la docencia y, al no encontrar ninguno, decidir los objetivos de este trabajo.
- Investigación de ZKP. Durante estos meses se llevo a cabo la investigación de los protocolos de conocimiento cero y de los diferentes algoritmos posibles. Básicamente, este periodo fue empleado para el desarrollo del [Marco teórico](#).
- Selección del algoritmo. Una vez estudiados los protocolos en general, se dedicó un tiempo a compararlos entre ellos y cuál podría ser el más interesante para implementar en este trabajo.
- Implementación de Bulletproofs. En el capítulo [Selección de algoritmo](#) se comentó que los *Bulletproofs* son unos de los algoritmos más interesantes. Debido a ello, fueron los primeros seleccionados para desarrollar. Sin embargo, fue durante este desarrollo que se vio que dichos algoritmos requieren el uso de números extremadamente grandes (de más de 64 cifras) debido a la curva generadora. Por esto, consideró descartarlos ya que el objetivo de este trabajo es facilitar el entendimiento del algoritmo, y números tan grandes hacen que sea muy complicado de seguir.
- Implementación de Square Decomposition. Tras decidir que los *Bulletproofs* no son los más apropiados para este trabajo, se decidió tomar el segundo algoritmo más interesante, la *Descomposición cuadrada* o *Square Decomposition*. Durante este periodo se realizó toda la implementación del algoritmo en Python viendo que los resultados eran los esperados.

- Desarrollo de la herramienta docente. Una vez que se implemento el algoritmo, lo siguiente era crear la herramienta que facilitase su entendimiento. La mayor parte de este periodo fue dedicado a estudiar como implementarlo usando Flask.
- Documentación. Finalmente, el último mes aproximadamente ha sido dedicado al desarrollo de este documento.

Estos plazos pueden verse resumidos en el gráfico de la [Figura 4.1](#).

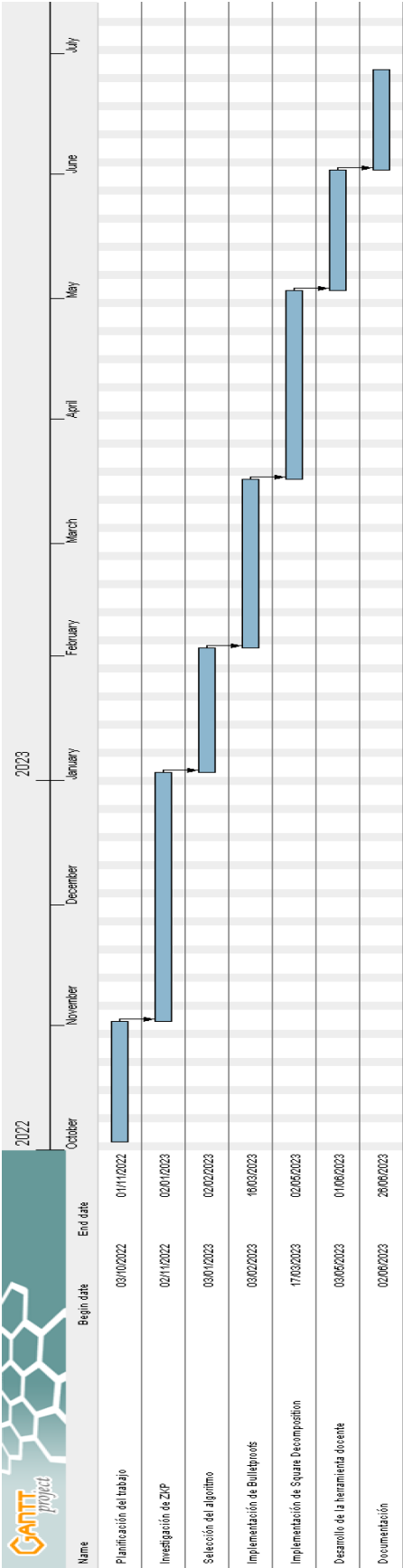


Figura 4.1: Tiempo del desarrollo de la propuesta

Capítulo 5

Desarrollo de la propuesta

Como se describió previamente en los objetivos, el desarrollo del presente trabajo puede dividirse en tres partes:

1. La implementación del algoritmo seleccionado, *Square Decomposition*.
2. La herramienta que permite visualizar y modificar los valores de la ejecución de dicho algoritmo.
3. Las pruebas funcionales del algoritmo.

Debido a ello, este capítulo estará dividida en tres partes, cada una correspondiente a un objetivo.

En cada parte se desarrolla el trabajo realizado para cumplir dicho objetivo, las tecnologías utilizadas para su implementación, se comentará en detalle algunas partes que pueden ser más interesantes y, en el caso de las pruebas, los resultados obtenidos.

Toda la implementación correspondiente a las tres partes puede encontrarse en GitHub en el siguiente enlace: <https://github.com/pedrors99/TFG>, así como este propio documento.

5.1. Implementación de *Square Decomposition*

En esta parte se estudia cómo queda la implementación del algoritmo en el que se decidió centrar este trabajo, *Square Decomposition*, así como el lenguaje utilizado

para su implementación, los distintos archivos y bibliotecas utilizadas y se entrará en detalle en los detalles más interesantes de la implementación.

5.1.1. Bibliotecas

El algoritmo ha sido implementado utilizando *Python 3.11*, además de las siguientes bibliotecas:

- *numpy*: Utilizada para almacenar los vectores con los distintos valores utilizados en el Algoritmo extendido de Euclides en el cálculo de inversos en un anillo.
- *random*: Utilizada para la generación aleatoria requerida por *Square Decomposition*. Es debido a esta aleatoriedad proporcionada por *random* que dos ejecuciones distintas nos darán resultados distintos.
- *math*: Utilizada para el cálculo de raíces cuadradas.
- *dataclasses*: Utilizada para crear objetos similares a los *struct* de otros lenguajes, que son classes con valores pero sin funciones. Estas estructuras servirán para representar los parámetros y las pruebas de los distintos algoritmos.

5.1.2. Implementación

Finalmente, se desarrolla cómo está implementado cada uno de los algoritmos. Como fue comentado previamente, cada fichero contiene la implementación tanto del algoritmo que genera la prueba como de su verificación, además de variantes de estas funciones que devuelven un diccionario con valores que normalmente son secretos para poder visualizarlos desde la herramienta. Debido a que el funcionamiento de estas variantes es básicamente el mismo, únicamente se entrará en detalles en uno de ellos para ver esta funcionalidad extra.

Para las pruebas y verificaciones se incluye un parámetro de entrada adicional, *debug*, que es una variable *booleana* por defecto en *False* que determina si imprimir los datos relacionados con el algoritmo, que incluye tanto la entrada, la salida, y algunos de los valores intermedios que utiliza, por pantalla.

Funciones adicionales

La mayoría de funciones implementadas son para las pruebas o las verificaciones de las mismas. Sin embargo, hay una serie de funciones adicionales implementadas en los ficheros *utils.py* y *mod.py*.

El primero, *utils.py*, contiene las dos siguientes funciones:

- *Hash*: Implementa la función *Hash* utilizada en el algoritmo. Por defecto, como se comentó previamente, será la identidad, es decir, una función tal que:

$$\text{Hash}(x) = x \quad \forall x$$

Esto se debe a que se prefiere utilizar una función más simple que permite seguir fácilmente el funcionamiento del algoritmo con mayor facilidad.

- *concat*: Concatena dos números que se le pasan como parámetros. La salida es el número resultante de dicha concatenación.

El segundo de ellos, *mod.py*, contiene una clase que representa un anillo módulo, formada por el valor del número y el valor del módulo. Contiene las funciones necesarias para poder realizar las operaciones básicas, como son la suma, resta, producto, potencias y comparación, así como el cálculo de inversos utilizando el algoritmo extendido de Euclides.

El constructor de esta clase es el siguiente:

```
1 class Mod:
2     def __init__(self, x, p):
3         self.x = x % p
4         self.p = p
```

Para la mayoría de funciones para las operaciones, como son similares, sólo se mostrará una de ellas, la suma, que servirá como ejemplo:

```
1     def __add__(self, y):
2         if isinstance(y, int):
3             return Mod(self.x + y, self.p)
4         if isinstance(y, Mod):
5             assert(self.p == y.p)
6             return Mod((self.x + y.x), self.p)
7         else:
8             self.assertFalse(True)
```

Estas funciones permiten tanto operar con otros objetos de esta misma clase, o con enteros.

Finalmente, se representan las dos funciones más complejas en detalle, que son las que se usan para el cálculo de inversos y para las potencias. La primera de ellas se basa en el algoritmo extendido de Euclides, y queda de la siguiente forma:

```

1      def inverse(self):
2          if self.x != 0:
3              q = np.empty(0)
4              r = np.array([self.p, self.x])
5              u = np.array([1, 0])
6              v = np.array([0, 1])
7
8              while True:
9                  q = np.append(q, int(r[len(r) - 2] / r[len(r) - 1]))
10                 r = np.append(r, int(r[len(r) - 2] % r[len(r) - 1]))
11                 if r[len(r) - 1] == 0:
12                     break
13                 u = np.append(u, int(u[len(u) - 2] - q[len(q) - 1] * u[len(u) - 1]))
14                 v = np.append(v, int(v[len(v) - 2] - q[len(q) - 1] * v[len(v) - 1]))
15
16             assert(r[len(r) - 2] == 1), "Error al calcular el inverso de {}".format(
self)
17             return Mod(v[len(v) - 1], self.p)
18         else:
19             return self

```

Para las potencias, tenemos tres casos:

1. El exponente es 0. Entonces simplemente se devuelve 1 en el módulo correspondiente.
2. El exponente es positivo. En este caso, se descompone la potencia como una serie de multiplicaciones, y se le aplica el módulo tras cada multiplicación. Esto, aunque no es muy eficiente en tiempo, evita que se alcancen valores demasiados altos que causen errores. Para optimizar el tiempo, usamos la siguiente propiedad: Si queremos calcular $a^b \pmod p$ y sabemos que $a^c \pmod p = 1$ con $c < b$, entonces podemos simplemente calcular $a^{b \% c} \pmod p$, donde $b \% c$ es el resto de dividir b entre c .
3. El exponente es negativo. Como $a^{-b} \pmod p$ con $b \in \mathbb{Z}^+$, es equivalente calcular $(a^{-1})^b \pmod p$, donde a^{-1} es el inverso de a en \mathbb{Z}_p , simplemente calculamos la potencia de su inverso con el exponente positivo.

Nótese que el cálculo de inversos sólo es posible si el valor x y el módulo p son primos relativos. En caso de que no lo sean, se detendrá la ejecución informando del error. Es por ello que se recomienda utilizar como valor para el módulo un número primo o al menos lo menos descomponible posible.

Todo esto queda implementado en la siguiente función:

```

1  def __pow__(self, n):
2      if self.x == 0 or self.x == 1:
3          return Mod(self.x, self.p)
4      else:
5          if n == 0:
6              return Mod(1, self.p)
7          elif n < 0:
8              self.x = self.inverse().x
9              n = -n
10
11         value = self.x
12
13         for i in range(1, n):
14             self.x = (self.x * value) % self.p
15             if self.x == 1:
16                 return Mod(value, self.p) ** (n % (i+1))
17
18         return Mod(self.x, self.p)
19         return self

```

Esquema

Para que sea más fácil entender que relación entre los distintos algoritmos, se incluyen unas imágenes que contienen los distintos algoritmos y cómo se llaman entre ellos.

Para la prueba de la Descomposición Cuadrada, el esquema es el siguiente:

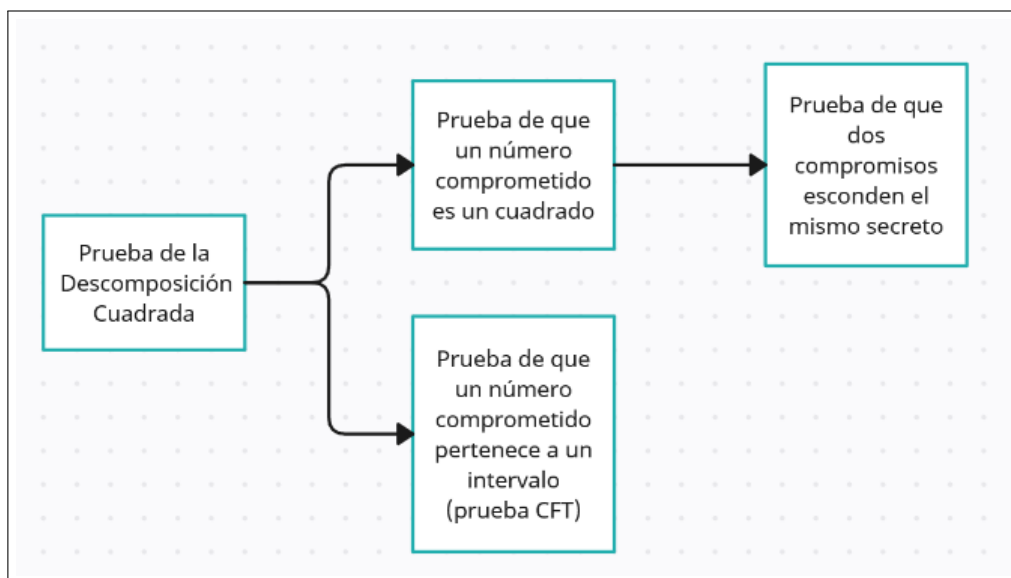


Figura 5.1: Esquema de la prueba

Y análogamente, el de la verificación es:

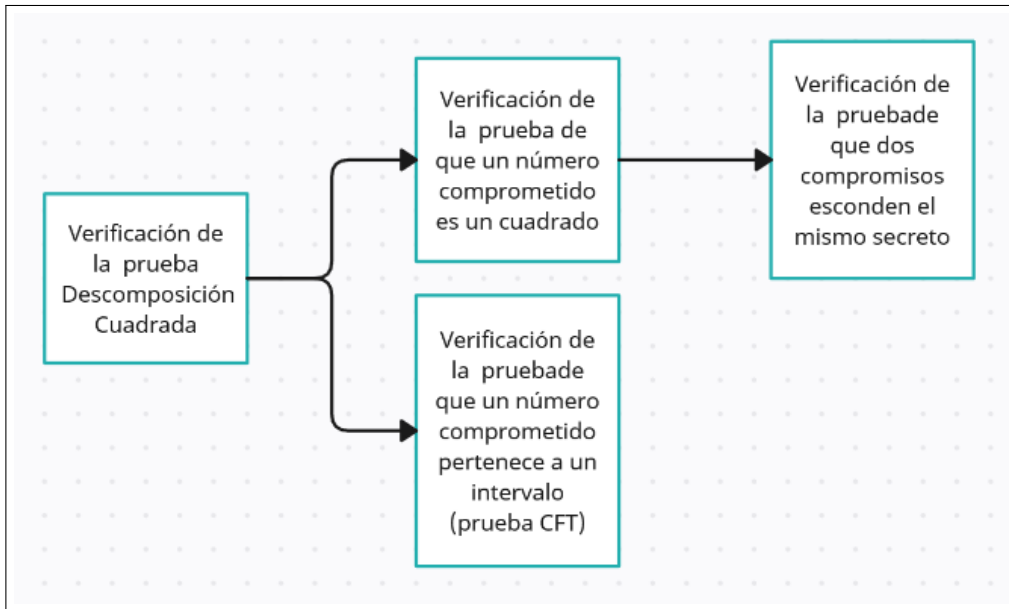


Figura 5.2: Esquema de la verificación

Prueba de que dos compromisos esconden el mismo secreto

Primero se presenta la implementación del algoritmo para la [Prueba de que dos compromisos esconden el mismo secreto](#), que queda implementada en el fichero *same_secret.py*. Utiliza dos estructuras, una para los parámetros y otra para la prueba, que son las siguientes:

```

1 @dataclass
2 class paramsSS:
3     t: int
4     l: int
5     s1: int
6     s2: int
7
8
9 @dataclass
10 class proofSS:
11     c: int
12     D: int
13     D1: int
14     D2: int
  
```

Se puede ver que todos los parámetros de ambas estructuras son enteros. Este algoritmo se diferencia de los demás ya que es el único que utiliza cuatro parámetros de seguridad: t , ℓ , $s1$ y $s2$; mientras que los demás utilizan t , ℓ y s . Esto se

debe a que cada uno de los parámetros s_1 y s_2 se utiliza para cada uno de los compromisos. Como se verá más adelante, dentro del algoritmo de *Square Decomposition* inicializamos ambos parámetros con el mismo valor, el de s , ya que ambos compromisos pertenecerán al mismo intervalo.

Recordemos que el algoritmo para dicha prueba era el siguiente:

Algorithm 27: Prueba del mismo secreto: Prove_{SS}

Input: $x, r_1, r_2, g_1, g_2, h_1, h_2, F, \text{params}_{\text{SS}}$

Output: proof_{SS}

$w \in_R [1, 2^{\ell+t}b - 1]$

$\eta_1 \in_R [1, 2^{\ell+t+s_1}n - 1]$

$\eta_2 \in_R [1, 2^{\ell+t+s_2}n - 1]$

$\Omega_1 = g_1^w h_1^{\eta_1} \pmod{n}$

$\Omega_2 = g_2^w h_2^{\eta_2} \pmod{n}$

$c = \text{Hash}(\Omega_1 || \Omega_2)$

$D = w + cx$

$D_1 = \eta_1 + cr_1$

$D_2 = \eta_2 + cr_2$

return $\text{proof}_{\text{SS}} = (c, D, D_1, D_2)$

La implementación es la siguiente:

```

1 def proveSS(x, n, r1, r2, g1, g2, h1, h2, b, params, debug=False):
2     w = random.randint(1, 2 ** (params.l + params.t) * b - 1)
3     eta1 = random.randint(1, 2 ** (params.l + params.t + params.s1) * n - 1)
4     eta2 = random.randint(1, 2 ** (params.l + params.t + params.s2) * n - 1)
5     omega1 = (Mod(g1, n) ** w * Mod(h1, n) ** eta1).x
6     omega2 = (Mod(g2, n) ** w * Mod(h2, n) ** eta2).x
7     c = utils.hash(utils.concat(omega1, omega2))
8     D = w + c * x
9     D1 = eta1 + c * r1
10    D2 = eta2 + c * r2
11
12    return proofSS(c, D, D1, D2)

```

Todas las variables aleatorias se generan utilizando la función *random.randint*, que genera un valor entero aleatorio a partir de los dos parámetros que se le pasen, que funcionan como límites para el intervalo al cuál pertenecerá el valor generado, con ambos extremos incluidos.

Como se comentó previamente, el algoritmo utiliza la clase *Mod* para las operaciones que requieren módulo, pasándole como argumentos el valor y el módulo. Además, en los casos que lo usa, para Ω_1 y Ω_2 se queda únicamente con x , que es el valor ignorando el módulo. Esto se debe a que las siguientes operaciones en las

que se utilizan no usaran el módulo.

Por último, se puede ver que la función *utils.concat* se utiliza para, como vimos en la explicación de los ficheros, concatenar dos enteros, es decir:

$$\text{utils.concat}(x, y) = x || y = xy$$

y la función *utils.hash* que, para facilitar el ejemplo, en nuestro caso funciona como la función identidad.

La verificación de la prueba tomará como entrada los parámetros E y F , que son los que la prueba quiere demostrar que esconden el mismo secreto, y junto con la prueba y algunos otros parámetros comprueba la veracidad de dicha prueba de la siguiente forma:

Algorithm 28: Prueba del mismo secreto: Verifyss

Input: $E, F, n, g_1, g_2, h_1, h_2, \text{proof}_{ss}$

Output: True o False

if $c == \text{Hash}(g_1^D h_1^{D_1} E^{-c} \pmod{n} || g_2^D h_2^{D_2} F^{-c} \pmod{n})$ **then**
 | **return** *True*

else

| **return** *False*

end

Esto queda implementada de la siguiente forma:

```
1 def verifySS(E, F, n, g1, g2, h1, h2, proof, debug=False):
2     val1 = Mod(g1, n) ** proof.D * Mod(h1, n) ** proof.D1 * Mod(E, n) ** (-proof.c)
3     val2 = Mod(g2, n) ** proof.D * Mod(h2, n) ** proof.D2 * Mod(F, n) ** (-proof.c)
4
5     return utils.hash(utils.concat(val1.x, val2.x)) == proof.c
```

La única diferencia es que dividimos el *if* en los dos cálculos para evitar tener una línea demasiado larga y difícil de comprender, y que en lugar de utilizar una estructura *if/else*, decidimos directamente devolver la igualdad, ya que el resultado es el mismo.

Prueba de que un número comprometido es un cuadrado

Similar al algoritmo de la prueba anterior, para la [Prueba de que un número comprometido es un cuadrado](#), que queda implementada en el fichero *square.py*, se utilizan estructuras para los parámetros y para la prueba, que son las siguientes:

```
1 @dataclass
2 class paramsS:
```

```

3     t: int
4     l: int
5     s: int
6
7
8 @dataclass
9 class proofS:
10     E: int
11     F: int
12     proof_ss: proveSS

```

Para este algoritmo, los parámetros de seguridad son diferentes, utilizando sólo tres variables, t, ℓ y s en lugar de los cuatro del algoritmo anterior, t, ℓ, s_1 y s_2 . En este caso, la prueba incluye un parámetro *proof_{ss}* que es la estructura utilizada para la prueba de algoritmo anterior.

El algoritmo de esta prueba era el siguiente:

Algorithm 29: Prueba de Cuadrado: Prove_S

Input: $x, n, E, r_1, g, h, b, \text{params}_S$

Output: proof_S

$r_2 \in_R [-2^s n + 1, 2^s n - 1]$

$F = g^x h^{r_2} \pmod{n}$

$r_3 = r_1 - r_2 x$

proof_{ss} = Prove_{SS}(x, r_2, r_3, E, F)

return proof_S = (E, F, proof_{ss})

que queda implementado:

```

1 def proveS(x, n, E, r1, g, h, b, params, debug=False):
2     r2 = random.randint(-2 ** params.s * n + 1, 2 ** params.s * n - 1)
3     F = (Mod(g, n) ** x * Mod(h, n) ** r2).x
4     r3 = r1 - r2 * x
5     params_ss = paramsSS(params.t, params.l, params.s, params.s)
6
7     proof = proveSS(x, n, r3, r2, F, g, h, h, b, params_ss, debug)
8     return proofS(E, F, proof)

```

donde *proveSS* es el algoritmo de la prueba anterior, la [Prueba de que dos compromisos esconden el mismo secreto](#).

La única diferencia es la inicialización de los parámetros de seguridad, ya que como hemos comentado, **proveSS** y **proveS** no utilizan los mismos.

Este algoritmo nos dará como salida los valores de la prueba, que se le pasará a la verificación junto con algunos otros parámetros, y comprobará si es válida con el siguiente algoritmo:

Algorithm 30: Prueba de Cuadrado: Verify_S

Input: n, g, h, proof_S

Output: True o False

return Verify_{SS}($E, F, n, F, g, h, h, \text{proof}_{ss}$)

y es implementado de la siguiente forma:

```

1 def verifyS(n, g, h, proof, debug=False):
2     return verifySS(proof.E, proof.F, n, proof.F, g, h, h, proof.proof_ss, debug)

```

Podemos ver que tanto la función de la prueba como de la verificación, al utilizar el algoritmo de la [Prueba de que dos compromisos esconden el mismo secreto](#), pasan el valor del parámetro *debug*, por lo que en caso de querer comprobar su funcionamiento interno o solucionar algún error, veremos los valores de ambos algoritmos imprimidos por pantalla.

Prueba de que un número comprometido pertenece a un intervalo (prueba CFT)

La [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#) está implementada en el fichero *interval.py*, utiliza nuevamente estructuras para los parámetros y la prueba, que son los siguientes:

```

1 @dataclass
2 class paramsLI:
3     t: int
4     l: int
5     s: int
6
7
8 @dataclass
9 class proofLI:
10    C: int
11    D1: int
12    D2: int
13    c: int

```

Recordemos que una prueba de pertenencia a un intervalo $[b_1, b_2]$ es una prueba de conocimiento que asegura al verificador que el probador conoce x que pertenece a $[B_1, B_2]$, un intervalo que contiene a $[b_1, b_2]$, y el índice de expansión de una prueba de pertenencia a un intervalo es la cantidad $\delta = (B_2 - B_1)/(b_2 - b_1)$, y que el algoritmo de dicha prueba, que demuestra que un número comprometido $x \in I$ pertenece a J , donde la tasa de expansión $\#J/\#I$ (donde $\#I$ representa el cardinal de I) es igual a $2^{t+\ell+1}$, era el siguiente:

Algorithm 31: Prueba de intervalo mayor: Prove_{LI}

Input: $x, n, g, h, r, b, \text{params}_{\text{LI}}$

Output: proof_{LI}

while $D_1 \notin [cb, 2^{t+\ell}b - 1]$ **do**
 $w \in_R [0, 2^{t+\ell}b - 1]$
 $\eta \in_R [-2^{t+\ell+s}n + 1, 2^{t+\ell+s}n - 1]$
 $\Omega = g^w h^\eta \pmod n$
 $C = \text{Hash}(\Omega)$
 $c = C \pmod{2^t}$
 $D_1 = w + xc$
 $D_2 = \eta + rc \in \mathbb{Z}$

end

return proof_{LI} = (C, D_1, D_2)

Este algoritmo queda implementado en la siguiente función:

```

1 def proveLI(x, n, g, h, r, b, params, debug=False):
2     loop = True
3
4     while loop:
5         w = random.randint(0, int(2 ** (params.t + params.l) * b))
6         eta = random.randint(-2 ** (params.t + params.l + params.s) * n + 1, 2 ** (
            params.t + params.l + params.s) * n - 1)
7
8         omega = (Mod(g, n) ** w * Mod(h, n) ** eta).x
9         C = utils.hash(omega)
10        c = C % (2 ** params.t)
11        D1 = w + x * c
12        D2 = eta + r * c
13
14        if c * b <= D1 <= 2 ** (params.t + params.l) * b - 1:
15            loop = False
16
17    return proofLI(C, D1, D2, c)

```

La principal diferencia entre esta implementación es que, como no tenemos inicializado el valor de D_1 , asumimos que no se cumple la condición $D_1 \in [cb, 2^{t+\ell}b - 1]$ para así entrar en el bucle, y hacemos la comprobación al final de cada iteración, una vez que ya se haya inicializado.

La verificación utiliza esta prueba para la verificación de la siguiente forma:

Algorithm 32: Prueba de intervalo mayor: $\text{Verify}_{\text{LI}}$

Input: $E, n, g, h, b, \text{proof}_{\text{LI}}$

Output: True o False

```

if  $D_1 \in [cb, 2^{t+\ell}b - 1] \wedge C == \text{Hash}(g^{D_1}h^{D_2}E^{-c}(\text{mod } n))$  then
  | return True
else
  | return False
end

```

que se implementa en la función:

```

1 def verifyLI(E, n, g, h, b, proof, params, debug=False):
2     cond1 = (proof.c * b <= proof.D1 <= 2 ** (params.t + params.l) * b - 1)
3     cond2 = (proof.C == utils.hash((Mod(g, n) ** proof.D1 * Mod(h, n) ** proof.D2 *
4         Mod(E, n) ** (-proof.c)).x))
5     return cond1 and cond2

```

Al igual que en la verificación de que dos compromisos esconden el mismo secreto, dividimos la condición en dos comprobaciones para mayor claridad del código.

Descomposición Cuadrada

Finalmente, se muestra la implementación del algoritmo que se estudia en este trabajo, [Square Decomposition](#), y que está implementado en *square_decomposition.py*. También se incluye el ejemplo de las funciones adicionales que comentamos al comienzo de la sección que utilizaremos en la herramienta para visualizar datos extra, tanto para las pruebas como para las verificaciones.

Al igual que las demás pruebas, utiliza estructuras para los parámetros y para las pruebas, que son las siguientes:

```

1 @dataclass
2 class paramsSD:
3     t: int
4     l: int
5     s: int
6
7
8 @dataclass
9 class proofSD:
10     Ea: int
11     Eb: int
12     Ea1: int
13     Ea2: int
14     Eb1: int
15     Eb2: int
16     proof_sa: proofS

```

```

17     proof_sb: proofS
18     proof_lia: proofLI
19     proof_lib: proofLI

```

El algoritmo final, que incluye las pruebas anteriores, era el siguiente:

Algorithm 33: Prueba con tolerancia: Prove_{WT}

Input: x, n, g, h, r, a, b, E

Output: proof_{WT}

$$E_a = E/g^a \pmod n$$

$$E_b = g^b/E \pmod n$$

$$x_a = x - a$$

$$x_b = b - x$$

$$x_{a_1} = \lfloor \sqrt{x - a} \rfloor$$

$$x_{a_2} = x_a - x_{a_1}^2$$

$$x_{b_1} = \lfloor \sqrt{b - x} \rfloor$$

$$x_{b_2} = x_b - x_{b_1}^2$$

while $r_{a_2} \notin [-2^s n + 1, 2^s n - 1]$ **do**

$r_{a_1} \in_R [-2^s n + 1, 2^s n - 1]$

$r_{a_2} = r - r_{a_1}$

end

Seleccionar r_{b_1} y r_{b_2} tales que $r_{b_1} + r_{b_2} = -r$.

$$E_{a_1} = g^{x_{a_1}^2} h^{r_{a_1}} \pmod n$$

$$E_{a_2} = g^{x_{a_2}} h^{r_{a_2}} \pmod n$$

$$E_{b_1} = g^{x_{b_1}^2} h^{r_{b_1}} \pmod n$$

$$E_{b_2} = g^{x_{b_2}} h^{r_{b_2}} \pmod n$$

$$\text{proof}_{S_a} = \text{Proves}(x_{a_1}, r_{a_1}, E_{a_1})$$

$$\text{proof}_{S_b} = \text{Proves}(x_{b_1}, r_{b_1}, E_{b_1})$$

$$\text{proof}_{LI_a} = \text{Prove}_{LI}(x_{a_2}, r_{a_2}, E_{a_2})$$

$$\text{proof}_{LI_b} = \text{Prove}_{LI}(x_{b_2}, r_{b_2}, E_{b_2})$$

return proof_{wt} = $(E_{a_1}, E_{a_2}, E_{b_1}, E_{b_2}, \text{proof}_{S_a}, \text{proof}_{S_b}, \text{proof}_{LI_a}, \text{proof}_{LI_b})$

queda implementado de la siguiente forma:

```

1  def proveSD(x, n, g, h, r, a, b, E, params, debug=False):
2      Ea = (Mod(E, n) * (Mod(g, n) ** a).inverse()).x
3      Eb = (Mod(g, n) ** b * Mod(E, n).inverse()).x
4
5      xa = x - a
6      xb = b - x
7
8      xa1 = math.floor(math.sqrt(x - a))
9      xa2 = xa - xa1 ** 2
10

```

```

11  xb1 = math.floor(math.sqrt(b - x))
12  xb2 = xb - xb1 ** 2
13
14  loop = True
15  while loop:
16      ra1 = random.randint(-2 ** params.s * n + 1, 2 ** params.s * n - 1)
17      ra2 = r - ra1
18
19      if -2 ** params.s * n + 1 < ra2 < 2 ** params.s * n - 1:
20          loop = False
21
22  rb1 = random.randint(-abs(r), abs(r))
23  rb2 = -r - rb1
24
25  Ea1 = (Mod(g, n) ** (xa1 ** 2) * Mod(h, n) ** ra1).x
26  Ea2 = (Mod(g, n) ** xa2 * Mod(h, n) ** ra2).x
27  Eb1 = (Mod(g, n) ** (xb1 ** 2) * Mod(h, n) ** rb1).x
28  Eb2 = (Mod(g, n) ** xb2 * Mod(h, n) ** rb2).x
29
30  proof_sa = proveS(xa1, n, Ea1, ra1, g, h, b, params)
31  proof_sb = proveS(xb1, n, Eb1, rb1, g, h, b, params)
32
33  proof_lia = proveLI(xa2, n, g, h, ra2, b, params)
34  proof_lib = proveLI(xb2, n, g, h, rb2, b, params)
35
36  return proofSD(Ea, Eb, Ea1, Ea2, Eb1, Eb2, proof_sa, proof_sb, proof_lia,
proof_lib)

```

La versión modificada que devuelve parámetros extra es:

```

1  def proveSD.Flask(x, n, g, h, r, a, b, E, params, debug=False):
2      Ea = (Mod(E, n) * (Mod(g, n) ** a).inverse()).x
3      Eb = (Mod(g, n) ** b * Mod(E, n).inverse()).x
4
5      xa = x - a
6      xb = b - x
7
8      xa1 = math.floor(math.sqrt(x - a))
9      xa2 = xa - xa1 ** 2
10
11     xb1 = math.floor(math.sqrt(b - x))
12     xb2 = xb - xb1 ** 2
13
14     loop = True
15     while loop:
16         ra1 = random.randint(-2 ** params.s * n + 1, 2 ** params.s * n - 1)
17         ra2 = r - ra1
18
19         if -2 ** params.s * n + 1 < ra2 < 2 ** params.s * n - 1:
20             loop = False
21
22     rb1 = random.randint(-abs(r), abs(r))
23     rb2 = -r - rb1
24

```

```

25 Ea1 = (Mod(g, n) ** (xa1 ** 2) * Mod(h, n) ** ra1).x
26 Ea2 = (Mod(g, n) ** xa2 * Mod(h, n) ** ra2).x
27 Eb1 = (Mod(g, n) ** (xb1 ** 2) * Mod(h, n) ** rb1).x
28 Eb2 = (Mod(g, n) ** xb2 * Mod(h, n) ** rb2).x
29
30 proof_sa, ext_sa, ext_ssa = proveS_Flask(xa1, n, Ea1, ra1, g, h, b, params)
31 proof_sb, ext_sb, ext_ssb = proveS_Flask(xb1, n, Eb1, rb1, g, h, b, params)
32
33 proof_lia, ext_lia = proveLI_Flask(xa2, n, g, h, ra2, b, params)
34 proof_lib, ext_lib = proveLI_Flask(xb2, n, g, h, rb2, b, params)
35
36 ext_wt = {'xa': xa, 'xb': xb, 'ra1': ra1, 'ra2': ra2, 'rb1': rb1, 'rb2': rb2, '
37          'xa1': xa1, 'xa2': xa2, 'xb1': xb1,
38          'xb2': xb2}
39
40 return proofSD(Ea, Eb, Ea1, Ea2, Eb1, Eb2, proof_sa, proof_sb, proof_lia,
41               proof_lib), ext_wt, ext_sa, ext_ssa, ext_sb, ext_ssb, ext_lia, ext_lib

```

Se puede ver que el funcionamiento es el mismo, pero utilizando las funciones modificadas del resto de pruebas, que nos devuelven diccionarios con los valores adicionales, además de un nuevo diccionario con los valores utilizados en esta prueba.

La verificación utiliza las verificaciones de las pruebas previas para comprobar si la prueba generada por este algoritmo es válida o no. Su algoritmo era:

Algorithm 34: Prueba con tolerancia: $\text{Verify}_{\text{WT}}$

Input: proof_{WT}

Output: True o False

if $E_{a_2} == E_a/E_{a_1}(\text{mod } n) \wedge E_{b_2} == E_b/E_{b_1}(\text{mod } n)$ **then**

$b_S = \text{Verifys}(\text{proofs}_a) \wedge \text{Verifys}(\text{proofs}_b)$
 $b_{LI} = \text{VerifyLI}(\text{proof}_{LI_a}) \wedge \text{VerifyLI}(\text{proof}_{LI_b})$
return $b_S \wedge b_{LI}$

end

return False

que se implementa de la siguiente forma:

```

1 def verifySD(n, g, h, b, proof, params, debug=False):
2     if proof.Ea2 == (Mod(proof.Ea, n) * Mod(proof.Ea1, n).inverse()).x and proof.
3         Eb2 == (Mod(proof.Eb, n) * Mod(proof.Eb1, n).inverse()).x:
4         bs = verifyS(n, g, h, proof.proof_sa, debug) and verifyS(n, g, h, proof.
5             proof_sb, debug)
6         bli = verifyLI(proof.Ea2, n, g, h, b, proof.proof_lia, params, debug) and
7             verifyLI(proof.Eb2, n, g, h, b, proof.proof_lib, params, debug)
8         return bs and bli
9     else:
10        return False

```

La versión modificada para mostrar valores adicionales en la herramienta es la

siguiente:

```

1 def verifySD_Flask(n, g, h, b, proof, params, debug=False):
2     cond1 = (proof.Ea2 == (Mod(proof.Ea, n) * Mod(proof.Ea1, n).inverse()).x)
3     cond2 = (proof.Eb2 == (Mod(proof.Eb, n) * Mod(proof.Eb1, n).inverse()).x)
4
5     if cond1 and cond2 and proof.Ea1 == proof.proof_sa.E and proof.Eb1 == proof.
        proof_sb.E:
6         cond3 = verifyS(n, g, h, proof.proof_sa, debug)
7         cond4 = verifyS(n, g, h, proof.proof_sb, debug)
8
9         cond5 = verifyLI(proof.Ea2, n, g, h, b, proof.proof_lia, params, debug)
10        cond6 = verifyLI(proof.Eb2, n, g, h, b, proof.proof_lib, params, debug)
11
12        bs = cond3 and cond4
13        bli = cond5 and cond6
14
15        ext = {'cond1': cond1, 'cond2': cond2, 'cond3': cond3, 'cond4': cond4, '
        cond5': cond5, 'cond6': cond6}
16
17        return (bs and bli), ext
18    else:
19        ext = {'cond1': cond1, 'cond2': cond2, 'cond3': '?', 'cond4': '?', 'cond5':
        '?', 'cond6': '?'}
20
21    return False, ext

```

Esta función divide la primera verificación en dos para poder mostrar los resultados de cada una en la herramienta. Además, nuevamente llama a las versiones modificadas de las demás verificaciones. En caso de que falle la primera comprobación, como las demás no se realizan, simplemente devuelve ? para indicar que el valor es desconocido.

5.2. Interfaz de la herramienta

Para facilitar el entendimiento del funcionamiento del algoritmo, se ofrece una interfaz web que permite visualizar los pasos que sigue el algoritmo, así como los valores y resultados de cada paso. Además, se podrán modificar los valores de entrada y la prueba obtenida para ver cómo modifican los resultados.

Para ello, se implementa una página web utilizando HTML para el Front-End, y la implementación desarrollada en la parte anterior como Back-End, utilizando *Flask* para comunicar el cliente con el servidor. Además, para mejorar el diseño de la página web, utilizaremos Bulma [27], que nos proporciona clases para los botones y otros elementos que directamente nos proporcionan una mejor apariencia.

Para toda esta implementación, se añade un nuevo fichero a Python:

- *flask_backend.py*: Funciona como servidor, leyendo los valores introducidos por el usuario en la página web, realizando los cálculos con la implementación previa, redirigiendo a las páginas correspondientes y mostrando los resultados. Consta de una función por cada una de las páginas web.

Además, se incluye una carpeta llamada *html* con los distintos archivos utilizados para cada una de las páginas, que son los siguientes:

- *input.html*: Es la primera página de la herramienta, en la que aparecen una serie de campos para que se puedan introducir los valores que utilizará el algoritmo. Estos valores están inicializados por defecto para poder realizar la prueba rápidamente, aunque pueden ser modificados a los que se deseen probar.

En la parte inferior encontraremos un botón para continuar, que realizará algunas comprobaciones, como que el intervalo tenga sentido (la cota inferior sea menor que la superior y ambos valores sean menores que el módulo) y que el secreto pertenece a dicho intervalo. En caso de que haya algún fallo, se avisará indicando como solucionarlo. En caso de que la entrada sea válida, continuará a la siguiente página, *prove_sd.html*.

- *prove_sd.html*: En esta página se hace un breve resumen del algoritmo *Square Decomposition*, se muestran los valores que toma por entrada, el funcionamiento del algoritmo, y los valores correspondientes a la prueba que se obtienen de salida.

Además, consta de una sección con botones para redirigir a las pruebas que utiliza el algoritmo internamente, para comprobar que un número comprometido es un cuadrado y para probar que un número comprometido pertenece a un intervalo.

Finalmente, en la parte inferior de la página tenemos un botón que nos redirigirá a la página anterior, *input.html*, y otro que nos redirigirá a la verificación de la prueba. Esta verificación tomará como entrada los valores que se encuentran en la salida del algoritmo, que pueden ser modificados en caso de que se quiera ver cómo la verificación cambia en caso de que la prueba no sea válida.

- *verify_sd.html*: Esta página contiene los resultados de la verificación, así como los valores que toma de entrada y el funcionamiento de esta verificación. Análogamente a la página anterior, como esta verificación utiliza la verificación de que un número comprometido es un cuadrado y de que un número

comprometido pertenece a un intervalo, hay una sección con botones para redirigirnos a las páginas correspondientes.

Finalmente, también hay un botón en la parte inferior que nos permite volver a la página anterior, *prove_sd.html*.

- *prove_s.html*: En esta página se muestra el funcionamiento de la prueba de que un número comprometido es un cuadrado, así como los valores que toma como entrada y los que optiene como salida. Análogamente a las páginas anteriores, como esta prueba utiliza la prueba de que dos compromisos esconden el mismo secreto, hay una sección con un botón para redirigirnos a la página de dicha prueba.

Al igual que *prove_sd.html*, tiene un botón en la parte inferior de la página que nos redirigirá a la página anterior y otro que nos redigirá a su verificación.

- *verify_s.html*: Similar a *verify_sd.html*, en esta página se realiza la verificación de la prueba de que un número comprometido es un cuadrado, mostrando los valores de entrada, el algoritmo y el resultado de dicha verificación. Al usar la verificación de que dos compromisos esconden el mismo secreto, constará de una sección con un botón para acceder a la página correspondiente a dicha verificación, *verify_ss.html*.

Finalmente, en la parte inferior de la página hay un botón para volver a la página anterior, que puede ser tanto *prove_s.html* o *verify_sd.html* dependiendo de la página que hayamos usado para acceder a esta.

- *prove_li.html*: Funcionamiento análogo a *prove_s.html*, pero con la prueba de que un número comprometido pertenece a un intervalo. Se muestra la entrada, el funcionamiento del algoritmo y el resultado obtenido. Al igual que las demás páginas, tenemos un botón para volver a la página anterior.
- *verify_li.html*: Muestra la verificación de la prueba de que un número comprometido pertenece a un intervalo, incluyendo las entradas de la verificación, su funcionamiento y el resultado obtenido. Además, incluye el botón para volver a la página anterior.
- *prove_ss.html*: Esta página muestra la prueba de que dos compromisos esconden el mismo secreto, detallando los valores de entrada, el algoritmo y la salida.

Similar a las otras pruebas, consta de un botón para acceder a la página de su verificación, *verify_ss.html*, y otro botón para volver a la página anterior.

- *verify_ss.html*: Finalmente, en esta última página se muestra la verificación de que dos compromisos esconden el mismo secreto, mostrando la entrada, el funcionamiento y el resultado.

Por lo que, en resumen, tenemos una página inicial para las entradas, y una para cada prueba y para cada verificación, disponiendo en todas menos en la página de entrada los valores de entrada, el funcionamiento del algoritmo, la salida y, en caso de que utilice algún otro algoritmo, un botón para acceder a la página correspondiente a dicho algoritmo. Además, las pruebas incluyen botones para acceder a sus verificaciones.

Un resumen de las diversas páginas y las conexiones entre ellas puede verse en el siguiente esquema de la [Figura 5.3](#). En este esquema, una flecha indica que la página señalada es accesible pulsando el botón correspondiente desde la página en la que se origina la flecha. Aunque, como todas las páginas tienen un botón para volver anterior, es posible recorrerlo en el sentido contrario.

Una guía detallada del funcionamiento de esta herramienta con imágenes de las diferentes páginas web puede encontrarse en el [Anexo A. Manual de usuario para la herramienta docente](#).

Con todo esto, disponemos de una herramienta que permite ver en detalle el funcionamiento del algoritmo, pudiendo modificar los valores de entrada, viendo los resultados obtenidos en cada paso, y pudiendo modificar las salidas para forzar errores en la verificación.

5.3. Verificación del funcionamiento del algoritmo

Finalmente, en esta parte se verifica que el algoritmo funciona tal y como se espera. Esto incluye realizar las dos siguientes comprobaciones:

1. Si los valores obtenidos de la prueba son correctos, la verificación debe devolver que es verdadero.
2. Si los valores obtenidos de la prueba son incorrectos, la verificación debe devolver que es falso.

Además, se va a probar que esto ocurre con distintos valores de entrada, tanto del secreto, como del intervalo, el módulo, los parámetros de seguridad y los generadores. Como extra, se calculan los tiempos de ejecución para ver si existe alguna

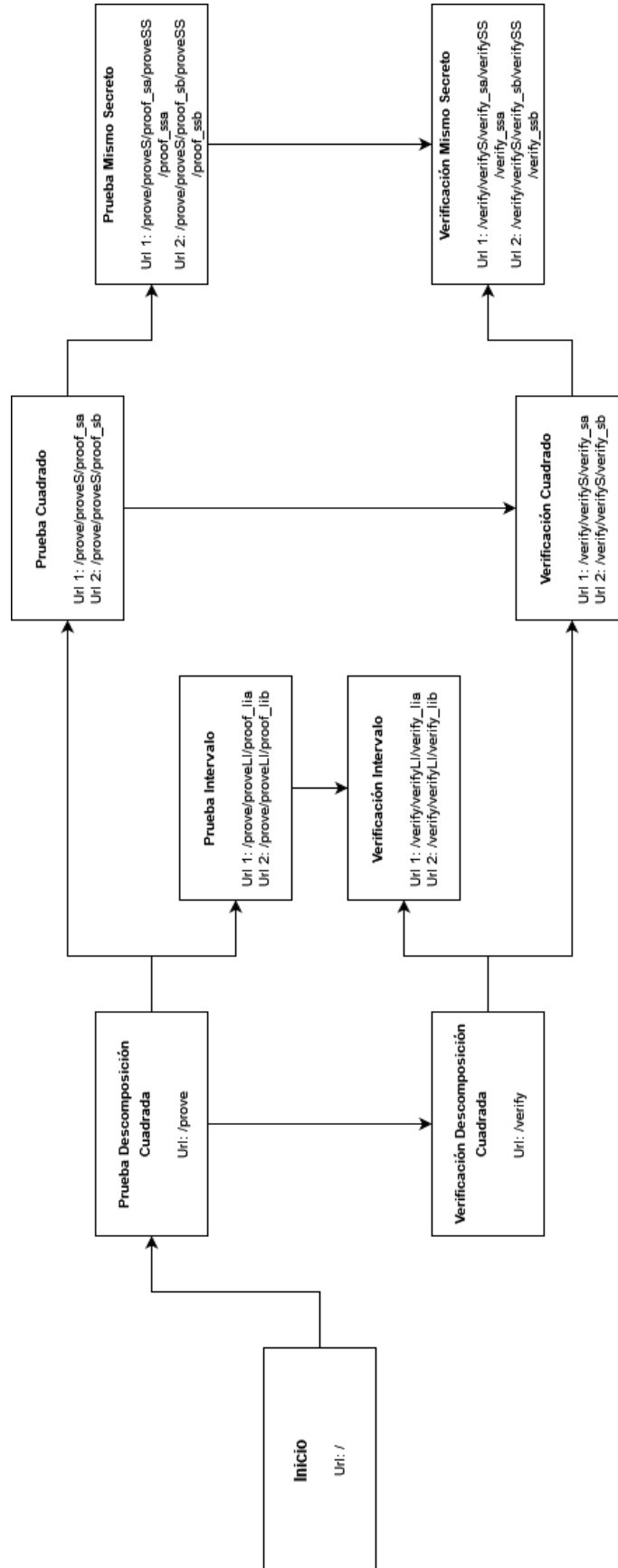


Figura 5.3: Esquema de las páginas web

relación entre estos y el tamaño del módulo.

Para ello, se añade un nuevo fichero a Python, *verification.py*, que es el encargado de realizar las pruebas. Para ello, inicializa un vector de números primos de distinto tamaño que se usa para los módulos de las distintas pruebas, que contiene:

[31, 607, 1291, 2053, 2803, 3637, 4481, 5351, 6203, 7057,
7963, 8867, 9769, 10709, 11699]

valores que se han seleccionado de forma aleatoria a partir de una lista de números primos intentando mantener una distancia similar entre ellos.

Luego, para cada uno de los módulos seleccionados se realizan 200 pruebas, para las cuales se seleccionan valores aleatoriamente en las que se asegura que el intervalo es correcto (la cota inferior es menor que la mayor), que el secreto pertenece al intervalo, y que todos estos valores y los generadores son menores que el módulo. Además, también se toman valores aleatorios para los parámetros de seguridad, que estarán entre 1 y 6 (ambos incluidos). Estos valores son mucho más pequeños porque el algoritmo siempre los usa como potencias, y si fueran mucho más grandes podrían ralentizar los cálculos o incluso provocar errores debido a que genera valores demasiado grandes.

Una vez seleccionados los valores, se utiliza el algoritmo para calcular la prueba y verificar que es verdadera y que, por lo tanto, se consigue la comprobación 1 de esta parte ya que con valores obtenidos de la prueba son correctos, la verificación devuelve que es verdadero.

Para la segunda comprobación, se usa una función a la que se le pasan todos los valores de la prueba, y los modificará de manera aleatoria. Esta aleatoriedad incluye el número de valores que modificará (al menos uno), y cómo lo modificará. Una vez modificados, se vuelve a verificar si esta nueva prueba es correcta.

Con todos estos resultados, se puede construir una matriz de confusión, que es una matriz en la que mostramos, de las pruebas verdaderas, cuáles verifica como correctas (verdaderos positivos, o *True Positives*) y cuáles verifica como incorrectas (falsos negativos, o *False Negatives*), y de las pruebas falsas, que hemos modificado, cuáles verifica como correctas (falsos positivos, o *False Positives*) y cuáles verifica como incorrectas (verdaderos negativos, o *True Negatives*). Los resultados que se obtienen para una ejecución son los siguientes:

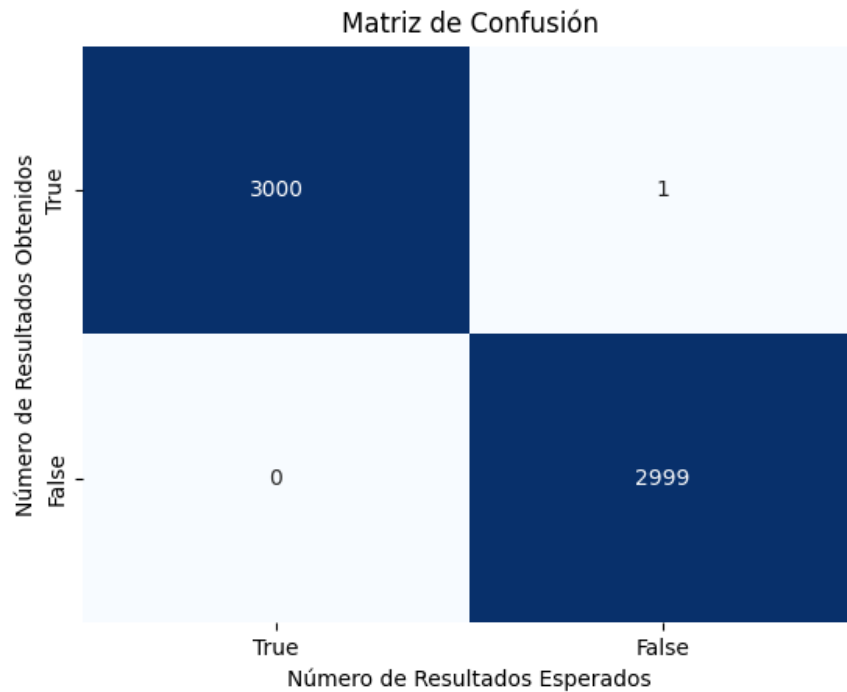


Figura 5.4: Matriz de confusión de los resultados

Estos resultados pueden variar debido a la aleatoriedad tanto en la selección de los valores discutidos en esta parte, como en la propia aleatoriedad de los algoritmos. Sin embargo, las pruebas verdaderas siempre las clasifica como verdaderas (es decir, satisface la completitud) y, de las pruebas falsas, casi todas son clasificadas como falsas.

Para ello, se realizan varias pruebas, y el número de errores que se obtienen en cada una son los de la [Figura 5.5](#). Esto demuestra que, aunque el número de errores puede variar, suele ser relativamente bajo, normalmente menor o igual a 3. Sabiendo que usa 15 valores distintos para el módulo, que se realizan 200 pruebas para cada uno de los valores y que para cada caso se realiza una prueba verdadera sin modificar y otra modificando los resultados, tenemos un total de 6000 pruebas, por lo que el error es inferior al 0.05 %.

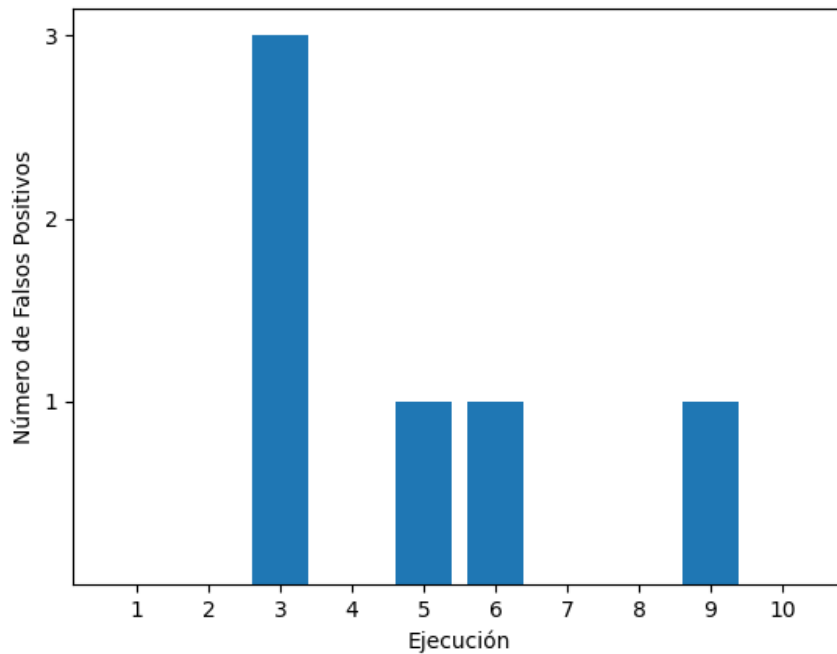


Figura 5.5: Número de errores

El hecho de que acepte estas pruebas falsas puede deberse a los siguientes motivos:

- Debido a la tolerancia. Como se vió en la prueba de que un número comprometido pertenece a un intervalo, tenemos una pequeña tolerancia, θ , que la verificación aceptaría como correcta sin serlo.
- Debido a que la modificación proporciona valores que coincidirían con otra prueba correcta. Esto principalmente ocurre porque, como nuestra intención es mostrar el funcionamiento del algoritmo y, por lo tanto, preferimos tomar valores pequeños que permiten seguir el funcionamiento del algoritmo con mayor facilidad y aceleran los cálculos, hacen que sea más probable que ocurra. Tomando valores más grandes, y modificando la función *Hash* por una más compleja que la identidad, como por ejemplo *md5*, estos casos serán mucho menos probables.

Finalmente se comprueba como afecta el tamaño del módulo al tiempo de ejecución. Para ello, se utiliza la biblioteca *time* para guardar el tiempo antes y después de las 200 pruebas con cada uno de los valores seleccionados, y se calcula la media. Esto incluye tanto el tiempo de generar la prueba y verificarla, como el tiempo de modificarla y volver a modificarla. Los tiempos obtenidos se pueden ver en la

Figura 5.6.

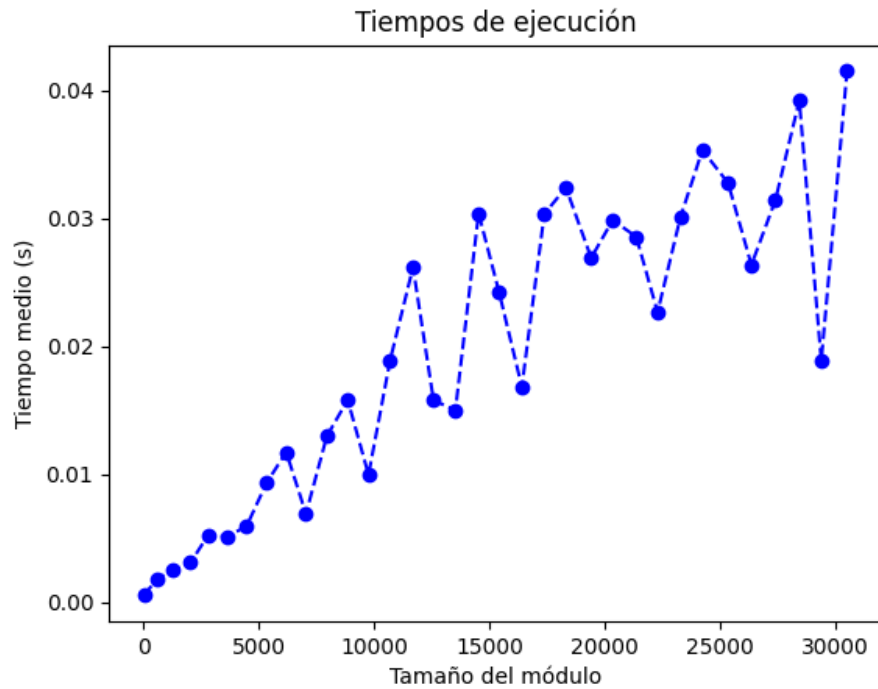


Figura 5.6: Tiempos de ejecución

En esta figura se puede ver como el tiempo aumenta al aumentar el tamaño del módulo de forma aproximadamente lineal, aunque con cierto error debido a la aleatoriedad.

Capítulo 6

Conclusiones y Líneas de Trabajo Futuro

6.1. Conclusiones

Este trabajo comienza viendo algunas aplicaciones de los protocolos de conocimiento cero, y cómo su uso ha ido en aumento en los últimos años. Además, estos usos están afectando cada vez a más campos distintos e influyen elevadas cifras de dinero. Debido a todo esto, es fácil darse cuenta de la importancia que tienen dichos algoritmos, y de lo fundamental que puede ser conocerlos. Sin embargo, no se encontró ningún artículo o herramienta que facilite la formación en este campo.

Todo esto motiva este TFG, que tiene como objetivo explicar el funcionamiento de los protocolos de conocimiento cero de una forma que sea fácil de entender y proporcionar herramientas que permitan ver cómo funcionan en detalle, con distintas entradas y explicando cada paso.

Primero, se comenzó viendo brevemente la historia de los protocolos de conocimiento cero, centrándose en su origen y en sus posibles aplicaciones. También se buscaron artículos relacionados, aunque no se encontró ninguno que cumpla un papel similar a este trabajo.

A continuación, se detalló la teoría detrás de estos protocolos, estudiando qué son y cuáles son los distintos tipos que existen. Además, se detalló el funcionamiento de algunos algoritmos, indicando paso a paso su funcionamiento incluso con algunos ejemplos, para así poder compararlos y elegir el más apropiado para implementar.

Con esto, se pudo detallar los objetivos del TFG, que se pueden resumir en lo que sigue:

1. Este documento, que incluye toda la base teórica, documentación e explicación del funcionamiento de las siguientes partes.
2. Implementación del algoritmo seleccionado en un lenguaje de programación.
3. Diseño de una herramienta que permita ver en detalle el funcionamiento del algoritmo.
4. Verificación de que el algoritmo seleccionado funciona tal y como esperamos, realizando distintas ejecuciones con distintos valores seleccionados de forma aleatoria.

Finalmente, se realizó toda la implementación requerida para esos objetivos, y se documentó su funcionamiento en este documento. El trabajo consigue realizar todos los objetivos planteados, teniendo una herramienta que facilita el entendimiento del algoritmo con la cuál una persona que no conozca el funcionamiento de los protocolos de conocimiento cero, tras hacerse una idea de qué son estos protocolos con la información incluida en la sección [Marco teórico](#), puede llegar a comprender su funcionamiento.

Además, como se vio en la verificación de dicho algoritmo, funciona tal y como deseamos ya que cumple los requisitos de los protocolos de conocimiento cero, aceptando las pruebas verdaderas, rechazando la gran mayoría de las falsas y escondiendo toda la información relevante al secreto.

6.2. Líneas de Trabajo Futuro

Aunque el trabajo ha sido capaz de conseguir todos los objetivos propuestos, hay una serie de mejoras que permitirían un mayor conocimiento de los protocolos de conocimiento cero y que habría sido interesante su estudio e implementación si hubiésemos dispuesto de mas tiempo.

Entre estas, cabe destacar:

- Estudiar otros algoritmos ZKP. A pesar de que el objetivo de este trabajo era estudiar los protocolos de conocimiento cero en general, finalmente sólo se centra en uno, en *Square Decomposition*. Aunque esto permite entender cuál es el objetivo de estos protocolos y algunos de los mecanismos utilizados para poder demostrar algo escondiendo el secreto, se habría conseguido un

mayor entendimiento pudiendo ver el funcionamiento de varios algoritmos distintos.

En particular, cabe destacar tal y cómo se indicó en la [Selección de algoritmo](#), las *Bulletproofs*, que son algoritmos mucho más potentes, pero tienen el gran inconveniente de usar número demasiado grandes que, aunque no son un problema para un ordenador, dificulta en parte entender su funcionamiento.

- Solucionar las pruebas falsas clasificadas como verdaderas. Como se comentó en la [Verificación del funcionamiento del algoritmo](#), esto puede deberse a dos motivos: la tolerancia y que las pruebas falsas coincidan con alguna verdadera. En este segundo caso, no hay forma de solucionarlo, pero el primero puede ser solucionado haciendo los números de mayor tamaño, aunque se decidió no hacerlo por priorizar que sea más fácil de entender y más rápido de ejecutar.

Bibliografía

- [1] Fabrice Boudot. “Efficient Proofs that a Committed Number Lies in an Interval”, n.d. <https://www.iacr.org/archive/eurocrypt2000/1807/18070437-new.pdf>.
- [2] Eduardo Morais, Tommy Koens, Cees van Wijk, and Aleksei Koren. “A Survey on Zero Knowledge Range Proofs and Applications - SN Applied Sciences.” SpringerLink, July 31, 2019. <https://link.springer.com/article/10.1007/s42452-019-0989-z>.
- [3] Geoffroy Couteau, Dahmun Goudarzi, Michael Kloof, and Michael Reichle. “Sharp: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.” ACM Conferences, November 1, 2022. <https://dl.acm.org/doi/10.1145/3548606.3560628>.
- [4] Johann Engelbrecht, Salvador Llinares, and Marcelo C. Borba. “Transformation of the Mathematics Classroom with the Internet - ZDM – Mathematics Education.” SpringerLink, June 26, 2020. <https://link.springer.com/article/10.1007/s11858-020-01176-4>.
- [5] Pranathi Rayavaram, Sreekriti Sista, Ashwin Jagadeesha, Justin Marwad, Nathan Percival, Sashank Narain, and Claire Seungeun Lee. “Designing a Visual Cryptography Curriculum for K-12 Education.” IEEE, February 22, 2023. <https://ieeexplore.ieee.org/abstract/document/10125191>.
- [6] 0xSage. “Understanding Zero-Knowledge Proofs through Simple Examples.” Medium, May 12, 2019. <https://blog.goodaudience.com/understanding-zero-knowledge-proofs-through-simple-examples-df673f796d99>.
- [7] Lupita. “Genuino Cloud: Correo Electrónico Corporativo.” Genuino Cloud — Correo electrónico corporativo, December 6, 2020.

<https://genuinocloud.com/blog/3-consejos-para-proteger-tu-identidad-en-internet/>.

- [8] Katherine Skiba. “Se Disparan Los Fraudes En Instragram Por La Pandemia.” AARP, November 4, 2020. <https://www.aarp.org/espanol/dinero/estafas-y-fraudes/info-2020/se-disparan-los-enganos-en-redes-sociales-por-pandemia.html>.
- [9] Electric Coin Company. “Zcash Privacy Remains Strongest of Any Cryptocurrency, Even with Recent Chainalysis, Elliptic Support.” Electric Coin Company, September 23, 2020. <https://electriccoin.co/blog/zbash-privacy-remains-strongest-of-any-cryptocurrency/>.
- [10] Jana Kane. “Monero (XMR) Price Prediction for 2023, 2024-2025 and Beyond.” LiteFinance, January 9, 2023. <https://www.litefinance.org/blog/analysts-opinions/monero-price-prediction-forecast/>.
- [11] Juan Fornell. “¿Qué Es Zero Knowledge Protocol (ZKP)?” Bit2Me Academy, May 4, 2023. <https://academy.bit2me.com/zkp-zero-knowledge-protocol/>.
- [12] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof-Systems”, February 1989. <https://people.csail.mit.edu/silvio/Selected%20Scientific>
- [13] Karim Bagheri. “CO6GC: Introduction to Zero-Knowledge Proofs (Part 1).” COSIC, June 8, 2022. <https://www.esat.kuleuven.be/cosic/blog/co6gc-introduction-to-zero-knowledge-proofs-1/>.
- [14] Cointelegraph Research. “Pushing Bitcoin to Become More Scalable with Zero-Knowledge Proofs.” Cointelegraph, August 17, 2022. <https://cointelegraph.com/news/pushing-bitcoin-to-become-more-scalable-with-zero-knowledge-proofs>.
- [15] Paul Razvan. “ZK-SNARKs vs. ZK-Starks vs. BulletProofs? (Updated).” Ethereum Stack Exchange, March 5, 2019. <https://ethereum.stackexchange.com/questions/59145/zk-snarks-vs-zk-starks-vs-bulletproofs-updated>.
- [16] “Zero-Knowledge Proofs.” ethereum.org, n.d. <https://ethereum.org/en/zero-knowledge-proofs/>.

-
- [17] Cathie Yun. “Building on Bulletproofs.” Medium, July 11, 2021. <https://cathieyun.medium.com/building-on-bulletproofs-2faa58af0ba8>.
 - [18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. “Bulletproofs: Short proofs for confidential transactions and more”, n.d. <https://eprint.iacr.org/2017/1066.pdf>.
 - [19] Boneh, Dan, Ben Lynn, and Hovav Shacham. “Short Signatures from the Weil Pairing.” SpringerLink, November 20, 2001. https://link.springer.com/chapter/10.1007/3-540-45682-1_30.
 - [20] Brown, Daniel R. L. “Sec 2: Recommended elliptic curve domain parameters.” January 27, 2010. <https://www.secg.org/sec2-v2.pdf>.
 - [21] “Elliptic Curve Cryptography (ECC).” Elliptic Curve Cryptography (ECC) - Practical Cryptography for Developers, n.d. <https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc>.
 - [22] Torben Pryds Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing.” SpringerLink, 1992. https://link.springer.com/chapter/10.1007/3-540-46766-1_9.
 - [23] “Zero-Knowledge Proof.” Wikipedia, May 27, 2023. https://en.wikipedia.org/wiki/Zero-knowledge_proof.
 - [24] “MD5 Hash: Generate MD5 Message Digests Online.” cryptii, n.d. <https://cryptii.com/pipes/md5-hash>.
 - [25] BarD Software s.r.o. “Free Project Management Tool for Windows, MacOS and Linux.” GanttProject, n.d. <https://www.ganttproject.biz/>.
 - [26] “AES Animation.” CrypTool Portal, n.d. <https://www.cryptool.org/en/cto/aes-animation>.
 - [27] “Free, Open Source, and Modern CSS Framework Based on Flexbox.” Bulma, n.d. <https://bulma.io/>.

Anexo A. Manual de usuario para la herramienta docente

El objetivo de este trabajo es hacer más fácil el comprender como funcionan los protocolos de conocimientos cero. Para ello, como se desarrolló en la [Selección de algoritmo](#), se decide implementar el algoritmo de la *Descomposición Cuadrada* (*Square Decomposition*), y proporcionar una herramienta que permita ver en detalle su funcionamiento con ejemplos numéricos, detallando la entrada y salida de cada uno de los algoritmos que usa.

Además, para poder experimentar más con el funcionamiento de dicho algoritmo, se permite que se puedan modificar la entrada y la prueba generada por el algoritmo, permitiendo así que falle la verificación.

Esta herramienta estará implementada como una página web con HTML, para que así una vez que este funcionando el servidor no se requieran elevadas especificaciones técnicas en el ordenador del cliente, lo cual facilita que cualquier persona pueda utilizarla.

La interfaz utiliza distintas páginas, una para las entradas de los valores que usará el algoritmo, y una para cada una de las pruebas y verificaciones que utiliza el algoritmo. Las relaciones entre estas páginas quedan resumidas en el siguiente esquema de la [Figura 5.3](#)

Este esquema puede ser algo complicado de entender si no se conoce la herramienta, por lo que a continuación se desarrolla explicando además el funcionamiento de cada una de las páginas.

Ejecución del servidor

Antes de desarrollar las distintas páginas que forman parte de la herramienta, es necesario explicar cómo poder acceder a ellas. Lo primero que hay que hacer es ejecutar el servidor, lo cuál simplemente requiere ejecutar el fichero *flask_backend.py*. Tras su ejecución, se obtiene la siguiente salida:

```
* Serving Flask app 'flask_backend'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 168-650-935
```

Figura: Salida tras la ejecución del servidor

Esto permite saber que el servidor se encuentra en funcionamiento, y por lo cual que se puede acceder a la herramienta de una de las siguientes formas:

- Haciendo click sobre la dirección IP que aparece en dicha salida. Esto abrirá la herramienta en una nueva pestaña del navegador por defecto.
- Accediendo a localhost:5000/ en un navegador.

Como la herramienta funciona como cliente/servidor, donde el navegador funciona como cliente y Python funciona como servidor, si se detiene la ejecución del fichero *flask_backend.py*, la herramienta dejará de funcionar.

Elementos en común

Todas las páginas tiene algunos elementos en común, por lo que en lugar de repetirlos y explicarlos en cada sección, se desarrollan a continuación.

Para saber a qué nos referimos, se incluye la [Figura 1](#) con una página cualquiera que se usará como ejemplo para explicar estos elementos.

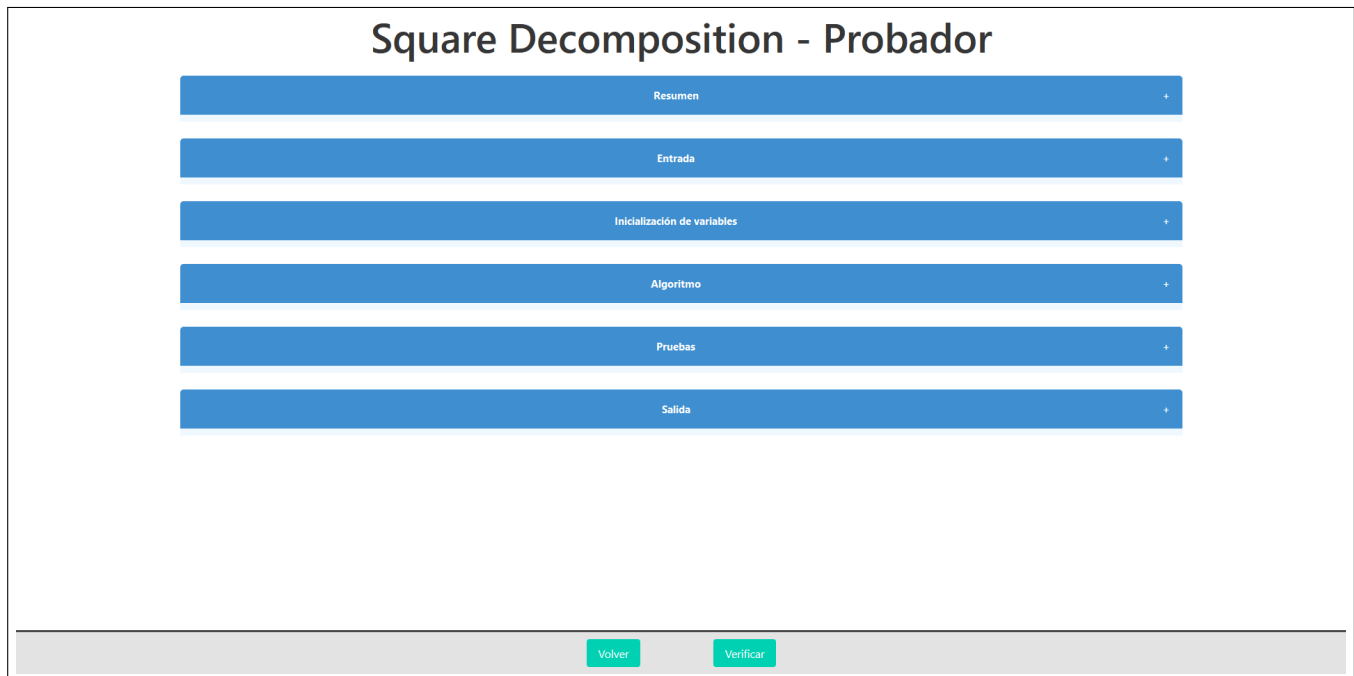


Figura 1: Página de la prueba de la Descomposición Cuadrada

Como se puede ver en la [Figura 1](#), lo primero que hay en la página es el título indicando qué se está realizando en dicha página. Por ejemplo, en esta página se muestra el algoritmo que realiza el probador al usar el algoritmo *Square Decomposition*.

Luego, el cuerpo de la página está formado por distintas cajas de color azul con un texto en medio. Este texto indica que se realiza o que datos se muestran en dicha caja. Por ejemplo, la caja con el nombre “Resumen” contiene un breve resumen del funcionamiento del algoritmo, mientras que la caja “Entrada” contiene la lista de valores que el algoritmo toma como entrada. Para ver el contenido de la caja, únicamente hay que hacer click sobre ella, lo cuál provocará que se expanda mostrando el contenido de la misma. Haciendo otro click, la caja volverá a cerrarse ocultando el contenido y permitiendo así centrarnos en la parte que nos interesa, sin necesidad de ver el contenido de otras cajas que puedan ser una distracción. La única diferencia a esta regla son en el resultado de las verificaciones, cuya caja se mostrará extendida por defecto.

Por ejemplo, el contenido de la página de la figura anterior con alguna caja extendidas es el de la [Figura 2](#).

Square Decomposition - Probador	
Resumen +	
Entrada -	
x:	13
n:	221
a:	0
b:	30
g:	7
h:	14
t:	5
l:	3
s:	4
Inicialización de variables +	
Algoritmo +	
Pruebas -	
Volver Verificar	

Figura 2: Página de la prueba de la Descomposición Cuadrada

En la [Figura 2](#) se puede ver como se ha extendido la caja con los valores de la entrada. Esto provoca que el resto de cajas que se encuentran debajo se desplacen para dejar el espacio suficiente. Además, en caso de no entrar en la página, como ocurre en la imagen, aparecerá una barra lateral que nos permitirá desplazarnos hacia arriba o hacia abajo en la página.

Otro elemento que tienen en común en todas las páginas es la barra inferior. Esta barra contendrá uno o varios botones dependiendo de la página en la que nos encontremos, y siempre se encontrará anclada al borde inferior de la página encima de los demás elementos de la página.

En todas las páginas menos en la página inicial hay un botón para volver a la página anterior. En el caso de que haya una página a la que sea posible acceder desde varias distintas, nos devolverá a la que hayamos usado para acceder.

Además, en todas las páginas en las que se incluya una prueba, también habrá un botón para acceder a la página que contiene la verificación de dicha prueba.

Inicio

Al iniciar la herramienta, la primera página a la que se accede es la siguiente:

Square Decomposition

Entrada

Intervalo: $x \in [a, b]$

Bases

Parámetros de seguridad

Continuar

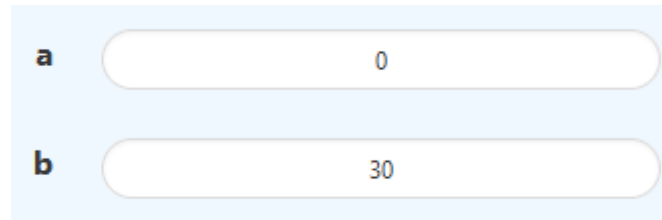
Figura 3: Página de inicio

Esta página sirve para poder indicarle los valores que tomará el algoritmo, divididos en las siguientes entradas:

- Entrada: Tiene las siguientes entradas:
 - x : Valor del secreto.
 - n : Valor del módulo, es decir, el algoritmo utilizará el anillo \mathbb{Z}_n .
- Intervalo: Tiene los siguientes valores del intervalo en el que queremos probar que x pertenece:
 - a : Valor inferior del intervalo.
 - b : Valor superior del intervalo.
- Bases: Tiene las entradas de las bases del algoritmo:
 - g .
 - h : Tiene que estar generada por g en \mathbb{Z}_n .
- Parámetros de seguridad: Contiene las entradas de los parámetros de seguridad a usar:
 - t .

- ℓ .
- s .

Las entradas son de la siguiente forma:



El formulario muestra dos campos de entrada. El primer campo, etiquetado con la letra 'a' en negrita, contiene el número 0. El segundo campo, etiquetado con la letra 'b' en negrita, contiene el número 30. Ambos campos están rodeados por un borde redondeado y se encuentran sobre un fondo azul claro.

Ejemplo de campo de entrada

Donde el texto que hay a la izquierda es el nombre de la entrada, que será uno de los indicados previamente, y la caja que hay a la derecha sirve para entrar el valor deseado. Para introducir un valor, simplemente hay que hacer click sobre la caja correspondiente, lo cuál nos permitirá modificarla leyendo los valores introducidos con el teclado.

Para poder ejecutar la herramienta de manera más rápida, todas las cajas están inicializadas con un valor por defecto que puede ser modificado en el fichero *input.html*.

Al pulsar el botón “Continuar” en el menú inferior, la herramienta leerá todos los valores introducidos y hará una serie de comprobaciones, como que ambos extremos del intervalo sean menores que el valor del módulo, que la cota inferior sea menor que la cota superior del intervalo, y que el secreto pertenezca al intervalo. Si alguna de estas comprobaciones falla, aparecerá un mensaje por pantalla indicando el error para que pueda ser solucionado. Si todas las comprobaciones son correctas, redigirá hacia la siguiente página, que contiene los resultados de la prueba.

Pruebas

En esta sección se detalla el funcionamiento de cuatro páginas, que sirven para las pruebas de los distintos algoritmos usados, ya que todas ellas tienen un funcionamiento. Estas son la prueba del algoritmo [Square Decomposition](#), la [Prueba de que un número comprometido es un cuadrado](#), la [Prueba de que dos compromisos esconden el mismo secreto](#) y la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#).

Para ello se verá en detalle la primera de ellas, y se comentaran las diferencias que incluyen las demás. La página para esta prueba es la siguiente:

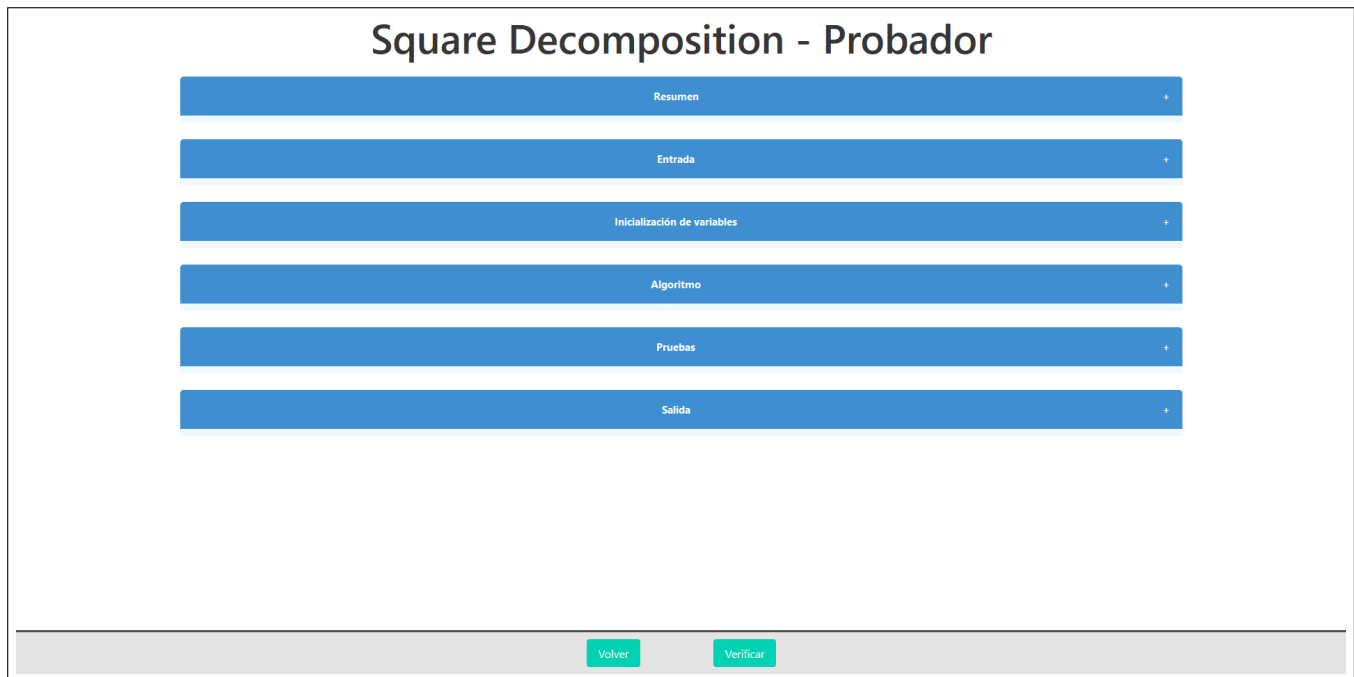


Figura 5: Página de la prueba de la Descomposición Cuadrada

Las distintas partes que incluyen son:

- **Resumen.** Aquí se hace un breve resumen del funcionamiento del algoritmo. Sin embargo, no tiene la intención de detallar su funcionamiento, lo cuál se hace en el [Marco teórico](#).
- **Entrada.** Contiene una lista con todos los valores que toma el algoritmo como entrada, que en el caso del ejemplo, en la prueba de la *Descomposición Cuadrada* será los que se hayan introducido en la página anterior, y en las demás pruebas serán los correspondientes según el algoritmo.
- **Iniciación de variables.** Esta parte es exclusiva de la prueba de la *Descomposición Cuadrada*. Antes del algoritmo, hay algunas variables que se inicializan de forma aleatoria, que se detallan en esta caja.
- **Algoritmo.** Se muestra la ejecución del algoritmo, incluyendo tanto las expresiones con el nombre de las variables como con los valores de la ejecución realizada. Con esto, se puede seguir fácilmente el algoritmo, sabiendo de dónde proceden los valores utilizados en el algoritmo.

- Pruebas. En el caso de que la prueba utilice otro algoritmo, se incluye botones para acceder a la página de dicha prueba. Esto ocurre en la prueba del algoritmo [Square Decomposition](#), que incluye botones a la [Prueba de que un número comprometido es un cuadrado](#) y a la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#); y en el caso de la [Prueba de que un número comprometido es un cuadrado](#), que incluye botones a la [Prueba de que dos compromisos esconden el mismo secreto](#).

En las otras pruebas, estos campos no existen.

- Salida. Aquí se muestra una lista con los valores obtenidos tras la ejecución del algoritmo. En el caso de la prueba de la *Descomposición Cuadrada*, la mayoría de estos valores se pueden modificar para forzar así que la prueba sea incorrecta, lo cuál permite ver como la verificación puede rechazar la prueba.

En el caso de que se modifique estas salidas, sobrescribirán los valores que se tenían previamente al pulsar cualquier botón que nos redirija a otra página, por lo que si queremos volver a obtener la prueba correcta será necesario volver a la página de inicio y volver a introducir los valores.

Verificaciones

Igual que en la sección anterior con las pruebas, a continuación se desarrollan todas las verificaciones en la misma sección. Esto incluye las verificaciones de [Square Decomposition](#), de la [Prueba de que un número comprometido es un cuadrado](#), de la [Prueba de que dos compromisos esconden el mismo secreto](#) y de la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#).

De nuevo, se usará la verificación de la primera de ellas como ejemplo, y se comentarán las diferencias que existan en las demás:

The screenshot shows a web interface titled "Square Decomposition - Square - Verificación". It features four blue expandable sections, each with a "+" icon on the right. The first section, "Resultado", is expanded and displays "• Correcto •" in green. The other sections are "Entrada", "Algoritmo", and "Verificación Verifys". At the bottom of the interface is a grey bar containing a green "Volver" button.

Figura 6: Página de la verificación de la Descomposición Cuadrada

Los campos que incluyen son:

- **Resultado.** Esta es la única caja que por defecto está expandida, para una mayor comodidad a la hora de comprobar los resultados. Contiene los resultados de la verificación, que puede ser correcta (en cuyo caso aparecerá en verde) o incorrecta (en cuyo caso aparecerá en rojo).
- **Entrada.** Contiene una lista con los valores que el algoritmo toma como entrada, que coincidirá en gran medida con la salida de la prueba correspondiente.
- **Algoritmo.** Igual en las pruebas, se incluye el funcionamiento paso por paso de la verificación, incluyendo tanto las fórmulas con los nombres de las variables como los valores de esta ejecución.
- **Verificaciones.** En caso de que la verificación requiera de otro algoritmo de verificación, se incluirá un botón en esta caja para acceder a la página de dicha verificación. Esto ocurre en el caso de la verificación de [Square Decomposition](#), que requiere las verificaciones de la [Prueba de que un número comprometido es un cuadrado](#) y de la [Prueba de que un número comprometido pertenece a un intervalo \(prueba CFT\)](#); y en la verificación de la [Prueba de que un número comprometido es un cuadrado](#) que requiere la verificación

de la Prueba de que dos compromisos esconden el mismo secreto.