Artículo Original / Original Research

Ciencias Matemáticas, Vol. 38, No. 2, 2024, Pag. 81-104
Recibido (*Received*): 08-01-2025, Revisado (*Reviewed*): 03-03-2025
Aceptado (*Accepted*): 13-07-2025, En línea (*Online*): 12-09-2025

zk-SNARK para desarrolladores *blockchain*
zk-SNARK for blockchain developers

Enmanuel Cabrera Bello[1]

Resumen Este trabajo presenta ZKATM, una aplicación descentralizada (DApp) que implementa transacciones privadas mediante pruebas de conocimiento cero (ZKP), optimizada para escalabilidad y accesibilidad. Desarrollada en *Scroll Sepolia* (ZKEVM) con árboles de Merkle históricos, ZKATM reduce los costos de transacción frente a soluciones similares en *Ethereum Mainnet*, manteniendo privacidad *end-to-end*. Como aporte educativo, incluimos una guía, que desglosa el flujo completo de construcción de pruebas: desde la compilación de circuitos aritméticos (R1CS) en *Circom* hasta la generación de pruebas y verificación. Los resultados muestran que es posible democratizar el uso de ZKP en aplicaciones reales.

Palabras Clave: blockchain, circom, criptografía, groth16, pruebas de conocimientos cero, snarkjs, zk, zk-snark, web3.

Abstract *This work presents ZKATM, a decentralized application (DApp) that implements private transactions using zero-knowledge proofs (ZKPs), optimized for scalability and accessibility. Built on Scroll Sepolia (ZKEVM) with historical Merkle trees, ZKATM reduces transaction costs compared to similar solutions on the Ethereum Mainnet while maintaining end-to-end privacy. As an educational contribution, we include a guide, which breaks down the complete proof construction flow: from arithmetic circuit compilation (R1CS) in Circom to proof generation and verification. The results show that it is possible to democratize the use of ZKPs in real-world applications.*

Keywords: *blockchain, circom, cryptography, groth16, zero knowledge proofs, snarkjs, zk, zk-snark, web3.*

Mathematics Subject Classification: 94A60, 68M25, 14H52, 13P15, 68W01.

[1] Instituto de Criptografía, Universidad de La Habana, La Habana, Cuba. Email: cenmanuel257@gmail.com.

[*] **Autor para Correspondencia (*Corresponding Author*)**

Editado por (*Edited by*): Damian Valdés Santiago, Facultad de Matemática y Computación, Universidad de La Habana, La Habana, Cuba.

Citar como: Cabrera Bello, E. (2024). zk-SNARK para desarrolladores *blockchain*. *Ciencias Matemáticas*, 38(2), 81-104. DOI: https://doi.org/10.5281/zenodo.17110031. Recuperado a partir de https://revistas.uh.cu/rcm/article/view/10616.

Introducción

En el ecosistema digital actual, donde las grandes empresas monetizan los datos recopilados de los usuarios, la privacidad se presenta como un desafío significativo. La seguridad en el mundo digital está seriamente comprometida. El abuso de datos se ha convertido en un problema global con consecuencias de gran alcance, dado que los datos constituyen un recurso fundamental en la sociedad digital contemporánea.

Los datos son manejados por corporaciones. Estas funcionan con sistemas obsoletos y altamente vulnerables, que ponen en riesgo la privacidad y seguridad de los datos personales. El almacenamiento de datos se está convirtiendo en un problema. La naturaleza segmentada de los datos significa que la integración se vuelve más complicada y costosa a medida que aumenta el volumen de estos datos.

Independientemente de la seguridad proporcionada por los esquemas de cifrado, el problema de la privacidad en el que puede recaer cualquier tipo de información al ser transmitida por un canal de comunicación, puede ser alterada, ya sea por un ataque *man-in-the-middle* o por la fuga de la información por parte del servicio o entidad receptora (Figura 1). Entonces, ¿de qué forma se pueden resolver estos desafíos?

Existe un área especial en la criptografía encargada de estudiar esta cuestión, conocida como prueba de conocimiento cero (*zero-knowledge proof*, ZKP, por sus siglas en inglés). En la que coexisten una familia de protocolos criptográficos probabilísticos, descritos por primera vez en 1985.

Esto permite a una probadora (denominada como Peggy), convencer a un verificador débil (nombrado como Victor), de que una afirmación es verdadera, sin filtrar ninguna información adicional sobre la afirmación más allá de su validez. De manera más general, los ZKP se pueden utilizar como componentes básicos para el cálculo verificable.

Figura 1. Un atacante observa el proceso de autenticación o intercambio de datos, los datos pueden ser filtrados [*An attacker observes the authentication process or data exchange, data can be leaked.*].

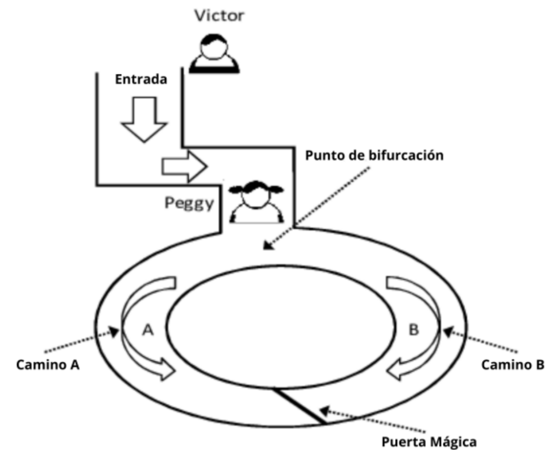

Figura 2. Protocolo ZKP en la cueva de Ali Babba, donde Peggy demuestra a Victor que conoce la contraseña sin revelarla [*ZKP protocol in Ali Babba's cave, where Peggy proves to Victor that she knows the password without revealing it.*].

Un protocolo ZKP debe satisfacer tres propiedades:

1. Integridad/Totalidad: si la afirmación es verdadera, entonces un probador puede convencer a un verificador.

2. Solidez/Solvencia: un probador que hace trampa no puede convencer a un verificador de una declaración falsa, al menos con una baja probabilidad.

3. Conocimiento cero: la interacción solo revela si una afirmación es verdadera.

Para ilustrar cómo funcionan las pruebas de conocimiento cero, en 1990 Guillou, Quisquater y Berson publicaron el artículo: *¿Cómo explicar los protocolos de conocimiento cero a sus hijos?* [13]. Una historia sobre cómo Ali Baba demostró que podía abrir la puerta de la cueva, sin revelar a nadie las palabras mágicas. Si bien no profundiza en los detalles técnicos, expone la variante de una manera interactiva y probabilística, demostrando el concepto de privacidad que preserva el intercambio de conocimientos.

En la cueva de Ali Babba, Peggy demuestra a Victor que conoce la contraseña de la puerta sin revelarla. Victor no ve a Peggy entrar. Cada vez que llame a Peggy a que se revele por un camino, Peggy saldrá siempre por el correcto si conoce la contraseña, pero eventualmente se equivocará sino la conoce, con una probabilidad del 50 %. Mientras más interacciones existan, aumenta la probabilidad en $1-\frac{1}{2^n}$ de que Peggy conozca la contraseña, siendo n la cantidad de interacciones (Figura 2).

Las pruebas de conocimiento cero (ZKP), y en particular los zk-SNARKs, han revolucionado la privacidad en *blockchain* al permitir transacciones verificables sin revelar información sensible. Sin embargo, su adopción enfrenta dos barreras críticas: (1) la complejidad técnica para desarrolladores, y (2) los desafíos regulatorios derivados de aplicaciones como *Tornado Cash*, sancionada en 2022 por facilitar lavado de activos[1]; recientemente las sanciones han sido retiradas de la OFAC [2].

Este trabajo presenta ZKATM, una aplicación descentralizada (DApp) con fines educativos que aborda ambos problemas mediante:

1. Un diseño técnico reproducible: se implementa un esquema de depósitos/retiros anónimos en *Scroll Sepolia* (ZKEVM), combinando circuitos zk-SNARK (*Groth16*) con árboles de Merkle históricos para reducir costos con respecto a *Ethereum Mainnet*.

2. Documentación estructurada que desglosa el proceso desde R1CS hasta la generación de pruebas.

Este artículo sigue un flujo lógico desde los fundamentos teóricos hasta la implementación práctica, estructurado en tres bloques clave: En la sección 1, se formaliza zk-SNARK como sistema de prueba interactivo, destacando su completitud, solidez y conocimiento cero. En la sección 2, se introduce *Circom* como DSL para compilar problemas computacionales en circuitos aritméticos (R1CS), con énfasis en la representación eficiente de restricciones. La sección 3 detalla la transformación R1CS → QAP mediante homomorfismos, demostrando cómo los vectores de coeficientes se mapean a polinomios $\mathbb{F}^n \to \mathbb{F}[x]/\langle x^k-1\rangle$). La sección 4 analiza *Groth16*, su configuración confiable (*trusted setup*), y optimizaciones para reducir el costo de verificación en un tiempo constante mediante

[1] https://home.treasury.gov/news/press-releases/jy0916

[2] https://home.treasury.gov/news/press-releases/sb0057

emparejamientos bilineales. La sección 5 presenta ZKATM, una DApp en *Scroll Sepolia*, una *blockchain* capa 2 que escala *Ethereum* que utiliza Merkle Trees históricos para minimizar costos en transacciones.

Relevancia del estudio

Este estudio resalta el valor de las pruebas de conocimiento cero (ZKP) como solución que combina transparencia y privacidad en *blockchain*. En un contexto donde la protección de datos es esencial, ZKP permite transacciones verificables sin exponer información sensible. La implementación en entornos como *Scroll Sepolia*, junto con optimizaciones para escalabilidad y costos, demuestra su aplicabilidad real. Además, su enfoque educativo, explicando desde la compilación de circuitos en *Circom* hasta la verificación con *Snarkjs*, fomenta la adopción en desarrolladores y organizaciones. Este trabajo sienta bases para aplicaciones descentralizadas seguras en sectores críticos, proyectando su expansión hacia otras tecnologías emergentes.

1. zk-SNARK

Hasta hace poco ZKP no era realmente práctico y la mayoría de las soluciones sugeridas requerían múltiples rondas de interacción entre las partes o pruebas enormes. Tras los trabajos de Groth y Sahai [11] y Ben-Sasson et al. [2] se conoció una familia particular de ZKP que se describe como argumentos (AR) de conocimiento (K) sucintos (S) y no interactivos (N) de conocimiento cero (ZK), conocidos como zk-SNARK. Los zk-SNARK se encuentran entre los protocolos de conocimiento cero más utilizados y la criptomoneda anónima *Zcash* y la plataforma de contratos inteligentes *Ethereum* se encuentran entre los primeros en adoptarlos, además son la columna vertebral de numerosas soluciones para escalar *blockchain* en la actualidad.

En zk-SNARK, cuando se hace referencia a sucinto, se dice que las pruebas generadas son pequeñas y ocupan poco espacio, facilitando las verificaciones rápidas y eficientes. Las pruebas son mucho más cortas que la forma matemática clásica de pruebas interactivas descrita anteriormente en el ejemplo de la cueva de Ali Babba. Al hacer que una prueba no sea interactiva esta se pueda enviar o presentar a otros verificadores para su validación inmediata. En la Figura 2, Peggy solo puede convencer a Victor con la prueba interactiva, pero no puede enviarla a otros para que la verifiquen.

A continuación se realiza una explicación introductoria a zk-SNARKs en un contexto semitécnico. La Figura 3 desglosa los componentes principales de zk-SNARKs mediante tres algoritmos fundamentales: el *generador*, que produce claves públicas y privadas; el *probador*, que utiliza el secreto para generar una prueba; y el *verificador*, que evalúa la validez de dicha prueba utilizando las claves públicas.

Además, se destacan las propiedades de seguridad de este sistema, como la imposibilidad de reutilizar o falsificar las pruebas en otros contextos, gracias a elementos secretos como el valor λ, que garantiza la unicidad de las claves y las pruebas.

El propósito de estas ilustraciones es hacer accesible un concepto técnico complejo, mostrando cómo los zk-SNARKs permiten verificar información sensible sin exponer los datos confidenciales.

La Figura 4 muestra el proceso completo, bajo las primitivas criptográficas, para generar pruebas de conocimiento cero, utilizando el sistema de pruebas *Groth16*, desde la conceptualización inicial hasta la creación de las pruebas.

Este enfoque de utilizar circuitos con lógicas específicas se sigue en múltiples aplicaciones, especialmente en *blockchain*. Debido a su eficiencia de rápida verificación en emparejamiento de curvas elípticas y seguridad basada en el problema de logaritmo discreto, el sistema de pruebas de *Groth16* [11] es el que se describe en este documento. A continuación, se detalla un resumen de cada paso.

El primer paso es diseñar un circuito aritmético, que representa el problema que se desea resolver o verificar. Estos circuitos están compuestos por puertas lógicas (restricciones), que definen las operaciones que deben cumplirse. En aplicaciones complejas, estos circuitos pueden contener miles de puertas.

El segundo paso es convertir el circuito en un sistema de ecuaciones llamado R1CS (*rank-1 constraint system*). Este sistema describe cada puerta lógica como una ecuación de la forma $a \cdot b = c$, utilizando tres vectores dispersos [a][b][c], que se describirán en la próxima sección. Esta representación permite transformar las operaciones lógicas en un marco algebraico.

Una vez que se tiene el sistema R1CS, este se transforma en un programa aritmético cuadrático (QAP, por sus siglas en inglés). En esta etapa, las ecuaciones se convierten en polinomios y se trabajan como productos escalares de estos polinomios. El QAP permite agrupar todas las ecuaciones en una única estructura matemática que puede ser evaluada de manera eficiente.

Después, los polinomios generados en el QAP se evalúan en puntos específicos sobre una curva elíptica (EC, por sus siglas en inglés). Este proceso utiliza parámetros generados en una configuración inicial confiable (*trusted setup*). Este paso es crítico, ya que garantiza la seguridad del sistema al tiempo que permite construir la prueba sin revelar los datos subyacentes.

Finalmente, se genera la prueba zk-SNARK utilizando el sistema *Groth16*. Este protocolo crea pruebas muy compactas y rápidas de verificar en tiempo constante. Lo que lo hace ideal para aplicaciones como contratos inteligentes en *blockchain*. Estas pruebas permiten demostrar que se cumplió un cálculo complejo sin revelar información confidencial.

Groth16 es uno de los protocolos más utilizados para zk-SNARKs debido a su tamaño de prueba reducido y rápida verificación. Sus pruebas son constantes en tamaño, características cruciales para entornos como *blockchain*. Sin embargo, requiere una configuración inicial confiable, lo que puede re-

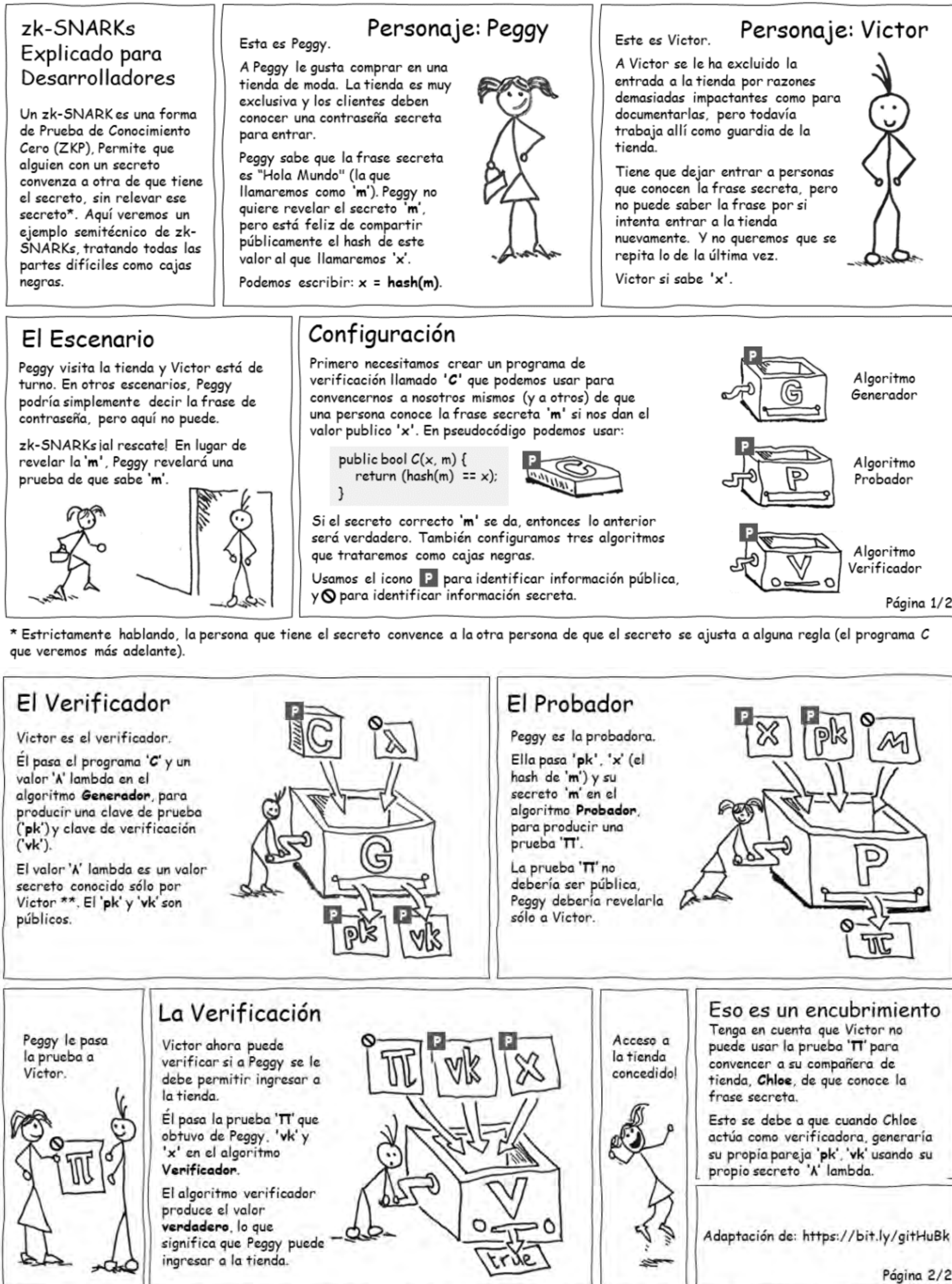

Figura 3. Funcionamiento de los zk-SNARKs: roles del verificador y probador, así como el proceso de verificación [*How zk-SNARKs Work: Roles of the verifier and prover, as well as the verification process*].

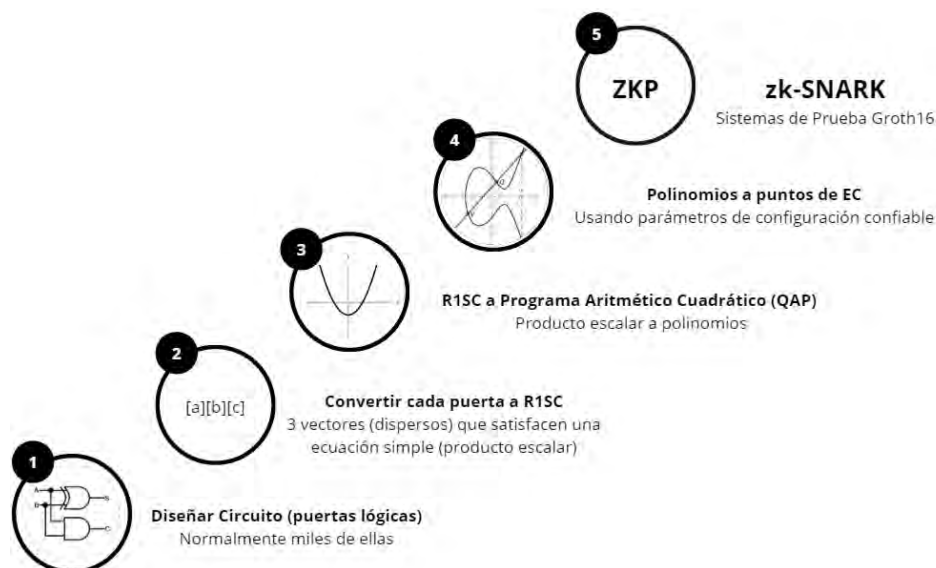

Figura 4. Flujo de generación de pruebas zk-SNARK con *Groth16* [*zk-SNARK proof generation flow with* Groth16].

presentar un riesgo si no se realiza correctamente. A pesar de ello, *Groth16* sigue siendo uno de los sistemas más populares, aunque existen alternativas como PLONK y HALO 2, que buscan superar algunas de sus limitaciones.

2. Lenguaje Específico de Dominio *Circom*

Los zk-SNARKs no se pueden aplicar directamente a ningún problema computacional; más bien, hay que convertir el problema en la forma correcta para que funcionen. Esto se realiza con un QAP. Para transformar el código de un problema en uno de estos se debe construir antes un sistema de restricciones conocido como R1CS, basadas en circuitos algebraicos. Este sistema se expresa como un conjunto de vectores y matrices que se puede ejecutar para que, si un probador tenga una entrada para una declaración del problema (circuito), cree una solución correspondiente llamada testigo (*witness*).

En este sección se pondrán en práctica fundamentos de la aritmética modular. Se implementará una aplicación con una herramienta avanzada de zk-SNARK de bajo nivel. *Circom* es un Lenguaje Específico de Dominio (DSL, por sus siglas en inglés) que se ha convertido en un estándar por los desarrolladores para el diseño de circuitos. Se construirá un R1CS de zk-SNARK, analizando los componentes indispensables de esta herramienta, para la aritmetización de la declaración de un probador. Se necesitan varias dependencias en su sistema para ejecutar *Circom* y sus herramientas asociadas, disponibles en `https://iden3.github.io/circom`.

Para mostrar algunas implementaciones de cálculos aritméticos se utiliza la herramienta *SageMath*[3], un software matemático de distribución gratuita basado en *Python*. Se puede usar *SageMath* para verificar rápidamente la exactitud de los resultados obtenidos, asegurando que los cálculos aritméticos sean correctos. Para conocer más sobre la herramienta ver la documentación en `https://doc.sagemath.org/html/en/tutorial`.

Con el testigo y el archivo R1CS, se puede calcular y verificar pruebas usando la librería *Snarkjs*[4]. La arquitectura (Figura 5), que se describe en el [1], se puede resumir como sigue:

- Primero, se crea un circuito en *Circom* con el cálculo a verificar de nuestra declaración o problema computacional.
- Para satisfacer todas las señales de entradas ante las restricciones que el compilarlo obtuvo en el archivo `.r1cs`. Se generá un archivo `.wtns` con el módulo *wasm* de *Circom* que contiene todos los testigos posibles de las restricciones del circuito dado los valores de entradas adecuado en formato JSON.
- Ambos archivos, `.wtns` y `.r1cs`, se emplean para calcular la prueba, utilizando la librería *Snarkjs*.

Circom describe procedimientos matemáticos, simplificando una ecuación matemática compleja en operaciones suma y multiplicacion modulares de un número primo grande, por ejemplo, $p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$.

Esto define una estructura algebraica de campo finito, que es el circuito aritmético (descripción del problema en un circuito). La satisfacibilidad de circuitos es un problema NP-completo en el que solo es posible verificar una solución candidata. Este sistema de verificación es el de ecuaciones

[3] `https://www.sagemath.org/download.html`

[4] `https://github.com/iden3/snarkjs`

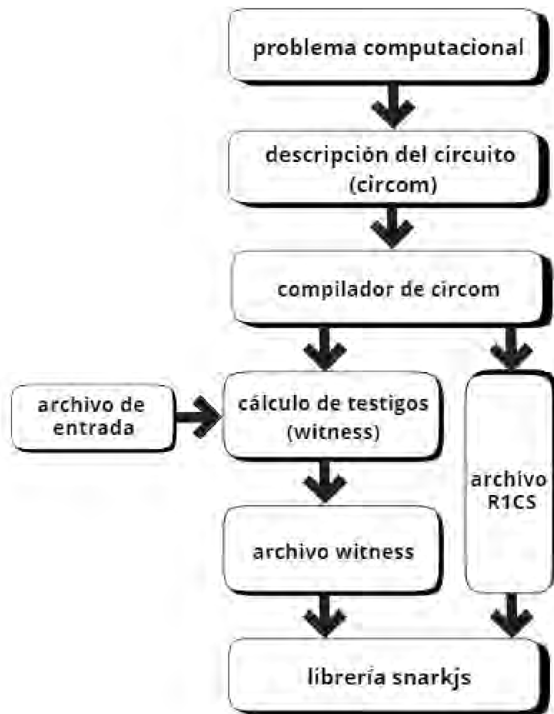

Figura 5. Flujo de trabajo de las libreria *Circom* y *Snarkjs* [*Circom and Snarkjs library workflow*].

R1CS que describe la relación entre las variables que intervienen en el circuito.

Código 1. Iniciando un nuevo proyecto con Node.js.

```
npm init -y
npm i --save circom snarkjs
```

2.1 Aritmética Modular

La división de un número entero por otro entero diferente de cero produce siempre un cociente y un resto. Por ejemplo, si se divide 35 por 12 dejaría cociente 2 y resto 11, ya que $35 = 2 \cdot 12 + 11$, esto es el algoritmo de la división y se asemeja al movimiento de las manecillas de un reloj, donde después de llegar a 12, se vuelve a comenzar desde 1.

Figura 6. Visualización en un reloj de la operación: 16 es congruentes con 4 ante la división por 12 [*Clock visualization of the operation: 16 is congruent to 4 when divided by 12*].

Si el reloj marca que son las 10 en punto, entonces con 6 horas más tarde serán las 4 en punto, no las 16 en punto. El número 16 no tiene significado en un reloj normal que muestra horas. El número en que se produce la envoltura (en este ejemplo, 12) se llama módulo. En este ejemplo se representa el resto 4 como el módulo de 16 y 12 (16 mod 12 = 4). De esta forma, dos números enteros son congruentes si dejan el mismo resto ante la misma división.

Definición 1 (Congruencia) *Sean $a, r, n \in \mathbb{Z}$ con $n > 0$. Se dice que, a es congruente a r módulo n si n divide a $a - r$.*

$$\textit{Notación: } a \equiv r \mod n.$$

¿Qué tienen en común los números 1, 13, 25, 37, ... ante la división entre 12? El resto es siempre $r = 1$. Al dividir números enteros arbitrarios entre 12, sólo son posibles los siguientes restos: $r = 0, 1, 2, \ldots, 11$. Los números enteros que dan como resultado el mismo resto r cuando se dividen por 12 se combinan para formar clases de restos r módulo 12. Dos números enteros que pertenecen a la misma clase de resto módulo 12 son congruentes módulo 12. Este tipo de operación es de vital importancia en *Circom*. La aritmética modular proporciona la infraestructura computacional para sistemas algebraicos que tienen ejemplos criptográficamente útiles de funciones unidireccionales.

En los zk-SNARK los números primos juegan un papel fundamental. Por definición, el número 1 no es un número primo. En lo adelante, p siempre denotará un número primo. La secuencia de números primos comienza con: $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, \ldots$ Los primeros 100 números incluyen exactamente 25 números primos. Después de esto, el porcentaje de números primos disminuye constantemente.

Definición 2 (Números primos) *Los números primos se pueden factorizar en una forma únicamente trivial: $p = 1 \cdot p$, es decir, un entero $p > 1$ es primo si sus únicos divisores son 1 y p.*

Como consecuencia cualquier número entero mayor que 1 y menor que cualquier número primo p no tiene divisores comunes con p. Todos los números que tienen 2 o más divisores distintos de 1 se llaman números compuestos. Éstas incluyen por ejemplo $4 = 2 \cdot 2$, $6 = 2 \cdot 3$ así como números que parecen primos, pero que en realidad son compuestos: $91 = 7 \cdot 13$, $161 = 7 \cdot 23$, $767 = 13 \cdot 59$. Todo entero positivo mayor que 1 puede ser escrito de forma única como producto de números primos. Por ejemplo $60 = 2 \cdot 2 \cdot 3 \cdot 5$.

En aritmética modular, existen dos aplicaciones o definiciones fundamentales para garantizar la seguridad y la eficiencia en este tipo de operaciones:

Definición 3 (Función de Euler) *La función de Euler, denotada como $\phi(n)$, cuenta la cantidad de números enteros positivos menores a n que son coprimos con n (es decir, no tienen factores primos comunes con n.*

Definición 4 (Pequeño Teorema de Fermat) *Si p es un número primo y a es un número entero coprimo con p, entonces: $a^{p-1} \equiv 1 \mod p$.*

Utilizamos los dos fundamentos anteriores para determinar inversos multiplicativos en *Circom*, estos se obtiene cuando un número se multiplica por otro y da como resultado 1. Son esenciales para resolver ecuaciones algebraicas y matrices. Operar con inversos multiplicativos nos evitamos las divisiones típicas como fracciones. Multiplicamos por el inverso de un número en el mismo conjunto formado por la clase de restos modulo un primo p y obtenemos el mismo resultado. Al utilizar un modulo primo garantizamos que todos los elementos de este conjunto tengan inversos multiplicativos, menos el 0. Cuando trabajamos con la suma y multiplicacion de números enteros en el módulo de un número primo cumplimos con todas las propiedades de un campo, una estructura algebraica en la cual operara los elementos del circuito.

2.2 Plantillas en *Circom*

Con *Circom*, se pueden diseñar nuestros circuitos aritméticos con nuestras propias restricciones y el compilador genera la representación R1CS que se necesita para construir una prueba. Algunos ejemplos de declaraciones que se pueden llevar a cabo con *Circom* y construir un R1CS para una prueba zkSNARK:

- Aplicación ⇒ Pruebo que conozco la preimagen de un hash.
- Aplicación ⇒ Piedra-Papel-Tijera: Pruebo que conozco una jugada secreta con respecto a cualquier jugada pública y gano el juego.
- Aplicación ⇒ Pruebo que conozco los valores secretos: (secret y nullifier) de un deposito en un árbol Merkle, así como su ruta.

A continuación se codifica en el lenguaje *Circom* el problema computacional que forma la siguiente declaración:

- Probar que conozco la solución de un problema sin mostrarla: esto sería la solución de una función multivariable que forma la siguiente ecuación: $f(x,y) = x^2y + y^2x - 12$. Manteniendo las entradas privadas x, y puedo demostrar que se cumple $f(x,y) \mapsto z$ para $z = 0$. Esto lo podemos hacer para cualquier función y cantidad de entradas.

Definimos un circuito aritmético que representa nuestra función: $f(x,y) \mapsto 0$. El circuito tiene 7 señales: las señales x y y son señales de entrada; $var1$, $var2$, $var3$ y $var4$ señales intermedia y la señal de salida del circuito, en este caso fijada en cero para probar que nuestras entradas son soluciones de nuestra función. Este proceso de expresar nuestro problema en un circuito es conocido como aplanamiento de código.

Para utilizar los protocolos zk-SNARK, necesitamos describir la relación entre señales como un sistema de ecuaciones que relacionan estas variables con puertas del circuito. Las puertas son las restricciones que se definen como una operación binaria de suma (puede recibir varias señales) o multiplicación (puede recibir solo 2 señales) y por esta razón las restricciones deben ser ecuaciones cuadráticas, lineales o constantes. Las ecuaciones que describen el circuito se llamarán restricciones, son condiciones que las señales de ese circuito deben satisfacer. Si se cumplen, demuestran que el cálculo se realizó correctamente. Inicialmente la figura 8 del circuito tiene cinco restricciones. Las operaciones dentro del circuito se realizan utilizando aritmética modular en un determinado número primo grande.

Código 2. Codificar el circuito poli_multi_var.circom

```
pragma circom 2.1.6;

template PoliMultiVar() {
        signal input x;
        signal input y;
        signal output out;
        signal var1 <== x * x;
        signal var2 <== var1 * y;
        signal var3 <== y * y;
        signal var4 <== var3 * x;
        out <== var4 + var2 - 12;
        0 === out;
}

component main {public [y]} =
    PoliMultiVar();
```

El código anterior especifica en la primera linea la versión del compilador *Circom* que se está utilizando (compruebe en consola con el comando **circom -version**). Creamos la plantilla PoliMultiVar() con la palabra reservada **template**, las plantillas en *Circom* funcionan igual que los componentes de un circuito eléctrico y pueden ser reutilizadas, lo veremos más adelante. El tipo de dato principal en *Circom* es **signal** y estas son inmutables (una vez que tienen un valor asignado, este valor ya no se puede cambiar) porque las entradas testigo en un R1CS tienen un valor fijo. Un vector de solución en un R1CS que cambia de valor no tiene sentido, ya que no se puede crear una prueba para ello.

Las señales de entradas siempre se consideran privadas (solo conocidas por el probador) y se declaran con la palabra reservada **signal input** a menos que lo especifiquemos en el **component main** como señal pública. Nuestra declaración como probador esta formada por dos señales de entradas, x privada y la señal y (conocida por cualquier verificador, visibles fuera del circuito). Utilizamos **input** y **out** para identificar señales de entradas y salidas respectivamente de lo contrario serán señales intermedias como aquí **var1**, **var2**, **var3**, **var4**. La señal de salida out por definición es pública.

El operador **<==** produce a la misma vez una asignación a señales en el lado izquierdo como muestra la linea 7 de 2 y aplica una restricción. El operador **===** solo aplica una restricción, pero no asigna valor. Todas las señales deben estar restringidas. Todas las restricciones deben ser a lo sumo cuadráticas de la forma $A * B + C = 0$, donde A, B y C son combinaciones lineales de señales, por lo que no podemos aplicar la restricción **signal var1 <== x * x * y**. Esto se debe

Figura 7. Flujo para generar y verificar una prueba zk-SNARK con *Circom* y snarkjs para la plantilla de circuito PoliMultiVar() [*Flow to generate and verify a zk-SNARK proof with Circom and snarkjs for the PoliMultiVar() circuit template*].

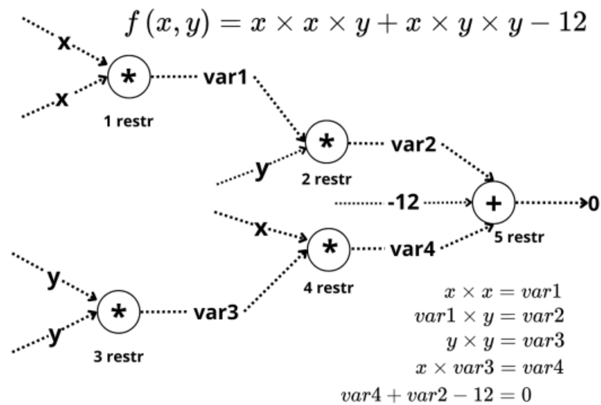

Figura 8. Circuito aritmético que verifica el conocimiento a soluciones de $f(x,y) = x^2y + y^2x - 12$ [*Arithmetic circuit that verifies knowledge of solutions of* $f(x,y) = x^2y + y^2x - 12$].

a que las puertas del circuito (restricciones) transforman las señales de entradas como elementos de una estructura algebraica conocida de campo, mediante las operaciones binaria de suma y multiplicación como se explicó anteriormente. Un R1CS válido debe tener exactamente una multiplicación por restricción. Para solucionar este inconveniente de error **signal var1 <== x * x * y** utilizamos la señal intermedia **var2**, en la linea 8 del Código 2.

Circom permite utilizar *arrays* y simplificar el código si es necesario. Se modifica el código, haciendo uso de otro tipo de datos que son las variables, y se realiza un ciclo **for**. Las variables no forman parte de R1CS y son mutables. Ahora la señal intermedia **var_aux** tiene almacenado los datos anteriores de **var3** y **var4**, en la posición uno y dos, respectivamente. Con este nuevo ejemplo se introducen algunas características sintácticas nuevas, manteniendo incluso las restricciones: argumentos y variables de plantilla.

Código 3. Insertando ciclo **for** con variables

```
1  pragma circom 2.1.6;
2
3  template PoliMultiVar(n) {
4         signal input x;
5         signal input y;
6         signal output out;
7         signal var_aux[n];
8         signal var1 <== x * x;
9         signal var2 <== var1 * y;
10
11        var_aux[0] <== x;
12        for (var i = 1; i < 3; i++) {
13            var_aux[i] <== var_aux[i - 1] *
                   y;
14        }
15        out <== var_aux[2] + var2 - 12;
16        0 === out;
17 }
18 component main {public [y]} =
19     PoliMultiVar(3);
```

Como comentábamos anteriormente podemos reutilizar y componer las plantillas como mismo cableamos un circuito eléctrico. Observe como por medio de la palabra reservada **component** instanciamos la plantilla **Potencias()**, para calcular aparte la multiplicación de un valor por si mismo. Es recomendable y convencional nombrar las señales de entradas como **in** y de salidas como **out**. Modificamos la restricción en la linea 8 de 3.

Código 4. Instanciando plantillas internas

```
1  pragma circom 2.1.6;
2
3  template Potencia() {
4         signal input in;
5         signal output out;
6
7         out <== in * in;
8     }
9
10    template PoliMultiVar(n) {
11        signal input x;
12        signal input y;
13        signal output out;
14        signal var_aux[n];
15
16        component var1 = Potencia();
17        var1.in <== x;
18        signal var2 <== var1.out * y;
19
20        var_aux[0] <== x;
21        for (var i = 1; i < 3; i++) {
22            var_aux[i] <== var_aux[i - 1] *
                   y;
23        }
24        out <== var_aux[2] + var2 - 12;
25        0 === out;
26    }
27    component main {public [y]} =
28        PoliMultiVar(3);
```

Para enriquecer nuestro circuito podemos incluir plantillas externas. Es frecuente utilizar el repositorio de plantillas de *Circom* llamado circomlib. Para agregarlo a cualquier proyecto ejecute **npm i circomlib**. Ahora se utiliza la plantilla **IsZero** que devuelve 1 si la señal de entrada es cero y 0 si la señal de entrada es distinta de cero.

Código 5. Plantilla IsZero() de circomlib

```
1  template IsZero() {
2         signal input in;
3         signal output out;
4         signal inv;
5         inv <-- in!=0 ? 1/in : 0;
6         out <== -in*inv +1;
7         in*out === 0;
8     }
```

En la plantilla IsZero(), **inv** es una señal intermedia que apoya, a que este circuito sea válido para una determinada

entrada. En la estructura algebraica de campo este es nuestro inverso multiplicativo para la entrada **in**, habíamos comentado de que todos los elementos que forman la estructura tienen inverso multiplicativos menos el 0, esta es la lógica que se lleva acabo en la linea 6 del Código 5. La expresión **1/in** no realiza una división de 1 y **in**, más bien es el elemento que cumple la función de inverso multiplicativo de **in**. El operador **<--** solamente asigna el valor a la señal pero no aplica restricción. En la linea 8 del Código 5 limitamos el resultado de **out**. Verá que solo es posible satisfacer las restricciones estableciendo que sea 1 cuando **in** sea 0 y **out** en 0 cuando **in** sea distinto de 0. La entrada **in**, debe seguir las reglas que imponen las restricciones. El circuito final se muestra como sigue.

Código 6. Instanciando plantillas externas

```
pragma circom 2.1.6;
   include "node_modules/circomlib/
       circuits/comparators.circom";

template Potencia() {
       signal input in;
       signal output out;

       out <== in * in;
   }

   template PoliMultiVar(n) {
       signal input x;
       signal input y;
       signal output out;
       signal var_aux[n];

       component var1 = Potencia();
       var1.in <== x;
       signal var2 <== var1.out * y;

       var_aux[0] <== x;
       for (var i = 1; i < 3; i++) {
           var_aux[i] <== var_aux[i - 1] *
                y;
       }
       out <== var_aux[2] + var2 - 12;

       component result_cero = IsZero();
       result_cero.in <== out;
       1 === result_cero.out;
   }
   component main {public [y]} =
       PoliMultiVar(3);
```

2.3 R1CS

Se ejecutan algunos *scripts* como partes de las librerías de *Circom* y snarkjs que nos permiten interactuar con el circuito diseñado.

Código 7. Interctuando con la librería circom y snarkjs

```
circom poli_multi_var.circom --inspect #
    inspeccionar codigo (errores,
    advertencias)
circom poli_multi_var.circom --r1cs --wasm
    --sym --json # compilar (sym - señales})
snarkjs r1cs info poli_multi_var.r1cs #
    info (curva, cantd. restricciones y
    entradas...)
snarkjs r1cs print poli_multi_var.r1cs
    poli_multi_var.sym # mostrar
    restricciones con señales
snarkjs r1cs export json poli_multi_var.
    r1cs poli_multi_var.r1cs.json # mejor
    lectura del r1cs
```

Cada restricción se representa como una terna de vectores (a,b,c) que satisface un vector solución w, tal que $(a\cdot w)*(b\cdot w)=(c\cdot w)$ donde $(\cdot)$ es el producto de vectores y $(*)$ producto de entradas, bajo aritmética modular del número primo $p=21888242871839275222246405745257275088548364400416034343698204186575808495617$ como define el archivo **poli_multi_var.r1cs.json**, obliguemos a los números a permanecer dentro del orden del campos. Esto se traduce finalmente en en un R1CS de la forma $(A\cdot w)*(B\cdot w)=(C\cdot w)$ donde A, B y C son matrices (conjunto de vectores) $n\times m$. Describen completamente el cálculo en el circuito aritmético, n igual al número de restricciones y m igual al número de señales que intervienen en el circuito más 1. Este elemento 1 se añade en la posición $w_0=1$, porque, de lo contrario, $w_0=0$ satisfaría todas las instancias de R1CS. En este ejemplo $n=6$ y $m=8$.

En el objeto *poli_multi_var.r1cs.json* la clave **map** establece las señales que generan el R1CS y de esta forma conocemos el valor de m. Un sistema de restricciones de rango 1 debe ser fijo e inmutable, esto significa que no podemos cambiar el número de filas o columnas una vez definido. No podemos cambiar los valores de las matrices, al no ser que modifiquemos el circuito.

Para observar mejor como quedan definidas las matrices A, B, C que forman las restricciones del archivos *poli_multi_var_constraints.json*, ejecute el script de Python matricesABC.py y pase como parámetro el valor de m (la cantidad de columnas de cada matriz o señales que intervienen en el circuito y forman el testigo w).

2.4 Testigos (Witness)

Para que un probador demuestre que conoce ciertos datos secretos, debe crear un archivo **input.json** que incluye esos valores. El siguiente archivo input.json con las señales de entradas x y y satisfacen el R1CS y genera un testigo válido para el circuito, pero el archivo input1.json no. Si evaluamos directamente el polinomio multivariables vemos que no se cumplen las restricciones para input1.json.

Código 8. Archivo input.json

```
{    "x": "3",
    "y": "1" }
```


Cabrera Bello, E.
https://doi.org/10.5281/zenodo.17110031

Código 9. Archivo input1.json

```
{    "x": "3",
   "y": "5" }
```

Al ejecutar los siguientes comandos podemos obtener los valores para algún testigo que satisface el circuito.

Código 10. Creando un testigo válido

```
cd poli_multi_var_js
nano input.json # agregar valores de las señales de entrada
node generate_witness.js poli_multi_var.wasm input.json witness.wtns # muestra log() si existen
snarkjs wtns export json witness.wtns witness.json # mejor lectura del testigo
```

Realicemos algunos cálculos interesantes en SageMath que nos muestran como el testigo:

$$w = [1,0,1,3,3,3,0,9],$$

generado con las señales de entradas válidas satisfacen nuestro R1CS como:

$$(A \cdot w) * (B \cdot w) = (C \cdot w). \tag{1}$$

A continuación se analiza la primera restricción, donde se parte de que $(a_1 \cdot w) * (b_1 \cdot w) = (c_1 \cdot w)$, donde:

$a_1 = [0,0,0,0,0,0,0,21888242871839275222246405745257275088548364400416034343698204186575808495616]$,

$b_1 = [0,0,1,0,0,0,0,0]$,

$c_1 = [21888242871839275222246405745257275088548364400416034343698204186575808495605, 21888242871839275222246405745257275088548364400416034343698204186575808495616, 0,0,0,1,0,0]$.

Note que a_1 representa el vector del lado izquierdo de la operación binaria (puerta del circuito), tenemos la constante que satisface la restricción al multiplicarla por el valor 7 que es la señal **var1.out** del vector w como muestra el archivo de símbolos **poli_multi_var.sym** y el **map**. En el lado derecho b_1 representa a la señal **y**. El resultado de la puerta, c_1 representa a las señales **out**, **var_aux[2]** y la constante que se representa como 1 en la posición cero del vector w.

$$(a_{11} \cdot w_1 + \dots + a_{18} \cdot w_8) * (b_{11} \cdot w_1 + \dots + b_{18} \cdot w_8) + (c_{11} \cdot w_1 + \dots + c_{18} \cdot w_8) = 0$$

a_{18} representa el elemento en la posición 7 del vector a_1 si se cuenta desde cero.

$$a_1 \cdot w * b_1 \cdot w = c_1 \cdot w$$
$$[a_1 \cdot (var1.out)] * [b_1 \cdot (y)] = [c_1 \cdot (out + var_aux[2] + 1)]$$

Si se comprueba el circuito en el Código 6, estos son los valores necesarios para calcular la restricción de la línea 25. Sustituimos **var2**, obteniendo **out <== var_aux[2] + (var1.out * y) - 12**. Las restricciones a lo sumo deben ser cuadráticas, entonces **out - var_aux[2] + 12 <== var1.out * y**. De aquí:

$$[a_1 \cdot (var1.out)] * [b_1 \cdot (y)] = [c_1 \cdot (out + var_aux[2] + 1)]$$

Sin pérdida de generalidad, se simplifica el producto escalar de dos vectores mostrando el resultado. Podemos realizar este cálculo aritmético con SageMath o también aquí. Se calcula la entrada izquierda a la puerta (restricción).

$$a_1 \cdot w = a_1 \cdot (var1.out)$$
$$a_1 \cdot w = 21888242871839275222246405745257275088548364400416034343698204186575808495616 \cdot 9$$
$$a_1 \cdot w = 21888242871839275222246405745257275088548364400416034343698204186575808495608$$

Se calcula la entrada derecha a la puerta.

$$b_1 \cdot w = b_1 \cdot (y) = 1 \cdot 1 = 1$$

Se calcula la salida de la puerta.

$$c_1 \cdot w = c_1 \cdot (out + var_aux[2] + 1)$$
$$c_1 \cdot w = 21888242871839275222246405745257275088548364400416034343698204186575808495605 \cdot 1 + 1 \cdot 3$$
$$c_1 \cdot w = 21888242871839275222246405745257275088548364400416034343698204186575808495608$$

Puede verse que el testigo w satisface esta restricción, así como todas las demás al chequearlo de la misma forma.

3. QAP

El R1CS representa nuestra prueba de conocimiento cero pero evaluarlo no es sucinto debido a las múltiples multiplicación de matrices. Un Programa de Aritmética Cuadrática (QAP) se define como un sistema de ecuaciones en el que los coeficientes son polinomios de una sola variable. Cuando se encuentra una solución válida para este sistema de ecuaciones, se obtiene una única igualdad polinómica. La característica “cuadrática” se refiere al hecho de que estos sistemas involucran exactamente una multiplicación polinómica. Los QAP desempeñan un papel fundamental en la concisión de los zk-SNARKs. Queremos evaluar los polinomios y luego comparar las evaluaciones.

Cuando un probador demuestra que ha realizado un cálculo, afirma que conoce un vector w para el R1CS, el verificador todavía tiene que hacer muchos cálculos multiplicando las matrices. Se necesita transformar un R1CS en una expresión polinómica única.

Antes de construir nuestro QAP, es importante comprender las estructuras algebraicas de Grupos, Anillos y Campos y sus propiedades. En resumen, se asocia los Grupos como el conjunto donde podemos realizar solamente una operación binaria con sus elementos como sumar o multiplicar. En los Anillos podemos realizar dos operaciones en el mismo conjunto pero no podemos dividir, es decir no tenemos inversos multiplicativos y por último en los Campos tenemos igual dos

operaciones binarias, además podemos dividir, todos los elementos tienen inversos multiplicativos menos el 0 (o elemento nulo).

Definición 5 (Grupo) *Un grupo $(G,*)$ es un conjunto G, junto con una operación binaria * en G tal que se satisface los siguientes tres axiomas:*

- *La operación binaria * es asociativa.*
- *Existe un elemento e en G tal que $e*x = x*e = x$ para todas las $x \in G$. Este elemento e se llama neutro (identidad) para * en G.*
- *Para cada elemento a en G existe un elemento a' en G con la propiedad de que $a'*a = a*a' = e$. El elemento a' es llama simétrico (inverso) de a respecto a *.*

*Si la operación binaria es conmutativa (a * b = b * a, para todas las $a,b \in G$.), se dice que el grupo es conmutativo o abeliano.*

Definición 6 (Anillo) *Un anillo $(A,+,\cdot)$ es un conjunto A junto con dos operaciones binarias + y ·, que se denomina suma y multiplicación, bien definidas en A tales que se satisfacen los siguientes axiomas:*

- *El grupo aditivo $(A,+)$ es un grupo abeliano. El elemento neutro lo denotaremos por 0.*
- *La multiplicación es asociativa, o sea $a\cdot(b\cdot c) = (a\cdot b)\cdot c$, para todas las $a,b,c \in A$.*
- *Para todos elementos $a,b,c \in A$, se cumple la ley distributiva izquierda $a\cdot(b+c) = (a\cdot b)+(a\cdot c)$ y la ley distributiva derecha $(a+b)\cdot c = (a\cdot c)+(b\cdot c)$.*

Si la multiplicación tiene un elemento identidad, se denota por 1: (anillo unitario). Si la multiplicación es conmutativa: (anillo conmutativo).

Definición 7 (Campo) *Si en un anillo conmutativo y unitario todos los elementos diferentes del 0 tienen simétricos para la operación multiplicación, diremos que $(A,+,\cdot)$ es un campo.*

3.1 Anillo de vectores

El anillo $V_n(\mathbb{R},+,\odot)$, de los vectores con elementos en $\mathbb{R}$ bajo la suma y producto Hadamard (no el producto de matrices) forman un anillo unitario. Por ejemplo, sea el anillo unitario $V_3(\mathbb{R},+,\odot)$ conjunto de vectores columna de 3 dimensiones. La suma de elementos produce otro vector de 3 dimensiones, es asociativa y conmutativa.

Conmutatividad:

$$\begin{bmatrix}-5\\2\\-1\end{bmatrix}+\begin{bmatrix}3\\1\\-3\end{bmatrix}=\begin{bmatrix}-2\\3\\-4\end{bmatrix}=\begin{bmatrix}3\\1\\-3\end{bmatrix}+\begin{bmatrix}-5\\2\\-1\end{bmatrix}$$

Asociatividad:

$$\left(\begin{bmatrix}-5\\2\\-1\end{bmatrix}+\begin{bmatrix}3\\1\\-3\end{bmatrix}\right)+\begin{bmatrix}3\\2\\0\end{bmatrix}=\begin{bmatrix}1\\5\\-4\end{bmatrix}$$

$$\begin{bmatrix}-5\\2\\-1\end{bmatrix}+\left(\begin{bmatrix}3\\1\\-3\end{bmatrix}+\begin{bmatrix}3\\2\\0\end{bmatrix}\right)=\begin{bmatrix}1\\5\\-4\end{bmatrix}$$

El elemento identidad del anillo para la adicción es un vector con todos sus elementos cero y los vectores inversos son justamente el signo opuesto.

$$\begin{bmatrix}-5\\2\\-1\end{bmatrix}+\begin{bmatrix}0\\0\\0\end{bmatrix}=\begin{bmatrix}-5\\2\\-1\end{bmatrix}$$

$$\begin{bmatrix}-5\\2\\-1\end{bmatrix}+\begin{bmatrix}5\\-2\\1\end{bmatrix}=\begin{bmatrix}0\\0\\0\end{bmatrix}$$

La multiplicación o producto de Hadamard, se denota con el operador $\odot$ para evitar confundirlo con el producto escalar. Tenga en cuenta que el producto escalar de dos vectores $a\cdot b = \sum_{i=0}^{n} a_i b_i$, al aplicar la suma se obtendrá un vector de un solo elemento por lo que no se cumpliría la propiedad de cerradura en la estructura. Ni tampoco podemos definirlo como el producto de matrices si se toman los vectores como matrices $n\times 1$ o $1\times n$ porque el número de columnas en la primera matriz es diferente al número de filas en la segunda matriz.

$$\begin{bmatrix}-5\\2\\-1\end{bmatrix}\odot\begin{bmatrix}-2\\-3\\-1\end{bmatrix}=\begin{bmatrix}10\\-6\\1\end{bmatrix}$$

$$\begin{bmatrix}-5\\2\\-1\end{bmatrix}\odot\begin{bmatrix}1\\1\\1\end{bmatrix}=\begin{bmatrix}-5\\2\\-1\end{bmatrix}$$

Si se comprueba la ley asociativa y distributivas para $\odot$, también se cumplen. Lo importarle a notar es que no tenemos inversos multiplicativos, razón por la cual nuestra estructura algebraica aquí es un anillo unitario y no un campo. No existen vectores diferentes de la unidad para $\odot$ que den el vector unidad.

3.2 Anillo de polinomios

Los polinomios con coeficientes en $\mathbb{R}$ bajo suma de polinomios y multiplicación de polinomios forman un anillo unitario. Por ejemplo, sea el anillo unitario $(P[x],+,\cdot)$, donde $P[x]$ es el conjunto de polinomios con coeficientes en $\mathbb{R}$. El grado de un polinomio está determinado por su máximo exponente de x, que en el caso de $x^3-6x^2+11x-6$ es 3, para ello se grafica su curva en SageMath.

Código 11. Funciones polinomiales en SageMath

```
poly = x^3 - 6*x^2 + 11*x - 6
plot(poly,(x,0,5))
```

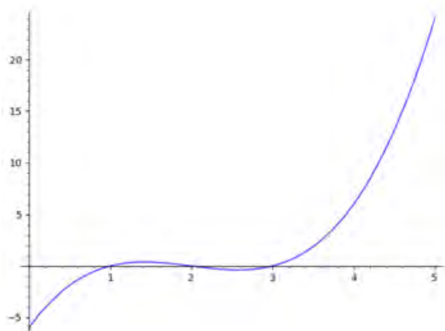

Figura 9. Gráfica de la función polinomial $f(x) = x^3 - 6x^2 + 11x - 6$ [*Graph of the polynomial function* $f(x) = x^3 - 6x^2 + 11x - 6$].

La suma de polinomios es nuevamente un polinomio (suma de coeficientes de la misma potencia) y obviamente es asociativa y abeliana. El polinomio identidad es el cero y los polinomios inversos son los signos opuestos de los coeficientes. Por ejemplo

$$(5x^2 + 3x - 4) + (-2x^2 + 7x + 4) = 3x^2 + 10x$$
$$(5x^2 + 3x - 4) + 0 = 5x^2 + 3x - 4$$
$$(5x^2 + 3x - 4) + (-5x^2 - 3x + 4) = 0$$

Podemos multiplicar polinomios, donde el grado del polinomio resultante puede ser mayor (la multiplicación de polinomios como una operación distributiva de sus coeficientes). El elemento identidad de los polinomios bajo multiplicación es 1 (1 es un polinomio de grado cero). No existe en este anillo de polinomios inversos multiplicativos, de hecho cada vez que se multiplican polinomios de grados mayor que uno, no podemos reducir el grado a cero y obtener el polinomio 1.

$$(x^2 + 3x + 1) \cdot (2x - 5) = 2x^3 + x^2 - 13x - 5$$

3.3 Multiplicación escalar de vectores y de polinomios.

La multiplicación escalar no es una tercera operación binaria. Podemos verla como una abreviatura de las operaciones binarias de la estructura algebraica.

$$\begin{bmatrix} -5 \\ 2 \\ -1 \end{bmatrix} \cdot 3 = \begin{bmatrix} -5 \\ 2 \\ -1 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -15 \\ 6 \\ -3 \end{bmatrix}$$

$$\begin{bmatrix} -5 \\ 2 \\ -1 \end{bmatrix} + \begin{bmatrix} -5 \\ 2 \\ -1 \end{bmatrix} + \begin{bmatrix} -5 \\ 2 \\ -1 \end{bmatrix} = \begin{bmatrix} -15 \\ 6 \\ -3 \end{bmatrix}$$

Para polinomios, multiplicar por un escalar es lo mismo que multiplicar por un polinomio de grado cero donde el coeficiente tiene el mismo valor que el escalar.

$$(x^2 + 3x + 1) \cdot 2 = (x^2 + 3x + 1) + (x^2 + 3x + 1) = 2x^2 + 6x + 2$$

3.4 Homomorfismo

Un homomorfismo es una función que preserva las propiedades de dos estructuras algebraicas.

Definición 8 (Homomorfismo de grupos) *Sean $(G, *)$, $(H, \triangle)$ dos grupos como estructura algebraica donde $x, y \in G$, entonces existe una función $f : G \to H$, tal que:*

$$f(x * y) = f(x) \triangle f(y)$$

La misma definición 8 es aplicable para estructuras de anillos y campos. Pasar de la multiplicación de vectores a polinomios es sencillo cuando el problema se plantea como un homomorfismo entre anillos algebraicos.

Definición 9 (Interpolación de Lagrange) *Dado un conjunto de puntos rectangulares*

$$S = (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), x_i \neq x_j, \forall\, i, j,$$

son suficientes para determinar el conjunto de pares $(x, f(x))$, que proporcionan un polinomio $f(x)$ de grado n con $f(x_i) = y_i$ par todos los pares (x_i, y_i) de S. Este polinomio se puede construir mediante la fórmula:

$$f(x) = \sum_{i=0}^{n} y_i L_i(x), \tag{2}$$

$$L_i(x) = \prod_{i=0}^{n} \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \forall i, j = \{0, 1, ..., n\}. \tag{3}$$

Existe un homomorfismo de anillo desde vectores columnas de dimensión n con elementos en $\mathbb{R}$ hasta polinomios con coeficientes en $\mathbb{R}$. Con la siguiente función φ homomórfica, resultado de la interpolación de Lagrange en puntos arbitrarios definidos por el probador pero conocidos por el verificador.

$$\varphi(A) = \{A_1(x), A_2(x), ..., A_m(x)\},\ A \in F_p^{n \times m}$$
$$\varphi(B) = \{B_1(x), B_2(x), ..., B_m(x)\},\ B \in F_p^{n \times m}$$
$$\varphi(C) = \{C_1(x), C_2(x), ..., C_m(x)\},\ C \in F_p^{n \times m}$$

Aquí $A_i(x)$, $B_i(x)$, $C_i(x)$, son polinomios que interpolan los vectores columnas de las Matrices A, B y C respectivamente. Dado que la multiplicación de un vectores columnas por escalares es homomórfica a la multiplicación de polinomios por escalares, cada polinomio puede escalarse mediante el elemento correspondiente del vector testigo w. Por ejemplo:

$$\varphi(A)\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

$$= \begin{bmatrix} A_1(x) & A_2(x) & \dots & A_m(x) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

$$= w_1A_1(x) + w_2A_2(x) + \dots + w_mA_m(x)$$

$$= \sum_{i=1}^{m} w_iA_i(x)$$

Note que este resultado final es un polinomio de a lo sumo grado $n-1$, ya que A tiene n filas.

3.5 Igualdad y Evaluación de Polinomios

Anteriormente hemos comentado que los polinomios son funciones. Por ejemplo dado el siguiente polinomio $f(x) = x^3 - 6x^2 + 11x - 9$, note que su grado es 3. Los polinomios

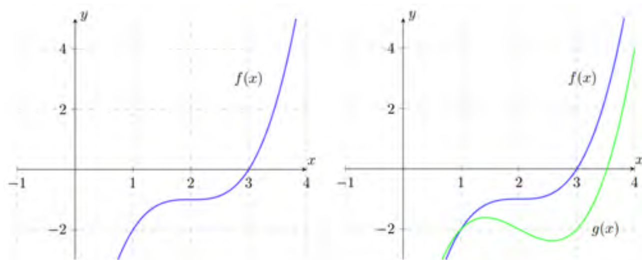

Figura 10. Funciones polinomiales $f(x) = x^3 - 6x^2 + 12x - 9$ y $g(x) = x^3 - 6x^2 + 11x - 8$ [*Polynomial functions* $f(x) = x^3 - 6x^2 + 12x - 9$ *and* $g(x) = x^3 - 6x^2 + 11x - 8$].

tienen una propiedad ventajosa, que puede notarse si se modifica el polinomio original $f(x)$ ligeramente a $g(x) = x^3 - 6x^2 + 11x - 8$ y se visualiza en verde. Se tienen ahora dos polinomios no iguales de grado como máximo 3. Note como en la figura 10 estos dos polinomios no pueden cruzarse en más de 3 puntos.

Un cambio tan pequeño produce resultados drásticamente diferente. Es imposible encontrar dos polinomios no iguales que comparten un trozo consecutivo de una curva. Si se quieren encontrar intersecciones de dos polinomios, es necesario igualarlos. Por ejemplo, para encontrar dónde se cruza un polinomio en el eje x (es decir, $f(x) = 0$), se igualan $x^3 - 6x^2 + 12x - 9 = 0$, y las soluciones de dicha ecuación serán esos puntos compartidos. Por ejemplo, si se encuentra una solución cuando $x = 3$. Esto puede observarse en la figura 10, donde la curva azul cruza la línea del eje x. Asimismo, es posible igualar nuestra versión original y modificada de polinomios para encontrar sus intersecciones.

$$x^3 - 6x^2 + 12x - 9 = x^3 - 6x^2 + \mathbf{11}x - \mathbf{8}$$
$$x - 1 = 0$$

El polinomio resultante es de grado 1 con una solución obvia cuando $x = 1$. Por lo tanto, sólo una intersección.

Definición 10 *Dados dos polinomios cualesquiera $f(x)$ y $g(x)$ de grados arbitrarios n, el hecho de igualar ambos polinomios es una ecuación que resulta en un nuevo polinomio $h(x)$ de grado como máximo n.*

Dado cualquier número o elemento perteneciente al mismo conjunto de los elementos de los coeficientes de un polinomio $f(x)$. El polinomio lo podemos evaluar en ese elemento. Lo que significa que sustituyamos el elemento dado para cada aparición de la variable indeterminada x en la expresión algebraica polinómica.

Lema 11 (Schwartz-Zippel) *Sea F un campo, $f(x_1, x_2, ..., x_n)$ un polinomio de grado máximo d diferente del polinomio nulo. Sea S un subconjunto finito de F. Sean $r_1, r_2, ..., r_n$ elegidos al azar de manera uniforme e independiente desde S. Entonces la probabilidad de que $f(r_1, r_2, ..., r_n) = 0$, es $\leq \frac{d}{|S|}$.*

El lema 11 destaca que diferentes polinomios son diferentes en la mayoría de los puntos. Dankrad Feist (2023) sostiene que: "Una conclusión concisa de Schwartz-Zippel es: en campos grandes, dos polinomios de "bajo grado" son idénticos o diferentes casi en todas parte".

Siempre existe un polinomio $f(x)$ de grado no mayor que n, que tome unos valores prefijados para $n+1$ distintos valores dados de la indeterminada.

3.6 R1CS a QAP

Transformemos nuestro R1CS a QAP con las ideas formalizadas anteriormente y apoyándonos en SageMath (este procedimiento se encarga de hacerlo la librería snarkjs). Aplicamos la función homomorfica descrita en el apartado 3.4.

Código 12. Instanciando vectores en SageMath

```
p = 218882428718392752222464057452572750885
    48364400416034343698204186575808495617
Fp = GF(p)

A = Matrix(Fp,
[[0, 0, 0, 0, 0, 0, 0, 2188824287183927522
    22464057452572750885483644004160343436
    98204186575808495616],
[0, 0, 0, 21888242871839275222246405745257
    275088548364400416034343698204186575808
    495616, 0, 0, 0, 0],
[0, 0, 0, 0, 21888242871839275222246405745
    257275088548364400416034343698204186575
    808495616, 0, 0, 0],
[0, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 21888242871839275222246405745257
    275088548364400416034343698204186575808
    495616, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0]]
)

```

```
B = Matrix(Fp,
[[0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0]]

C = Matrix(Fp,
[[21888242871839275222246405745257275088548
     364400416034343698204186575808495605,
    21888242871839275222246405745257275088854
     8364400416034343698204186575808495616,
    0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 21888242871839275222246405745
    257275088548364400416034343698204186575
    808495616, 0, 0, 0],
[0, 0, 0, 0, 0,
    21888242871839275222246405745
    257275088548364400416034343698204186575
    808495616, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0,
    21888242871839275222246405745
    257275088548364400416034343698204186575
    808495616],
[0, 21888242871839275222246405745
    257275088548364400416034343698204186575
    808495616, 0, 0, 0, 0, 0, 0]]
)

w = vector(Fp,[1, 0, 1, 3, 3, 3, 0, 9])
```

El vector solución (testigo) contiene 8 elementos, por lo que podemos construir 8 polinomios. Cada restricción aporta 1 punto a cada polinomio. Tenemos 6 restricciones, obtenemos 6 puntos por polinomio. Por Interpolación de Lagrange 6 puntos nos permiten definir un polinomio de grado máximo 5. Cada columna de A, B y C tiene 6 elementos. Entonces, cada una de estas columnas se puede convertir en un polinomio de grado 5. Por ejemplo, la quinta columna de C es:

$$[1, 0, 21888242871839275222246405745257275088548\,364400416034343698204186575808495616, 0, 0, 0]$$

Podemos considerar cada elemento como la coordenada y correspondiente a $x \in [1,2,3,4,5,6]$. Entonces obtenemos 4 conjuntos de puntos:

(1, 1), (2, 0), (3, 21888242871839275222246405745 257275088548364400416034343698204186575808495616), (4, 0), (5, 0), (6, 0).

Esta es una interpretación arbitraria que utilizamos para convertir el R1CS al formato QAP. Podemos encontrar el polinomio que pasa por estos 6 puntos.

Código 13. Interpolación de Lagrange en SageMath

```
Rp.<x> = PolynomialRing(Fp)
points = [(1, 1), (2, 0), (3, 218882428718
    392752222464057452572750885483644004160
    34343698204186575808495616), (4, 0), (5,
     0), (6, 0)]
polinomio = Rp.lagrange_polynomial(points)
polinomio
```

De esta forma obtenemos el polinomio correspondiente a la quinta columna de C:

Código 14. Obtener todos los polinomios en SageMath

```
 M = [A, B, C]
    PolyM = []

    for m in M:
        PolyList = []
        for i in range(m.ncols()):
            points = []
            for j in range(m.nrows()):
                points.append([j+1,m[j,i]])

            Poly = Rp.lagrange_polynomial(
                points).coefficients(sparse=
                False)

            if(len(Poly) < m.nrows()):
                # si el grado del polinomio
                     es menor que 6
                # agregamos ceros para
                    representar los términos
                     omitidos
                dif = m.nrows() - len(Poly)
                for c in range(dif):
                    Poly.append(0);

            PolyList.append(Poly)
        PolyM.append(Matrix(Fp, PolyList))
```

Código 15. Mostrar los polinomios del QAP en SageMath

```
Ax = Rp(list(w*PolyM[0]))
Bx = Rp(list(w*PolyM[1]))
Cx = Rp(list(w*PolyM[2]))
print("A(x)␣=␣" + str(Ax))
print("B(x)␣=␣" + str(Bx))
print("C(x)␣=␣" + str(Cx))
Tx = Ax*Bx - Cx
print("T(x)␣=␣" + str(Cx))
```

$$R1CS \Rightarrow A \cdot w * B \cdot w = C \cdot w$$
$$QAP \Rightarrow T(x) = A(x) * B(x) - C(x)$$

El verificador no conoce el polinomio $T(x)$, ni puede calcularlo ya que no conoce el vector solución w. Entonces, el probador tiene que demostrarle al verificador que $T(x) = 0$ para $x \in 1,\ldots,6$. Evaluamos $T(x)$ en $x = 1,2,3,4,5,6$.

Código 16. Evaluar polinomio T(x) en SageMath

```
 print("T(0)␣=␣" + str(Tx(0)))
```

```
2 print("T(1)␣=␣" + str(Tx(1)))
3 print("T(2)␣=␣" + str(Tx(2)))
4 print("T(3)␣=␣" + str(Tx(3)))
5 print("T(4)␣=␣" + str(Tx(4)))
6 print("T(5)␣=␣" + str(Tx(5)))
7 print("T(6)␣=␣" + str(Tx(6)))
8 print("T(7)␣=␣" + str(Tx(7)))
```

Se observa que al evaluar cualquier otro punto como 0 y 7, no es una solución para este polinomio $T(x)$. Como es de esperar por las propiedades de los polinomios, un polinomio de grado d puede tener como máximo d soluciones y por tanto como máximo d puntos compartidos. Si evaluamos el polinomio en varios puntos, conocemos de ante mano en que valores el polinomio se anula es decir cuales son sus raíces.

Eso también significa que existe un polinomio $H(x)$, tal que: $T(x) = H(x) \cdot Z(x)$ donde $Z(x) = (x-1)(x-2)...(x-5)$. En otras palabras, la división $T(x)/Z(x)$ no tiene resto y el resultado $H(x)$ es un polinomio $H(x) = T(x)/Z(x)$. Entonces creamos un nuevo polinomio $Z(x)$ conocido tanto por el Probador como por el Verificador. El Teorema Fundamental del Álgebra establece que cualquier polinomio se puede factorizar en polinomios lineales.

$$Z(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6)$$

Código 17. Chequear el resto de la división polinomial en SageMath

```
1 Zx = Rp((x - 1)*(x - 2)*(x - 3)*(x - 4)*(x
      - 5)*(x - 6))
2 Hx = Tx.quo_rem(Zx)
3 print("Cociente␣de␣Tx/Zx␣=␣", end="")
4 print(Hx[0])
5 print("Resto␣de␣Tx/Zx␣=␣", end="")
6 print(Hx[1])
```

En principio el verificador tiene que chequear lo siguiente:

$$\frac{E(A(\tau)) \cdot E(B(\tau)) - E(C(\tau))}{E(Z(\tau))} = E(H(\tau)),$$

donde E cifra homomorficamente a los polinomios mediante compromisos polinomiales.

3.7 Compromisos Polinomiales

Los polinomios son el núcleo de zk-SNARK aunque es probable que también existan otros medios de pruebas. [12]

Si un probador (Peggy) afirma conocer algún polinomio (sin importar cuán grande sea su grado), que el verificador (Victor) también lo sabe, puede seguir un protocolo simple para verificar la declaración: como se muestra en la figura 11. Si, por ejemplo, consideramos un rango entero de x de 1 a 10^{50}, el número de puntos donde las evaluaciones son diferentes, serian, $10^{50} - g$, donde g es el grado del polinomio. De ahí en adelante, la probabilidad de que x "golpe" accidentalmente cualquiera de los g puntos compartidos es igual, $\frac{g}{10^{50}}$, lo cual se considera insignificante. Note que este simple protocolo es interactivo como lo son en principio todos los protocolos de zk-SNARK y requiere solo una ronda.

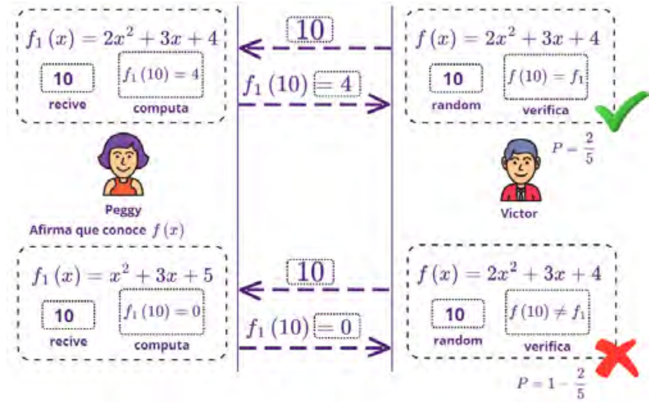

Figura 11. Protocolo sobre el conocimiento de un polinomio con evaluaciones mód 5 [*Protocol on knowledge of a polynomial with evaluations* mód 5].

Tenemos algunos inconvenientes con esta idea de prueba, en la que las partes tienen que confiar entre sí porque todavía no hay medidas para hacer cumplir las reglas del protocolo. Por ejemplo, no se requiere que el probador conozca un polinomio, y puede utilizar cualquier otro medio disponible para llegar a un resultado correcto. Además, si la amplitud de las evaluaciones polinómicas del verificador no es grande, digamos 10, el verificador puede adivinar un número, y existe una probabilidad no despreciable de que sea aceptado.

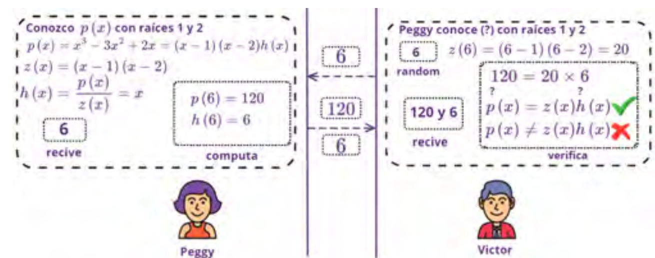

Figura 12. Protocolo sobre el conocimiento de un polinomio con propiedades de divisibilidad [*Protocol on the knowledge of a polynomial with divisibility properties*].

Podemos comprobar las propiedades de divisibilidad específicas de un polinomio sin aprender el polinomio en sí, por lo que esto ya nos da cierta forma de conocimiento cero y concisión. No obstante pueden haber ciertos problemas con este enfoque:

- Puede ser posible que el probador no conozca en absoluto el polinomio $p(x)$ en cuestión. Puede calcular la evaluación $z = z(x)$, seleccionar un número aleatorio $h = h(x)$ y establecer $p = h \cdot z$, que será aceptado por el verificador como válido, ya que la ecuación se cumple.
- Como el demostrador conoce el punto aleatorio $x = r$, puede construir cualquier polinomio que tenga un punto compartido en r con $h(r) \cdot z(r)$.

- En la declaración del protocolo simple en la figura 11, el probador afirma conocer un polinomio de un grado particular? en el protocolo en la figura 12 no se exige ningún grado. Por lo tanto, el probador puede hacer trampa utilizando un polinomio de grado superior que también satisfaga la verificación de cofactores.

Los inconvenientes anteriores se dan porque los valores se presentan como textos claros. Sería ideal si esos valores se dieran de manera cifrada, de modo que no se pueda alterar el protocolo, pero aún así se puedan calcular operaciones sobre esos valores cifrados. Esto lo podemos garantizar empleando cifrados homomórficos, que nos permita cifrar un valor y poder aplicar operaciones aritméticas a dicho cifrado. Existen múltiples formas de conseguir propiedades homomórficas del cifrado a través de funciones homomorficas. Este es un mecanismo bastante poderoso. Podemos tener evaluaciones en un polinomio cifrado en un valor x desconocido para nosotros.

$$\begin{aligned} E(p) &= E(x^3)^1 \cdot E(x^2)^{-3} \cdot E(x)^2 \\ &= \left(g^{x^3}\right)^1 \cdot \left(g^{x^2}\right)^{-3} \cdot (g^x)^2 \\ &= g^{x^3} \cdot g^{-3x^2} \cdot g^{2x} \\ &= g^{x^3-3x^2+2x} \end{aligned}$$

$$\boxed{g^p = \left(g^h\right)^z(s) = g^{z(s)\cdot h}}$$

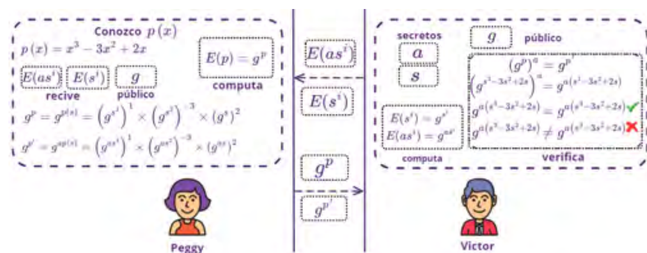

Figura 13. Esquema interactivo de conocimiento cero sobre polinomios [*Interactive zero-knowledge scheme on polynomials*].

Si bien en dicho protocolo la agilidad del probador es limitada, aún puede usar cualquier otro medio para falsificar una prueba sin usar realmente los cifrados provistos de potencias de s. Es decir, el verificador necesita la prueba de que solo se utilizaron los cifrados suministradas de las potencias de s para calcular g^p y g^h y nada más. Una forma de hacer esto es requerir que se realice la misma operación en otro valor cifrado desplazado junto con el original, actuando como un análogo aritmético de "suma de verificación", asegurando que el resultado sea la exponenciación del valor original.

4. Sistema de pruebas (*Groth16*)

En *Groth16*, operamos en grupos de curvas elípticas sobre campos finitos $\mathbb{F}_p$. Los zk-SNARK basados en emparejamiento siguen un paradigma simple donde el probador calcula una cantidad de elementos de grupo utilizando operaciones de grupo genéricas y el verificador verifica la prueba utilizando una cantidad de ecuaciones de producto de emparejamiento.

Definición 12 (Emparejamientos bilineales) *Definimos grupos bilineales (p, G_1, G_2, G_T) tales que G_1, G_2, G_T son grupos de orden primo p. El emparejamiento*

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$$

es un mapeo bilineal. Dado $G_1 \in \mathbb{G}_1, G_2 \in \mathbb{G}_2, G_T \in \mathbb{G}_T$, existe una función determinista $e(G_1, G_2) = G_T$. G_1 es un generador para $\mathbb{G}_1$, G2 es un generador para $\mathbb{G}_2$ y $e(G1, G2)$ es un generador para $\mathbb{G}_T$. [5]

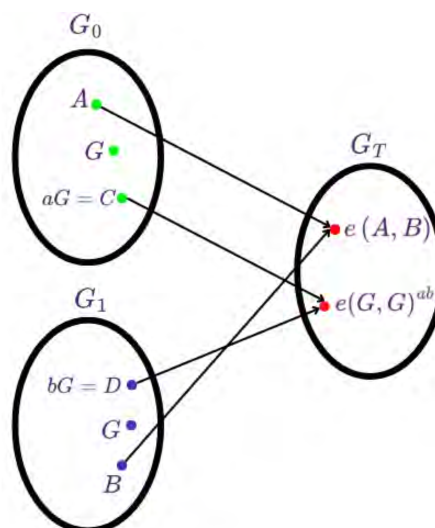

Figura 14. Emparejamiento bilineal simétrico, $\mathbb{G}_0 = \mathbb{G}_1$, $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$ [*Symmetric bilinear pairing* $\mathbb{G}_0 = \mathbb{G}_1$, $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_T$].

Proposición 13 (Bilinealidad) *Dados $A, B \in \mathbb{G}_1$ y $B, D \in G_2$, se cumple que:*

$$\begin{aligned} e(A+C, B) &= e(A, B) \cdot e(C, B) \\ e(A, B+D) &= e(A, B) \cdot e(A, D) \end{aligned}$$

Proposición 14 (Bilinealidad) *Sean n y m enteros escalares*

entonces:

$$
\begin{aligned}
e(nA, mB) &= e((n-1)A + A, mB) \\
&= e((n-1)A, mB) + e(A, mB) \\
&= \ldots = \\
&= e(A, mB)^n \\
&= e(A, (m-1)B + B)^n \\
&= \ldots = \\
&= [e(A, (m-1)B) \cdot e(A, B)]^n \\
&= \ldots = \\
&= e(A, B)^{nm} \\
&= \ldots = \\
&= e(A, nmB) \\
&= \ldots = \\
&= e(nmA, B)
\end{aligned}
$$

$$\boxed{e(nA, mB) = e(nmA, B) = e(A, nmB)}$$

Dado que los grupos de curvas elípticas tienen propiedades homomorficamente aditivas, podemos por ejemplo utilizar los emparejamientos en ciertos grupos para verificar cálculos.

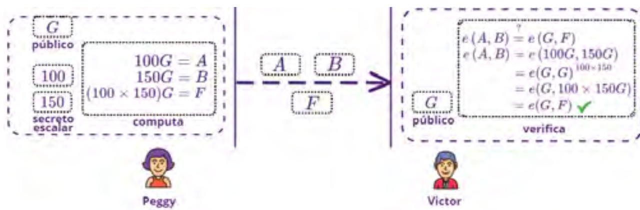

Figura 15. Emparejamiento para verificar cálculos en grupos de curvas elípticas [*Pairing to verify calculations on groups of elliptic curves*].

Groth16 [11] usa notación donde los elementos del grupo están representados por sus logaritmos discretos: $G_1 \cdot a$ se representa como $[a]1$, $G_2 \cdot b$ como $[b]_2$ y $e([a]_1, [b]_2)_c$ como $[c]_T$. El grupo $\mathbb{G}_T$ es distinto tanto de $\mathbb{G}_1$ como de $\mathbb{G}_2$ y, por lo tanto, $[c]_T$ no puede ser una entrada al mapa bilineal e.

4.1 Configuración de confianza (Trusted Setup)

Eliminar múltiples rondas de comunicación no solo mejora el rendimiento del protocolo, sino que también elimina el requisito de que ambas partes estén disponibles simultáneamente. Para lograr esta reducción en las rondas, se utilizan comúnmente dos enfoques: el modelo de oráculo aleatorio que define la heuristica Fiat-Shamir [4] y la cadena de referencia común (CRS). Groth16 utiliza una configuración confiable de dos pasos para generar una cadena de referencia común (CRS) como se muestra en la figura 16. La primera fase es genérica donde se crea el CRS mediante un calculo de múltiples partes (MPC) y la segunda es específica del circuito. En teoría, un tercero honesto y confiable es responsable de generar un valor aleatorio “r” y usarlo para crear parámetros CRS, incluida la clave de prueba “pk” y la clave de verificación “vk”. Es importante tener en cuenta que cualquiera que tenga acceso a la aleatoriedad “r” puede generar potencialmente una prueba fraudulenta. Para mejorar la seguridad del protocolo zk-SNARK, a menudo se aprovecha el cómputo entre varias partes, donde cada una de estas calculan de manera colaborativa una función sin la necesidad de una autoridad de terceros de confianza. Siempre que haya al menos una parte honesta, los parámetros finales son seguros. [6]

En zk-SNARKs, la configuración confiable es un protocolo crítico que genera parámetros criptográficos iniciales (como la Structured Reference String, SRS) necesarios para construir pruebas. Su objetivo es permitir la evaluación de polinomios en un punto secreto τ sin revelar dicho valor. La CRS tiene diferencias sutiles ante SRS que se describen en [3].

1. Supongamos un polinomio $p(x) = a_2x^2 + a_1x + a_0$
2. En lugar de evaluarlo directamente en τ (lo que expondría el secreto), usamos una CRS precomputada:
$$\mathrm{CRS} = (G, \tau G, \tau^2 G, \tau^3 G),$$
donde G es un generador de una curva elíptica.
3. Ahora, $p(\tau) \cdot G$ se calcula como:
$$a_2(\tau^2 G) + a_1(\tau G) + a_0(G),$$
sin conocer τ.

La seguridad del sistema depende de que τ sea eliminado después de generar la CRS. Si un atacante lo recupera, podría:

- Falsificar pruebas: Generar pruebas falsas que parezcan válidas.

Zcash, la ceremonia inicial de trusted setup requirió que múltiples participantes destruyeran sus claves parciales. Aunque se auditaron, la necesidad de confianza sigue siendo una debilidad teórica.

Una de las alternativas sin configuración confiable que se presentan es Halo 2, una actualización de Halo [7] que hace uso del protocolo PLONK [9]. Aprovecha recursión para evitar secretos iniciales. Halo2 se utiliza en producción para asegurar transacciones protegidas en la blockchain de Zcash. En la tabla 1 enmarcamos algunos protocolos que difieren ante la configuración confiable. Más recientemente se han dado a conocer protocolos modernos como la variante Hyperplonk [8] que ofrece pruebas con una eficiencia mejorada, mientras que zk-STARK (Zero-Knowledge Scalable Transparent ARguments of Knowledge) es otro tipo de prueba que presenta características distintas en términos de escalabilidad. SNARK utiliza criptografía de curva elíptica, vulnerable a ataques cuánticos. STARK, por otro lado, utiliza funciones hash criptográficas resistentes a ataques cuánticos.

Como ilustramos en la figura 17 para construir CRS, generamos la tupla de los elementos de campo aleatorios

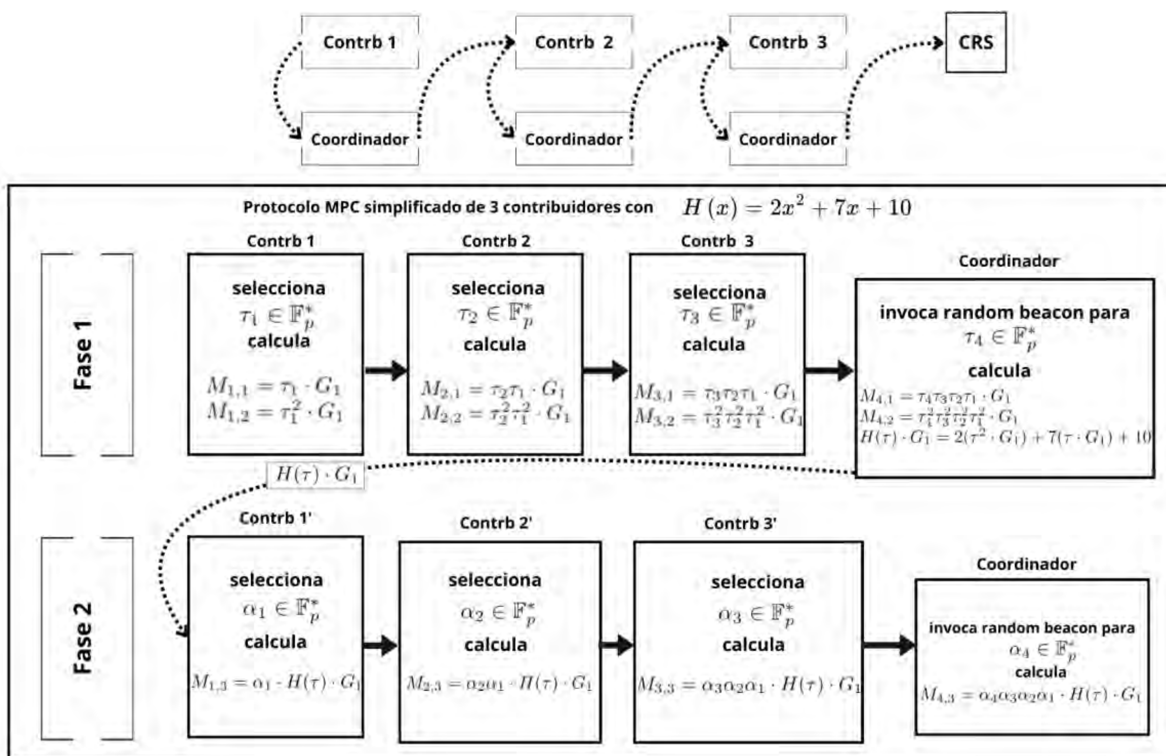

Figura 16. Configuración confiable para Groth16 [*Trusted setup for Groth16*].

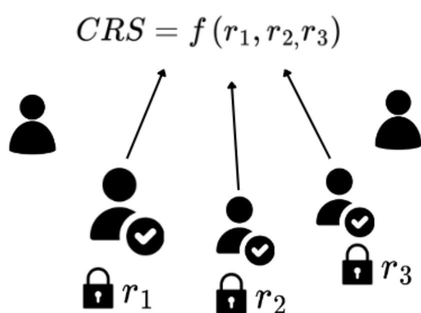

Figura 17. CRS como un calculo seguro entre múltiples partes [*CRS as a secure multi-party calculation*].

$(\alpha, \beta, \gamma, \delta, \tau) \in F_p$, construida mediante un calculo MPC, conocidos también como residuos tóxicos. Los elementos α, β se utilizan para garantizar que A, B, C sean consistentes entre sí cuando se usa el vector w. El producto $A \cdot B$ tiene una dependencia lineal de α, β y sólo se equilibra con C cuando w es consistente en los tres A, B, C. El papel de los otros elementos del campo, γ y δ se utilizan para hacer la entrada pública independiente de los otros testigos.

- Potencias de Tau: $[\tau]_1 = \tau \cdot G_1, \mathbb{G}_1 =< G_1 >$ (inviable logaritmo discreto)

$$([\tau^0]_1, [\tau^1]_1, [\tau^2]_1, ..., [\tau^{n-1}]_1)$$
$$([\tau^0]_2, [\tau^1]2, [\tau^2]_2, ..., [\tau^{n-1}]_2)$$
$$[\alpha]_1 \cdot ([\tau^0]_1, [\tau^1]1, [\tau^2]_1, ..., [\tau^{n-1}]_1)$$
$$[\beta]_1 \cdot ([\tau^0]_1, [\tau^1]1, [\tau^2]_1, ..., [\tau^{n-1}]_1)$$

Tabla 1. Comparación de protocolos zk-SNARKs: Trusted Setup

Protocolo	Trusted Setup	Inconvenientes
Groth16	Requerido (por circuito)	Riesgo de fuga de τ
PLONK	Universal (una vez)	Dependencia inicial
Halo 2	No requerido	Mayor costo comput.

En el ejemplo dado en la figura 16, el CRS es $(\tau \cdot G_1, \tau^2 \cdot G_1, \alpha H(\tau) \cdot G_1)$ dónde $\tau = \tau^3\tau^2\tau^1$ y $\alpha = \alpha^3\alpha^2\alpha^1$ son múltiplos escalares que se han calculado acumulativamente de forma serial. Con cada contribución sucesiva, la aleatoriedad independiente elegida por un contribuyente se acumula en la transcripción.

De esta forma podemos obtener evaluaciones para cada polinomio de la siguiente forma:

$$[A]_1 = \sum_{i=0}^{m} w_i [A(\tau)]_1 = \sum_{i=0}^{m} w_i \sum_{j=0}^{n} A_{i,j} [(\tau^j G_1)]_1 \tag{4}$$

$$[B]_2 = \sum_{i=0}^{m} w_i [B(\tau)]_2 = \sum_{i=0}^{m} w_i \sum_{j=0}^{n} B_{i,j} [(\tau^j G_2)]_2 \tag{5}$$

$$[C]_1 = \sum_{i=0}^{m} w_i [C(\tau)]_1 + [H(\tau)Z(\tau)]_1 =$$

$$= \sum_{i=0}^{m} w_i \sum_{j=0}^{n} C_{i,j}[(\tau^j G_1)]_1 + \sum_{i=0}^{gr(H(x))} c_i[\tau^j Z(\tau) G_1]_1 \quad (6)$$

Para este resultado el verificador inicialmente calcularía

$$e([A]_1,[B]_2) = e([\alpha]_1[\beta]_2) + e([C]_1,[G_2]_2) \quad (7)$$

que es equivalente a calcular solamente $\alpha\beta$ en la siguiente expresión con los polinomios ya cifrados.

$$(\alpha + [A]_1)(\beta + [B]_2) =$$

$$= \alpha\beta + \sum_{i=0}^{m} w_i(\beta A_i(\tau)) + \sum_{i=0}^{m} w_i(\alpha B_i(\tau)) + [C]_1 \quad (8)$$

Código 18. Ejecutar con la librería snarkjs una configuración confiable

```
snarkjs powersoftau new bn128 14
    pot14_0000.ptau -v
snarkjs powersoftau contribute pot14_0000.
    ptau pot14_0001.ptau --name="First␣
    contribution" -v
snarkjs powersoftau contribute pot14_0001.
    ptau pot14_0002.ptau --name="Second␣
    contribution" -v -e="some"
snarkjs powersoftau verify pot14_0002.ptau
     #verificar el protocolo MPC
snarkjs powersoftau beacon pot14_0002.ptau
     pot14_beacon.ptau 0102030405060708090
    a0b0c0d0e0f101112131415161718191a1b1c1
    d1e1f 10 -n="Final␣Beacon"
```

El Random Beacon [6] es necesario en el final de la Fase 1 como mecanismo de prevención de ataques laterales, es una fuente de aleatoriedad pública que no está disponible antes de un tiempo determinado. Proporcionar la mayor cantidad de entropía posible a partir de fuentes que sean realmente difíciles de replicar a la hora de realizar una contribución. El problema es que si algún contribuyente en una posición intermedia, maliciosamente no desea continuar con la Fase 2 y elimina sus residuos tóxicos. Existen contribuyentes que construyeron a partir de su contribución y a la vez en las anteriores, y lo que esto significa es que todas esas contribuciones ya no se podrían usar porque nadie conoce los residuos tóxicos y por tanto no podrían continuar. Entonces se necesitaría borrar la contribución y todas las anteriores de las que depende.

En la historia de las configuraciones más conocidas que se han realizado (Zcash Sapling 90 contribuidores, Tornado Cash 1114 participantes), AZTEC Ignition, Semaphore) este problema no suele suceder, incluso este tipo de comportamiento es detectable en contraste con alguien que utiliza mala aleatoriedad o se confabula con otros participantes. Se puede saber exactamente quien creó el problema y lo peor que podría pasar es que se reinicie la configuración o una parte de ella.

4.2 Fase 1. Potencias perpetuas de Tau (PPOT)

Una de las ceremonias en rondas secuenciales más grande y conocida que se sigue utilizando al día de hoy como beneficio de todos los proyectos zk-SNARK, es PPOT. Una ceremonia de Fase 1 para toda la comunidad con el fin de reducir la carga a todos los equipos de desarrollo. La figura 18 ilustra como realizar este proceso

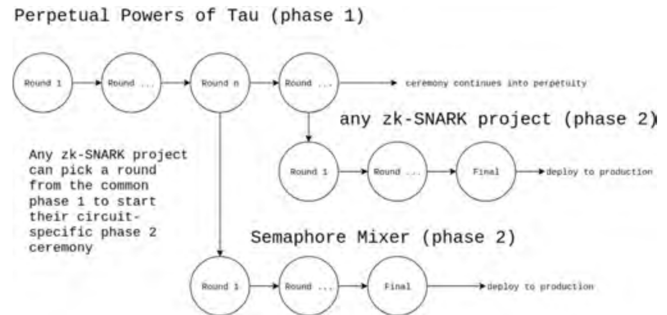

Figura 18. Ceremonia de Poderes Perpetuos de Tau para beneficiar a todos los proyectos zk-SNAR [*Perpetual Powers of Tau Ceremony to benefit all zk-SNARK projects*] Fuente de la imagen, Koh Wei Jie.

4.3 Fase 2. Específica del circuito

Esta fase define el cálculo de la evaluación cifrada de los polinomios de la Interpolación de Lagrange para $\alpha\tau$ y $\beta\tau$. Dados A_i, B_i, C_i, definimos polinomios L_i

$$L_i(x) = \beta \cdot A_i(x) + \alpha \cdot B_i(x) + C_i(x) \quad (9)$$

No podemos calcular $L_i(x)$ directamente porque α y β son privados, por lo que construimos $Li(\tau) \cdot G_1$ usando los valores calculados en (Fase 1):

Código 19. Ejecutar con la librería snarkjs Fase 2

```
snarkjs powersoftau prepare phase2
    pot14_beacon.ptau pot14_final.ptau -v
```

El archivo *.ptau* final, se utilizará para generar las claves de prueba y de verificación con respecto al circuito. En resumen el procedimiento de configuración produce parámetros públicos a partir de

$$R = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, G_1, G_2, l, \{A_i(x), B_i(x), C_i(x)\}_{i=0}^{m}, Z(x))$$

aquí l define la cantidad de entradas públicas.

$$(pk, vk) \leftarrow Setup(R)$$

1. pk (clave de prueba): el CRS que define la declaración, generada por el cálculo MPC.
2. vk (clave de verificación): una simulación trapdoor para R con información del circuito.

Estas claves están disponibles públicamente y solo deben generarse una vez para un programa o circuito determinado. [10]

Generamos las claves sin confianza, pero se recomienda hacer contribuciones mediante MPC.

Código 20. Generamos de claves sin confianza

```
snarkjs zkey new poli_multi_var.r1cs
    pot14_final.ptau circuit_final.zkey
snarkjs zkey verify poli_multi_var.r1cs
    pot14_final.ptau circuit_final.zkey
snarkjs zkey export verificationkey
    circuit_final.zkey verification_key.
    json
```

4.4 Generar pruebas

$$\pi \leftarrow Prove(R, pk, w)$$

El probador toma la cadena de referencia común pk que forma parte del archivo **circuit_final.zkey**, el testigo w archivo **witness.wtns**, dos elementos aleatorios (r,s) del campo y devuelve un argumento $\pi = ([A]_1, [C]_1, [B]_2)$ en el archivo **proof.json**. Usamos r,s para hacer aleatoria la generación de pruebas.

Código 21. Generar pruebas con la librería snarkjs

```
snarkjs groth16 prove build/circuit_final.
    zkey poli_multi_var_js/witness.wtns
    proof.json public.json
snarkjs groth16 fullprove
    poli_multi_var_js/input.json
    poli_multi_var_js/poli_multi_var.wasm
    build/circuit_final.zkey proof1.json
    public1.json # generar testigo y prueba
```

4.5 Verificar pruebas

$$0/1 \leftarrow Verify(R, vk, \pi)$$

El verificador rechaza (0) o acepta (1) la prueba π dada. La función devolverá 1 si el testigo w se cumple.

$$[A]_1 \cdot [B]_2 =$$

$$= [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^{l} w_i \left[\frac{\beta A_i(\tau) + \alpha B_i(\tau) + C_i(\tau)}{\gamma}\right]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2 \tag{10}$$

Intuitivamente, $[A]_1 \cdot [B]_2$ representa el emparejamiento $e : G_1 G_2 \to G_T$. Ahora, comprobemos que lo anterior realmente verifica

$$A(\tau)B(?) = C(\tau) + H(\tau)Z(\tau).$$

El verificador debe calcular 3 emparejamientos: $e([A]_1, [B]_2)$, $e([C]_1, [\delta]_2)$ y

$$e\left(\sum_{i=0}^{l} w_i \left[\frac{\beta A_i(\tau) + \alpha B_i(\tau) + C_i(\tau)}{\gamma}\right]_1, [\gamma]_2\right).$$

Asumimos que $[\alpha \cdot \beta]_T = e([\alpha]_1, [\beta]_2)$ ya está dado en la clave de verificación.

Código 22. Verificación onchain con librería snarkjs

```
snarkjs groth16 verify build/
    verification_key.json prover/public.
    json prover/proof.json
snarkjs zkey export solidityverifier build
    /circuit_final.zkey contracts/Verifier.
    sol
snarkjs zkey export soliditycalldata
    prover/public.json prover/proof.json #
    parámetros llamada al contrato Verifier
    .sol
```

5. ZKATM

Un mixer (o mezclador) es un protocolo que permite a los usuarios realizar transacciones privadas en blockchain al "mezclar" fondos con otros participantes, rompiendo el vínculo entre el depositante y el beneficiario. ZKATM es una abstracción de Tornado Cash homologando un cajero automático, que implementa este concepto mediante:

5.1 Arquitectura Técnica de ZKATM

Backend: Circuitos zk-SNARK con *Circom* y SnarkJS

1. Generación del circuito.
   - Diseñamos un circuito en *Circom* que verifica:
     - La inclusión de un *commitment* en el árbol de Merkle.
     - La validez del $nullifier$ para evitar reutilización.
   - Ejemplo de restricción en R1CS:

```
signal input root;
signal input secret;
component poseidon = Poseidon(2);
poseidon.inputs[0] <== secret;
poseidon.inputs[1] <== nullifier;
root === poseidon.out;
```

2. Generación de pruebas
   - Usamos Snarkjs para:
     - Compilar el circuito a R1CS → QAP.
     - Generar las claves (*trusted setup* con *Groth16*).
     - Crear pruebas *off-chain* (optimizadas para *Scroll Sepolia*).
3. Desafíos técnicos:
   - Compatibilidad de versiones: Tuvimos que usar circomlibjs@0.0.8 para hacer match con el circuito Poseidon de circomlib.
   - Parsing de datos: Ajustar conversiones bigInt ? uint256 en el cliente para la función withdraw.

Frontend: Scaffold-Eth 2

- Razones para elegirlo:
  - Integración nativa con wagmi/Next.js para conexión de billeteras.
  - Plantillas preconfiguradas para contratos inteligentes (Hardhat + Ethers).
  - Soporte para pruebas zk-SNARKs en el navegador.

5.2 Desafíos y Soluciones

1. Generación de pruebas en cliente:
   - **Problema**: Las pruebas zk-SNARK son computacionalmente costosas (?1000 restricciones en nuestro circuito).
   - **Solución**: Generación off-chain en el navegador (Vercel) levemente lento, con optimizaciones para reducir tiempos a <3 segundos realizamos pruebas en servidor (practica no recomendada).
   - "Pudimos haber usado WASM o servicios como Irreducible (FPGA).
2. Escalabilidad del árbol de Merkle:
   - **Actual**: Soporta hasta 2^3 hojas (suficiente para MVP).
   - **Extensible**: Basta aumentar la altura del árbol en el contrato (MerkleTree.sol).
3. Costos en Scroll Sepolia:
   - **Gas fees**: ≈ \$0,02 USD por transacción (vs. ≈ \$0,15 USD en Ethereum Mainnet).

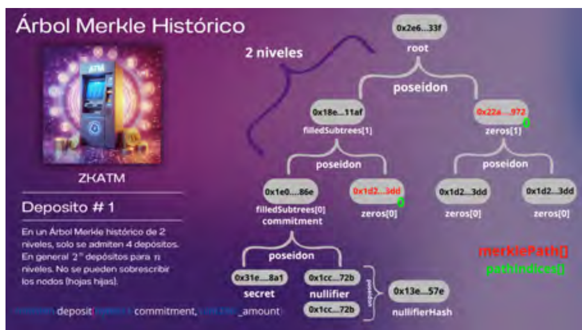

Figura 19. Cada depósito genera un commitment (hoja) que se integra al árbol mediante hashes Poseidon [*Each deposit generates a commitment (leaf) that is integrated into the tree using Poseidon hashes.*].

La imagen 19 muestra el estado del árbol tras un depósito, destacando los nodos *filledSubtrees* y *zeros* usados para calcular el *root*. Limitado inicialmente a 2^n hojas. El diseño permite escalabilidad al incrementar la altura del árbol.

La imagen 21 muestra: (1) el archivo *input.json* con las señales públicas ($root$, $nullifierHash$) y privadas ($secret$,

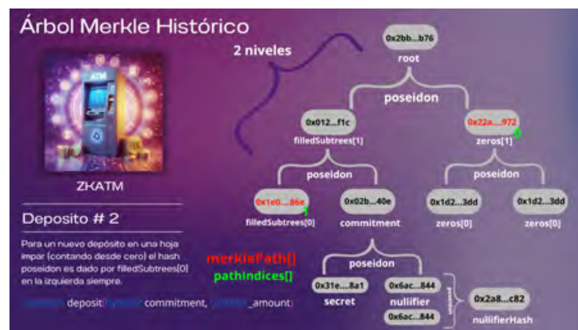

Figura 20. Árbol de Merkle histórico modificado tras un segundo depósito [*Historical Merkle tree modified after a second deposit*].

Figura 21. Flujo de trabajo de generación y verificación de pruebas zk-SNARK en ZKATM [*Workflow for generating and verifying zk-SNARK proofs in ZKATM*].

$nullifier$); (2) la estructura del circuito *Circom* que procesa estas entradas; y (3) la función withdraw del contrato inteligente que verifica la prueba on-chain mediante el verificador *Groth16*. Las pruebas garantizan la validez del retiro sin revelar información sensible.

La imágenes 22, 23, 24 continuación muestra las funciones principales de la DApp: depósito (ATM Deposit), retiro (ATM Withdraw) y visualización de *tokens*. Incluye un módulo para interactuar con el contrato inteligente (Debug Contracts) y detalles de transacciones, como la cantidad de *tokens* a aprobar (50 ZKATM) y el saldo en ETH (1.4565 ETH). La interfaz está optimizada para operaciones privadas y eficientes en *Scroll Sepolia*.

Conclusiones

El desarrollo de ZKATM demuestra que es posible integrar privacidad *end-to-end* en aplicaciones descentralizadas manteniendo escalabilidad y costos competitivos frente a implementaciones en redes como *Ethereum Mainnet*. La elección de *Scroll Sepolia* (ZKEVM) y el uso de árboles de Merkle históricos permitieron optimizar el rendimiento sin sacrificar la verificabilidad, mostrando que las ZKP no son solo una tecnología experimental, sino una herramienta viable para casos de uso reales.

A nivel educativo, este trabajo ofrece un recurso práctico para comprender la construcción de circuitos aritméticos, la

Figura 22. Interfaz de usuario de ZKATM desarrollada con Scaffold-ETH 2 (aprobación de token ZKATM para la DApp por parte de los usuarios) [*ZKATM UI developed with Scaffold-ETH 2 (ZKATM token approval for the DApp by users)*].

Figura 23. Interfaz de depósito en ZKATM [*ZKATM Deposit UI*].

generación de pruebas y su verificación, reduciendo la barrera de entrada para desarrolladores interesados en criptografía aplicada. La metodología paso a paso, combinada con fundamentos matemáticos claros, aporta valor tanto a la comunidad académica como al sector profesional.

Los resultados obtenidos confirman que las pruebas de conocimiento cero tienen un papel central en el futuro de la *blockchain*, especialmente en áreas donde privacidad y transparencia deben coexistir. Como línea de trabajo futuro, se proyecta la investigación y desarrollo de circuitos en otros lenguajes y entornos especializados – como Noir, Cairo y sistemas de pruebas alternativos – con el fin de ampliar el espectro de aplicaciones y optimizar aún más su eficiencia. Esto abrirá oportunidades para explorar nuevos paradigmas de seguridad y escalabilidad, impulsando la adopción de soluciones de conocimiento cero en ámbitos financieros, logísticos y de gobernanza descentralizada.

Suplementos

Este artículo contiene un suplemento de información en: https://github.com/manudev97/dapp-zkatm.

Conflictos de interés

Se declara que no existen conflictos de interés.

Referencias

[1] Bellés-Muñoz, M., M. Isabel, J.L. Muñoz-Tapia, A. Rubio, and J. Baylina: *Circom: A circuit description language for building zero-knowledge applications*. IEEE Transactions on Dependable and Secure Computing, 20(6):4733–4751, 2022. https://doi.org/10.1109/TDSC.2022.3232813.

[2] Ben-Sasson, E., A. Chiesa, E. Tromer, and M. Virza: *Succinct {Non-Interactive} zero knowledge for a von Neumann architecture*. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014. https://dl.acm.org/doi/10.5555/2671225.2671275.

[3] Benarroch, D., L. Brandão, M. Maller, and E. Tromer: *ZKProof Community Reference*. Technical report, ZKProof, July 2022. https://docs.zkproof.org/reference.

[4] Bernhard, D., O. Pereira, and B. Warinschi: *How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios*. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 626–643. Springer, 2012. https://doi.org/10.1007/978-3-642-34961-4_38.

[5] Beuchat, J. L., J.E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya: *High-speed software implementation of the optimal ate pairing over Barreto–Naehrig curves*. In *Pairing-Based Cryptography-Pairing 2010: 4th International Conference, Yamanaka Hot Spring, Japan, December 2010. Proceedings 4*, pages 21–39. Springer, 2010. https://doi.org/10.1007/978-3-642-17455-1_2.

[6] Bowe, S., A. Gabizon, and I. Miers: *Scalable multi-party computation for zk-snark parameters in the random beacon model*. Cryptology ePrint Archive, 2017. https://eprint.iacr.org/2017/1050.pdf.

[7] Bowe, S., J. Grigg, and D. Hopwood: *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Paper 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

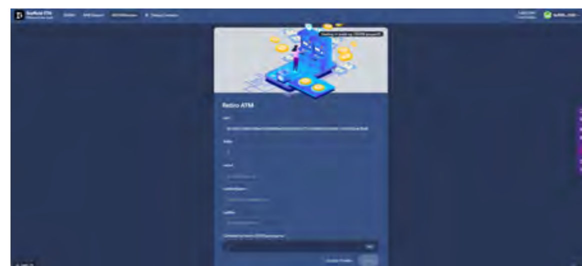

Figura 24. Interfaz de retiro en ZKATM [*Withdrawal UI at ZKATM*].

[8] Chen, B., B. Bünz, D. Boneh, and Z. Zhang: *Hyperplonk: Plonk with linear-time prover and high-degree custom gates*. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 499–530. Springer, 2023. https://doi.org/10.1007/978-3-031-30617-4_17.

[9] Gabizon, A., Z.J. Williamson, and O. Ciobotaru: *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

[10] George, K.: *The mathematical mechanics behind the groth16 zero-knowledge proving protocol*, 2022. https://kayleegeorge.github.io/math110_WIM.pdf.

[11] Groth, J. and A. Sahai: *Efficient non-interactive proof systems for bilinear groups*. In *Advances in Cryptology–EUROCRYPT 2008: 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings 27*, pages 415–432. Springer, 2008. https://doi.org/10.1007/978-3-540-78967-3_24.

[12] Petkus, M.: *Why and how zk-snark works*, 2019. https://arxiv.org/abs/1906.07221.

[13] Quisquater, J.J., M. Quisquater, M. Quisquater, M. Quisquater, L. Guillou, M.A. Guillou, G. Guillou, A. Guillou, G. Guillou, and S. Guillou: *How to explain zero-knowledge protocols to your children*. In *Conference on the Theory and Application of Cryptology*, pages 628–631. Springer, 1989. https://doi.org/10.1007/0-387-34805-0_60.