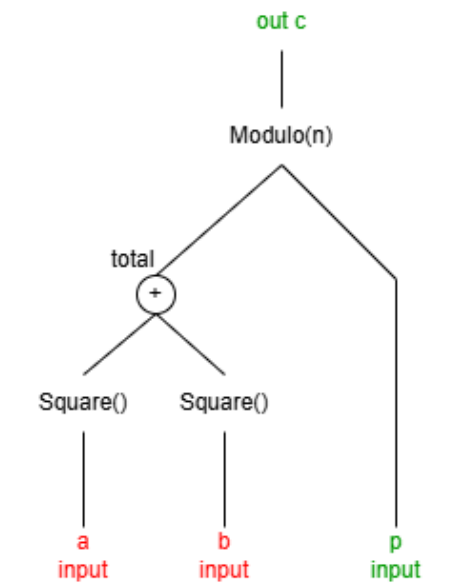
Circuito aritmético para verificar operaciones matemáticas sin revelar los valores de entrada, utilizando pruebas de conocimiento cero.

Utilizaremos el lenguaje "Circom" para implementar el circuito aritmético que pueda verificar la operación $c = (a^2 + b^2) \mod p$ donde p es un número primo público. Este circuito expone c como salida pública mientras que mantiene a y b como entradas privadas.

En Circom definimos los circuitos con la palabra reservada *template* (se permite la composición de *templates*); dentro de ellos definimos *señales* tanto de entrada como de salida de manera que podamos trabajar con ellas junto con las operaciones de aritmética para obtener un resultado. Finalmente, necesitamos crear una instancia (*component*) del *template* para realmente crearlo y utilizarlo.

Representación gráfica



Dentro del *template* **Modulo(n)** realizamos $c = total \mod p$ donde nos aseguramos de que la operación es correcta utilizando un *template* **LessThan(n)** que comprueba que $0 \leq c < p$. Para lograr esto de forma eficiente, los números se descomponen en bits para que las restricciones puedan operar sobre bits individuales.

El *template* **LessThan(n)** utiliza **Num2Bits(n)** para comparar dos números comprobando si su diferencia es negativa (utilizando el bit más significativo). Donde **n** significa número de bits, en este circuito utilizamos $n = 256bits$, lo que permite números primos de hasta 2^{256}.

Los sistemas de prueba de conocimiento cero (como Groth16) requieren que los circuitos aritméticos utilicen restricciones cuadráticas. **Num2Bits(n)** descompone las operaciones no cuadráticas (por ejemplo, la manipulación de bits) en restricciones cuadráticas.

Finalmente, la estructura del circuito nos queda así:

- Main()
  - Square()
  - Modulo(n)
    - LessThan(n)
      - Num2Bits(n)

A continuación mostramos el código completo del circuito, escrito en el lenguaje Circom y comentado detalladamente:

parcial1.circom

```
pragma circom 2.0.0;

// Convierte un número entero en su representación binaria y aplica restricciones para validar la corrección
template Num2Bits(n) {

    signal input in;          // Entrada a descomponer
    signal output out[n];     // Array de salida de n bits
    var lc = 0;               // Acumulador para el valor reconstruido

    for (var i = 0; i < n; i++) {
        out[i] <-- (in >> i) & 1;        // Extrae el bit i-ésimo
        out[i] * (out[i] - 1) === 0;     // Nos aseguramos de que el bit sea 0 o 1
        lc += out[i] * (1 << i);         // Reconstruimos el entero a partir de bits
    }
    lc === in;     // Verificamos que la suma coincida con la entrada

}

// Comprueba si in[0] < in[1] usando descomposición de bits
template LessThan(n) {
```

```
    signal input in[2];     // Entradas: [a, b]
    signal output out;      // Salida: 1 si a < b, de lo contrario 0

    component n2b = Num2Bits(n+1);           // Descomponer en n+1 bits
    n2b.in <== (1 << n) + in[0] - in[1];     // Calculamos (2^n + a - b)
    out <== 1 - n2b.out[n];                  // MSB es 0 si a < b

}

// Calcula c = total % p y comprueba que 0 ≤ c < p
template Modulo(n) {

    signal input total;    // Valor de entrada a reducir
    signal input p;        // Módulo primo
    signal output c;       // Resto (c = total % p)

    signal q;                // Cociente (q = total / p)
    q <-- total \ p;         // Asignar cociente (división entera)

    c <== total - q * p;     // Asignar resto

    total === q * p + c;     // Verificamos total = q*p + c

    component lt = LessThan(n);     // Comprobación de rango: c < p
    lt.in[0] <== c;                 // Comparamos c y p
    lt.in[1] <== p;
    lt.out === 1;                   // Debe ser verdadero (1)

}

// Calcula el cuadrado de un número
template Square() {

    signal input in;       // Número de entrada
    signal output out;     // Salida (in²)
    out <== in * in;       // Asignamos out = in²

}

// Nivel mas alto del circuito para calcular c = (a² + b²) % p.
template Main() {
```

```
    signal input a;      // Entrada privada a
    signal input b;      // Entrada privada b
    signal input p;      // Primo público p
    signal output c;     // Salida pública (a² + b²) % p

    component a_sq = Square();     // Cuadrado de a
    a_sq.in <== a;                 // Asignar "a" a la entrada de Square

    component b_sq = Square();     // Cuadrado de b
    b_sq.in <== b;                 // Asignar "b" a la entrada de Square

    signal total;                      // Suma de cuadrados
    total <== a_sq.out + b_sq.out;     // Asignar la suma de cuadrados

    component mod = Modulo(256);       // Calcular el total % p
    mod.total <== total;               // Asignar total
    mod.p <== p;                       // Asignar primo
    c <== mod.c;                       // Salida (resto)

}

// Crea una instancia del template Main y marca p como pública.
component main {public [p]} = Main();
```

Ahora que tenemos este circuito aritmético, podemos usarlo en pruebas ZK-SNARKs; para eso necesitamos primeramente compilar el circuito para obtener un sistema de ecuaciones aritméticas que lo represente. Lo hacemos con el siguiente comando:

```
circom parcial1.circom --r1cs --wasm --sym --c
```

Con estas opciones, generamos tres tipos de archivos:

- **--r1cs:** Genera el archivo parcial1.r1cs, que contiene el sistema de restricciones R1CS del circuito en formato binario.
- **--wasm:** Genera el directorio parcial1_js, que contiene el código WASM (parcial1.wasm) y otros archivos necesarios para generar el witness.
- **--sym:** Genera el archivo parcial1.sym, un archivo de símbolos necesario para la depuración o para imprimir el sistema de restricciones en modo anotado.
- **--c:** Genera el directorio parcial1_cpp, que contiene varios archivos (parcial1.cpp, parcial1.dat y otros archivos comunes para cualquier programa compilado, como main.cpp, MakeFile, etc.) necesarios para compilar el código C++ y generar el testigo. Por defecto, esta opción genera un archivo asm incompatible con algunas arquitecturas.

> **¿Qué es el witness?** Es la colección de valores que coinciden con todas las restricciones de un circuito sin revelar ninguno de los valores excepto las entradas y las salidas públicas.

Para poder generar pruebas necesitamos generar un witness; para esto, antes necesitamos un archivo con las entradas de las cuales queremos un witness. Definimos dicho archivo como sigue:

input.json

```
{
    "a": "3",
    "b": "4",
    "p": "5"
}
```

Ahora podemos generar el witness con WebAssembly una vez situados en la carpeta **parcial1_js** con el siguiente comando:

`node generate_witness.js parcial1.wasm ../input.json witness.wtns`

Con esto generaremos el archivo **witness.wtns**. Este archivo está codificado en un formato binario compatible con snarkjs.

Finalmente podemos proceder a generar y verificar pruebas. Utilizaremos un protocolo SNARK llamado Groth16 (es uno de los protocolos compatibles con Circom). Es uno de los esquemas más eficientes, pero requiere una configuración de confianza inicial (trusted setup).

El trusted setup consiste en dos partes:

- The powers of tau, que es independiente del circuito.
- La fase 2, que depende del circuito.

A continuación llamaremos "ceremonia" a todo el proceso de un trusted setup, realizándola con los siguientes pasos:

Comenzando una nueva ceremonia de "Powers of Tau": `snarkjs powersoftau new bn128 12 pot12_0000.ptau -v`

Contribuyendo a la ceremonia: `snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau --name="First contribution" -v`

> Aquí se solicita una entropía, la **entropía** en un trusted setup es un texto aleatorio que asegura que los parámetros generados sean seguros.

Preparando la fase 2 de la ceremonia: `snarkjs powersoftau prepare phase2 pot12_0001.ptau pot12_final.ptau -v`

Generando .zkey: `snarkjs groth16 setup parcial1.r1cs pot12_final.ptau parcial1_0000.zkey`

> Este archivo contiene las keys de prueba y verificación, junto con todas las contribuciones de la fase 2.

Contribuyendo a la fase 2 de la ceremonia: `snarkjs zkey contribute parcial1_0000.zkey parcial1_0001.zkey --name="1st Contributor Name" -v`

> Aquí también se solicita una entropía.

Finalmente **exportamos la clave de verificación**: `snarkjs zkey export verificationkey parcial1_0001.zkey verification_key.json`

Damos por terminada la ceremonia y seguimos con la generación y verificación de pruebas. Podemos generar una prueba zk asociada al circuito y al witness con el siguiente comando:

```
snarkjs groth16 prove parcial1_0001.zkey parcial1_js/witness.wtns proof.json public.json
```

Este comando genera una prueba de Groth16 y genera dos archivos:

- **proof.json:** contiene la prueba. La prueba contiene el protocolo y la curva elíptica usada junto con puntos en la curva (A, B, C) y compromisos sobre polinomios.
- **public.json:** contiene los valores de las entradas y salidas públicas.

> **¿Qué es una curva elíptica?** Una curva elíptica es una ecuación que define un conjunto de puntos con propiedades algebraicas. En ZK-SNARKs, las curvas elípticas permiten construir pruebas verificables con emparejamientos bilineales.

Para verificar la prueba, ejecutamos el siguiente comando:

```
snarkjs groth16 verify verification_key.json public.json proof.json
```

El comando utiliza los archivos verification_key.json, proof.json y public.json que exportamos anteriormente para comprobar la validez de la prueba. Si la prueba es válida, el comando emite: `[INFO] snarkJS: OK!`

Una prueba válida no solo demuestra que conocemos un conjunto de valores que satisfacen el circuito, sino también que las entradas y salidas públicas que utilizamos coinciden con las descritas en el archivo public.json.

Con esto damos por finalizado todo el proceso de generación de pruebas y verificación para los valores de entrada definidos en el archivo input.json.

En breve proporcionaremos distintos input.json de manera que sirvan como ejemplos de uso con distintos valores. Para probar estos ejemplos, basta con modificar el archivo input.json y volver a ejecutar el comando de generación de witness y los de generación de pruebas y verificación. El orden de ejecución quedaría de la siguiente manera:

1. Dentro de la carpeta **parcial1_js**, generamos el witness para el input modificado: `node generate_witness.js parcial1.wasm ../input.json witness.wtns`
2. Generamos la prueba: `snarkjs groth16 prove parcial1_0001.zkey parcial1_js/witness.wtns proof.json public.json`
3. Verificamos la prueba: `snarkjs groth16 verify verification_key.json public.json proof.json`

A continuación mostramos una tabla con ejemplos con diferentes datos de entrada, el archivo public.json que genera la prueba y el proceso que se realiza dentro del circuito aritmético.

input.json	public.json	Circuito aritmético
{ "a": "3", "b": "4", "p": "5" }	["0", "5"]	1. a² = 9 , b² = 16 → total = 25 . 2. 25 % 5 = 0 → c = 0 . 3. Output: c = 0 .
{ "a": "2", "b": "3", "p": "5" }	["3", "5"]	1. a² = 4 , b² = 9 → total = 13 . 2. 13 % 5 = 3 → c = 3 . 3. Output: c = 3 .
{ "a": "3", "b": "4", "p": "3" }	["1", "5"]	1. a² = 9 , b² = 16 → total = 25 . 2. 25 % 3 = 1 → c = 1 . 3. Output: c = 1 .

Con estas instrucciones e información debería de poder ser capaz de realizar distintas pruebas con distintos valores de entradas de manera que logre entender el funcionamiento del conocimiento cero junto con la generación de pruebas y verificación a partir de un witness generado con valores a raíz de un circuito aritmético.

Este mini-proyecto destacó la precisión requerida en el diseño de circuitos ZKP y el poder de las pruebas de conocimiento cero para validar cálculos de forma confidencial.

Realizado por: *Abigail Mercedes Nuñez Fernandez y Jonathan Sebastian Godoy Silva.*