Workshop: Circom

"From Zero to Hero" (zkintro)

ZKET Core Program 2025

October 3, 2025

Agenda

¿Qué es una prueba de conocimiento cero?

- Permite demostrar que sabes algo sin revelar el secreto.
- Propiedades clave:
  - *Zero knowledge* (privacidad)
  - *Succinctness* (la prueba permanece pequeña)
- Aplicaciones: privacidad, escalabilidad, identidades, blockchains, firmas grupales.

Visión general del flujo con Circom + Groth16

Write → Build → Setup → Prove → Verify

- **Write**: escribir el circuito (constraints)
- **Build**: compilar el circuito (r1cs, wasm)
- **Setup**: trusted setup para generar claves
- **Prove**: generar la prueba a partir del input privado
- **Verify**: verificar la prueba con input público

Circuito de multiplicación: ejemplo básico

- Vamos a crear un **“Hola Mundo”** en ZKPs: un programa que demuestra conocimiento de dos números secretos cuya multiplicación es pública, sin revelarlos.
- Por ejemplo, si el número público es 33, los secretos podrían ser 11 y 3.
- Esta idea sirve como base para lo que vendrá luego.

Circuito de multiplicación: ejemplo básico

- Vamos a crear un **“Hola Mundo”** en ZKPs: un programa que demuestra conocimiento de dos números secretos cuya multiplicación es pública, sin revelarlos.
- Por ejemplo, si el número público es 33, los secretos podrían ser 11 y 3.
- Esta idea sirve como base para lo que vendrá luego.

Coding time

Problema: trivialidad con $1 \times c = c$

- Si permitimos $a = 1$ o $b = 1$, el circuito es siempre cierto (trivial).
- Necesitamos prohibir que alguno de los factores sea 1.

Ejemplos de inputs para el output público $c = 33$

- **Válidos (sin trivialidad)**: $a = 3,\ b = 11 \quad (\text{o } a = 11,\ b = 3)$.
- **Inválidos por la nueva regla**: $a = 1,\ b = 33 \quad (\text{o } a = 33,\ b = 1)$.
- *Nota*: si no restringimos signo, también valen pares como $a = -3,\ b = -11$ (en $\mathbb{F}_p$, $-3 \equiv p - 3$), pero el objetivo es evitar los factores triviales.

Uso de `IsZero()` para condición "no igual a 1"

¿Qué hace `IsZero`?

- Es un *gadget* de Circom que recibe una señal x y produce:

$$\texttt{IsZero}(x) = \begin{cases} 1 & \text{si } x = 0 \\ 0 & \text{si } x \neq 0 \end{cases}$$

- Se usa para convertir una comparación con cero en una restricción booleanas dentro del circuito.

Idea para evitar la trivialidad $1 \times c = c$

- Queremos asegurar $a \neq 1$ y $b \neq 1$.
- Consideramos el producto $(a-1)(b-1)$.
- Si alguno es 1, entonces $(a-1)(b-1) = 0$ y `IsZero` devolvería 1.
- Forzando la salida de `IsZero` a 0, obligamos a que $(a-1)(b-1) \neq 0$ $\Rightarrow$ ni a ni b pueden ser 1.

Como implementamos isZero?

No hace falta, podemos usar Circomlib

`circomlib` y `circomlibjs`

- **circomlib**: colección de plantillas de circuitos reutilizables (comparadores, *hashes* ZK-friendly como Poseidon/MiMC, BabyJubJub, etc.) para incluir en circuitos Circom.
- **circomlibjs**: implementaciones y utilidades en JavaScript (por ejemplo, Poseidon en JS) para testear, generar entradas/salidas y validar fuera del circuito (scripts de pruebas, tooling).
- Beneficio: evita "reinventar la rueda" y asegura componentes optimizados y probados.

Como implementamos isZero?

No hace falta, podemos usar Circomlib

`circomlib` y `circomlibjs`

- **circomlib**: colección de plantillas de circuitos reutilizables (comparadores, *hashes* ZK-friendly como Poseidon/MiMC, BabyJubJub, etc.) para incluir en circuitos Circom.
- **circomlibjs**: implementaciones y utilidades en JavaScript (por ejemplo, Poseidon en JS) para testear, generar entradas/salidas y validar fuera del circuito (scripts de pruebas, tooling).
- Beneficio: evita "reinventar la rueda" y asegura componentes optimizados y probados.

Coding time

De multiplicar secretos a firmar mensajes

- Ya demostramos conocimiento de dos secretos cuyo producto es público.
- Ahora queremos algo más útil: **firmar mensajes**.
- Meta: probar autoría de un mensaje sin revelar la identidad secreta.

Recordatorio: firma digital tradicional

- **Keygen**: generar clave privada + clave pública.
- **Sign**: firmar mensaje con clave privada.
- **Verify**: verificar firma con la clave pública.

Con ZKPs podemos imitar esto usando **hashes y compromisos**.

Hash y compromiso

- **Hash**: función unidireccional, eficiente en ZK (ej. Poseidon).
- **Commitment**: equivalente a la clave pública.
  $identity\_commitment = Poseidon(identity\_secret)$
- Propiedades: *hiding* (oculta el secreto), *binding* (no se puede cambiar).

El esquema que implementaremos

- **Compromiso**: $identity\_commitment = Poseidon(identity\_secret)$
- **Firma**: $signature = Poseidon(identity\_secret,\ message)$
- **Verificación**: usar $(commitment, message, signature)$ y el proof.

El esquema que implementaremos

- **Compromiso**: $identity\_commitment = Poseidon(identity\_secret)$
- **Firma**: $signature = Poseidon(identity\_secret,\ message)$
- **Verificación**: usar $(commitment, message, signature)$ y el proof.

Coding time

¿Qué logramos con este circuito?

- **Autenticidad**: sólo quien conoce el secreto puede firmar.
- **Privacidad**: el secreto nunca se revela, sólo se usa en el ZKP.
- **Eficiencia**: Poseidon hace viable este esquema dentro del circuito.

Conclusión

- Aprendimos cómo construir ZKPs desde cero usando Circom + Groth16
- Recorrimos tres iteraciones: multiplicador simple, mejoras, y firma digital
- Con esta base puedes explorar construcciones más avanzadas (identidades ZK, firmas grupales, escalabilidad, etc.)

¡Gracias! ¿Preguntas?