Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Demostrando la ejecución de un programa de alto nivel con Plonky2

Tesis de Licenciatura en Ciencias de la Computación

Bruno Weisz

Director: Agustín Garassino
Codirector: Matías López y Rosenfeld
Buenos Aires, 2025

DEMOSTRANDO LA EJECUCIÓN DE UN PROGRAMA DE ALTO NIVEL CON PLONKY2

En este trabajo se estudia la posibilidad de demostrar la ejecución de programas de alto nivel utilizando Plonky2, un sistema de pruebas criptográficas de tipo ZK-SNARK que no requiere de un *trusted setup*. El trabajo se enmarca en el área de las pruebas de conocimiento cero (Zero Knowledge Proofs), presentando una introducción a las primitivas matemáticas y criptográficas necesarias para comprender estos protocolos, así como un análisis superficial de los sistemas relevantes como PLONK.

Se propone y desarrolla una herramienta que traduce programas de ACIR (la representación intermedia de Noir) a Plonky2. De esta manera, se logra la construcción de pruebas verificables de ejecución en entornos con recursos limitados de almacenamiento, superando una de las principales restricciones de Barretenberg.

El documento incluye la descripción formal de la traducción de primitivas y operaciones, una evaluación experimental del desempeño obtenido y un análisis comparativo en términos de tiempos de ejecución y tamaños de artefactos generados. Finalmente, se discuten los resultados alcanzados y se plantean líneas de trabajo futuro.

Palabras clave: Pruebas de conocimiento zero, zero knowledge proofs, ZK, SNARK, PLONK, Plonky2, Noir, ACIR, Barretenberg, Criptografía.

Índice general

Parte I

INTRODUCCIÓN

Las pruebas de conocimiento cero, o en inglés **Zero Knowledge** (**ZK**) Proofs [1], son protocolos criptográficos interactivos entre 2 actores. Permiten a una de las partes (el *prover*) demostrar a otra parte (el *verifier*) que una afirmación matemática es verdadera sin revelar nada más que la veracidad de la misma. En el uso cotidiano tanto el prover como el verifier son programas informáticos y por ende todas las partes del protocolo están automatizadas.

Por otro lado, otra familia de protocolos llamados **SNARKS** (Succint Non-Interactive ARgument of Knowledge) permiten demostrar la ejecución de un programa conocido por ambas partes. Esto se consigue modelándolo como un sistema de ecuaciones que representa todos los estados intermedios del cómputo y demostrando que se conoce una solución para ese sistema de ecuaciones. Estas familias de protocolos no son disjuntas, ya que combinando las ideas de ambas se obtienen protocolos que denominamos **ZK-SNARK**.

En los últimos años, la investigación sobre estos protocolos tuvo un crecimiento exponencial, y esto se debe principalmente 2 factores: (1) las computadoras alcanzaron un poder de cómputo que transformó estas familias de tecnologías de prohibitivo a tratable; (2) el hecho de que los SNARKs proveen un método para escalar eficientemente las tecnologías Blockchain [2]. Si bien estas son áreas que tuvieron muchos avances últimamente, todavía presentan desafíos y están en constante movimiento.

Hoy en día existen diversos protocolos e implementaciones que proveen distintas propiedades, como pueden ser mayor velocidad de ejecución, menor tamaño de las pruebas generadas, necesidad de Trusted Setup [3], etc. A su vez cada sistema tiene su propia estrategia para darle forma a las afirmaciones que se desean probar. Un protocolo muy influyente fue PLONK [4], que aportó entre otras cosas una forma genérica para expresar programas como restricciones aritméticas.

En este contexto surgen los Domain Specific Languages (**DSL**): lenguajes de programación de alto nivel que pueden ser interpretados por distintos sistemas de pruebas. En particular, Noir [5] es un DSL cuya sintaxis es similar a la de Rust. Noir pretende ser el lenguaje universal de **ZK** y su estrategia para lograrlo es compilar a una representación intermedia llamada *Abstract Circuit Intermediate Representation* (**ACIR**) [6]. A partir de esta representación intermedia, cualquier sistema de pruebas (llamados comúnmente *proving systems*) debería tener una representación del programa completo que le permita construir una prueba de su ejecución. Sin embargo, de momento sólo existe un sistema de pruebas capaz de interactuar con código ACIR, llamado Barretenberg [7]. Este sistema de pruebas requiere de un *Common Reference String* (CRS) para funcionar, lo cual es limitante en algunos dispositivos ya que puede requerir mucho espacio de almacenamiento para generar pruebas dependiendo de los parámetros del protocolo.

Plonky2 [8] es un sistema de pruebas basado en PLONK (*Plonkish Arithmetization*) pero que a diferencia de Barretenberg, no requiere de un Trusted Setup (y por ende de un CRS) para funcionar. Sin embargo, Plonky2 en este momento no posee la capacidad de interpretar código ACIR, pero cuenta con una API para generar manualmente programas arbitrarios demostrables.

A partir del limitante de almacenamiento de Barretenberg y la posibilidad de usar la API de Plonky2, se pretende construir una herramienta que permita demostrar la ejecución de programas de alto nivel en dispositivos con almacenamiento limitado. Más específicamente se propone extender Plonky2 con la posibilidad de interpretar código ACIR.

El trabajo estará estructurado de forma tal que el conocimiento previo necesario para entenderlo sea mínimo, es decir, que alcance con ser graduado de CS de la computación.

Primero se brindará background suficiente para tener un entendimiento completo del problema a solucionar y luego se dará lugar al desarrollo.

1. PRIMITIVAS CRIPTOGRÁFICAS

En esta sección se van a presentar algunos de los tópicos básicos necesarios para entender la problemática que se intenta resolver. Se va a buscar la mayor completitud posible en los temas no abarcados por la carrera de ciencias de la computación. Entre ellos se encuentran tópicos de la matemática, de la criptografía y más específicamente de la criptografía **ZK**.

1.1. Aritmética modular y cuerpos finitos

La primitiva más básica en el funcionamiento de un sistema de pruebas es la aritmética modular. Voy a dar un breve repaso.

Definición. Sea $m > 1$ un entero, decimos que los enteros a y b son *congruentes módulo* m si su diferencia $(a - b)$ es divisible por m. Lo notamos

$$a \equiv b \ (m)$$

y llamamos a m el *módulo*.

Algunos ejemplos:

- $1 \equiv 8 \ (7)$ porque $7 \mid (8 - 1)$.
- $24 \equiv 4 \ (5)$ porque $5 \mid (24 - 4)$.

Coloquialmente vamos a decir que hacemos operaciones *módulo* m si al completar una operación, tomamos como resultado el resto de la operación en la división por m. Por ejemplo, la operación 10+15 módulo 5 nos daría 0 como resultado, ya que $(10+15)\,\%5 = 0$. Para referirnos al conjunto de los enteros módulo m vamos a usar la notación $\mathbb{Z}/m\mathbb{Z}$, y todas las operaciones de $\mathbb{Z}/m\mathbb{Z}$ quedan definidas *módulo* m. Por ejemplo, el conjunto de los enteros módulo 7 es:

$$\mathbb{Z}/7\mathbb{Z} = \{0, 1, 2, 3, 4, 5, 6\}$$

La gran mayoría de las operaciones aritméticas que ocurrirán en los protocolos van a hacer uso de la aritmética modular. Otras nociones importantes que tenemos que tener son las de **grupo**, **anillo** y **cuerpo**.

Definición. Un **grupo** es un par $(G, \cdot)$ donde G es un conjunto y $\cdot$ es una operación binaria, a la que llamamos *producto*, que cumple las siguientes propiedades:

1. **Clausura:** $\forall a, b \in G$, $(a \cdot b) \in G$, es decir, la operación entre dos elementos de G siempre vuelve a G.
2. **Asociatividad:** $\forall a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, es decir, el producto cumple la propiedad asociativa.
3. **Elemento neutro:** $\exists e \in G$ tal que $\forall a \in G$ se cumple $e \cdot a = a \cdot e = a$, es decir, hay un elemento neutro respecto al producto.

4. **Elemento inverso:** $\forall a \in G, \exists a^{-1} \in G$ tal que $a \cdot a^{-1} = e$, es decir, cada elemento de G tiene un inverso respecto al elemento neutro.

5. Decimos que un grupo es **conmutativo** o **abeliano** si su producto satisface la conmutatividad, es decir que $\forall a, b \in G, a \cdot b = b \cdot a$.

Algunas definiciones importantes sobre un grupo $(G, \cdot)$ son:

1. Si G tiene una cantidad finita de elementos decimos que $(G, \cdot)$ es un grupo finito. El **orden** de G se nota como $|G|$.

2. La **exponenciación** se define como la aplicación sucesiva del producto a varias copias de un mismo elemento. Formalmente, dados $g \in G, x \in \mathbb{Z}^+$, $g^x = \underbrace{g \cdot g \cdot ... \cdot g}_{x \text{ veces}}$.

3. Sean $a \in G, d \in \mathbb{Z}^+$, si d es el menor número tal que $a^d = e$ decimos que d es el **orden** de a en $(G, \cdot)$. Si no existe tal d entonces decimos que a es de orden *infinito*.

   Propiedades:

   - Si G es de orden finito entonces todos los elementos de G tienen orden finito.
   - Sean $a \in G,\ k \in \mathbb{Z}^+$ y e el elemento neutro. Si a tiene orden d y $a^k = e$ entonces $d|k$.

Vamos a poner un ejemplo concreto con aritmética modular. El par $((\mathbb{Z}/7\mathbb{Z}), +_{(7)})$ es un grupo, ya que cumple las 4 propiedades mencionadas anteriormente. Recordemos que en este caso la suma está definida módulo 7:

1. Siempre que sumemos 2 elementos de $(\mathbb{Z}/7\mathbb{Z})$ obtendremos 1 elemento de $(\mathbb{Z}/7\mathbb{Z})$, por lo tanto cumple la propiedad de **clausura**.

2. La suma es asociativa por lo tanto cumple la propiedad de **Asociatividad**.

3. Hay un **elemento neutro** que es el 0.

4. Siempre podemos obtener el **elemento inverso** de un numero a, ya que alcanza con tomar $a^{-1} = (7 - a)\,\%7$.

5. Este es un grupo **conmutativo** ya que la suma cumple la propiedad conmutativa.

Revisando las propiedades, también tenemos que

1. $(\mathbb{Z}/7\mathbb{Z})$ es un grupo finito, ya que $|(\mathbb{Z}/7\mathbb{Z})| = 7$.

2. La *exponenciación* en $((\mathbb{Z}/7\mathbb{Z}), +)$ sería lo que coloquialmente conocemos como multiplicación, es decir, la aplicación sucesiva de la suma.

3. Podemos ver que todos los elementos de $(\mathbb{Z}/7\mathbb{Z})$ tienen un orden finito ya que $\forall i \in \{1, 2, 3, 4, 5, 6\}$, $\underbrace{i + i + i + i + i + i + i}_{7 \text{ veces}} = 0$ exceptuando al 0 que tiene orden 1.
   Notamos que se cumple que en todos los casos el orden del grupo divide al módulo (en este caso, el orden siempre es 7).

Definición. Un anillo es una tripla $(R, +, \cdot)$ donde R es un conjunto y $(\cdot, +)$ son operaciones binarias que llamamos *producto* y *suma* respectivamente. Se tiene que cumplir que:

1. El par $(R, +)$ es un grupo conmutativo.
2. El par $(R, \cdot)$ es un grupo pero donde no necesariamente los elementos tienen un inverso multiplicativo ni se cumple la propiedad conmutativa.
3. Se cumple la **propiedad distributiva**: $\forall a, b, c \in R, \quad (a + b) \cdot c = a \cdot c + b \cdot c$

Si bien todos los elementos de R tienen un inverso aditivo, un anillo no requiere que todos sus elementos tengan un inverso respecto al producto. Tomemos por ejemplo el anillo $(\mathbb{Z}, +, \cdot)$, es decir, los números enteros con la suma y el producto. Todo elemento $n \in \mathbb{Z}$ tiene un inverso aditivo que podemos obtener tomando $-n$. Si buscamos un inverso multiplicativo, tomamos por ejemplo al 2 y notamos que $2^{-1} = \frac{1}{2} \notin \mathbb{Z}$.

Definición: Un **cuerpo** es una tripla $(F, +, \cdot)$ donde F es un conjunto y $(\cdot, +)$ son operaciones binarias que llamamos *producto* y *suma* respectivamente. Se tiene que cumplir que:

1. El par $(F, +)$ es un grupo conmutativo.
2. El par $(F - \{0\} \cdot)$ es un grupo conmutativo.
3. Se cumple la **propiedad distributiva**.

En otras palabras, un cuerpo es un anillo donde el grupo $(F, \cdot)$ tiene inverso para todos sus elementos excepto el 0 y se cumple la propiedad conmutativa.

Notación: Sea $p \in \mathbb{N}$ primo, llamamos $\mathbb{F}_p = \big((\mathbb{Z}/p\mathbb{Z}), +_{(p)}, \cdot_{(p)}\big)$ al cuerpo finito conformado por los enteros módulo p, con la suma $+_{(p)}$ y el producto $\cdot_{(p)}$ definidos módulo p.

Los cuerpos finitos van a ser centrales en todo el desarrollo posterior, en particular aquellos de la forma $\mathbb{F}_p$. Ahora vamos a revisar algunas definiciones y propiedades de los cuerpos que van a resultar útiles cuando estemos explorando el protocolo PLONK.

Definición. Decimos que un elemento de un cuerpo F es una **unidad** si tiene un inverso multiplicativo. Luego, F^* es el conjunto de todos los elementos de F que tienen inverso multiplicativo.

Propiedad. En $\mathbb{F}_p$ todos los elementos distintos de 0 son unidades. Por ende, $\mathbb{F}_p^* = \{1, 2, \cdots, p - 1\}$.

Definición. Sea $\mathbb{F}_p$ un cuerpo finito, decimos que $g \in Fp$ es un **generador** de $\mathbb{F}_p$ si todo elemento de $\mathbb{F}_p^* = \mathbb{F}_p - \{0\}$ es igual a algún g^x con $g \in \mathbb{N}$. En otras palabras, si se cumple que

$$\mathbb{F}_p^* = \{1, g, g^2, g^3, ..., g^{p-2}\}$$

Coloquialmente decimos que g *genera* a $\mathbb{F}_p$.

Teorema. $\forall p$ primo, existe un generador en $\mathbb{F}_p$.

Veamos esto con el ejemplo de $\mathbb{F}_7$ donde $F = \{0, 1, 2, 3, 4, 5, 6\}$. Para cada uno de los elementos de F, vemos que si aplicamos sucesivamente el producto por sí mismo obtenemos los siguientes conjuntos de elementos:

- $1 \to 1, 1, 1, \cdots = \{1\}$.
- $2 \to 2, 4, 1, 2, 4, 1, \cdots = \{1, 2, 4\}$.
- $3 \to 3, 2, 6, 4, 5, 1, 3, 2, \cdots = \{1, 2, 3, 4, 5, 6\}$
- $4 \to 4, 2, 1, 4, 2, 1, \cdots = \{1, 2, 4\}$
- $5 \to 5, 4, 6, 2, 3, 1, 5, 4, 6, \cdots = \{1, 2, 3, 4, 5, 6\}$
- $6 \to 6, 1, 6, 1, \cdots = \{1, 6\}$

En el ejemplo anterior, vemos que $\{3, 5\}$ es el conjunto de generadores de $\mathbb{F}_7$, no así el resto de sus elementos. Coloquialmente vamos a decir que cada uno de ellos es un **generador de grado n**, siendo n el tamaño del conjunto de elementos que generan.

Teorema de Lagrange. Sea $\mathbb{F}_p$ un cuerpo finito y a un generador de grado n. Siempre se cumple que n divide a $|\mathbb{F}_p|-1$. Vemos que esto se cumple en ejemplo, ya que los conjuntos generados son de tamaño $1, 3, 6, 3, 6$ y 2, todos dividen a $7 - 1$.

1.1.1. Problema del logaritmo discreto

El Problema del Logaritmo Discreto es un problema matemático que surge en criptografía como una solución a muchos problemas y va a ser una pieza clave de todos los protocolos que vamos a ver a continuación. Muchos protocolos criptográficos se volverían inseguros si se hallara una solución tratable de este problema.

Definición. Sea g un generador de $\mathbb{F}_p$ y $h \neq 0 \in \mathbb{F}_p$. El Problema del Logaritmo Discreto (**DLP** por sus siglas en inglés) es el problema de encontrar un exponente $x \in \mathbb{N}$ tal que

$$g^x \equiv h \ (p)$$

El número x se conoce como logaritmo discreto de h en base g y se denota $log_g(h)$. Hay una cantidad infinita de soluciones al problema, ya que si $g^x = h$ entonces $g^{x+k\cdot(p-1)} = h$. Por esto mismo, el problema está definido módulo $p - 1$ y existe una única solución.

DLP es un problema difícil en teoría de la computación, ya que los únicos algoritmos que se conocen para resolverlo para un cuerpo genérico son de tiempo exponencial en función del tamaño de p. Esto quiere decir que si p es suficientemente grande entonces no es posible hallar x en un tiempo tratable.

El algoritmo trivial para solucionar el problema es computar todos los exponentes de g hasta hallar un exponente x tal que $g^x = h$. Sin embargo, si analizamos la complejidad computacional y suponemos que p se representa con k bits (es decir que $p \in [2^{k-1}, 2^k]$), entonces el algoritmo tiene un costo de $\mathcal{O}(3^k)$.

A continuación vamos a ver un protocolo criptográfico básico cuya seguridad depende de la dificultad de DLP: el intercambio de claves de Diffie-Hellman.

1.1.2. Diffie-Hellman

El intercambio de claves de Diffie-Hellman es un problema conocido de la criptografía simétrica. El problema que resuelve es el siguiente: supongamos que tenemos 2 partes (Alice y Bob) que quieren tener una clave compartida para usar un cifrado simétrico, pero el único medio que tienen para comunicarse es inseguro. Una forma de interpretar un medio inseguro es pensar que hay un adversario (llamémoslo Malcom) que puede observar toda la informacion que Alice y Bob intercambian. La dificultad de DLP en $\mathbb{F}_p$ provee una solución a este problema. El protocolo es el siguiente:

1. Alice y Bob se ponen de acuerdo en un primo p suficientemente grande y un valor $g \neq 0 \in \mathbb{F}_p$. No importa si p y g son valores públicos, es decir, Malcom puede conocer estos valores. Esta idea de que se debe asumir que el adversario conoce el protocolo se conoce como **principio de Kerckhoff**.

2. Alice y Bob eligen valores secretos $a \in \mathbb{N}$ y $b \in \mathbb{N}$ respectivamente y estos valores también deben ser suficientemente grandes.

3. Cada uno va a calcular
$$A \equiv g^a \ (p) \quad y \quad B \equiv g^b \ (p)$$
respectivamente.

4. Cada uno comparte el valor que calculó con el otro, es decir, Alice le transmite A a Bob a través del medio inseguro, y Bob hace lo mismo con B.

5. Por ultimo, cada uno va a calcular el valor de K (la clave compartida) de la siguiente forma: Alice va a tomar el valor obtenido B y va a calcular
$$K \equiv B^a \ (p)$$
y lo mismo va a hacer Bob con A y b, es decir
$$K' \equiv A^b \ (p).$$
Es fácil ver que $K \equiv K' \ (p)$ y que por ende ahora tienen una clave en común.

La pregunta entonces es ¿que hace que Malcom no tenga la clave K si pudo observar todas las comunicaciones? La respuesta es que la seguridad de este intercambio de claves depende de la dificultad de resolver DLP. Los valores que Malcom pudo observar son p, g, A y B, sin embargo
$$K \equiv g^{a \cdot b} \equiv A^b \equiv B^a \ (p),$$
pero K no puede calcularse sin conocer a o b, que son valores privados. Uno podría pensar entonces que el valor de a se puede deducir de saber que $A \equiv g^a \ (p)$ pero ahí es donde DLP entra en juego y nos asegura que Malcom no puede realizar este despeje en un tiempo razonable.

Sin embargo, Malcom puede tener suerte, y con suerte me refiero a una probabilidad de $\frac{n}{p}$, si decide hacer n intentos por encontrar el valor de K. Esto es lo que hace que este intercambio de claves, así como todos los protocolos cuya seguridad depende de DLP, sean protocolos probabilísticos. Sin embargo, como suele pasar que $p \gg n$ entonces $\frac{n}{p} \approx 0$.

1.2. Polinomios

Vamos a introducir brevemente el concepto de polinomios sobre un cuerpo finito $\mathbb{F}_p$. Decimos que $\mathbb{F}_p[x]$ es el anillo de polinomios sobre la variable libre x, compuesto por todos los símbolos de la forma

$$a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_m \cdot x^m$$

donde m es un entero no negativo y $\forall\ 0 \leq i \leq m,\ a_i \in \mathbb{F}_p$. El **grado** de un polinomio p es el mayor exponente del mismo, es decir, el mayor k tal que $a_k \neq 0$. Lo notamos $deg(p)$.

Dados 2 polinomios $p, q \in \mathbb{F}_p[x]$ tales que

$$p(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$$

y

$$q(x) = b_0 + b_1 \cdot x + ... + b_m \cdot x^m :$$

- Decimos que $p(x) = q(x)$ sii todos sus coeficientes son iguales, es decir si $\forall\ 0 \leq i \leq m,\ a_i = b_i$.
- La suma de $p(x)$ y $q(x)$ es igual a

  $$p(x) + q(x) = a_0 + b_0 + (a_1 + b_1) \cdot x + ... + (a_m + b_m) \cdot x^m + ... + a_n \cdot x^n$$

  asumiendo sin pérdida de generalidad que $m < n$. Es importante notar que la operación de suma $a_i + b_i$ es sobre el cuerpo $\mathbb{F}_p$.
- El producto de $p(x)$ y $q(x)$ se obtiene a través de la multiplicación formal de sus símbolos, distribuyendo la suma. De esto se deduce que que el grado resultante es igual a la suma de los grados de p y q. Nuevamente notamos que la operación de producto $a_i \cdot b_j$ es sobre el cuerpo $\mathbb{F}_p$.

  Por ejemplo, si tomamos $p(x) = 1 + x - x^2$ y $q(x) = 2 + x^2 + x^3$, obtenemos que $p(x) \cdot q(x) = 2 + 2 \cdot x - x^2 + 2 \cdot x^3 - x^5$.

Dado un polinomio $p(x)$ podemos definir funciones asociadas a él. Por ejemplo, podemos tomar una función $f(x) : \mathbb{F}_p \to \mathbb{F}_p$ sobre el polinomio mencionado. Es importante esta distinción entre el polinomio como objeto algebraico y la función asociada sobre un dominio.

Las raíces de un polinomio p sobre el dominio $\mathbb{F}_p$ son aquellos valores $x^* \in \mathbb{F}_p$ en donde la función asociada a p se anula, es decir $p(x^*) = 0$.

Propiedad 1: Sean $K \subset \mathbb{F}_p$ el multiconjunto de raíces de $p \in \mathbb{F}_p[x]$ y $k \in K$, entonces el polinomio resultante de $p(x)/(x - k)$ tiene como raíces a $K - \{k\}$. En otras palabras, preserva al resto de sus raíces.

Propiedad 2: Sean $p, q \in \mathbb{F}_p[X]$ polinomios mónicos tal que $deg(p) = deg(q) = n$. Si el multiconjunto de raíces de p es el mismo que el de q entonces $p = q$.

1.2.1. Interpolación

En numerosas ocasiones nos va a interesar obtener el **polinomio interpolador** de una lista de elementos de $\mathbb{F}_p$ sobre un dominio determinado. Esto quiere decir que dado un dominio $D = \{d_0, d_1, d_2, ..., d_n\}$ y una serie de valores $p_0, p_1, p_2, ..., p_n \in \mathbb{F}_p$, nos interesa hallar el polinomio cuya función asociada $f : \mathbb{F}_p \to \mathbb{F}_p$ cumple que

$$f(d_i) = p_i \ \forall \ 0 \leq i \leq n$$

No me voy a adentrar en profundidad en los mecanismos para obtener dicho polinomio, pero me interesa hablar de algunas de sus propiedades:

- **Existencia**: existe al menos un polinomio interpolador para cualquier conjunto de puntos.
- **Unicidad**: dados $n + 1$ puntos, el polinomio de grado a lo sumo n que lo interpola es único.

1.2.2. Lema de Schwartz-Zippel

El Lema de Schwartz-Zippel es una herramienta matemática usada en las Pruebas de Identidad de Polinomios. Estas van a resultar centrales para los proving systems y será usado numerosas veces en diversos protocolos, dándoles a su vez la caracterización de pruebas probabilísticas.

Sea p un primo grande, dos polinomios $P, Q \in \mathbb{F}_p[X]$ de grado n, y $k \in \mathbb{F}_p$ un valor aleatorio tomado de una distribución uniforme de $\mathbb{F}_p$. Si las funciones asociadas a P y Q en $\mathbb{F}_p$ ($p(x)$ y $q(x)$ respectivamente) cumplen que $p(k) = q(k)$ entonces la probabilidad de que P y Q sean distintos es de $\frac{n}{p}$. En un entorno donde $p \gg n$ (por ejemplo $p > 2^{200}$ y $n = 2^{32}$) podemos decir que si $p(k) = q(k)$ entonces es altamente probable que P y Q sean iguales, ya que $1 - \frac{2^{32}}{2^{200}} \approx 1$.

La intuición detrás de esto está dada por la propiedad de los polinomios de grado n tienen como mucho $n - 1$ puntos críticos, por lo tanto 2 polinomios con coeficientes en $\mathbb{F}_p$ se pueden cruzar a lo sumo en n puntos. Podemos ver una intuición de esto en la figura 1.1.

Pensemos por un segundo qué pasaría si el valor elegido no fuera aleatorio, o más bien si el valor fuera elegido maliciosamente por un agente que conoce P y Q. Supongamos también que $P \neq Q$. Si este agente conociera un valor k^* tal que $p(k^*) = q(k^*)$ (alguno de los puntos donde los polinomios intersecan) podría alegar que $P = Q$ a los ojos de un protocolo que usa el lema de Schwartz-Zippel, dando lugar a una vulnerabilidad. Más adelante veremos cómo se resuelve esta obtención de números aleatorios en el contexto de los protocolos ZK. Para que este protocolo sea seguro se pide que $p \gg n$.

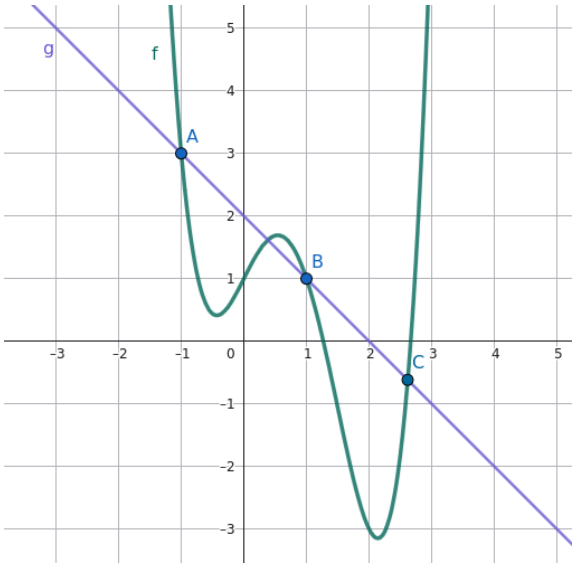
1.3. Curvas elípticas

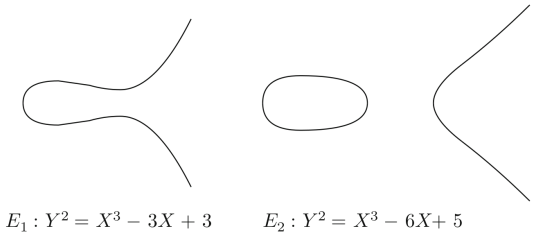
Una curva elíptica es el conjunto de soluciones (x, y) a una ecuación de la forma

$$y^2 = x^3 + Ax + B$$

donde A y B son constantes. Dos ejemplos de curva elíptica pueden verse en la figura 1.2. Un **punto** en una curva elíptica es un par ordenado (x, y) que cumple con la ecuación

Fig. 1.1: Intuición Schwartz-Zippel





$E_1 : Y^2 = X^3 - 3X + 3$ $E_2 : Y^2 = X^3 - 6X + 5$

Fig. 1.2: Dos ejemplos de curvas elípticas en el eje cartesiano de los reales

de la curva. Las curvas elípticas, al igual que los cuerpos finitos como $\mathbb{F}_p$, van a ser herramientas muy útiles para los protocolos criptográficos que siguen. También, tienen algunas propiedades similares como vamos a ver a continuación.

Operaciones básicas

Vamos a explicar la **suma** de 2 puntos de curva elíptica a través de su intuición geométrica. Sean $P = (x_p, y_p)$ y $Q = (x_q, y_q)$ dos puntos sobre la curva. Primero proyectemos una recta que corta a ambos puntos. La naturaleza de la ecuación nos garantiza que siempre existe un tercer punto sobre la curva que interseca con esta recta, llamado $R = (x_r, y_r)$. Lo siguiente es reflejar este punto sobre el eje x, obteniendo $R' = (x_r, -y_r)$. Finalmente definimos

$$P \oplus Q = R'$$

Una representación visual de la operación realizada puede verse en la figura 1.3.

Un caso especial para la suma es en el cual queremos sumar un punto P consigo mismo. En esa situación, alcanza con tomar la recta tangente a P y la ecuación nos garantiza que siempre existe un segundo punto de la curva sobre esta recta. Un ejemplo se puede ver en la figura 1.4.

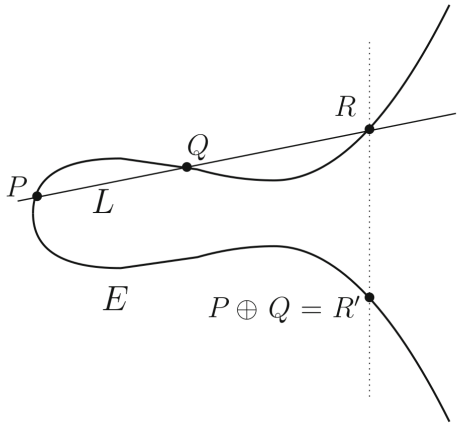


Fig. 1.3: Suma de P y Q en una curva elíptica.

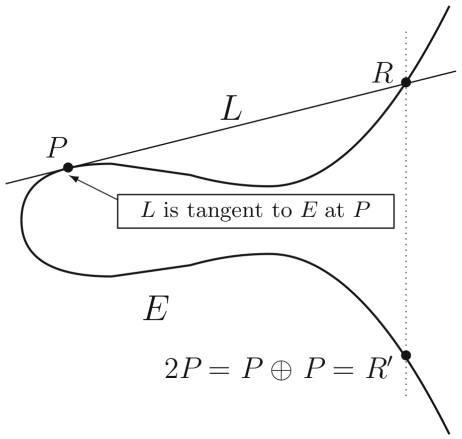


Fig. 1.4: Suma de P consigo mismo en una curva elíptica

Hay otro caso especial en el cual queremos sumar a un punto $P = (a, b)$ con su punto espejado en el eje x, es decir $P' = (a, -b)$. La recta definida entre estos puntos es paralela al eje vertical pero P y P' son los únicos puntos que tienen en común esta recta y la curva elíptica. ¿Dónde está el tercer punto? La solución es crear un tercer punto, un *punto en el infinito* que no existe en el plano pero sí en el infinito de toda linea vertical. Finalmente, definimos que

$$P \oplus P' = \infty$$

Paralelamente, definimos la suma de cualquier punto P con ∞ como

$$P + \infty = P$$

ya que si trazamos la recta entre P y ∞, el tercer punto intersecado es P', por lo tanto su punto espejado vuelve a ser P. Con este punto en el infinito, vamos a definir nuevamente una curva elíptica.

Definición. Una curva elíptica E es el conjunto

$$E = \{(x, y) \mid y^2 = x^3 + Ax + B\} \cup \{\infty\} \quad A, B \in \mathbb{Z}$$

Los valores A y B deben cumplir que $4A^3 + 27B^2 \neq 0$. Esta condición se debe a que si factorizamos $X^3 + AX + B$ como $y = (x - x_0)(x - x_1)(x - x_2)$ entonces E no tiene raíces repetidas ($x_o \neq x_1 \neq x_2$) sii $4A^3 + 27B^2 \neq 0$.

Sea E una curva elíptica, la **multiplicación** entre un punto de curva $P \in E$ y un $n \in \mathbb{N}$ se define como la suma sucesiva de P consigo mismo n veces, es decir:

$$nP = \underbrace{P + P + \cdots + P}_{n \text{ veces}}$$

Curvas Elípticas como grupos conmutativos

Sea E una curva elíptica. La operación de **suma** ($\oplus$) cumple con las propiedades:

- **Clausura:** $P + Q \in E \quad \forall\ P, Q \in E$
- **Elemento neutro:** $P + \infty = \infty + P = P \quad \forall\ P \in E$
- **Elemento inverso:** $P + (-P) = \infty \quad \forall\ P \in E$
- **Asociatividad:** $(P + Q) + R = P + (Q + R) \quad \forall\ P, Q, R \in E$
- **Conmutatividad:** $P + Q = Q + P \quad \forall\ P, Q \in E$

En otras palabras, dada la definición de la sección 1.1, podemos decir que la tupla $(E, +)$ es un grupo conmutativo.

La **resta** de dos puntos de curva P y Q podemos pensarla como la suma del inverso aditivo, es decir, dados $P, Q \in E$, decimos que $P - Q = P + (-Q)$ siendo $-Q$ el punto espejado a Q en el eje horizontal.

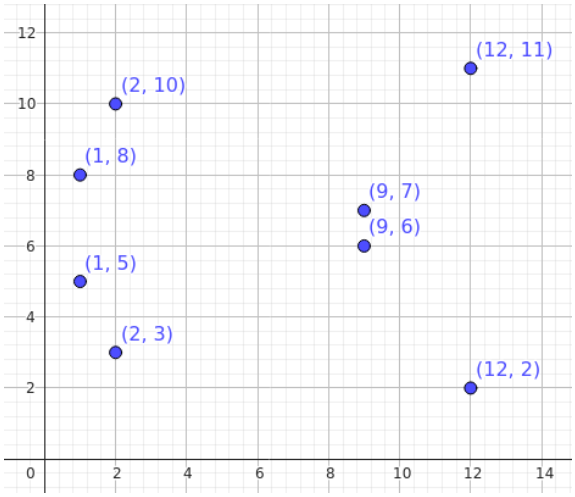


Fig. 1.5: Visualización de $E(\mathbb{F}_{13})$

1.3.1. Curvas elípticas sobre cuerpos finitos

Para usar la teoría de curvas elípticas en el contexto de protocolos criptográficos tenemos que pensarlas como conjuntos de puntos con coordenadas en $\mathbb{F}_p$. Alteramos levemente la definición:

Definición. Sea $p \geq 3 \in \mathbb{N}$. Una curva elíptica sobre $\mathbb{F}_p$ es un conjunto

$$E(\mathbb{F}_p) = \{(x, y) : x, y \in \mathbb{F}_p \mid y^2 = x^3 + Ax + B\} \cup \{\infty\}$$

Dado un p fijo y una ecuación de curva elíptica E, para obtener los puntos de la curva debemos recorrer todos los valores $x = \{0, 1, \cdots, p-1\}$ y ver para cuáles de ellos, $E(x)$ es un cuadrado en $\mathbb{F}_p$ (es decir, que exista un y tal que $y^2 = E(x)$). Si esto ocurre, decimos que $(x, y) \in E(\mathbb{F}_p)$.

Por ejemplo, si tomamos $p = 13$ y $E : y^2 = x^3 + 3x + 8$, el conjunto de puntos que obtendríamos sería $E(\mathbb{F}_{13}) = \{\infty, (1, 5), (1, 8), (2, 3), (2, 10), (9, 6), (9, 7), (12, 2), (12, 11)\}$. Podemos ver una representación visual en la figura 1.5. Definimos al **orden** de la curva elíptica como la cantidad de puntos que la componen, incluyendo al punto en el infinito. Por ejemplo, la curva recién mencionada tiene orden 9.

Notamos que en este caso no podemos usar una intuición geométrica para sumar 2 puntos de curva, sin embargo podemos derivar el algoritmo necesario para calcularla, como se puede ver en el algoritmo 1. Se puede demostrar que el punto obtenido también forma parte de la curva, en otras palabras,

$$P_1 \in E(\mathbb{F}_p) \wedge P_2 \in E(\mathbb{F}_p) \implies P_1 + P_2 \in E(\mathbb{F}_p)$$

También podemos afirmar que $(E(\mathbb{F}_p), +)$ es un grupo conmutativo, ya que cumple con todas las propiedades requeridas.

Definición. Dado un punto $P \in E(\mathbb{F}_p)$, definimos el **orden** de P como el menor $s > 1 \in \mathbb{N}$ tal que $sP = \infty$.

El orden de P siempre existe ya que la cantidad de puntos en $P \in E(\mathbb{F}_p)$ es finita, por lo tanto si tomamos los valores $P, 2P, 3P, \cdots$ necesariamente vamos a tener valores repetidos. Sean $i > j \in \mathbb{N}$ los valores más chicos tales que $jP = iP$, tomamos $s = i - j$. Por la propiedad 3, sabemos que s divide al orden de E.

Algorithm 1 Suma de puntos de curva elíptica

Entrada: $P_1 : E$, $P_2 : E$
Salida: $P_1 + P_2$
if $P_1 = \infty$ **then**
    return P_2
end if
if $P_2 = \infty$ **then**
    return P_1
end if
$P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$
if $x_1 == x_2$ **y** $y_1 == -y_2$ **then** // Son puntos espejados en el eje horizontal
    return ∞
else
    if $P_1 \neq P_2$ **then** // Calcular la recta que pasa por ambos puntos
        $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$
    else // Calcular la recta tangente
        $\lambda = \frac{3x_1^2 + A}{2y_1}$
    end if
    $x_3 = \lambda^2 - x_1 - x_2$
    $y_3 = \lambda(x_1 - x_3) - y_1$
    return (x_3, y_3)
end if

1.3.2. Problema del logaritmo discreto para curvas elípticas

Recordamos que el problema del logaritmo discreto de la sección 1.1.1 está definido sobre un cuerpo finito $\mathbb{F}_p$. Queremos algo similar para $E(\mathbb{F}_p)$ que también sirva a nuestros propósitos.

Definición. Sea E una curva elíptica sobre $\mathbb{F}_p$ y sean $P, Q \in E(\mathbb{F}_p)$. El problema **ECDLP** (*Problema del Logaritmo Discreto para Curvas Elípticas*) es el problema de encontrar un $n \in \mathbb{N}$ tal que

$$Q = \underbrace{P + P + \cdots + P}_{\text{n veces}} = nP$$

Notamos $n = log_P(Q)$. Puede ocurrir que el problema no esté definido para algunos pares $P, Q \in E(\mathbb{F}_p)$, ya que Q no necesariamente es un múltiplo de P. En la práctica, un agente de un protocolo criptográfico tomará un punto P público y un valor secreto n y computará $Q = nP$, por lo tanto el problema estará bien definido.

La dificultad ECDLP es similar a la del DLP para grupos genéricos, en el sentido de que dados $P, Q \in E(\mathbb{F}_p)$ conocidos, el mejor algoritmo conocido que computa $n = log_P(Q)$ es equivalente a sumar P sucesivamente a sí mismo hasta alcanzar el valor de Q. Sin embargo, hay algunas curvas elípticas conocidas para las cuales existen algoritmos más eficientes para resolver ECDLP.

1.3.3. BLS y Pairings

Paralelamente, también existen curvas elípticas más seguras y con propiedades deseables, como puede ser la curva BLS12-385 [9]. Esta curva está definida como

$$E(\mathbb{F}_p) : Y^2 = X^3 + 4 \qquad p = 2^{255} - 13$$

Esta curva posee la propiedad de que calcular ***pairings*** es una operación poco costosa respecto a otras curvas. Ahora bien, ¿qué son los pairings? No voy a presentar una definición formal, más bien enunciar algunas de las propiedades que nos interesan en el contexto de los SNARKs.

Sea $E(\mathbb{F}_p)$ una curva elíptica, un pairing es una función e que cumple lo siguiente:

- $e : E(\mathbb{F}_p) \times E(\mathbb{F}_p) \to \mathbb{F}_p$.
- $e(P, Q)$ puede computarse de manera eficiente.
- $e(a \cdot P, b \cdot Q) = e(P, Q)^{a \cdot b} \quad \forall\ a, b \in \mathbb{F}_p$, propiedad conocida como **bilinearidad**.

La definición de pairing es más general, pero esto es todo lo que nos va a interesar en el contexto de los SNARKs.

2. PROVING SYSTEMS

Llegó el momento de hablar de uno de los temas centrales de la tesis: los **sistemas de pruebas** o *proving systems* en inglés. Vamos a mencionar en particular una familia de protocolos llamados **SNARKs** (**S**uccint **N**on-Interactive **AR**gument of **K**nowledge) que sirven para demostrar la "ejecución" de un programa con determinados inputs.

La idea general es modelar a un programa cualquiera con matrices. Algunas de estas matrices van a representar al programa en sí y otras van a representar la ejecución concreta del programa modelado. A estas últimas las llamamos la **traza** del programa. Luego tomaremos las columnas de estas matrices de forma independiente y las interpolaremos sobre un dominio específico para obtener un conjunto de polinomios que caractericen al programa y a la ejecución. Luego de eso, vamos a comprometernos a estos polinomios ante un observador externo usando algún **Polynomial Commitment Scheme** y con esta información, aprovechándonos de la heurística de Fiat-Shamir tendremos una prueba de tamaño reducido que contiene toda la información encriptada de la ejecución del programa. Esta está armada para ser verificada a través de operaciones rápidas sobre cuerpos finitos. Esta verificación será de carácter probabilístico, lo cual quiere decir que habrá una probabilidad despreciable de que se demuestre correctamente una ejecución incorrecta o que falle la demostración de una ejecución correcta.

La mayoría de los conceptos mencionados (como los Polynomial Commitment Schemes o la heurística de Fiat-Shamir) serán tratados a continuación. Mi intención es descomponer poco a poco el funcionamiento de esta familia de protocolos, hablar de sus propiedades y dar un ejemplo: **PLONK**.

2.1. Provers y verifiers

En general en estos protocolos vamos a tener 2 agentes representados por programas informáticos: un **prover** y un **verifier**. El prover es quien ejecuta un programa, genera una prueba de su ejecución y se la pasa al verifier para que la verifique y se convenza de que la prueba es correcta. El prover y el verifier deberían tener un setup para ponerse de acuerdo en un conjunto de elementos. Estos elementos dependen del protocolo y suelen estar comprimidos dentro de una **Verifiying Key**. Algunos de estos elementos son comunes entre protocolos:

- Un cuerpo finito sobre el cual operar, es decir, un primo p suficientemente grande para dar seguridad al protocolo que será el grado de $\mathbb{F}_p$. En general el cuerpo finito usado depende del sistema de pruebas elegido.
- Un generador de $\mathbb{F}_p$ al cual vamos a llamar g.
- Un modelo del programa (lo que coloquialmente llamamos 'circuito') que depende del sistema de pruebas y la forma de aritmetización, algo que vamos a ver más adelante. Es intuitivo pensar que ambas partes deben conocer el programa cuya ejecución se está demostrando, ya que una prueba no tendría sentido de otro modo.

2.2. Protocolo de Schnorr

Antes de entrar de lleno en la descripción de un protocolo complejo como PLONK, me gustaría introducir un protocolo interactivo más simple pero que sirve para poner en práctica algunos de los conceptos que mencionamos con anterioridad, como los cuerpos finitos y la idea de un agente que prueba algo y otro que lo verifica. El Protocolo de Schnorr permite a un prover comprometerse a un valor en $\mathbb{F}_p$ ante un verifier.

Sean los valores conocidos tanto por el prover como por el verifier:

- p un primo suficientemente grande
- $\mathbb{F}_p$ un cuerpo finito
- g un generador de $\mathbb{F}_p$ tal que $\mathbb{F}_p = \{g^i : \ 0 \leq i < p\}$

El protocolo se dispone de la siguiente forma:

Round 1 (commitment): el prover genera un valor privado $a \in \mathbb{F}_p$ y computa $A = g^a$. A continuación le envía A al verifier. Notamos que gracias a la dificultad del problema del logaritmo discreto, el verifier no puede deducir a ni aún conociendo A y g.

Round 2 (random sampling): El prover elige un valor aleatorio $r \in \mathbb{F}_p$ y computa $R = g^r$. A continuación le envía R al verifier. Este valor aleatorio r tampoco puede ser despejado por el verifier en un tiempo razonable.

Round 3 (challenge): el verifier elige un valor $c \in \mathbb{F}_p$ y se lo envía al prover.

Round 4: el prover calcula $s = r + a \cdot c$ y se lo envía al verifier.

Round 5 (verificación): el verifier comprueba la igualdad $R \cdot A^c = g^s$ para convencerse de que el prover conoce un a tal que $g^a = A$. Notemos que el verifier puede realizar esta cuenta porque conoce todos los valores involucrados: R fue recibido en el round 2, A fue recibido en el round 1, c fue elegido por el verifier en el round 3 y s fue recibido en el round 4.

Vamos a descomponer esta última operación:

$$R \cdot A^c = g^s \Leftrightarrow$$

$$g^r \cdot (g^a)^c = g^s \Leftrightarrow$$

$$(\text{como } s = r + a \cdot c)$$

$$g^r \cdot g^{a \cdot c} = g^{r + a \cdot c} \Leftrightarrow$$

$$g^{r + a \cdot c} = g^{r + a \cdot c}$$

Más allá del detalle de las cuentas, tratemos de entender lo que está pasando. El prover conoce una pieza de información privada (a) y ambos conocen una pieza de información pública (A). El prover quiere demostrarle al verifier que conoce a tal que $A = g^a$, pero no quiere revelar ninguna información sobre a más que la que ya es de público conocimiento. Para esto, el round 2 introduce un factor de aleatoriedad que sirve para ocultar el valor de a en la verificación. El verifier quiere convencerse de que el prover conoce a sin ser engañado. Es por eso que el round 3 introduce un challenge por parte del verifier. Este challenge no puede ser conocido de antemano por el prover, ya que le permitiría generar una prueba de conocimiento de a sin conocer a verdaderamente.

La pregunta que cabe hacernos entonces es **¿pudo el prover haber generado un valor s tal que la verificación se cumpla sin conocer a?** Veamos primero el caso mencionado anteriormente donde el prover conoce de antemano el challenge c y puede hacer trampa. En este escenario, el prover generó un valor A que no es el resultado de ninguna cuenta y desconoce el valor de a, sin embargo quiere hacer creer al verifier que conoce un a tal que $A = g^a$. El objetivo del prover es que verifique la ecuación $R \cdot A^c = g^s$ y como en este escenario conoce c, también conoce A^c. Luego, puede tomar un valor de s cualquiera, llamado s^* y calcular $S = g^{s^*}$. Finalmente, solo tiene que despejar el valor de R para que se cumpla la ecuación. Concretamente, pasaría que

$$R \cdot A^c = g^s$$

A^c es conocido por el prover, fijamos $s := s^*$ y calculamos $S := g^{s^*}$ para obtener

$$R \cdot A^c = S$$

por lo tanto tomamos

$$R := S \cdot \frac{1}{A^c}$$

y la ecuación se verifica, engañando de esa manera al verifier.

Dicho esto, el protocolo impide que el prover conozca el challenge de antemano, en particular no conoce el challenge al momento de comprometerse a los valores de A y R. Esta temporalidad impide hacer este truco, ya que por el orden de los rounds el verifier conoce el valor de R antes de mandar el challenge c, y por ende el prover no puede hacer cuentas previas en función de c.

Se llama **Soundness** a la propiedad de un protocolo criptográfico que dicta que un verifier honesto no puede ser convencido por un prover deshonesto de que una afirmación falsa es verdadera. En este caso, el protocolo de Schnorr tiene soundness, ya que el prover no puede demostrar al verifier la afirmación "conozco a tal que $A = g^a$".

¿Qué pasa entonces con el ocultamiento de a?¿Es posible para el verifier deducir el valor de a con los datos provistos por el prover? El verifier conoce A, pero no puede deducir a a partir de ello por la dificultad del problema del logaritmo discreto. Por otro lado, conoce R, pero tampoco puede calcular r a partir de ese dato por el mismo motivo. Conoce c porque él mismo lo generó, pero eso no le aporta información de por sí. Por último, conoce $s = r + a \cdot c$, sin embargo no conoce r, entonces no puede deducir a con esa información (para cada valor fijo de r correspondería un valor de a distinto). Acá es donde se ve la importancia de la aleatoriedad introducida en el round 2, sin la cual este protocolo estaría revelando información confidencial (a).

Se llama **Zero Knowledge** a la propiedad de un protocolo criptográfico que dicta que el verifier no aprende nada más que la afirmación que se está probando. Esta es una categorización coloquial que alcanza por el momento, para no adentrarnos en definiciones más complejas. En este caso podemos decir que el prover pudo demostrar al verifier el conocimiento de a sin revelar ningún tipo de información nueva sobre a. La ecuación $s = r + a \cdot c$ no aporta ningún tipo de información, ya que r es un valor aleatorio, convirtiendo a s en otro valor aleatorio.

2.3. Propiedades de los SNARKS

Cuando hablamos de SNARKs, no es enteramente correcto hablar de "demostrar la ejecución de un programa" aunque sea una frase que coloquialmente usamos con frecuencia.

En la realidad lo que ocurre es que el prover **demuestra que conoce la traza de una ejecución válida del programa**. Es decir, no es necesario que el prover ejecute el programa en cuestión, aunque en general esa es la forma de obtener dicha traza. Por ahora podemos pensar en la traza como los valores que toman todas sus variables a lo largo de su ejecución. Dado un programa fijo, la traza define inequívocamente su ejecución.

Veamos entonces como se descompone el término SNARK:

- **Succint**: la traducción más correcta de esta palabra sería 'conciso'. La propiedad de succintness de los SNARKs nos dice que las pruebas generadas tienen que tener un tamaño pequeño respecto al de la ejecución o 'traza' del programa cuya ejecución demuestra. También nos dice que la verificación de dicha prueba tiene que ser rápida, por ejemplo, en tiempo constante $\mathcal{O}(1)$. Esto no establece ninguna limitación sobre el tiempo de generación de una prueba, o sobre el overhead de almacenamiento que generar dicha prueba conlleva.

- **Non-interactive**: esta propiedad nos habla de la ausencia de interactividad en el protocolo, es decir, una prueba puede ser generada de forma offline y luego ser verificada en cualquier momento del futuro. La heurística de Fiat-Shamir que será vista en la sección 2.4 es lo que permite que estos protocolos no sean interactivos.

- **ARgument of Knowledge**: esto hace referencia a la pieza de conocimiento que tiene el prover, cuya posesión quiere demostrarle a un verifier.

Una pregunta que puede surgir es: si el objetivo de un SNARK es que un prover demuestre que conoce una traza válida de ejecución de un programa, ¿por qué simplemente la demostración no es la traza en sí? o más concretamente, ¿por qué simplemente el verifier no ejecuta el mismo programa? La respuesta a eso es que la verificación en esos caso no sería succint, tanto la ejecución del programa como la verificación elemento a elemento de la traza tendrían un costo mucho más alto de lo que se busca. Sin embargo, también hay otro motivo que tiene que ver con el ocultamiento.

En este punto es importante distinguir entre un SNARK y un ZK-SNARK. Un ZK-SNARK cumple la propiedad de **Zero Knowledge**. Esta propiedad busca que la pieza de conocimiento no sea revelada al verifier, permitiendo sin embargo que este último pueda convencerse de que el prover la posee. Concretamente, esto se puede materializar en la existencia de inputs privados en la ejecución de un programa. Dicho de otra forma y para resumir, un ZK-SNARK permite a un prover demostrar que **conoce una traza válida correspondiente a un programa sin revelar esta traza**.

2.4. Heurística de Fiat-Shamir

La heurística de Fiat-Shamir [10] es un truco usado frecuentemente que permite convertir protocolos interactivos en **no interactivos** usando una función de hash que se comporta como un oráculo aleatorio. Un **oráculo aleatorio** es una función que se comporta como una caja negra y para cada input produce un output aleatorio y uniformemente distribuido de forma determinista. Podemos asumir que una función de hash se comporta como un oráculo aleatorio.

La idea general consiste en tener un *transcript* append-only: un log de todo lo que circularía entre el prover y el verifier si el protocolo fuera interactivo. Cada vez que el prover necesita un *challenge* por parte del verifier, en lugar de recurrir a la interactividad

lo que va a hacer es hashear el contenido del transcript y usar el digest del hash para samplear un valor, que va a ser el reemplazo del challenge. Inmediatamente después, se realiza un append del sample obtenido al final del transcript.

Para que el protocolo sea seguro es necesario inicializar el transcript con un string común, conocido por ambos agentes del protocolo. También notamos que como el transcript solo contiene aquello que viaja del prover al verifier y vice-versa, ambos agentes pueden reconstruirlo. Veamos cómo se vería la heurística de Fiat-Shamir aplicada al protocolo de Schnorr descrito en la sección 2.2:

1. Setup: El prover **inicializa** el transcript con un string común $init$.
2. Round 1: El prover **extiende** el transcript con el valor binario de $A = g^a$.
3. Round 2: El prover **extiende** el transcript con el valor binario de $R = g^r$.
4. Round 3: El prover **calcula el hash** del transcript hasta el momento, que es $init||A||R$, obteniendo así $c = hash(init||A||R)$. **Extiende** el transcript de forma que queda $init||A||R||c$.
5. Round 4: El prover calcula $s = r + a \cdot c$ y extiende el transcript con ese valor, de manera que queda $init||A||R||c||s$.
6. Generación de prueba: El prover debe **enviar al verifier toda la información necesaria para que este pueda replicar sus acciones**. La prueba entonces es $\pi := \{A, R, s\}$.

Luego de todo esto, le toca al verifier replicar todas las acciones que hizo el prover respecto al transcript. Esto es necesario para obtener el valor de c, el cual no forma parte de la prueba π. En otras palabras, el verifier no fue partícipe de la elección del challenge c, pero este fue generado por una función de hash que se comporta como un oráculo aleatorio, lo cual para nuestros fines es suficiente. Entonces, el verifier:

1. **recibe** A, R y s.
2. **inicializa** el transcript con $init||A||R$.
3. **hashea** el contenido del transcript para obtener $c = hash(init||A||R)$.
4. **verifica** la ecuación $R \cdot A^c = g^s$ igual que en el protocolo original.

Las ventajas de usar un protocolo no interactivo son evidentes. Un prover puede generar una prueba en cualquier momento y entregarla al verifier cuando desee. Esto no requiere una conectividad sincrónica entre ambos agentes, y por sobre todas las cosas, si lo aplicamos a protocolos SNARK nos da la propiedad de **Non-interactive** mencionada en la sección 2.3.

2.5. PLONK

Dijimos anteriormente que para demostrar la ejecución de un programa primero tenemos que tener ese programa modelado como un sistema de ecuaciones que tanto el prover como el verifier tienen que conocer. Demostrar su ejecución es equivalente a demostrar

que conocemos una solución para este sistema de ecuaciones. La forma que este sistema tiene depende del proving system, y a continuación vamos a presentar uno que resultó muy influyente en el área: **PLONK**. Las siglas de PLONK no son importantes ya que son un retroacrónimo. PLONK es un protocolo que provee principalmente 2 cosas:

1. Una convención para expresar programas como sistemas de ecuaciones.
2. Una forma de tomar estos programas y sus trazas y convertirlos en algo útil para demostrar su conocimiento.

2.5.1. Matrices de PLONK

Para comenzar, vamos a centrarnos en el modelado del programa a través de matrices y sistemas de ecuaciones. Partamos de un programa de ejemplo P con las siguientes características:

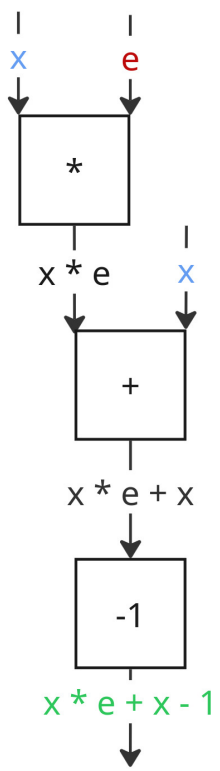
- **Public Input:** x
- **Private Input:** e
- **Cómputo:** $e * x + x - 1$

En la familia de programas que vamos a tratar a partir de ahora vamos a hacer una distinción entre aquellos inputs que son públicos (conocidos tanto por el prover como por el verifier) como los privados (conocidos solo por el prover). Estamos trabajando con un caso sencillo, pero un caso de uso real podría ser uno donde el input publico es el resultado de un hash, el input privado es la preimagen de ese hash y el cómputo está definido como $hash(e) == x$. Ese es un caso donde tiene sentido ocultar el valor de e y donde es no trivial obtenerlo por medios propios.

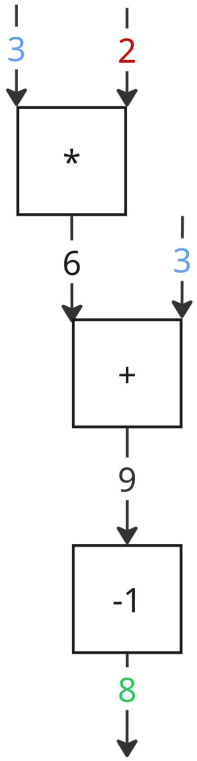
Otra forma útil de ver los programas es la de un circuito binario. En este caso para el programa mencionado sería algo como lo que se ve en la figura 2.1a. En esta representación, cada caja es una operación, los inputs entran por encima y el output sale por abajo. Notemos que como el -1 es una constante en nuestro programa, en el circuito lo representamos como una operación unaria en sí misma. Esta representación es ideal ya que en la aritmetización que propone PLONK vamos a representar a un programa como una serie de compuertas, con un operando izquierdo, uno derecho y un output. En nuestro caso tenemos 3 compuertas: una de multiplicación, una de suma, y una de suma constante.

En la práctica lo que vamos a tener son valores concretos para estas variables, en este caso el circuito estaría completo como se puede ver en la figura 2.1b.

Para modelar nuestro programa de ejemplo, PLONK usa **matrices**. La primera matriz que vamos a mencionar es la matriz T: la **Traza**. En nuestro caso esta sería la que vemos en la tabla 2.1. En ella podemos ver cómo cada fila de esta matriz se corresponde con una de las compuertas mencionadas anteriormente. En todos los casos la columna A es el operando izquierdo, B el derecho y C el output. Por el momento no nos vamos a preocupar por el hecho de que no existe un segundo operando para la tercera operación, el símbolo $-$ quiere decir que puede ir cualquier valor en ese lugar, y que este no será tomado en cuenta. La traza plasmada en la tabla 2.1 es válida para nuestro programa, sin embargo no todas las posibles trazas lo son. Entonces ¿qué significa que sea válida para el protocolo? La pregunta surge porque no tenemos modelado nuestro programa todavía.



(a) Representación en forma de circuito del programa de ejemplo



(b) Representación en forma de circuito del programa de ejemplo, con valores concretos

Fig. 2.1: Programa de ejemplo, con y sin valores concretos

A	B	C
2	3	6
6	3	9
9	-	8

Tab. 2.1: Matriz T del programa de ejemplo.

Para empezar a modelar nuestro programa de juguete vamos a necesitar 2 matrices: Q y V. Estas matrices son públicas tanto para el prover como para el verifier, y nos definen el programa en su totalidad.

La matriz Q

La matriz Q tiene 5 columnas: Q_L, Q_R, Q_M, Q_O, Q_C. Con estas columnas podemos codificar todas las compuertas habilitadas por PLONK y están diseñadas para relacionarse con la traza bajo la siguiente ecuación:

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0 \quad \forall i$$

Cada una de las columnas tiene un rol, como se puede deducir de la ecuación:

- Q_L es un valor que multiplica al operando izquierdo (A en la traza)
- Q_R es un valor que multiplica al operando derecho (B en la traza)
- Q_M es un valor que multiplica al producto entre el operando izquierdo y el derecho.
- Q_O es un valor que multiplica al output. En general este valor va a ser negativo porque queremos que operaciones sobre los inputs menos el output sea igual a 0.
- Q_C es un valor constante que se suma al resto de los términos.

Compuerta de suma

Antes de analizarla en detalle veamos un ejemplo de qué pinta podría tener una fila de Q, en particular una que defina la compuerta de suma entre 2 inputs. Esta se puede ver en la tabla 2.2.

Q_L	Q_R	Q_M	Q_O	Q_C
1	1	0	-1	0

Tab. 2.2: Fila de la matriz Q con operación de suma.

¿Por qué esta disposición nos define una compuerta de suma? Veamos que pasa cuando reemplazamos los valores en la ecuación:

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0$$

$$1 \cdot A_i + 1 \cdot B_i + 0 \cdot A_i \cdot B_i + (-1) \cdot C_i + 0 = 0$$

$$1 \cdot A_i + 1 \cdot B_i + (-1) \cdot C_i = 0$$

$$A_i + B_i = C_i$$

Compuerta de multiplicación

Veamos ahora qué elección de valores resultaría en una compuerta de multiplicación. El resultado es el que se puede ver en la figura 2.3. Paralelamente, podemos ver que esto se verifica:

Q_L	Q_R	Q_M	Q_O	Q_C
0	0	1	-1	0

Tab. 2.3: Fila de la matriz Q con operación de multiplicación.

Q_L	Q_R	Q_M	Q_O	Q_C
1	0	0	-1	-1

Tab. 2.4: Fila de la matriz Q con operación de suma constante del valor -1.

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0$$
$$0 \cdot A_i + 0 \cdot B_i + 1 \cdot A_i \cdot B_i + (-1) \cdot C_i + 0 = 0$$
$$1 \cdot A_i \cdot B_i + (-1) \cdot C_i = 0$$
$$A_i \cdot B_i = C_i$$

Compuerta de suma constante

Por último, vamos a ver cómo sería una compuerta para sumar una constante a un número. El resultado es el que se puede ver en la figura 2.4. Veamos nuevamente que se cumple:

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0$$
$$1 \cdot A_i + 0 \cdot B_i + 0 \cdot A_i \cdot B_i + (-1) \cdot C_i - 1 = 0$$
$$1 \cdot A_i + (-1) \cdot C_i - 1 = 0$$
$$A_i - 1 = C_i$$

El programa completo

Podemos ver entonces que nuestro programa completo está dado por la matriz Q que se puede ver en la tabla 2.5. La primera linea se corresponde con un producto, la segunda con una suma y la tercera con un suma constante de -1.

Q_L	Q_R	Q_M	Q_O	Q_C
0	0	1	-1	0
1	1	0	-1	0
1	0	0	-1	-1

A	B	C
2	3	6
6	3	9
9	-	8

Tab. 2.5: Matriz Q y matriz T del programa de ejemplo.

La matriz V

Es trivial ver en la figura 2.5 cómo en cada fila independiente se cumple la ecuación

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0 \quad \forall i$$

Sin embargo, ¿alcanza Q para definir el programa que estamos queriendo representar? Veamos la traza alternativa de la figura 2.6. En el ejemplo se puede ver cómo cada fila cumple con su propia compuerta, pero visto como un todo esa traza no tiene sentido. No es suficiente con que cada fila en sí misma sea consistente con una compuerta. Adicionalmente,

Q_L	Q_R	Q_M	Q_O	Q_C
0	0	1	-1	0
1	1	0	-1	0
1	0	0	-1	-1

A	B	C
2	3	6
10	10	20
3	-	2

Tab. 2.6: Matriz Q del programa de ejemplo, con una traza inválida.

A	B	C
2	3	6
6	3	9
9	-	8

Tab. 2.7: Matriz T del programa de ejemplo con consistencias resaltadas.

necesitamos alguna forma de dar **consistencia entre las filas**. Necesitamos la capacidad de expresar que **el output de una compuerta tiene que tener el mismo valor que el input de otra** o que dos operandos pueden provenir de la misma variable y por ende deben tener el mismo valor. En otras palabras, nos gustaría tener una consistencia como la que se ve en la tabla 2.7.

En este contexto entra en juego la matriz **V**. Esta nos permite ´´nombrar" operandos y volverlos a referenciar más adelante. Estas matrices tienen la forma que se ve en la tabla 2.8. Los casilleros que tienen el mismo número en la matriz V deberán tener el mismo valor en la matriz T (la traza). Finalmente con esta nueva matriz podemos definir la validez de una traza respecto a un programa expresado como matrices de PLONK.

L	R	O
1	2	3
3	2	4
4	-	5

Tab. 2.8: Matriz V del programa de ejemplo.

Definición. Sea T una traza con columnas A, B y C. Esta refleja una evaluación del circuito dado por las matrices Q y V si y solo si:

1. Se satisface la ecuación

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0 \quad \forall i$$

2. $\forall i, j, k, l,\ V_{i,j} = V_{k,l} \implies T_{i,j} = T_{k,l}$

Este segundo chequeo impide que el ejemplo de la traza en 2.6 no sea válido, dada la matriz V de 2.8.

Compuertas personalizadas

Hasta ahora vimos algunas configuraciones que pueden tomar las filas de la matriz Q, en particular para operaciones básicas como sumar y multiplicar. Sin embargo, PLONK nos permite definir compuertas personalizadas dándole una forma arbitraria a la matriz Q para conseguir la operación deseada. En particular, vemos que podemos comprimir todo nuestro programa de ejemplo en una sola fila de la matriz Q, quedándonos de esta manera un conjunto de matrices como se puede ver en la tabla 2.9. Podemos hacer esto porque hasta ahora no estábamos aprovechando al máximo la expresividad que poseen las ecuaciones de PLONK, que nos permiten describir nuestro programa por completo en una sola ecuación.

Q_L	Q_R	Q_M	Q_O	Q_C
0	1	1	-1	-1

L	R	O
0	1	2

A	B	C
2	3	8

Tab. 2.9: Matrices Q, V y T comprimidas del programa de ejemplo.

Recordemos que nuestro programa busca computar $p(x, e) = e \cdot x + x - 1$, veamos que las matrices de 2.9 reflejan esto:

$$Q_{L_i} \cdot A_i + Q_{R_i} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0$$
$$1 \cdot A_i + 0 \cdot B_i + 1 \cdot A_i \cdot B_i + (-1) \cdot C_i - 1 = 0$$
$$A_i + A_i \cdot B_i - C_i - 1 = 0$$
$$A_i + A_i \cdot B_i - 1 = C_i$$

Si tomamos a x como el operando izquierdo y e como el derecho, la fórmula final obtenida es la que queríamos, y las matrices de 2.9 son las que vamos a seguir usando a partir de ahora. En la práctica no siempre vamos a poder comprimir nuestro programa en una sola fila.

Public Inputs y Output

Reflexionemos un poco sobre lo que tenemos hasta ahora y el objetivo que estamos persiguiendo. Queremos permitir que un prover le demuestre a un verifier que posee una matriz T (una traza) que sea válida respecto al programa dado por las matrices Q y V. La validez está dada por la definición 2.5.1. Sin embargo, también queremos poder demostrar que el programa fue ejecutado con ciertos inputs públicos. Actualmente, teniendo en cuenta que la traza T no será revelada en ningún momento al verifier, no tenemos herramientas para demostrar esto. Además, queremos poder demostrar que el resultado de un cómputo es el que decimos ser, y tampoco tenemos forma de hacer esto.

PLONK modela al output del cómputo como uno o más inputs públicos, y esto se debe a que el resultado del programa modelado tiene que ser conocido tanto por el prover como por el verifier y funcionar como un input para el software verificador. Tomando esto en cuenta, solo nos queda solucionar el problema de los Public Inputs. Para esto vamos a agregar una nueva matriz pública llamada PI para almacenar los públic inputs explicitamente, y además vamos a agregar filas al comienzo de todas las matrices para

Q_L	Q_R	Q_M	Q_O	Q_C
-1	0	0	0	0
-1	0	0	0	0
0	1	1	-1	-1

L	R	O
0	-	-
1	-	-
2	0	1

A	B	C
3	0	0
8	0	0
2	3	8

PI
3
8
0

Tab. 2.10: Matrices Q, V, T y PI del programa de ejemplo.

asegurarnos de que los public inputs están ocupando los lugares debidos en la traza. Esto también va a requerir modificar ligeramente la ecuación de 2.5.1. Las matrices resultantes pueden verse en las tablas de 2.10.

Para que las modificaciones en las matrices tengan sentido, la ecuación tiene que ser modificada acordemente:

$$Q_{L_i} \cdot A_i + Q_{R_i} \cdot B_i + Q_{M_i} \cdot A_i \cdot B_i + Q_{O_i} \cdot C_i + Q_{C_i} + PI_i = 0 \quad \forall i$$

Al comienzo de su única columna, la matriz PI tiene los valores de los inputs públicos y luego 0 de ahí en adelante. A partir de las filas agregadas en Q y en V, respetando la ecuación modificada, podemos asegurarnos de que los valores de los public inputs están siendo usados para el cómputo. Esto lo podemos ver en 2 partes:

1. Las dos primeras filas de la matriz Q junto con las dos primeras filas de la Traza T y las dos primeras filas de PI le aseguran al verifier que los valores de la traza son los mismos que los public inputs, en el mismo orden.

2. La matriz V nos asegura que esos valores son los mismos que se usan en el cómputo. Particularmente en el caso del ejemplo, le asegura al verifier que el segundo input público es el resultado final, que es lo que queríamos ver.

Veamos que el punto 1. se cumple con los valores concretos de las matrices de 2.10.
Fila 0:

$$Q_{L0} \cdot A_0 + Q_{R0} \cdot B_0 + Q_{M0} \cdot A_0 \cdot B_0 + Q_{O0} \cdot C_0 + Q_{C0} + PI_0 = 0$$

$$-1 \cdot 3 + 0 \cdot 0 + 0 \cdot 0 \cdot 0 + 0 \cdot 0 + 0 + 3 = 0$$

$$-3 + 3 = 0$$

$$0 = 0$$

Fila 1:

$$Q_{L1} \cdot A_1 + Q_{R1} \cdot B_1 + Q_{M1} \cdot A_1 \cdot B_1 + Q_{O1} \cdot C_1 + Q_{C1} + PI_1 = 0$$

$$-1 \cdot 8 + 0 \cdot 0 + 0 \cdot 0 \cdot 0 + 0 \cdot 0 + 0 + 8 = 0$$

$$-8 + 8 = 0$$

$$0 = 0$$

Fila 2:

$$Q_{L2} \cdot A_2 + Q_{R2} \cdot B_2 + Q_{M2} \cdot A_2 \cdot B_2 + Q_{O2} \cdot C_2 + Q_{C2} + PI_2 = 0$$

$$0 \cdot 2 + 1 \cdot 3 + 1 \cdot 2 \cdot 3 + (-1) \cdot 8 + (-1) + 0 = 0$$

$$3 + 6 - 8 - 1 = 0$$

$$0 = 0$$

Por último, es fácil ver como la consistencia requerida por la matriz V se cumple en la matriz T.

2.5.2. De matrices a polinomios

Vamos a darle uso a toda la teoría de cuerpos y polinomios que tratamos en las secciones 1.1 y 1.2 en lo que sigue del protocolo: ahora hay que armar polinomios que interpolen los valores de las matriz. Por el momento vamos a dejar de lado a la matriz V y centrarnos en un protocolo limitado que le va a permitir demostrar al prover que conoce una traza que cumple cada compuerta del programa *por separado*.

- Tenemos las columnas Q_L, Q_R, Q_M, Q_O, Q_C, A, B, C y PI por separado.
- Asumamos que nuestro programa tiene $2^n = N$ filas, con $n \in \mathbb{N}$. Si esto no es así, completamos las filas de todas las matrices con 0 (es fácil ver que las ecuaciones se cumplen para todas las filas, alcanza con tener reservado el índice 0 en la matriz V para el valor 0).
- Tenemos un cuerpo dado por $\mathbb{F}_p$, con un generador de grado N al que llamamos ω. Esto se vio en la sección 1.1.
- Tenemos un dominio de interpolación $D = [\omega^i : 0 \leq i < N] = [1, \omega, \omega^2, ..., \omega^{N-1}]$.

Vamos a interpolar cada una de las columnas sobre el dominio D. Esto nos va a dejar como resultado los 9 polinomios interpoladores, los cuales vamos a nombrar respectivamente (al igual que sus funciones asociadas en $\mathbb{F}_p$) q_L, q_R, q_M, q_O, q_C, a, b, c y pi. Conceptualmente, lo que acabamos de hacer es crear 9 polinomios que cumplen una propiedad muy interesante. Tomemos por ejemplo el polinomio a, obtenido a partir de interpolar los valores de la columna A. Vamos a notar $A[i]$ al valor en la i-ésima fila de la columna A. Dado el dominio de interpolación elegido y los valores que estamos interpolando, podemos ver que por definición se cumple la igualdad

$$a(\omega^i) = A[i]$$

ya que el polinomio a está armado para que se cumpla. Esto mismo pasa en el resto de las columnas. A continuación, vamos a armar un nuevo polinomio

$$f = q_L \cdot a + q_R \cdot b + q_M \cdot a \cdot b + q_O \cdot c + q_C + pi$$

Si resulta familiar, es porque es una formula que vimos anteriormente predicando sobre las columnas de las matrices, pero en este caso aplicada a los polinomios asociadas a ellas. Si tomamos $f(x)$ la función asociada a f en $\mathbb{F}_p$ obtenemos que

$$f(x) = 0 \quad \forall x \in D$$

si la traza T está bien formada respecto al programa Q y a los Public Inputs PI. ¿Por qué? Bueno, veamos que pasa si evaluamos $f(x)$ en un ω^i cualquiera:

$$\begin{aligned} f(\omega^i) = \\ q_L(\omega^i) \cdot a(\omega^i) + q_R(\omega^i) \cdot b(\omega^i) + q_M(\omega^i) \cdot a(\omega^i) \cdot b(\omega^i) + q_O(\omega^i) \cdot c(\omega^i) + q_C(\omega^i) + pi(\omega^i) = \\ Q_L[i] \cdot A[i] + Q_R[i] \cdot B[i] + Q_M[i] \cdot A[i] \cdot B[i] + Q_O[i] \cdot C[i] + Q_C[i] + PI[i] = 0 \\ \forall\ 0 \leq i < N \end{aligned}$$

Otra forma de expresar esta propiedad es diciendo que el polinomio f tiene raíces en todos los valores de D, que son N raíces distintas. Esto implica directamente por la propiedad 1 que

$$\begin{gathered} f(x) = 0 \qquad \forall x \in \{\omega^i : 0 \leq i < N\} \qquad \implies \\ (x - \omega^i) \mid f(x) \qquad \forall x \in \{\omega^i : 0 \leq i < N\} \qquad \implies \\ \left[\prod_{i=0}^{N-1} (x - \omega^i)\right] \mid f(x) \qquad \forall x \in \{\omega^i : 0 \leq i < N\} \qquad \implies \end{gathered}$$

$$\text{Si nombramos } z_D := \textstyle\prod_{i=0}^{N-1}(x - \omega^i)$$

$$\exists t, \quad f(x) = t(x) * z_D(x) \qquad \forall x \in \{\omega^i : 0 \leq i < N\}$$

Más aún, podemos afirmar que

$$z_D(x) = x^N - 1$$

Para probar esto vamos a notar que z_D es de grado N y sus raíces son los elementos de D. El polinomio $x^N - 1$ también es de grado N y si podemos demostrar que sus raices son las mismas que las de z_D, entonces por la propiedad 2 de los polinomios podemos probar que $z_D(x) = x^N - 1$.

Para llegar a eso vamos a usar el hecho de que ω es una raíz de $\mathbb{F}_p$ de grado N. Esto nos dice que

$$1 = \omega^0 = \omega^N = \omega^{2N} = \omega^{kN} \quad \forall k \in \mathbb{N}$$

Vemos que 1 es una raíz de $x^N - 1$, ya que $1^N - 1 = 0$. También vemos que ω es una raíz de $x^N - 1$ ya que por lo dicho anteriormente $\omega^N = 1$ y por ende $\omega^N - 1 = 0$. Lo mismo ocurre para ω^i en general. Por ende, podemos afirmar que todos los elementos de D son raíz de $x^N - 1$ y de esta forma terminamos de verificar la igualdad buscada.

Finalmente, podemos expresar a f como

$$f = t \cdot z_D$$

A	B	C
$a(1)$	$b(1)$	$c(1)$
$a(\omega)$	$b(\omega)$	$c(\omega)$
$a(\omega^2)$	$b(\omega^2)$	$c(\omega^2)$
...	...	...

Tab. 2.11: Matriz T recompuesta por el verifier.

Oráculos de polinomios

Sea $p \in \mathbb{F}_p[X]$, definimos a $[p]$ como el **oráculo** de p. Este oráculo es un observador de p al cual podemos pedirle la evaluación de la función asociada $p(x)$ en un valor aleatorio z. Notamos $[p]_z$ al valor devuelto por el oráculo, tal que $[p]_z = p(z)$.

Dado un conjunto de k polinomios $p_1, p_2, ..., p_k \in \mathbb{F}_p[x]$, podemos pedirle al conjunto de oráculos $[p_1], [p_2], ..., [p_k]$ una evaluación conjunta, que nos devuelve para un mismo z su evaluación en todos los polinomios, es decir $[p_1]_z, [p_2]_z, ..., [p_k]_z$ que equivale a $p_1(z), p_2(z), ..., p_k(z)$.

Esta abstracción nos va a resultar útil en el protocolo para simplificar algunos temas que veremos más adelante. La motivación concretamente está relacionada a que el prover no quiere dar al verifier los polinomios a, b y c, ya que haciendo esto estaría revelando los valores de la traza. El verifier simplemente podría completar la traza T haciendo las operaciones que se pueden ver en la matriz 2.11. En cambio, al pasar un oráculo polinomio se evita este problema, permitiendo hacer la verificación deseada con las limitadas capacidades del oráculo. Veremos más adelante cómo estos oráculos se implementan aprovechando el lema de Schwartz-Zippel.

2.5.3. Protocolo parcial

Veamos entonces cómo funciona el protocolo usando estos oráculos y todas las propiedades vistas anteriormente. Empecemos por ver qué elementos tienen el prover y verifier.

- Ambos conocen $\mathbb{F}_p$ y ω un generador de grado N.
- Ambos conocen la matriz Q y la matriz PI (el programa propiamente dicho).
- Solo el prover conoce la matriz T (la ejecución).

Podemos ver los pasos a continuación:

1. El prover computa los polinomios p_i, q_L, q_R, q_M, q_O, q_C, a, b, c
2. El verifier computa los polinomios pi, q_L, q_R, q_M, q_O, q_C
3. El prover computa el polinomio $f = q_L \cdot a + q_R \cdot b + q_M \cdot a \cdot b + q_O \cdot c + q_C + pi$
4. El prover computa el polinomio $t(x) = \frac{f(x)}{x^n - 1}$
5. El prover le pasa al verifier los oráculos $[a], [b], [c], [t]$

6. El verifier toma un valor aleatorio z y verifica la ecuación

$$q_L(z) \cdot [a]_z + q_R(z) \cdot [b]_z + q_M(z) \cdot [a]_z \cdot [b]_z + q_O(z) \cdot [c]_z + q_C(z) + pi(z) = [t]_z \cdot (z^N - 1)$$

Los pasos 1-4 deberían ser triviales y en el paso 5 estamos tomando el modelo de oráculo explicado en 2.5.2, lo importante ahora es entender la verificación del paso 6 como la culminación de todas las explicaciones anteriores.

Primero, notemos que si en lugar de tomar la evaluación de oráculos y polinomios en la ecuación hubiéramos comparado directamente a los polinomios, la igualdad sería trivial y el verifier podría convencerse fácilmente de que el prover posee a, b y c válidos para el programa dado por q_L, q_R, q_M, q_O y q_C. Sin embargo hacer esto sería revelar la totalidad de la traza, cosa que queremos evitar. El prover quiere que el verifier se convenza de esto mismo pero sin revelar a, b y c. Utilizando el lema de Schwartz-Zippel de la sección 1.2.2 observamos que si 2 polinomios que desconocemos evalúan al mismo valor en un mismo elemento aleatorio del dominio, entonces con probabilidad abrumadora ambos polinomios son iguales. En esto nos estamos apoyando ahora mismo: el verifier no necesita conocer los polinomios completos para convencerse de la igualdad, necesita conocer una evaluación de todos ellos en un mismo punto generado aleatoriamente, que es lo que nos proveen los oráculos.

El prover no podría haber falseado a, b, c y t para que la igualdad se mantenga solo en un punto, ya que no puede saber de antemano cuál es el punto z en el que se van a evaluar. De esto se deduce que a, b y c tienen que ser validos y por ende el prover conoce una traza válida del programa.

El protocolo para la matriz V se puede encontrar en el apéndice C, ya que no es tan central para el desarrollo del trabajo.

2.5.4. Blindings

Un **blinding** es un proceso por el que pasa un polinomio generado por el prover para permanecer indescifrable para el verifier, haciendo que todos sus valores evaluados sean aparentemente aleatorio. Tomemos por ejemplo el polinomio a (el que interpola la columna A de la traza). El prover provee al verifier con un oráculo de a llamado $[a]$ y el verifier puede evaluar este polinomio a través de su oráculo en un punto aleatorio $[a]_z = a(z)$. Al hacer esto, el verifier está obteniendo información parcial sobre los puntos de la traza, haciendo que se pierda la propiedad de Zero Knowledge.

Para preservar esta propiedad, los polinomios que deban permanecer secretos deben pasar por un proceso de *blinding* (ocultamiento). Este proceso toma a un polinomio original a de grado N y devuelve un nuevo polinomio $a_{blinding}$ de grado $M \geq N$. Estos polinomios cumplen la propiedad de que

$$a(\omega^i) = a_{blinding}(\omega^i) \quad \forall i \in D = \{\omega^0, \omega^1, \cdots, \omega^{N-1}\}$$

Para obtener este nuevo polinomio, tomemos nuevamente al polinomio $x^N - 1$, el cual sabemos que se anula en todos los puntos de D. Sea $k = M - N$, el prover va a generar $k + 1$ valores aleatorios $b_0, b_1, \cdots, b_k$ y definir

$$a_{blinded}(x) := (b_0 + b_1 \cdot x + \cdots + b_k \cdot x^k)(X^N - 1) + a$$

Vemos que la propiedad deseada se cumple ya que el término de la derecha se anula en todos los puntos del dominio D y en todo el resto de los puntos la evaluación del

polinomio devuelve valores calculados a partir de b_i aleatorios. Adicionalmente, se pueden establecer restricciones en el protocolo para que esos polinomios no puedan ser evaluados directamente en los valores de ningún ω_i.

2.6. Turbo-PLONK

Turbo-PLONK [7] extiende PLONK con 2 recursos para el armado de programas. Estos recursos son **custom gates** y **lookup tables**. La idea no es meterse en detalle con estos temas pero vamos a mencionar brevemente qué son y cómo funcionan ya que tienen relevancia en el desarrollo del trabajo y en las tecnologías concretas que serán usadas.

2.6.1. Custom Gates

En la sección 2.5.1 hablamos de *compuertas personalizadas* en el contexto de PLONK, sin embargo estas *Custom Gates* son de naturaleza distinta. Antes mencionamos la posibilidad de elegir valores personalizados para Q_L, Q_R, Q_M, Q_O, Q_C que nos permitieran definir operaciones en el marco de la ecuación

$$Q_{Li} \cdot A_i + Q_{Ri} \cdot B_i + Q_M \cdot A_i \cdot B_i + Q_O \cdot C_i + Q_C = 0 \quad \forall i$$

Sin embargo, Turbo-PLONK permite la definición de ecuaciones distintas, es decir, salirse del esquema definido por la matriz Q y definir ecuaciones **arbitrarias**. En este esquema, la traza no se ve limitada a las columnas A, B y C sino que el estado del programa tiene un ancho arbitrario Θ definido por el usuario o el protocolo. A su vez, la matriz Q tiene otra semántica: ahora va a tener un ancho arbitrario y cada columna representará una *custom gate*, con un 1 si está encendida y un 0 si está apagada. En otras palabras, la matriz Q nos permite encender y apagar restricciones polinomiales en cada fila.

Dado un estado $v \in \mathbb{F}_p^{\Theta}$, una traza válida de Turbo-PLONK es una matriz $T \in \mathbb{F}_p^{\Theta \times n}$ que cumple con la función de transición P y respeta las restricciones impuestas por la matriz V. La función de transición P es un polinomio de grado a lo sumo d en $2\Theta + l$ variables, donde l es el número de selectores (por ende, el ancho de la matriz Q). Estos selectores nos permiten "prender" y "apagar" compuertas. La restricción establecida sobre P es la siguiente:

$$P(T_{i,1}, T_{i,2}, \cdots, T_{i,\Theta}, T_{i+1,1}, T_{i+1,2}, \cdots, T_{i+1,\Theta}, q_{i,1}, \cdots, q_{i,l}) = 0 \quad \forall 0 \leq i < n$$

Analizando los índices, podemos ver la función de transición predica sobre la fila i-esima de la traza T y también sobre la siguiente. A su vez, los selectores permiten encender y apagar ciertas restricciones como veremos a continuación.

Supongamos que tenemos una traza T de ancho 3 al igual que en PLONK y queremos 2 restricciones distintas, digamos por ejemplo $A_i^3 = B_i$ y $A^2 + B^2 = C^2$. Para esto, alcanza con definir una matriz Q con dos columnas $CUBO$ y $NORMA$: la primera estará asignada a la primera restricción ($A_i^3 = B_i$) y la segunda a la segunda ($A^2 + B^2 = C^2$). Usaremos solo unos y ceros para llenar la matriz Q. Con todo esto, definimos la función de transición P como

$$\begin{aligned} &P(A_i, B_i, C_i, A_{i+1}, B_{i+1}, C_{i+1}, CUBO_i, NORMA_i) \\ &= CUBO_i \cdot (A^3 - B) + NORMA_i \cdot (A^2 + B^2 - C^2) \end{aligned}$$

Es decir, si la matriz Q se ve como la de la tabla 2.12 significa que está restringiendo los valores de la primera fila de la traza T a respetar la restricción $CUBO$ y los valores de la segunda fila a respetar la restricción $NORMA$. Por eso llamamos a las columnas de Q **selectoras**.

$CUBO$	$NORMA$
1	0
0	1

Tab. 2.12: Matriz Q de turbo-PLONK con operaciones $CUBO$ y $NORMA$.

Notemos que en el ejemplo no estamos usando el hecho de que podemos predicar sobre 2 filas consecutivas de la traza al mismo tiempo, pero eso no es más que una decisión de diseño. Es fácil ver también cómo el esquema de PLONK descrito anteriormente es un caso particular de Turbo-PLONK, solo que usando la matriz Q de una forma distinta: en lugar de encender o apagar compuertas, se usa para parametrizar una ecuación que se repite en todas las filas. La matriz T se comporta igual en ambos protocolos.

2.6.2. Lookup Tables

Las *lookup tables* [11] son una herramienta que permite chequear de forma eficiente que un conjunto de elementos de la traza es un subconjunto de una lista prefijada de valores. Esto es útil en el contexto de chequear el **rango** de un valor (ver que cierto elemento tiene como mucho n bits) y calcular **operaciones bit a bit** como xor, and, or, etc. Por ejemplo, para ver que un número e es de 8 bits alcanza con definir una tabla

$$t_{u8} = \{0, 1, 2, ..., 255\}$$

y luego verificar que $\{e\} \subset t_{u8}$. Otro ejemplo sería el de definir una tabla de múltiples columnas para una operación xor de 1 bit, por ejemplo:

$$t_{xor} = \{(0,0,0), (0,1,1), (1,0,1), (1,1,0)\}$$

para luego verificar que una operación $\{(x, y, x \oplus y)\} \subset t_{xor}$.

Para hacer esto existe un protocolo interactivo que puede incluirse dentro del protocolo principal. Dada una lookup table $\{t_i\}$ y un multiconjunto de valores en la traza $\{f_i\}$, expresamos ambos multiconjuntos como polinomios donde sus raíces son sus elementos respectivamente.

$$F(X) := \prod_{0 \leq i < |t|} (X - f_i), \quad G(X) := \prod_{0 \leq i < |f|} (X - t_i)$$

El protocolo consiste en ver que ambos polinomios tienen las mismas raíces, ignorando multiplicidades, pero no vamos a entrar en más detalles en este trabajo.

3. POLYNOMIAL COMMITMENT SCHEMES

En la sección 2.5.2 definimos a los oráculos como un objeto matemático casi mágico. En la práctica, por supuesto, esto no es así, sino que son implementados a través de un protocolo interactivo entre el prover y el verifier. Muchas de las propiedades de los SNARKs se derivan directamente de las propiedades que proveen estos protocolos, como los tamaños de las pruebas. La familia de protocolos para resolver el problema de los oráculos se conoce como ***Polynomial Commitment Scheme***. La idea general es la descrita anteriormente, pero ahora vamos a dar una idea un poco más bajo nivel.

Un prover posee un polinomio privado y un verifier quiere conocer la evaluación de este polinomio en un punto. Los Polynomial Commitment Schemes son una herramienta que permite a un prover **comprometerse** a un polinomio, de manera tal que el verifier pueda hacer consultas sucesivas sobre sus valores en cualquier punto z del dominio de la función asociada y obtener un resultado que puede estar convencido de que se corresponde con la evaluación de la función asociada a ese polinomio en z.

La interfaz de estos protocolos está compuesta por 3 funciones:

1. $Commit(p) \to [p]$: Acción que realiza el prover cuando se compromete a un polinomio p frente a un verifier. En general esta acción consiste en pasar algún elemento matemático $[p]$ al verifier, al cual llamaremos **commit** de p.

2. $Open(p, x) \to y, \pi$: Acción que realiza el prover cuando el verifier le envía un valor x en el dominio. El prover calcula $y = p(x)$ y se lo entrega al verifier, junto con una prueba π de que la evaluación es correcta en p.

3. $Verify([p], x, y, \pi) \to \{true, false\}$: Acción que realiza el verifier para verificar que efectivamente $p(x) = y$. Para llevarla a cabo, debe hacer uso del commit $[p]$ obtenido previamente y la prueba para la pareja x, y en particular.

En algunos casos, para generar tanto el commit como la prueba es necesario contar con un CRS (*Common Reference String*). Este es un objeto matemático que tanto el prover como el verifier conocen. A continuación vamos a ver un protocolo de ejemplo que usa un CRS: **KZG** [12].

3.1. KZG

KZG (Kate-Zaverucha-Goldberg) es un Polinomial Commitment Scheme publicado en el año 2010 que se basa en el uso de curvas elípticas, introducidas en la sección 1.3. Veamos a continuación como funciona.

Supongamos que tenemos un polinomio $p \in \mathbb{F}_p[X]$ de grado n. Tomemos la curva elíptica $E := BLS$ y un punto $G \in E$ de grado n. Recordemos que esto significa que $G = n \cdot G = 2n \cdot G = \cdots$. Tanto E como G son conocidos por el prover y el verifier. Para que este protocolo funcione, necesitamos generar una lista de puntos de E a partir de un $\tau \in \mathbb{F}_p$ desconocido. Esta lista será:

$$CRS = [G, \tau G, \tau^2 G, \tau^3 G, \cdots, \tau^{n-1} G]$$

Es muy importante que nadie conozca el valor de τ ya que de lo contrario habría una falla de seguridad crítica: quien conozca τ podrá demostrar evaluaciones incorrectas del polinomio al cual se compromete, generando un problema de *soundness* en el protocolo. Para crear dicha lista sin que nadie tenga nunca el valor de τ existen las opciones:

- Un agente de confianza podría crearla a partir de un $\tau \in \mathbb{F}_p$ generado de forma aleatoria y luego descartar el valor. Este enfoque requiere una confianza absoluta en este agente, que sería una condición más en el modelo de seguridad del protocolo y por ende, indeseable.
- Se puede genera una ceremonia entre varios agentes con protocolos de Multi Party Computation (MPC) para crear dicha lista sin que ninguno de ellos posea el valor de τ. Esta es la opción que más suele usarse, aunque requiere la participación varios agentes, de los cuales por lo menos uno tiene que ser honesto.

Es importante notar que el valor de τ no puede deducirse a partir del CRS: aunque contemos con el valor de τG, es computacionalmente dificil extraer τ por el problema del logaritmo discreto para curvas elípticas introducido en la sección 1.3.2.

Veamos una intuición de lo que viene a continuación. Supongamos que tenemos un polinomio $p \in \mathbb{F}_p[X]$ de grado a lo sumo $n-1$ y un valor $\tau \in \mathbb{F}_p$ desconocido, como mencionamos anteriormente. Primero, observamos que

$$p(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_{n-1} \cdot x^{n-1}$$

Notamos que para obtener $p(\tau) \cdot G$ no necesitamos conocer τ alcanza con usar valores de la lista CRS:

$$\begin{aligned} p(\tau) \cdot G = \\ (c_0 + c_1 \cdot \tau + c_2 \cdot \tau^2 + \cdots + c_{n-1} \cdot \tau^{n-1}) \cdot G = \\ c_0 \cdot G + c_1 \cdot \tau \cdot G + c_2 \cdot \tau^2 \cdot G + \cdots + c_{n-1} \cdot \tau^{n-1} \cdot G \end{aligned}$$

Notemos que los valores remarcados son puntos de la curva E que forman parte del CRS, y los valores $c_0, c_1, \cdots, c_{n-1}$ son constantes conocidas por el prover, ya que conoce al polinomio p. Por lo tanto, alcanza con hacer una serie de sumas y multiplicaciones de puntos de curva elíptica para obtener $p(\tau) \cdot G$, pero no es necesario conocer τ.

Veamos ahora cómo se implementa cada una de las funciones de la interfaz del protocolo:

1. $Commit(p)$: el commit $[p]$ será un punto de curva. Tan simple como eso. El punto de curva será $p(\tau) \cdot G$ y puede ser calculado como mencionamos anteriormente.
2. $Open(p, k) \to y, \pi$:
   a) El prover computa $y := p(x)$, el resultado de la evaluación.
   b) El prover computa un polinomio $q(x) := \frac{p(x)-y}{x-k}$.
   c) Finalmente, el prover computa $\pi := q(\tau) \cdot G$ y se lo pasa al verifier

3. $Verify([p], x, y, \pi) \rightarrow \{true, false\}$: dado un *pairing bilineal* e sobre E, el verifier comprueba que

$$e((p(\tau) - y) \cdot G, G) = e(q(\tau) \cdot G, (\tau - k) \cdot G)$$

para convencerse de que la evaluación es correcta.

Descompongamos algunos de los pasos anteriores. El paso 2.b) crea un nuevo polinomio q de una forma muy particular. Sea $y = p(k)$, pensemos en el polinomio $p(x) - y$. Este polinomio tiene una raíz en k, ya que si $p(k) = y$ entonces $p(k) - y = 0$, entonces $(x - k)|(p(x) - y)$. Esto solo ocurre si $y = p(k)$; si esto no fuera así, el prover no podría crear el polinomio q. El grado de q es uno menos que el grado de p, por lo tanto, podemos usar el truco mencionado con el CRS para calcular $q(\tau)$ en el paso 2.c).

Por último, veamos por qué la igualdad del paso 3. nos garantiza que la evaluación es correcta. Notemos que todos los valores involucrados en la verificación son accesibles para el verifier:

- $(p(\tau) - y) \cdot G = p(\tau) \cdot G - y \cdot G$, que son dos puntos que fueron dados por el prover en el paso 1. y que puede ser calculado respectivamente.
- $q(\tau) \cdot G$ fue dado al verifier en el paso 2.c)
- $(\tau - k) \cdot G = \tau \cdot G - k \cdot G$ son dos puntos que están en el CRS y que puede ser calculados respectivamente.

Para entender esta verificación es importante recordar la propiedad de bilinealidad que posee el pairing e. Esto quiere decir que $e(a \cdot P, b \cdot Q) = e(P, Q)^{ab}$. Con esto en mente, desarrollemos un poco la igualdad:

$$\begin{aligned}
e((p(\tau) - y) \cdot G, G) &= e(q(\tau) \cdot G, (\tau - k) \cdot G) \\
e(G, G)^{p(\tau) - y} &= e(G, G)^{q(\tau) \cdot (\tau - k)} \\
&\iff \\
p(\tau) - y &= q(\tau) \cdot (\tau - k) \\
p(\tau) - y &= \frac{p(\tau) - b}{\tau - k} \cdot (\tau - k) \\
p(\tau) - y &= p(\tau) - y
\end{aligned}$$

Por el lema de Schwartz-Zippel mencionado en la sección 1.2.2, una igualdad de polinomios en $\mathbb{F}_p[X]$ evaluados en un punto aleatorio o **desconocido** (en este caso τ) equivale con probabilidad altísima a una igualdad de polinomios. En esencia, lo que el verifier está verificando es la igualdad de polinomios del paso 2.b) i.e.

$$q(x) = \frac{p(x) - y}{x - k}$$

Recordemos que $q(x)$ existe sii $p(k) = y$, que es lo que el prover quiere demostrar. Por ende, al verificar esta igualdad de polinomios implícitamente se está verificando el hecho de que $p(k) = y$, y el lema de Schwartz-Zippel nos permitió que en ningún momento el verifier necesite explícitamente p o q para verificar esto.

Vulnerabilidad con τ conocido

Notemos por otro lado que si el prover tuviera acceso a τ podría hacer algo parecido a lo que vimos en el protocolo de Schnorr en la sección 2.2 y demostrar una evaluación incorrecta. Para esto, alcanzaría con hallar un polinomio q tal que $p(\tau) - y = q(\tau) \cdot (\tau - k)$, pero este polinomio no tiene que ser obtenido a través de la cuenta $q(x) = \frac{p(x)-y}{x-k}$ sino que podría ser cualquiera. En este caso se está explotando la vulnerabilidad de Schwartz-Zippel al conocer el punto exacto donde los polinomios se van a evaluar.

3.1.1. Costo del CRS

Vamos a ver ahora uno de los motivos centrales de la tesis: las implicancias de tener un $CRS = [G, \tau G, \tau^2 G, \tau^3 G, \cdots, \tau^{n-1} G]$. Primero vamos a evaluar el tamaño (computacionalmente hablando) de dicha lista.

¿Cuánto pesa un punto de curva?

Un punto de curva se representa a través de un par ordenado de componentes, tal que $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$. Recordemos que usando una curva como BLS12-381, $p = 2^{255} - 13$ es decir que es un número de 256 bits. Sin embargo, la representación usada para estos números usa 381 bits por razones de seguridad y eficiencia a la hora de calcular pairings. Por ende, un punto de curva se representa con 762 bits.

¿Qué tan largo es el CRS?

Pensemos ahora qué valor puede llegar a tener n. En el contexto de PLONK, el valor de n representa la cantidad de filas del programa o de la traza. Estos valores pueden llegar al orden de los millones para un programa grande. También recordemos que n solo puede tomar valores que sean potencias de 2, veamos entonces cómo se verían los costos.

Costo Final

En la tabla 3.1 podemos ver como a medida que crece el grado del polinomio, el CRS se vuelve cada vez más costoso de mantener. Todo prover que quiera comprometerse a un polinomio o generar una prueba usando KZG debería contar con un CRS en su dispositivo, y si el polinomio es muy grande podría resultar limitante el tipo de dispositivo que puede ser usado para generar este tipo de pruebas.

3.2. Merkle Trees

Los **Merkle Trees** [13] son estructuras de datos ampliamente utilizadas en protocolos criptográficos. Funcionan como huellas digitales de ciertas estructuras de datos ordenadas. Además, así como KZG nos da un protocolo para comprometernos a un polinomio y abrirlo en cualquier punto del dominio, los Merkle Trees son una primitiva que resultará útil para para comprometernos a **listas**.

Esta estructura de datos tiene la forma de un árbol binario perfectamente balanceado y se construye a partir de una lista. Por simplicidad, vamos a asumir que esta lista tiene un tamaño $N = 2^n$. Sea entonces $A = [a_1, \cdots a_N]$ una lista de elementos de algún tipo

n	peso
1	762 bits
2	1524 bits
4	381 bytes
8	762 bytes
...	...
2^i	$2^i \cdot$ 762 bits
...	...
2^{20}	95 MB
2^{25}	3 GB
2^{30}	95 GB

Tab. 3.1: Costo del CRS en función del grado del polinomio.

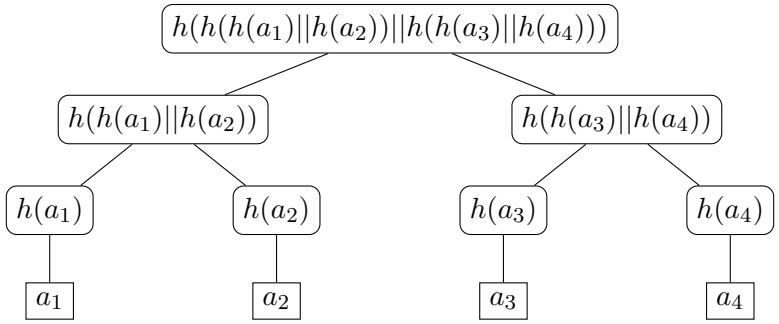


Fig. 3.1: Merkle Tree de 3 niveles

$a_i \in S$, $h : S \to S$ una función de hash y $|| : S \times S \to S$ alguna función de concatenación, vamos a definir la estructura de datos de la siguiente manera:

- Las **hojas** del árbol serán los hashes de los elementos ordenados de A, es decir,

$$[h(a_1), \cdots, h(a_N)]$$

- Los **niveles intermedios** (nivel $n-1$ al nivel 2), donde cada nodo será el hash de la concatenación de sus hijos. Por ejemplo, para el nivel $n-1$, los nodos serán

$$[h(h(a_1)||h(a_2)), \cdots, h(h(a_{N-1})||h(a_N))]$$

- Siguiendo la definición recursiva, la **raíz** del merkle tree será una huella digital de la lista.

Visualmente, un merkle tree de 3 niveles se puede ver en la figura 3.1.

Hay otras operaciones que nos interesan realizar sobre un Merkle Tree además de construirlo para obtener su raíz. Una de usos más comunes es **comprometerse a una lista** para luego revelar elementos a pedido sin revelar la lista entera. ¿Qué quiere decir esto?

Supongamos que un prover tiene una lista secreta A de tamaño $N = 2^n$ y necesita convencer a un verifier de que esa lista posee ciertos elementos en determinadas posiciones. Desde el punto de vista del verifier, quiere convencerse de que el prover tiene una lista A con los elementos $e_1, \cdots, e_k$ en las posiciones $i = 1, \cdots k$ ($k \leq N$). Para cada uno de estos elementos, el prover puede brindar una prueba criptográfica π_{i,e_i}. Veamos cómo es el protocolo:

1. $Commit(A) \to [A]$: el prover computa el Merkle Tree de A y le pasa al verifier la raíz del árbol, que va a funcionar como el commit o "huella digital" de la lista A. Si asumimos que la función de hash se comporta como un oráculo aleatorio, entonces la probabilidad de que otra lista tenga la misma huella es casi nula.

2. $Open(A, i) \to a_i, \pi_{i,a_i}$: la idea acá es mandar un **camino de autenticación** de la posición i-esima. Para esto alcanza con mandar un nodo interno del árbol por nivel. ¿Cómo funciona esto? Consideremos una lista A de largo $N = 4$ como la de la figura 3.1. Supongamos que el verifier quiere conocer el valor de la posición 3, es decir, a_3. ¿Cuáles son los valores que el prover tiene que darle para que pueda llegar hasta la raíz? En este caso, esos valores serían $\pi_{3,a_3} = h(a_4)$ y $h(h(a_1)||h(a_2))$. ¿Por qué es suficiente con esto? Veamos los pasos que realiza el verifier en la verificación.

3. $Verify([A], i, a_i, \pi_{i,a_i}) \to \{true, false\}$:

   a) Calcula $h(a_3)$ a partir del valor obtenido de a_3. Recordemos que la función de hash es pública para el protocolo.

   b) Teniendo $h(a_4)$ que fue provisto por el prover, puede calcular $h(h(a_3)||h(a_4))$.

   c) Teniendo $h(h(a_1)||h(a_2))$ que fue provisto por el prover, puede calcular $h(h(h(a_1), h(a_2))||h(h(a_3), h(a_4)))$, habiendo llegado de esta manera hasta la raíz. Llamemos al valor obtenido de la raíz $[A]'$.

   d) Finalmente, verifica que $[A] = [A]'$ para convencerse de que efectivamente a_3 es el elemento en el índice 3 de A.

   Notemos que la cantidad de pasos a realizar de forma recursiva es del orden de n, es decir, logarítmica respecto al tamaño de la lista o lineal respecto a la altura del árbol. Una representación del proceso de verificación puede verse en la figura 3.2. En rojo están los valores provistos por el prover y en verde los valores calculados por el verifier (la raíz de el árbol entra en ambas categorías).

3.3. FRI

Otro protocolo que implementa oráculos de polinomios es FRI. Este protocolo tiene la ventaja de que no necesita un CRS para funcionar, sin embargo las pruebas que genera ocupan más espacio de almacenamiento que las de KZG para alcanzar el mismo nivel de seguridad. FRI también es un protocolo interactivo, con muchas más interacciones que KZG y que se apoya en los commitments de listas de los Merkle Trees vistos en la sección 3.2.

Concretamente, FRI es una Low Degree Proof: un protocolo que sirve para demostrar que el prover conoce un polinomio de grado **menor estricto a** un cierto $N = 2^n$. El polinomial commitment scheme se apoya en este protocolo para dar garantías de Soundness.

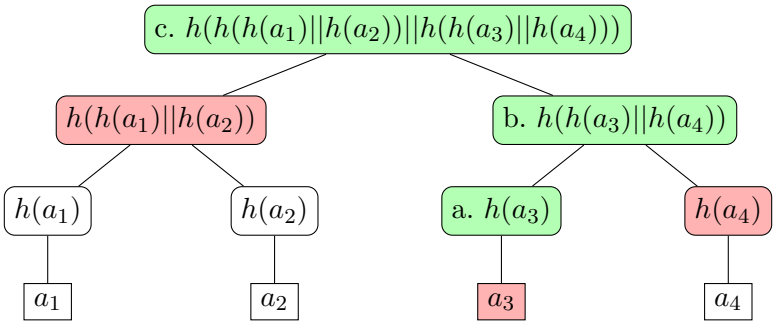


Fig. 3.2: Merkle Path para a_3

3.3.1. Polinomial Commitment Scheme sobre FRI

Nuevamente partimos de que el prover conoce un polinomio p de grado $N = 2^n$ y quiere comprometerse a ese polinomio para que el verifier pueda abrirlo en algún punto que desee, convenciéndose de que la evaluación obtenida se corresponde al polinomio comprometido inicialmente. Vamos a comenzar tomando un valor $M = 2^{n+b}$, donde b es un parámetro de seguridad (o *blowup-factor*) y tomar una raíz M-ésima de la unidad ω para armar un dominio de evaluación

$$D = \{1, \omega, \omega^2, \cdots, \omega^{M-1}\}$$

El protocolo es como sigue:

1. $Commit(p) \rightarrow [p]$: Lo primero que va a hacer el prover es inicializar un Merkle Tree con hojas
$$[p(1), p(\omega), p(\omega^2), \cdots, p(\omega^{M-1})]$$
De esta manera va a obtener un Merkle Hash de su raíz $[p]$, es decir, un Merkle Hash de las evaluaciones de p en el dominio D. A continuación va a darle $[p]$ al verifier.

   Podemos volver nuevamente a la abstracción de que el prover le está dando al verifier un oráculo de p, acotado a los valores del dominio D.

2. $Open(p, z) \rightarrow p(z)$: El verifier va a pedir la evaluación de p en un valor $z \in D$, por lo tanto va a darle al prover ese valor. El prover va a retornar $p(z)$ al verifier.

3. $Verify([p], p(z)) \rightarrow \{true, false\}$: El verifier va a pensar en el polinomio
$$q(x) = \frac{p(x) - p(z)}{x - z}$$
No tiene acceso a este polinomio, sin embargo puede usar algo llamado *oráculo virtual* (o $[q]$), obteniéndolo a través de $[p]$. Es decir, si el verifier puede obtener evaluaciones de p usando el oráculo, también puede obtener evaluaciones de q usando el mismo oráculo $[p]$. El truco para hacer la verificación es el siguiente: si efectivamente la evaluación que el prover dio de $p(z)$ es correcta, entonces q debería tener un grado menos que p. Si la evaluación no es correcta, entonces q va a tener un grado notablemente más alto. Al verifier le alcanza entonces con convencerse de que q tiene grado *a lo sumo* $N - 1$ para ver que p tiene grado N. Para garantizarse esto, va a recurrir al protocolo FRI explicado en el apéndice D.

Un detalle importante es que al aplicar la heurística de Fiat-Shamir, todo el protocolo FRI estará comprimido en una única prueba, armada de forma offchain por el prover. La verificación ocurre en tiempo succint.

4. PROBLEMA A RESOLVER

Noir es un lenguaje de programación de alto nivel que compila a una representación intermedia llamada ACIR y permite generar pruebas de ejecución usando un prover llamado Barretenberg. Barretenberg usa KZG como Polynomial Commitment Scheme, por lo que requiere un CRS que puede crecer arbitrariamente en tamaño; por ende, necesita una ceremonia con varios agentes involucrados y espacio de almacenamiento suficiente para generar las pruebas. La propuesta de la tesis es adaptar Noir para que pueda ser usado con un prover llamado Plonky2, el cual utiliza FRI en lugar de KZG y por ende no tiene los problemas anteriores.

Parte del trabajo consiste en investigar ambas herramientas con la profundidad suficiente para poder implementar y explicar la herramienta generada, por lo tanto el desarrollo contará con algunas secciones dedicadas a indagar en Plonky2 y Noir. El grueso del trabajo consiste en el desarrollo de la herramienta, justificando las decisiones de diseño tomadas y brindando desarrollos matemáticos suficientes para demostrar la correctitud de las soluciones. Además, para verificar que la herramienta cumple con su propósito, el código será acompañado de un conjunto exhaustivo de tests.

Finalmente, haremos un reporte de la performance de la herramienta, midiendo magnitudes como tiempos de ejecución o tamaños de artefactos generados que se detallarán en la sección 8.2. Por último, se realizará una comparación entre los tamaños de las pruebas de Plonky2 con el CRS mínimo que Barretenberg necesita para algunos casos, para evaluar si esta alternativa presenta alguna ventaja espacial o no.

Parte II

DESARROLLO

5. NOIR

Noir es un lenguaje de programación que permite generar pruebas de ejecución usando un protocolo ZK-SNARK. Podemos escribir programas tradicionales, librerías y Smart Contracts, aunque vamos a enfocarnos en los programas tradicionales. El prover que usa para esto se llama **Barretenberg** y está escrito en C++. La versión de Noir usada en este trabajo es la 0.47.0 por motivos que veremos más adelante. A su vez, la versión de Barretenberg usada es la 0.61.0. En el apéndice A vamos a realizar un análisis en profundidad del flujo que realiza Noir para pasar de un código de alto nivel a una prueba de ejecución a través de su compilador: **Nargo**. A continuación vamos a extraer los aspectos más relevantes de este flujo.

Una vez que instalamos Nargo y creamos un proyecto, hay que escribir un programa de Noir. Noir tiene una particularidad frente a otros lenguajes: permite distinguir entre inputs públicos y privados. Los inputs privados serán conocidos únicamente por el prover mientras que los públicos serán conocidos también por el verifier. A continuación se ve un ejemplo:

```
fn main(x: pub Field, y: Field) {
    assert(x == y*y);
}
```

¿Qué es lo que hace este programa? `main` es el punto de entrada, recibe un parámetro público y uno privado y no tiene un valor de retorno. Su propósito en lenguaje coloquial es generar una prueba de ejecución de que el prover conoce el valor de $\sqrt{x}$ en $\mathbb{F}_p$, pero sin revelar su valor.

A continuación, el prover deberá brindar una asignación de valores concretos para las variables de input, en este caso x e y. Esto es necesario para obtener la traza de una ejecución concreta del programa. Se puede tomar por ejemplo `x = 4, y = 2`. Una vez provistos estos valores, el programa se puede ejecutar para obtener 2 artefactos:

- un sistema de restricciones en formato **ACIR** (*Abstract Circuit Intermediate Representation*). Este está compuesto por un conjunto de **witnesses**, que son las variables inmutables del programa, y un sistema de ecuaciones que establece restricciones sobre ellas. En términos de PLONK, son las matrices Q y V. Veremos más en detalle el código ACIR en la sección 5.1.
- un diccionario que asigna valores concretos a todos los witnesses, que tienen que ver con la ejecución concreta del programa dados los inputs suministrados por el prover. En términos de PLONK, son las matrices T y PI.

.

Para este programa concreto, los witnesses son:

- w_0 representando a la variable x (pública)
- w_1 representando a la variable y (privada)

y el sistema de restricciones está dado por una única ecuación:

$$-1 \cdot w_1 \cdot w_1 + 1 \cdot w_0 = 0,$$

que es equivalente a

$$w_0 = w_1^2,$$

Conceptualmente podemos pensarlo en términos de PLONK como que ya tenemos las matrices T (traza), Q y V (programa) y PI (public inputs). El siguiente paso es generar la prueba de ejecución con un **prover** específico, en este caso Barretenberg. Alcanza con suministrar a Barretenberg los artefactos que generamos para que este genere la prueba deseada.

Ahora bien, el verifier quiere verificar esta prueba. Va a tener acceso al programa y a los inputs públicos, pero no a los inputs privados. Una prueba no puede ser verificada en el vacío, sino que tiene que ser contextualizada para que tenga significado o semántica. Concretamente, el verifier adempás de la prueba necesita una **Verifying Key**: un artefacto que contiene la información sobre el programa y los inputs públicos. La VK puede ser armada por el prover y pasada al verifier, o puede ser armada directamente por el verifier. Barretenberg tiene herramientas para crear la VK sin necesidad de generar una prueba. Esto se puede ver en más detalle en el apéndice A.

Los comandos usados en cada paso son:

- **prove**: para generar la prueba
- **write_vk**: para generar la Verification Key
- **verify**: para verificar la prueba (requiere haber generado la VK)

5.1. ACIR

El *Abstract Circuit Intermediate Representation* (ACIR) es el bytecode de alto nivel en el cual deriva todo programa escrito en Noir. Esencialmente, un código ACIR determina un sistema de ecuaciones **en abstracto**. Esto quiere decir que no siempre va a proveer explícitamente todas las ecuaciones polinomiales, sino que muchas veces simplemente va a decir qué características deben cumplir estas ecuaciones, delegando en el proving system la implementación concreta.

Las variables de este sistema de ecuaciones son los **witnesses**. Un código ACIR describe qué variables van a ser **inputs públicos**, **inputs privados** y **valores de retorno**. Esto no significa nada para el sistema de ecuaciones en sí, pero provee información para que el proving system pueda replicar el programa escrito en Noir con su propio modelo. Los witnesses están numerados del 0 en adelante, y los notamos w_i o simplemente $\_i$. Podemos pensar a los witnesses como variables **inmutables** del programa, aunque en realidad sabemos que su inmutabilidad se debe a que estamos parados en otro modelo: en el de los sistemas de ecuaciones.

Las instrucciones que describen este sistema de ecuaciones se denominan **opcodes** y los hay de varios tipos. Ya nos topamos con uno antes cuando estábamos viendo el flujo de un programa de Noir en la figura A.1. Todos los opcodes predican sobre witnesses, estableciendo restricciones sobre ellos. Veamos a los opcodes uno por uno:

5.1.1. AssertZero

Un opcode AssertZero determina de forma explícita una restricción polinomial sobre los witnesses. Estas tienen 3 campos posibles:

- **Términos cuadráticos**: una lista M de $|M|$ triplas de la forma $(k_{Mi}, w_{M_{i,l}}, w_{M_{i,r}})$ con $k_{Mi} \in \mathbb{F}_p$. Esta lista describe una combinación lineal de productos cuadráticos entre witnesses:
$$\sum_{i=0}^{|M|}(k_{Mi} \cdot w_{M_{i,l}} \cdot w_{M_{i,r}})$$
- **Términos lineales**: una lista L de $|L|$ tuplas de la forma (k_{Li}, w_{L_i}) con $k_{Li} \in \mathbb{F}_p$. Esta lista describe una combinación lineal de witnesses:
$$\sum_{i=0}^{|L|}(k_{Li} \cdot w_{L_i})$$
- **Constante**: un valor constante $C \in \mathbb{F}_p$.

Finalmente, un opcode AssertZero establece la siguiente restricción sobre los witnesses:

$$\sum_{i=0}^{|M|}(k_{Mi} \cdot w_{M_{i,l}} \cdot w_{M_{i,r}}) + \sum_{i=0}^{|L|}(k_{Li} \cdot w_{L_i}) + C = 0$$

Un ejemplo concreto sería

$$3 \cdot w_3 \cdot w_2 + 5 \cdot w_1 \cdot w_2 \; + \; 4 \cdot w_1 + 1 \cdot w_0 \; + \; 22 = 0$$

5.1.2. Operaciones de memoria

En un programa tradicional usamos arreglos para almacenar datos. Noir también provee esa posibilidad, pero ¿cómo esto se traduce a código ACIR? El caso donde queremos leer y escribir posiciones con un indice constante es sencillo: el código ACIR simplemente modela estos casos con AssertZero. El caso intrincado es cuando las posiciones que leemos y escribimos son dinámicas, es decir, cuando no dependen del circuito estático sino de la ejecución concreta del mismo. Para esto, hay operaciones de memoria **abstractas** para crear un bloque de memoria, escribir en una posición y leer de esa posición.

Es importante aclarar que las listas en Noir son de tamaño estático: no se pueden agregar ni quitar elementos. También son creadas con valores iniciales para todas sus posiciones. Una intuición de por qué existe esta limitación se verá más adelante en la sección 5.1.5

MemoryInit

Este opcode dicta la creación de un bloque de memoria. El opcode tiene 3 campos:

- **blockId**: es un $n \in \mathbb{N}$ que toma valores del 0 en adelante para distintos bloques de memoria. Permite identificar y referenciar al bloque más adelante.
- **init**: una lista L con n witnesses que contienen los valores iniciales de todas las posiciones del arreglo. Dado que $L = [w_{l0}, w_{l1}, \cdots, w_{ln-1}]$, los valores del arreglo A estarán restringidos por el proving system a tomar los valores $A[0] = w_{l0}$, $A[1] = w_{l1}, \cdots, A[n-1] = w_{ln-1}$ inicialmente.

```
fn main(arr: [Field; 4], idx: pub Field) -> pub Field {
    arr[idx]
}
```

Fig. 5.1: Programa de Noir minimal con lectura de memoria

```
private parameters indices : [0, 1, 2, 3]
public parameters indices : [4]
return value indices : [5]
INIT (id: 0, len: 4, witnesses: [_0, _1, _2, _3])
MEM (id: 0, read at: _4, value: _6)
EXPR [ (1, _5) (-1, _6) 0 ]
```

Fig. 5.2: ACIR para el programa de con lectura de memoria

MemoryRead

Este opcode permite "leer" una posición de memoria. Tiene los siguientes campos:

- **blockId**: una constante para identificar el índice del bloque previamente creado.
- **index**: un witness que representa el índice que se quiere leer del arreglo. Es importante notar que este es un witness y no una constante, y que por ende no es conocida en tiempo de compilación.
- **value**: un witness que está restringido a tomar el valor de la posición de memoria indicada.

Podemos pensar una operación de lectura como una restricción de la forma

$$w_{value} := B_{blockId}[w_{index}]$$

Veamos un ejemplo minimal con un programa de Noir como el de la figura 5.1. Este programa recibe un arreglo de tamaño 4 y un índice público, y retorna el valor del arreglo en ese índice. Semánticamente, como el arreglo es privado y tanto el índice como el valor de retorno son públicos, este programa está demostrando la afirmación "poseo un arreglo de 4 elementos de $\mathbb{F}_p$ que tiene un valor x en la posición idx" donde tanto x como idx son conocidos por el verifier.

Podemos ver el ACIR generado para este programa en la figura 5.2. Vemos que el arreglo es representado inicialmente como 4 witnesses privados independientes, pero luego hay una operación de `INIT` que usa a esos 4 witnesses para inicializar un bloque de memoria con $id = 0$. Luego, hay una operación de lectura que dice que se va a leer el bloque de memoria con $id = 0$ en la posición denotada por el witness w_4 (que en nuestro caso, es la variable `idx`), y el valor leido será el mismo que el de w_6. Por último, el programa usa un AssertZero para restringir $w_5 = w_6$, ya que w_5 es el witness que representará el valor de retorno.

MemoryWrite

Este opcode permite "escribir" una posición de memoria. Tiene campos similares a los del MemoryRead, pero con distínta semántica:

```
fn main(mut arr: [Field; 4], idx: pub Field, val: pub Field) -> pub Field{
    arr[idx] = val;
    arr[idx]
}
```

Fig. 5.3: Programa de Noir minimal con escritura de memoria

```
private parameters indices : [0, 1, 2, 3]
public parameters indices : [4, 5]
return value indices : [6]
INIT (id: 0, len: 4, witnesses: [_0, _1, _2, _3])
MEM (id: 0, write x5 at: x4)
MEM (id: 0, read at: x4, value: x7)
EXPR [ (1, _6) (-1, _7) 0 ]
```

Fig. 5.4: ACIR para el programa de ejemplo con escritura de memoria

- **blockId**: una constante para identificar el índice del bloque previamente creado.
- **index**: un witness que representa el índice que se quiere leer del arreglo.
- **value**: un witness que está restringido al valor que se quiere escribir en el arreglo.

Podemos pensar una operación de lectura como una restricción de la forma

$$B_{blockId}[w_{index}] := w_{value}$$

Veamos nuevamente un ejemplo minimal con un programa de Noir, como el de la figura 5.3. Este programa recibe nuevamente un arreglo de tamaño 4 y un índice público, pero también un valor público que es el que será escrito en el arreglo. Este programa escribe el valor val en la posición idx y finalmente retorna el valor escrito.

Podemos ver el ACIR generado para este programa en la figura 5.4. Notamos que en este caso tenemos nuevamente 4 witnesses privados pero 2 públicos (uno para el índice y otro para el valor), y el witness w_6 reservado para el valor de retorno. Comenzamos al igual que en el ejemplo anterior con la inicialización del bloque de memoria. El siguiente opcode en este caso es un MemWrite que escribe en el bloque de memoria inicializado en el índice notado por el witness w_4 (idx) el valor del witness w_5 (val). Lo siguiente es una operación de lectura, que "guarda" el valor de ese índice en el witness w_7 y por último un AssertZero para igualarlo al witness de retorno con $w_6 = w_7$.

5.1.3. BlackBoxFunctions

Las funciones de caja negra son una herramienta muy abstracta que tiene el código ACIR para delegar la implementación de una función concreta al proving system. La filosofía detrás de este opcode es la misma que las operaciones de memoria: un código ACIR podría modelar cualquier programa de Noir con una cantidad suficientemente grande de AssertZeros, sin embargo esto es potencialmente muy costoso y puede desaprovechar características propias del proving system que utilicen, como pueden ser las lookup tables mencionadas en la sección 2.6.2.

Cuando hablamos de "delegar la implementación de una función concreta al proving system", estamos diciendo que le decimos al proving system "quiero restricciones en tu modelo para esta función especifica, donde los inputs van a ser estos witnesses y los outputs estos otros witnesses". Podemos pensarlas como restricciones extremadamente complejas de un sistema de ecuaciones cuya implementación es responsabilidad del prover que interprete el código ACIR.

Algunos ejemplos de estas funciones son:

- `RangeCheck(w, num_bits)`: opcode que aparece cuando queremos restringir un witness a determinado rango de valores $[0, 2^{num\_bits})$. Es útil por ejemplo cuando queremos trabajar con `u8, u16, u32`. Un ejemplo minimal de programa de Noir que dispararía el uso de un opcode `BlackBoxFunction::RangeCheck` sería el que se puede ver en la figura 5.5. Su ACIR generado se puede ver en 5.6.

```
fn main(a: u32) -> pub u32 {
    a
}
```

Fig. 5.5: Programa de Noir minimal con RangeCheck

```
private parameters indices : [0]
public parameters indices : []
return value indices : [1]
BLACKBOX::RANGE [(_0, num_bits: 32)] [ ]
EXPR [ (-1, _0) (1, _1) 0 ]
```

Fig. 5.6: ACIR para el programa de ejemplo con RangeCheck

- `XOR(num_bits, w_left, w_right, w_out)`: restringe w_{out} a ser el resultado de la operación de XOR bit a bit entre los witnesses w_{left} y w_{right}, asumiendo que estos tienen num_bits bits.
- `AND(num_bits, w_left, w_right, w_out)`: restringe w_{out} a ser el resultado de la operación de AND bit a bit entre los witnesses w_{left} y w_{right}, asumiendo que estos tienen num_bits bits.
- Funciones de hash como Sha256, Poseidon2, Keccak256, PedersenHash, Blake2s, Blake3, etc.
- Operaciones con puntos de curvas elípticas.

5.1.4. BrilligCall

Para explicar el opcode `BrilligCall` primero tenemos que hablar de las **unconstrained functions** en Noir. Noir permite definir funciones que no establecen restricciones sobre el código ACIR (de ahí el nombre *unconstrained*). Estas se definen y se usan como

```
fn main(a: Field, b: Field) -> pub Field{
    suma_insegura(a, b)
}

unconstrained fn suma_insegura(a: Field, b: Field) -> Field {
    a+b
}
```

Fig. 5.7: Programa de Noir con función *unconstrained*

```
private parameters indices : [0, 1]
public parameters indices : []
return value indices : [2]
BRILLIG CALL inputs: [_0,_1] outputs: [_3]
EXPR [ (1, _2) (-1, _3) 0 ]
```

Fig. 5.8: ACIR para el programa de ejemplo con función *unconstrained*

se puede ver en la figura 5.7. El ACIR generado es bastante particular, y lo podemos ver en la figura 5.8.

El ACIR en la última linea genera una restricción de la forma $w_2 = w_3$, donde w_2 es el output del programa y w_3 es el output del `BrilligCall` de la linea anterior. Pero ¿qué significa esta Brillig Call para el prover que va a leer el código ACIR? Recordemos que en el flujo de Noir descrito en la sección 5 el compilador de Noir hace 2 cosas: por un lado genera el código ACIR y por otro, calcula a partir de los inputs provistos por el usuario un diccionario con valores para todos los witnesses. Un valor obtenido como resultado de un llamado a una función no restringida es una **pista** para el prover, es decir, un valor que no tiene restricciones. No es más que un witness que toma un valor dado. El prover no tiene que hacer nada respecto al este witness, no tiene que establecer ninguna restricción nueva de por sí, solo tiene que poder acceder a su valor al momento de generar la prueba. Vamos a re-visitar este tema cuando veamos el flujo concreto con Plonky2.

Por lo pronto, vamos a comentar que también existen usos implícitos de brillig call en la generación de código ACIR, por ejemplo cuando en un programa de Noir calculamos el inverso multiplicativo $\frac{1}{n}$ de un valor en $\mathbb{F}_p$. Esta cuenta requiere muchas más operaciones para hacerse que para verificarse, por lo tanto la estrategia de optimización que utiliza Noir es resolver el valor por un lado (al momento de resolver los witnesses) y verificarlo por otro (a través de AssertZeros). Este es un caso donde la verificación no sigue el mismo camino que todas las operaciones que se hicieron para llegar al resultado. Así como esta operación está incorporada en Noir, las `unconstrained functions` permiten que el usuario pueda elegir qué operaciones serán verificadas y cuáles no.

5.1.5. Ciclos en Noir

Noir permite al programador usar ciclos como cualquier lenguaje de alto nivel, con la salvedad de que la cantidad de iteraciones tiene que estar definida en tiempo de generación

de circuito. Esto es algo que también ocurre con la definición de listas en el lenguaje, las cuales tienen que tener un tamaño conocido en tiempo de generación de circuito. Pensemos durante un segundo por qué esta restricción existe.

Un programa escrito en Noir va a traducirse en última instancia a un sistema de ecuaciones, donde estas están fijadas y no dependen de los valores que tomen las variables sobre las que predican. En cierto sentido, podemos pensar que la cantidad de operaciones está fija. Hacer un ciclo de tamaño variable (es decir, determinado por un input) es equivalente a definir un sistema de ecuaciones donde la cantidad de ecuaciones que lo conforman depende de los valores que tomen las variables. En otras palabras, no tiene sentido conceptualmente hablando.

Es posible diseñar programas que tengan ciclos con cantidad de iteraciones variable, pero estableciendo un máximo de iteraciones. También hay que tener en cuenta en esos casos que todas las ejecuciones que resulten de ese programa tendrán la misma cantidad de operaciones (equivalente a la cantidad máxima de iteraciones declarada). Esto es similar a lo que ocurre cuando usamos un término condicional: ambas partes de la guarda serán "evaluadas" en términos del sistema de ecuaciones generado.

Para pensarlo en términos de compiladores más tradicionales, los ciclos en Noir siempre son desdoblados en tiempo de generación de circuito. Todas las restricciones pertenecientes al cuerpo del ciclo se repetirán tantas veces como cantidad de iteraciones haya. Para que el compilador pueda hacer esto tiene que conocer esta cantidad de iteraciones.

5.2. Barretenberg

La mayoría del background técnico de este trabajo tiene como objetivo poder decir lo siguiente y que tenga sentido: **Barretenberg es un proving system basado en Turbo-PLONK**[1] **que usa KZG como su Polynomial Commitment Scheme**. Esta misma frase no hubiera significado nada sin haber pasado por todo lo anterior, y si bien hay muchos detalles implementativos y de protocolo involucrados en el desarrollo de la herramienta, ahora por lo menos tenemos los recursos para entender conceptualmente qué está ocurriendo por debajo.

Un proving system de estas características necesita un CRS para KZG, cuyo tamaño va a depender del largo de los programas que se deseen demostrar (ya que estos afectan directamente el tamaño de la traza). No hay casi documentación sobre Barretenberg, por lo tanto la mayoría de la investigación relacionada a su funcionamiento está en el código fuente. Primeramente, Barretenberg tiene que tener la capacidad de leer e interpretar código ACIR y los witnesses generados por Noir. No casualmente, fue creado con este propósito, y esta funcionalidad está incorporada en la herramienta.

Como ya mencioné en la introducción, la idea de este trabajo es atacar el problema del CRS en Barretenberg, habilitando a Noir para ser usado con un proving system que no lo requiera. El proving system elegido para esto es Plonky2.

[1] Actualmente Barretenberg ya no usa Turbo-PLONK sino que migró a otro proving system basado en el protocolo UltraHonk [14]. Hasta el momento que finalizó el desarrollo, Turbo-PLONK era la implementación por defecto.

6. PLONKY2

Plonky2 es un proving system basado en Turbo-PLONK que usa FRI como su Polynomial Commitment Scheme. El cuerpo finito usado por Plonky2 se llama **Goldilocks Field**, y su orden es

$$p_{goldilocks} = 2^{64} - 2^{32} + 1$$

Esto quiere decir que todas las operaciones que ocurran sobre elementos nativos en un programa bajo el modelo de Plonky2 se harán sobre el cuerpo de Goldilocks.

Plonky2 está implementado como una biblioteca de Rust [15]. A diferencia de Barretenberg, Plonky2 no fue creado con el propósito ni la funcionalidad de interpretar código ACIR. A pesar de esto, Plonky2 posee una API para creación de programas genéricos, que de ahora en adelante llamaremos **circuitos**. El nombre "circuito" se debe a la estructura con la que nos invita a pensar la API este tipo de programas.

La API nos provee herramientas para construir programas: crear variables, declararlas como inputs, combinarlas en operaciones binarias para tener operaciones de suma, resta, multiplicación, generar condicionales, etc. La comprensión y el uso de esta API es central en el trabajo realizado, por lo tanto voy a hacer un resumen que resulte suficiente para describir las acciones tomadas a través de un ejemplo de cómo se ve un circuito de Plonky2. El ejemplo puede verse en el programa 1:

Veamos el código poco a poco. Se comienza definiendo una **configuración** para el circuito. En este caso estamos tomando una configuración default `standard_recursion_config`. La configuración define varios atributos del circuito a construir, algunos ejemplos son:

- Ancho de la traza: 135 columnas
- Cuántas columnas forman parte del argumento de permutación: 80
- Constantes habilitadas por cada gate: 2
- Configuración de FRI

Con esta configuración vamos a crear un `CircuitBuilder`, un objeto que encapsula toda la creación del circuito. Las "variables" del circuito van a ser los **targets**. Podemos crear un target con el método `builder.add_virtual_target()` y registrarlo como input con el método `builder.register_public_input(x)`. En nuestro ejemplo, el programa tiene 2 public inputs: el que sería el **input** en un programa tradicional y el que sería su **output**. Desde el punto de vista del protocolo, esto se debe a que el prover va a querer demostrar la ejecución de ese programa para un input público pero también va a querer que el verifier corrobore que el resultado es el esperado, por lo tanto el output del programa es considerado un "input público" por Plonky2.

Como vemos, cada operación crea un nuevo target, y esto se debe a que los targets son inmutables. Podemos pensarlos como los cables en un circuito como el de la figura 2.1a. Al mismo tiempo, esta metáfora no es del todo correcta ya que los targets pueden aparecer más de una vez en el circuito, entonces quizás simplemente es más sencillo pensarlo como variables en un sistema de ecuaciones. De esta manera, cuando vemos una operación como `let c = builder.add(a, b)`, podemos pensar que se agrega una restricción al sistema

```rust
fn main() {
    // Configuración que va a tener el circuito
    let config = CircuitConfig::standard_recursion_config();

    // Circuit Builder, la herramienta para construir circuitos
    let mut builder = CircuitBuilder::new(config);

    // Crear una variable y registrarla como input público.
    let x = builder.add_virtual_target();
    builder.register_public_input(x);

    // Operaciones del circuito
    let a = builder.mul(x, x);
    let b = builder.mul_const(F::from_canonical_u32(4), x);
    let c = builder.add(a, b);
    let d = builder.add_const(c, F::from_canonical_u32(7));

    // Registrar al output como un public input, igual que en PLONK
    builder.register_public_input(d);

    // Construir el circuito a partir de las operaciones propuestas
    let circuit_data = builder.build();

    // Crear un PartialWitness con los valores que no tienen dependencias
    let mut patial_witness = PartialWitness::new();
    patial_witness.set_target(x, F::from_canonical_u32(1));

    // Crear la prueba
    let proof = circuit_data.prove(patial_witness)?;

    // Verificar la prueba
    circuit_data.verify(proof)
}
```

Listing 1: Ejemplo de armado de un circuito simple en Plonky2

de ecuaciones de la forma $a+b-c=0$. La API dispone de muchas operaciones que iremos encontrando a medida que las usemos para nuestros propósitos.

Una vez que la forma del circuito es la que deseamos, tenemos que construir el circuito propiamente dicho con `let circuit_data = builder.build()`. El objeto de tipo `CircuitData` que obtendremos estará compuesto por la **proving key** y la **verifying key**, que contienen toda la información necesaria para generar y verificar la prueba respectivamente.

El siguiente elemento importante en este flujo es el `PartialWitness`, un objeto que servirá para proveer valores concretos a ciertos targets para generar la prueba $\pi_{plonky2}$. En esencia, es un diccionario potencialmente incompleto que mapea targets a elementos de $\mathbb{F}_p$. Podríamos proveer al `PartialWitness` con el valor para todos los targets, resolviendo el valor de cada uno de ellos por nuestra cuenta. Sin embargo, Plonky2 sabe resolver los valores de los targets. Esto es posible porque todos los targets tienen dependencias para ser resueltos: por ejemplo, para resolver el valor del target c en la expresión `let c = builder.add(a, b)` se necesita conocer los valores asignados a a y b. Más generalmente, Plonky2 solo necesita conocer los valores asignados a los targets que **no tienen dependencias** y con eso puede resolver el resto. Si hay targets que no puede resolver, significa que 1. el circuito está mal formado o 2. los valores provistos no fueron suficientes para resolver todas las dependencias.

En este caso particular alcanza con proveer al `PartialWitness` con el valor del target nombrado x, que es lo que hace la expresión

```
patial_witness.set_target(x, F::from_canonical_u32(1))
```

Notamos que como Plonky2 solo trabaja nativamente con elementos de $\mathbb{F}_p$, los valores que demos para los targets tienen que estar en $\mathbb{F}_p$, que es lo que consigue la expresión `F::from_canonical_u32(1)`. Finalmente el `circuit_data` nos abstrae la acción de generar y verificar la prueba.

6.1. Lookup tables en Plonky2

La API de Plonky2 también nos provee mecanismos para definir y realizar consultas en lookup tables, tales como están definidas en la sección 2.6.2. En realidad, las lookup tables en Plonky2 presentan leves diferencias y restricciones respecto a la definición general. Se definen como `vec<(u16,u16)>`, es decir, dos columnas con un máximo de 2^{16} elementos de tamaño máximo $2^{16}-1$.

Una vez que tenemos definido un listado de valores

```
let table: vec<(u16,u16)> = [...]
```

podemos usar el método

```
let table_index = builder.add_lookup_table_from_pairs(table)
```

para indicar al `CircuitBuilder` que queremos una tabla definida con dichos valores. Estos valores tienen que ser conocidos en tiempo de generación de circuito, es decir, deben ser valores explícitos que están hardcodeados en el circuito y no dependen de la ejecución concreta.

Para usar estas lookup tables necesitamos un target t. Podemos ver en tiempo de resolución de circuito si el valor de este target se encuentra en la primera componente de alguna de las tuplas definidas en la tabla, usando el método

```
let lookup_result = builder.add_lookup_from_index(t, table_index)
```

donde `table_index` es el índice de la tabla creada con el método

```
table_index = builder.add_lookup_table_from_pairs().
```

En este caso el resultado de llamar al método será otro target `lookup_result` que estará restringido por el circuito a tomar el valor de la segunda componente de la tupla.

En pocas palabras: una lookup table en Plonky2 nos permite definir un diccionario $u16 \to u16$ que opera de forma eficiente, pero que tiene que estar definido en tiempo de generación de circuito para realizar consultas sobre valores dinámicos en tiempo de resolución de circuito. Si en tiempo de resolución de circuito (mientras se genera la prueba) se realiza un lookup para un valor que no está en la primera componente de ninguna tupla, la generación de la prueba fallará.

6.2. Gates y Generators

En Plonky2 y cualquier otro proving system es muy importante para entender la diferencia entre hacer una cuenta matemática y verificar que la misma está bien hecha. Tomemos por ejemplo el caso de la descomposición de un número en n bits. Un algoritmo básico para realizar esta operación es iterar n veces sobre un número, que va a ir dividiéndose sucesivamente por 2 y guardando en cada iteración su resto módulo 2. Si se quisiera expresar esta serie de operaciones como un sistema de ecuaciones implicaría el uso de muchas restricciones. Más aún, si se quisiera representar una operación de XOR de n bits habría que descomponer primero ambos términos, después aplicar la operación bit a bit y por último afirmar que el resultado de la operación está formado por la descomposición obtenida. Sin embargo, si queremos verificar la operación alcanza con hacer únicamente el último paso.

Supongamos que tenemos un valor k y queremos obtener su descomposición en n bits $(k_0, k_1, \cdots, k_{n-1})$. La verificación de esta cuenta puede hacerse con una única ecuación de la forma

$$k_0 \cdot 2^0 + k_1 + 2^1 + \cdots + k_{n-1} * 2^{n-1} = k$$

Entonces nace la pregunta "¿de dónde sacamos los valores de $k_0, k_1, \cdots, k_{n-1}$?". Recordemos que el proceso de generar una prueba de tipo PLONK se basa en 2 pasos fundamentales: armar el sistema de ecuaciones y proveer una solución válida para el mismo. Esta idea de "proveer una solución válida" es la que separa la implementación de la verificación. El proving system no tiene la necesidad de realizar paso por paso las operaciones que llevaron a la obtención de un número, sino que su única responsabilidad es verificar que esta cuenta es correcta.

Volviendo concretamente a Plonky2, esta distinción está dada por las **Gates** y los **Generators**. Las Gates son abstracciones del modelo de Plonky2 que encapsulan la verificación de las operaciones, mientras que los Generators son abstracciones que se ocupan de calcular los valores que van a satisfacer a las Gates. Por ejemplo: Plonky2 cuenta con una ArithmeticGate y un ArithmeticGenerator que van de la mano, de manera que las operaciones calculadas por el Generator son Verificadas por la Gate. Esta dualidad se ve también en operaciones más complejas, como puede ser hallar el inverso multiplicativo de un valor en $\mathbb{F}_p$. También existe para la descomposición en bits.

En el caso general, un programa de Plonky2 puede requerir la descomposición en bits de un número que no es conocido en tiempo de compilación (cuyo valor va a conocerse recién cuando se trate de una ejecución concreta, un input de usuario). Un Generator en este contexto es una forma de establecer cómo va a ser realizada una cuenta, para luego ser verificada por una Gate. Los Generators son los que resuelven el sistema de ecuaciones en su totalidad, exceptuando a los valores iniciales brindados por el usuario.

7. NOIRKY2

Haber recorrido Noir y Plonky2 nos permite re-definir el problema que estamos intentando resolver en términos más concretos. Dijimos en la sección 4 que la intención es adaptar Noir para que pueda usar Plonky2 para generar pruebas de ejecución. Concretamente, el trabajo consiste en hacer un programa (el cual llamaremos **Noirky2** de ahora en adelante) que tiene como dependencias a Nargo (el compilador de Noir) y a Plonky2. Tanto Nargo como Plonky2 están escritos en Rust, por lo tanto Noirky2 también estará programado en Rust. La figura 7.1 muestra un esquema en alto nivel de la interacción de los programas involucrados.

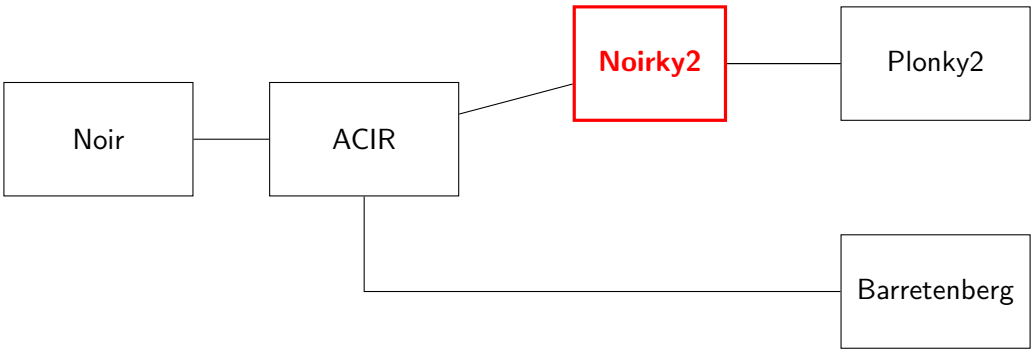


Fig. 7.1: Flujo de compilación de Noir hacia distintos backends

Noirky2 tiene que poder:

- Leer los artefactos generados por Noir, es decir, el ACIR y el diccionario de witnesses a elementos de cuerpo. Para parsear estos elementos correctamente vamos a usar de forma directa el código de Nargo, ya que nos vamos a manejar con el mismo sistema de objetos que Nargo usa internamente.
- Generar un circuito de Plonky2 que sea semánticamente equivalente al código ACIR.
- Responder a los mismos comandos que Barretenberg provee: **prove**, **create_vk** y **verify**. La semántica de estos debe ser la misma que en Barretenberg.

7.1. Cuerpo finito utilizado

Esta solución haría que tanto Noir como Plonky2 sean agnósticos a la existencia de Noirky2, ya que Noirky2 solo usa a Nargo y Plonky2 como dependencias y no al revés. Sin embargo hay lidiar con una dificultad extra: el cuerpo finito usado por Noir no es el mismo que el usado por Plonky2. Adicionalmente, el desarrollo de Noirky2 se produce en simultaneo con el desarrollo de Nargo por parte de otro equipo de trabajo (Aztec [16]), por lo tanto el repositorio de Nargo está en constante cambio. La decisión respecto a esto fue crear un **fork** del repositorio de Nargo y realizar las modificaciones pertinentes sobre el mismo. La cantidad de cambios en el compilador hacía muy engorrosa cualquier otra modalidad de trabajo; aunque al principio se intentó mantener una versión actualizada rápidamente se descartó esta alternativa.

La primera traba del trabajo fue la siguiente: hacer que la resolución de witnesses de Nargo se realizara con un cuerpo finito con $p_{goldilocks} = 2^{64} - 2^{32} + 1$ (primo utilizado por Plonky2) en lugar de $p_{grumpkin} = 2^{254} + 2^{224} + 2^{192} + 2^{96} - 1$ (primo usado por Nargo y Barretenberg).

Pensemos por qué esto es un problema con un ejemplo más simple. Supongamos que $p_{nargo} = 7$ y $p_{plonky2} = 5$. Hacer un programa en Noir que verifique $a + b = c$ implicaría que todas las operaciones ocurren módulo 7, por lo tanto si proveemos inputs como $a = 5, b = 5, c = 3$, el programa verificaría correctamente ya que $5 + 5 \equiv 3$ (7). Sin embargo, si trasladamos estos mismos inputs a Plonky2 e hiciéramos un programa que hace lo mismo, la ecuación no verificaría porque $5 + 5 \not\equiv 3$ (5). Esto hace que los witnesses resueltos por Noir no sean utilizables por Plonky2 a menos que estén usando el mismo cuerpo finito. Vamos a necesitar el valor de esos witnesses para generar la prueba de Plonky2.

La descripción de la solución es puramente implementativa y no es de interés para este trabajo, así que simplemente nos va a alcanzar con decir que esta modificación fue realizada con éxito, pero al mismo tiempo el cambio en el compilador generó la necesidad de usar una versión forkeada de Nargo durante el resto del desarrollo.

Otro problema relacionado con el uso de $p_{goldilocks} = 2^{64} - 2^{32} + 1$ en Plonky2 es que no permite representar fácilmente valores de 64 bits, una funcionalidad que Noir provee. Esto se debe a que $p_{goldilocks} < 2^{64} - 1$ por lo tanto un `u64` no "entra" en un elemento de $\mathbb{F}_p$. Esta dificultad se podría saltear usando dos elementos de $\mathbb{F}_p$ para representar un `u64`, donde cada elemento de cuerpo representa los 32 bits menos significativos o los 32 bits más significativos del `u64`. Sin embargo, esto sigue resultando insuficiente ya que habría que modificar la resolución de Witnesses por parte de Nargo y su forma de armar el código ACIR. En particular, al usar `u64` en un programa de Noir e intentar compilarlo con nuestro Nargo modificado para usar $p_{goldilocks}$, el compilador genera un error "`Range constraint of 64 bits is too large for the Field size`". Por estas cosas vamos a impedir el uso de `u64` y `u128` en Noirky2.

7.2. Tiempos de ejecución

Hay un concepto que es tan importante para este proyecto que merece una sección aparte. Noirky2 cuenta con 3 momentos de ejecución distintos. Entender la diferencia entre ellos es crucial para entender conceptualmente donde estamos parados y qué recursos tenemos en cada momento. Notar que esto es algo abstracto y no nos estamos refiriendo literalmente a la cantidad de tiempo que lleva cada fase, sino al momento del flujo del programa en el que estamos parados:

1. **Tiempo de compilación**: es el momento donde se compila Noirky2 y no es más que la compilación de un programa de Rust. En esta instancia contamos con nuestro código, pero no sabemos nada acerca de los programas ACIR que vamos a recibir ni de los inputs concretos con los que va a ser ejecutado.

2. **Tiempo de generación de circuito**: es el momento en la **ejecución** de Noirky2 en donde el circuito de Plonky2 está siendo construido. En esta instancia tenemos el código ACIR del programa a traducir, pero no usamos todavía los valores concretos con los que el circuito va a ser ejecutado. Esta idea va de la mano de la independencia entre el circuito y los inputs concretos con los que se ejecuta. No podemos usar la información de los inputs para construir el circuito de Plonky2 de ninguna forma.

3. **Tiempo de resolución de circuito**: es el momento en que se genera la prueba, cuando los valores concretos son inyectados en el circuito. Plonky2 tiene la capacidad de asignar valores a todos los targets, resolviéndolos a través de sus dependencias.

Es importante entender también que Noirky2 empieza su flujo cuando Noir ya construyó el código ACIR. También es importante percibir la independencia entre la creación del ACIR y la resolución de witnesses: se podría compilar un código fuente de Noir y solamente generar un código ACIR, sin la resolución de los witnesses. Paralelamente, también se debería poder ejecutar Noirky2 para traducir un código ACIR en un circuito de Plonky2 sin contar con estos valores concretos, solamente con el objetivo de generar un circuito vacío sin la prueba asociada.

Toda la traducción que veremos a continuación ocurre durante el tiempo de generación de circuito. El tiempo de resolución de circuito comienza cuando decidimos proveer el `PartialWitness` con los valores concretos para generar la prueba, pero Noirky2 no tiene control sobre eso, sino que toda esa parte del flujo queda delegada en Plonky2 y su mecanismo interno.

7.3. Traducción de ACIR a Plonky2

En esencia lo que queremos hacer es traducir un lenguaje a otro preservando la semántica de los programas. El código ACIR tiene

- Un conjunto de witnesses $W = \{w_i\}$ sobre los cuales opera. Llamamos $val_e : W \to \mathbb{F}_p$ a la función que recibe un witness y devuelve el valor asignado a ese witness en una ejecución e con valores concretos. Para simplificar, vamos a decir val en lugar de val_e.

- Una lista OP de opcodes (AssertZero, operaciones de memoria, BlackBoxFunction y BrilligCall). Notamos $OP[i]$ al i-esimo opcode. Decimos también que $w_i \in OP[j]$ si el witness w_i es referenciado por el j-esimo opcode del código ACIR.

Lo primero que notamos es que dado un programa P de ACIR, todos los witnesses de P van a tener un target equivalente en un circuito de Plonky2 P' que tenga la misma semántica. Esto no va a ocurrir a la inversa, ya que un opcode de ACIR podría tener una implementación arbitrariamente compleja en Plonky2 que requiera el uso de muchos más targets. Por ende, siendo W el conjunto de witnesses de P y T el conjunto de targets en P', vamos a tener una función inyectiva

$$witness\_target\_map : W \to T$$

que asigna a cada witness de P un target de P'. Concretamente, esta función va a ser implementada con un diccionario que tendremos que mantener durante toda la generación de P'. ¿Por qué es necesario esto? Hay varios motivos para tener este mapeo:

1. Supongamos que en el código ACIR tenemos el listado OP de opcodes y que un witness w_k aparece mencionado en más de un opcode. Si nosotros asociamos un target t a w queremos referirnos al mismo t cuando vuelva a aparecer w más adelante. Para eso definimos la función tal que

$$witness\_target\_map(w) = t$$

2. Cuando generamos la prueba de Plonky2, tenemos que proveer al `CircuitBuilder` con una función

$$partial\_witness : T \to \mathbb{F}_p$$

que asigna a los targets de P' un valor y también es representada como un diccionario. Pero ¿cómo sabemos cuál es el valor correspondiente a un target? Para resolver esto vamos a tener que usar la información de las funciones

$$witness\_target\_map : W \to T \text{ y } val : W \to \mathbb{F}_p$$

Con ambas, dado un witness $w \in P$ podemos saber cual es su target correspondiente en P' y su valor en una ejecución concreta, por lo tanto también podemos construir la función *partial_witness* para que

$$partial\_witness(witness\_target\_map(w)) = val(w)$$

Otra cosa que notamos es que la traducción se puede hacer de forma secuencial sobre el listado de opcodes siguiendo las siguientes reglas:

1. Al principio vamos a registrar los public inputs. Dado el ACIR

```
private parameters indices : PUB
public parameters indices : PRIV
return value indices : OUT
```

Donde $PUB, PRIV, OUT \subset W$ con $PUB \cap PRIV = \emptyset$, $PUB \cap OUT = \emptyset$ y $OUT \cap PRIV = \emptyset$. Vamos a crear targets para todos esos witnesses y registrarlos en la función $witness\_target\_map(w)$ haciendo que los witnesses en PUB y OUT sean públicos para el circuito de Plonky2.

2. Cada vez que aparezca un witnesses nuevo en un opcode, vamos a registrarlo en la función $witness\_target\_map(w)$ (sin hacerlo público ya que los únicos targets públicos deberían ser aquellos que son declarados públicos por el código ACIR).

Esas dos reglas nos garantizan que siempre que necesitemos acceder al target correspondiente a un witness con la función *witness_target_map* este va a estar disponible. A continuación, vamos a hacer un recorrido por todos los opcodes para ver cómo se traduce cada uno en nuestro modelo, generando un circuito de Plonky2 consistente y que respeta la semántica del programa en ACIR.

7.3.1. Traducción de AssertZero

Este es posiblemente el opcode con la traducción más directa, pero va a servir para explicar por qué la función *witness_target_map* no es biyectiva. Supongamos que recibimos un opcode AssertZero de la forma

- Términos cuadráticos: $[(a_0, w_{l,0}, w_{r,0}), (a_1, w_{l,1}, w_{r,1}), \cdots (a_n, w_{l,n}, w_{r,n})]$
- Términos lineales: $[(b_0, w_0), (b_1, w_1), \cdots, (b_m, w_m)]$

- Constante: c

donde $\{a_i\}, \{b_i\}, \{c\} \subset \mathbb{F}_p$. Vamos a comenzar asignando targets a todos los witnesses que aún no estén asignados en el diccionario *witness_target_map*. Podemos pensar entonces que tenemos en el modelo de Plonky2

- Términos cuadráticos: $CUAD = [(a_0, t_{l,0}, t_{r,0}), (a_1, t_{l,1}, t_{r,1}), \cdots (a_n, t_{l,n}, t_{r,n})]$
- Términos lineales: $LIN = [(b_0, t_0), (b_1, t_1), \cdots, (b_m, t_m)]$
- Constante: c

La API que nos provee Plonky2 para construir circuitos nos invita a pensar con sus funciones en un circuito binario, donde las operaciones tienen 2 inputs y un output. En este caso las operaciones necesarias para llevar esto a cabo son la **suma**, la **multiplicación**, el **assert_zero** y la creación de targets con valor **constante**[1]. El algoritmo a llevar a cabo es el que se puede ver en 2.

Algorithm 2 Traducción del opcode AssertZero

```
1: Entrada: CUAD, LIN, c, builder
2: let accumulator = builder.constant(c)
3: for (b_i, t_i) in LIN do
4:     let linear_combination_target = builder.mul_const(b_i, t_i)
5:     accumulator = builder.add(linear_combination_target, accumulator)
6: end for
7: for (a_i, t_li, t_ri) in CUAD do
8:     let cuadratic_target = builder.mul(t_li, t_ri)
9:     cuadratic_target = builder.mul_const(a_i, cuadratic_target)
10:    accumulator = builder.add(cuadratic_target, accumulator)
11: end for
12: builder.assert_zero(accumulator)
```

La idea es simple: se itera por todos los términos lineales y cuadráticos, combinando targets a través de operaciones binarias. Notamos que $\{a_i\}, \{b_i\}$ y c son constantes en tiempo de generación de circuito; son valores que forman parte del ACIR y por ende son constantes del sistema de ecuaciones. El último paso del algoritmo consiste en usar la operación `builder.assert_zero(accumulator)` para establecer la restricción $accumulator = 0$.

Veamos cómo se traduce todo esto a un ejemplo simple como el que vimos en la sección 2.5. Partamos del programa $result = e \cdot x + x - 1$, donde $result$ y x son públicos y e es privado. El ACIR que generaría esto sería como el que podemos ver en la figura 7.2. Esto derivaría en nuestro sistema en una función

$$witness\_target\_map : \{w_0, w_1, w_2\} \to \{t_0, t_1, t_2\} \text{ , donde}$$

$$witness\_target\_map(w_i) = t_i \quad i = 1, 2, 3$$

Podemos pensar al producto de Noirky2 como un circuito o un sistema de ecuaciones, como podemos ver en la figura 7.3. Ambas representaciones también permiten visualizar

[1] El target en sí no contiene un valor constante porque los targets son solo "cables en el circuito". El `CircuitBuilder` contiene un diccionario que mapea ciertos targets a valores constantes y los va a usar en el tiempo de resolución de circuito.

```
private parameters indices : [0]
public parameters indices : [1]
return value indices : [2]
EXPR [ (1, _0, _1) (1, _1) (-1, _2) -1 ]
```

Fig. 7.2: ACIR para el programa de ejemplo

claramente a qué nos referimos con que no todos los targets tienen un witness correspondiente: t_0, t_1 y t_2 se corresponden con los witnesses w_0, w_1 y w_2 respectivamente, pero todos los targets t_aux_i son "intermedios", en el sentido de que no se corresponden con inputs ni outputs de la operación.

En este punto pueden empezar a surgir dudas sobre qué tan óptima es esta construcción. Vamos a tratar el tema de las optimizaciones más adelante, sin embargo el mecanismo interno con el que Plonky2 arma la traza tiene optimizaciones que vale la pena mencionar ahora. Antes dijimos que la configuración del circuito con la que instanciamos al `CircuitBuilder` establece una traza con 135 columnas, sin embargo todas nuestras las ecuaciones del ejemplo tienen a lo sumo 3 términos. Plonky2 empaqueta operaciones similares en las mismas filas, aprovechando el espacio total de la traza para reducir la cantidad de filas totales. Esto es muy importante a la hora de evitar cierto tipo de optimizaciones que en la práctica no tendrán ningún impacto real.

¿A qué nos referimos con operaciones similares? Plonky2 usa Custom Gates para todas sus operaciones. Todas las operaciones que vimos hasta el momento son distintas configuraciones de una misma `ArithmeticGate`. Esta compuerta sirve para establecer restricciones de la forma

$$t_{result} = const_1 \cdot t_1 \cdot t_2 + const_2 \cdot t_3$$

Podemos ver que estableciendo las constantes correctas, todas las operaciones que vimos hasta el momento (suma, multiplicación, multiplicación por constante, igualdad a 0) pueden obtenerse usando esta compuerta. Esto es más parecido a lo que veníamos viendo con PLONK (2.5) y Turbo-PLONK (2.6 y 2.6.1), dejando de lado detalles implementativos.

La `ArithmeticGate` no es la única compuerta que vamos a utilizar a lo largo de la traducción.

7.3.2. Asignación condicional en Plonky2

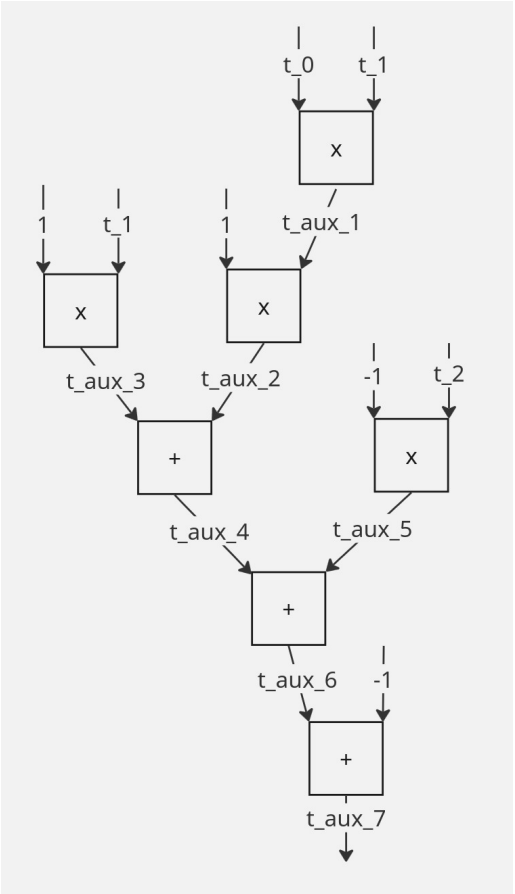
Antes de incursionar en la lectura y escritura de bloques de memoria, veamos cómo podemos implementar una asignación condicional en el modelo de Plonky2. Más concretamente, nos interesa tener una restricción de la forma

```
res = if (x = y) then a else b
```

donde `x,y,a,b` y `res` son targets, y por ende su valor no se conoce en tiempo de generación de circuito. Esta expresión puede ser separada en 2 partes:

```
1.(res = if equal then a else b) | 2.(equal = if (a == b) then 1 else 0)
```

donde `equal` es un target booleano.



(a) Visto como circuito

$$\begin{cases} t_0 \cdot t_1 & = t_{aux_1} \\ 1 \cdot t_1 & = t_{aux_3} \\ 1 \cdot t_{aux_1} & = t_{aux_2} \\ t_{aux_2} + t_{aux_1} & = t_{aux_4} \\ -1 \cdot t_2 & = t_{aux_5} \\ t_{aux_4} + t_{aux_5} & = t_{aux_6} \\ -1 + t_{aux_6} & = t_{aux_7} \\ t_{aux_7} & = 0 \end{cases}$$

(b) Visto como sistema de ecuaciones

Fig. 7.3: Vistas del Assert Zero

Podemos expresar **1.** como un sistema de ecuaciones

$$\begin{cases} not\_z &= 1 - z \\ conditional\_a &= a \cdot z \\ conditional\_b &= b \cdot not\_z \\ res &= conditional\_a + conditional\_b \end{cases}$$

o más simplemente como la ecuación

$$res = z \cdot a + (1 - z) \cdot b$$

Para **2.** el proceso es conceptualmente distinto. Primero vamos a calcular un valor

$$\texttt{diff\_inverse := if } (x \neq y) \texttt{ then } \frac{1}{x - y} \texttt{ else 0}$$

Pero ¿cómo podemos calcularlo si no conocemos los valores que toman x e y en tiempo de generación de circuito? Antes mencionamos que Plonky2 tiene mecanismos para hacer operaciones en tiempo de resolución de circuito: es lo que usa para resolver los valores de todos los targets. La condición de la operación solo puede resolverse en tiempo de resolución de circuito, y eso es lo que va a ocurrir. Conceptualmente no estamos estableciendo una restricción, estamos **fijando** una forma con la que un valor va a ser calculado cuando haya valores concretos. Esta cuenta, hecha por un Generator, viene acompañada por el siguiente sistema de ecuaciones asociado como se vio en la sección 6.2:

$$\begin{cases} equal \cdot (equal - 1) &= 0 \\ equal \cdot (x - y) &= 0 \\ (x - y) \cdot diff\_inverse + equal &= 1 \end{cases}$$

Estas ecuaciones sirven restringir a la variable *equal* a que sea un valor binario: 0 si x e y son distintos y 1 si son iguales.

1. La primera ecuación se ocupa de que *equal* sea binaria.
2. Para analizar la segunda ecuación separemos en casos:
   - **Si x e y son iguales** entonces la ecuación se cumple porque se anula el factor de la derecha.
   - **Si son distintos** entonces la variable *equal* debería tener el valor 0, anulando nuevamente el término.

   Si x e y son distintos pero $equal \neq 0$ la ecuación no se cumpliría, por lo tanto esta ecuación nos da la implicación

   $$x \neq y \implies equal = 0$$

   pero nada más.
3. Para analizar la tercera ecuación también separemos en casos:

- **Si x e y son iguales** entonces la ecuación se cumple si solo si $equal = 1$.
- **Si son distintos** entonces $(x - y) \cdot diff\_inverse = 1$ por lo tanto la ecuación se cumple si solo si $equal = 0$.

Uno podría preguntarse para qué está la segunda ecuación, ¿no alcanza con la primera y la tercera? Pensándolo con cuidado vemos que $diff\_inverse$ no está restringido, por lo que a priori no tendría por qué ser lo que dice ser (eg. $\frac{1}{x-y}$). Si tratamos de romper este sistema de ecuaciones podríamos hacer una asignación que cumpla

- `diff_inverse := 0`
- `equal := 1`
- `x` $\neq$ `y`

Esta asignación cumple la primera y la tercera ecuación, sin embargo es incorrecta respecto a lo que buscamos. Necesitamos que $equal = 1 \iff x = y$ y esto no ocurre. La segunda ecuación nos salva en este caso ya que no hay forma de que cumpla bajo estas condiciones, ya que no existe ningún valor en $z \in \mathbb{F}_p$ no nulo tal que $z \cdot 1 = 0$.

Recién vimos cómo podemos modelar asignaciones condicionales en Plonky2. Esta es una herramienta que nos va a servir para modelar operaciones de memoria, ya que justamente vamos a necesitar asignar condicionalmente un valor dependiendo de la igualdad entre dos índices.

7.3.3. Traducción de operaciones de memoria

Para implementar las operaciones de memoria vamos a pensarlas a través del lente de los tiempos de ejecución mencionados en la sección 7.2. Dijimos cuando presentamos los opcodes de memoria en 5.1.2 que las operaciones de memoria en el código ACIR aparecían en el contexto de accesos a bloques de memoria en posiciones dinámicas, por ejemplo, con un índice como input. ¿Qué significa esto respecto a los tiempos de ejecución? Significa que el índice con el que vamos a acceder al bloque de memoria **no es conocido en tiempo de generación de circuito** sino recién en tiempo de resolución de circuito, por ende no podemos indexar de una manera tradicional.

La estrategia va a ser como sigue: así como teníamos un diccionario `witness_target_map` para mantener una relación entre witnesses y targets, también vamos a tener un listado de bloques de memoria representados como listas de targets que vamos a mantener actualizado a lo largo de toda la traducción. Podemos pensarlo como un objeto global a la traducción

```
memory_blocks: Vec<Vec<Target>>
```

que inicialmente está vacío.

Cuando procesamos un opcode $MemoryInit(idx, [w_{m_1}, \cdots, w_{m_n}])$ significa que el programa requiere la creación de un bloque nuevo de tamaño n. Como los índices son secuenciales y comienzan en 0, podemos simplemente asumir que el i-esimo bloque va a ser el que se encuentre en la posición i-esima de `memory_blocks`. Con esto en mente, la creación de un nuevo bloque de memoria es como sigue:

1. Por cada witness $\{w_i\}$ mencionado en el opcode que no haya sido registrado todavía en la función $witness\_target\_map$ vamos a registrarlo como target $\{t_i\}$, al igual que hacíamos en la implementación de AssertZero.

2. Vamos a crear un vector de targets con todos los targets mencionados en el opcode en ese mismo orden y lo vamos a agregar a `memory_blocks`. La acción concreta sería `memory_blocks.append(`$[t_{m_1}, \cdots, t_{m_n}]$`)`.

Lectura de memoria

Veamos cómo traducir un opcode de lectura de memoria. Este está conformado por un $blockId$ (índice del bloque que se va a leer), w_{index} (witness que contiene al índice que se va a leer) y w_{value} (witness que tomará el valor leído). El índice `block_idx` del bloque que vamos a leer es una constante conocida en tiempo de generación de circuito, por lo tanto podemos obtener fácilmente el bloque con

```
block := memory_blocks[blockId]
```

Recordemos en qué consiste una lectura de memoria en este contexto. Visto como una asignación tendríamos algo como

```
res := block[index]
```

por lo que queremos una restricción de la forma

$$res = block[index]$$

donde res es el target correspondiente al witness w_{value} e $index$ es el target correspondiente al witness w_{index}. La solución es construir un circuito que considere todos los casos (es decir, si se tiene que modificar el índice 0, índice 1, indice 2, etc.). Se recorren todas las posiciones del bloque de memoria, comparando el índice con el valor de $index$ y asignándole condicionalmente a res su propio valor o el valor del target en el índice dependiendo de si la igualdad se cumple o no. El pseudocódigo se puede ver en la figura 3.

Algorithm 3 Traducción del opcode `MemRead`

```
1: Entrada: builder, block, index
2: Salida: res
3: let res: Target
4: for i = 0..block.length() do
5:    let target_position_i = builder.constant(i)
6:    let is_current_position = builder.is_equal(index, target_position_i)
7:    res = builder.conditional_assign(is_current_position, block[i], res)
8: end for
9: return res
```

El pseudocódigo usa 2 funciones que encapsulan el comportamiento descrito anteriormente:

1. `builder.is_equal(a,b)` es un método que recibe 2 targets y devuelve un tercer target que está restringido a tomar el valor 1 si $a = b$ y el valor 0 si $a \neq b$.

2. `builder.conditional_assign(z,a,b)` es un método que recibe un target booleano z y 2 targets a y b. Devuelve a si z toma el valor de 1 y b si z toma el valor de 0.

Escritura de memoria

Veamos cómo traducir un opcode de escritura de memoria. Este está conformado por un $blockId$ (índice del bloque que se va a leer), w_{index} (witness que contiene al índice que se va a leer) y w_{value} (witness que contiene el valor que se va a escribir).

El enfoque que vamos a tomar en este caso es parecido al de la operación de lectura, con la diferencia de que tenemos que modificar el contenido del bloque de memoria. Recordemos que los targets toman un único valor a lo largo de toda la ejecución, por lo tanto para cambiar el valor en una posición del bloque vamos a tener que reemplazar un target por otro. Pero no solo eso: reemplazar un único target sería posible si supiéramos en tiempo de generación de circuito cuál es el índice que tenemos que reemplazar, pero no lo sabemos. En cambio, lo que vamos a hacer es **cambiar todos los targets** de todas las posiciones. Todos los targets excepto uno van a estar restringidos a tomar el mismo valor que su antecesor. La excepción va a ser en el índice a modificar, donde va a tomar el nuevo valor.

Conceptualmente podemos pensar todas las posiciones de manera independiente. En cada posición se hace la pregunta ¿es esta la posición que tiene que ser modificada? Solamente una va a tener un sí como respuesta. En el caso que la respuesta es no, el nuevo target que tomará esa posición tendrá el mismo valor que el anterior. En el caso que la respuesta es sí, el nuevo target que tomará esa posición tendrá el valor nuevo. Es otro caso más donde podemos aplicar la asignación condicional. Se pueden ver gráficamente las acciones a tomar en la figura 7.4 y el pseudocódigo en la figura 4.

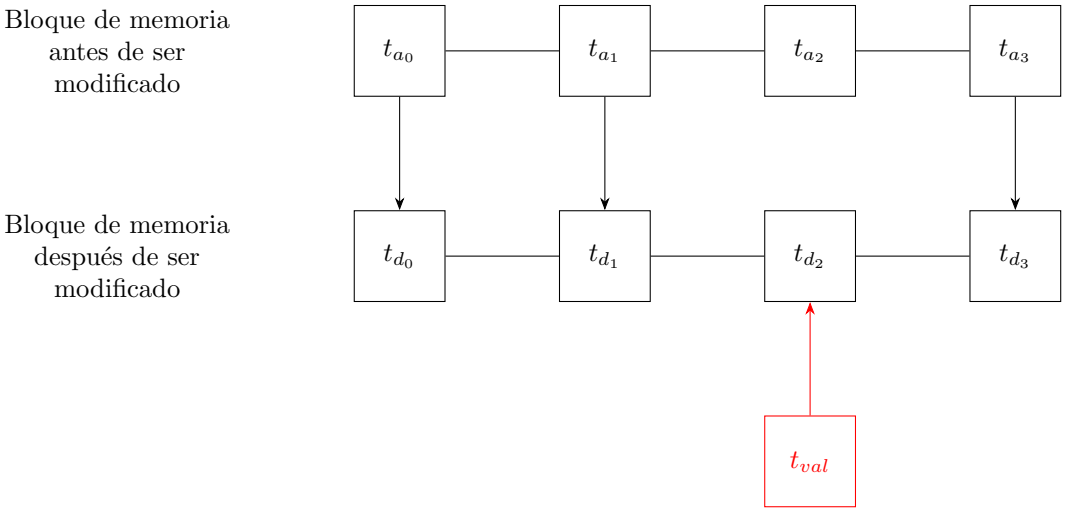


Fig. 7.4: Actualización de un bloque de memoria

Algorithm 4 Traducción del opcode `MemWrite`

```
1: Entrada: builder, block, index, value
2: for i = 0..block.length() do
3:    let target_position_i = builder.constant(i)
4:    let is_current_position = builder.is_equal(index, target_position_i)
5:    let new_target_i = builder.conditional_assign(is_current_position,
   value, block[i])
6:    block[i] = new_target_i
7: end for
```

7.3.4. Traducción de Range-Check

La primera BlackBox Function que vamos a analizar va a ser `BlackBoxFunction::RangeCheck`. Este opcode aparece cuando queremos establecer una cantidad máxima de bits en la representación binaria de una variable. Sus argumentos son `RangeCheck(w, num_bits)` donde `w` es el witness que queremos restringir y `num_bits` la cantidad máxima de bits permitidos. Esto quiere decir que la restricción establecida debería ser

$$0 \leq w < 2^{num\_bits}$$

Este opcode existe como una BlackBoxFunction porque expresar la restricción con ecuaciones polinomiales de forma directa tiene un costo alto. Una solución trivial sería expresar esta condición con varias operaciones de AssertZero, pero la filosofía de Noir en estos casos es delegar operaciones complejas al prover.

Para traducir esta operación vamos a usar 2 estrategias distintas: descomposición en bits y Lookup Tables. La idea es explicar la filosofía detrás de cada una y compararlas en términos de eficiencia.

Range-Check con descomposición en bits

Supongamos que tenemos un opcode `RangeCheck(w, n)`. La idea básica es crear (u obtener si ya existe) al target t correspondiente a w, asumir que puede ser expresado con n bits y establecer ecuaciones para descomponerlo en los bits $b_0, b_1, \cdots, b_{n-1}$. La ecuación sería

$$b_0 + 2^1 \cdot b_1 + 2^2 \cdot b_2 + \cdots + 2^{n-1} \cdot b_{n-1} = t$$

$$\iff$$

$$\prod_{i=0}^{n-1} 2^i \cdot b_i = t$$

En Plonky2 podemos hacer esto a través de una Custom Gate llamada `BaseSumGate`, que se encarga de la descomposición de números en distintas bases (en este caso, base 2). Si no es posible generar esa descomposición entonces podemos afirmar que el número concreto con el que se está ejecutando no está en rango. Notamos que $b_0, b_1, \cdots, b_{n-1}$ deben ser targets binarios, sino la ecuación no nos daría la información que queremos obtener. Por eso también vamos tener que agregar las restricciones

$$b_i \cdot (b_i - 1) = 0 \quad \forall i \in [0, n-1]$$

input	output
0	-
1	-
...	...
255	-

Tab. 7.1: Lookup table para un range-check de 8 bits

Es importante destacar que los valores de b_i van a ser calculados por Plonky2 en tiempo de resolución de circuito en base al valor concreto que se le asigne a t. Le vamos a indicar a Plonky2 que lo descomponga en una cantidad fija de elementos, y este va a intentar hacerlo partiendo del bit menos significativo (b_0), sin embargo va a fallar si es incapaz de completar la descomposición del número.

Recordemos también que dado que el primo que usa Plonky2 es $p_{goldilocks} = 2^{64} - 2^{32} + 1$ no podemos trabajar nativamente con números de más de 63 bits. Por esto, también hubo que restringir la operación de range-check.

Range-Check con lookup tables

Otra estrategia posible para resolver esta operación es usar las lookup tables vistas en la sección 6.1. ¿Cómo las usamos? Tenemos que definir una lookup table que contenga en la primera componente de cada tupla todos los valores entre 0 y el rango deseado, y cualquier valor en la segunda componente (ya que solo queremos verificar la presencia de cierta clave en la tabla).

Por ejemplo, para chequear que un target está restringido a un valor $u8$ definimos una tabla como la que se ve en la figura 7.1. Finalmente, el target t asociado al witness que queremos restringir sería el input para el método `builder.add_lookup_from_index(t, table_index)`, donde el resultado puede ser ignorado ya que solo nos interesa la presencia del elemento en la tabla.

Este enfoque nos establece una restricción que no se presentaba en la estrategia anterior: los lookups pueden hacerse hasta valores de $2^{16} - 1$, con lo cual este método solamente puede usarse para el chequeo de rango de hasta 16 bits inclusive. No solo eso, sino que si tenemos una tabla con el rango $[0, 2^n)$, esa tabla solo se puede usar para chequeos de n bits y no menores. Por ejemplo, si tenemos una tabla con rango $[0, 2^{16})$ no podemos usarla para valores $u8$, sino que tendríamos que tener otra tabla definida con rango $[0, 2^8)$.

También hay que tener en cuenta el overhead de tener la lookup table: las consultas son computacionalmente baratas pero la creación de la tabla puede ser costosa para Plonky2. Especialmente se puede dar una situación donde tengamos un programa pequeño y una lookup table muy grande, en cuyo caso quizás no vale la pena definirla. La idea es entrar más en detalle con estos temas cuando hagamos un análisis y comparación de la performance de los distintos métodos.

Range-Check con descomposición en limbs y lookup tables

Una alternativa intermedia entre las dos opciones consideradas es hacer una descomposición del número, pero no en base 2 (bits), sino en una base más grande. Por ejemplo,

un número de 32 bits puede pensarse como la composición de cuatro números de 8 bits; entonces alcanzaría con verificar que esas 4 componentes (o *limbs*, como solemos llamarlas) están en el rango de un $u8$ y juntas conforman al número que queremos restringir. Para verificar el rango de $u8$ podemos reutilizar la idea de las lookup tables.

¿Cómo separamos al target t en limbs de 8 bits cada una? Podríamos pensar en usar la compuerta que usamos para la separación en bits: `BaseSumGate`, pero esta no soporta bases mayores a 9 y necesitamos una base 256. Esto nos obliga a meternos más adentro de Plonky2 para entender cómo construye en tiempo de resolución de circuito los valores para los targets intermedios, y cómo podemos aprovecharnos de este mecanismo para calcular valores en tiempo de resolución. Recordamos entonces que Plonky2 usa `Generators`, como vimos en la sección 6.2. Estos objetos se suelen asociar a las Gates pero también pueden usarse de manera independiente a ellas. Los Generators:

- calculan el valor que se va a asociar los targets en tiempo de resolución de circuito.
- tienen dependencias de Targets: siempre ocurre que el valor de 0 o más targets tiene que estar resuelto para poder calcular el valor de un target nuevo. En este caso tiene sentido pensar en el circuito de la figura 7.3 a). Por ejemplo, para resolver el valor del target t_{aux_4} es necesario tener antes el valor de t_{aux_2} y t_{aux_3}.
- implementan el método `run_once()`, donde se ejecuta la operación.
- pueden tener una cantidad arbitraria de dependencias y de targets calculados. Por ejemplo, en el ejemplo anterior teníamos 2 dependencias (t_{aux_2} y t_{aux_3}) y un resultado (t_{aux_4}).
- se usan de la siguiente forma: en tiempo de generación de circuito se instancian con targets (de entrada y de salida), haciendo que quede pre-seteado en tiempo de generación de circuito el cálculo que será realizado en tiempo de resolución de circuito.
- no son suficiente para garantizar un circuito correcto: los Generators generan valores que el prover va a usar para llenar la traza, pero no verifican con ecuaciones que los resultados calculados sean correctos. Si simplemente confiamos en que el Generator va a calcular el valor pero no restringimos sus operaciones, estaríamos creando una vulnerabilidad en el sistema.

Con esto en mente, creamos un Generator propio de la forma

```
struct LimbDecomposition8BitsGenerator<const N: usize>
```

Para aquellos no tan familiarizados con Rust, la parametrización generada por `N` permite instanciar al `struct` con distintos valores para el parámetro. Estos valores tienen que ser conocidos en tiempo de compilación. En particular, nos va a interesar una descomposición en 2 limbs para números de 16 bits y una descomposición en 4 limbs para números de 32 bits (recordemos que números de 64 bits no son soportados nativamente por Noirky2).

Internamente, el Generator recibe 1 target `full_number` con su valor asociado y N targets para los limbs $(l_0, \cdots, l_{N-1})$. Su trabajo es calcular el valor de los N limbs. El algoritmo puede verse en la figura 5.

Algorithm 5 Descomposición en N limbs de 8 bits

1: **Entrada 1:** `witness_value_mapping:` $Target \to \mathbb{F}_p$
2: **Entrada 2:** `full_number_target: Target`
3: **Entrada 3:** `limb_targets: [Target; N]`
4: **let mut** `full_number_value = target_value_mapping.get(full_number_target)`
5: **for** `i = 0..N` **do**
6: `let current_limb_value = full_number_value` $\%\ 2^8$
7: `full_number_value = full_number_value` $/\ 2^8$
8: `witness_value_mapping.set(limb_targets[i], current_limb_value)`
9: **end for**

Internamente, Plonky2 cuenta con un `target_value_mapping` donde almacena los valores resueltos de los targets en tiempo de resolución de circuito. Ese diccionario es el que usamos para almacenar los nuevos valores y obtener los valores de las dependencias. Un detalle curioso es que si tratamos de sobreescribir el valor de alguno de los targets, el sistema generará un error en tiempo de resolución de circuito ya que un target no puede tomar 2 valores distintos en la misma ejecución.

Más allá de la ejecución del Generator, mencionamos anteriormente que el circuito tiene que tener las restricciones polinomiales para la ejecución realizada, por lo tanto, debemos establecer una restricción de la forma

$$full\_number_{32} = limb_0 + 2^8 \cdot limb_1 + 2^{16} \cdot limb_2 + 2^{24} \cdot limb_3$$

para números de 32 bits y

$$full\_number_{16} = limb_0 + 2^8 \cdot limb_1$$

para números de 16 bits. Por último tenemos que definir una lookup table con los valores en el rango $[0, 2^8)$ y verificar que cada uno de los limbs se encuentre en este rango. Para resumir, los pasos serían:

- Usar un Generator para establecer el cálculo correcto de los limbs en tiempo de resolución de circuito.
- Descomposición: verificar con restricciones que los limbs efectivamente componen al número.
- Rango parcial: verificar con lookups que los limbs son efectivamente valores $u8$.

Esta es una sofisticación por sobre la alternativa de usar solo lookup tables, ya que nos permite chequear también valores de 32 bits solamente definiendo una única tabla de 2^8 elementos. Se podrían explorar alternativas de descomposición en limbs que no sean de 8 bits, por ejemplo, de 4 bits. Esta opción, aunque posible, va a quedar por fuera del alcance del trabajo.

7.3.5. Traducción de XOR y AND

La forma de traducir la operaciones `BlackBoxFunction::AND` y `BlackBoxFunction::XOR` son muy similares en casi todos los aspectos, por lo tanto ahondaremos únicamente en la explicación del XOR sin perder generalidad más que para la operación subyacente.

a	b	c	$(a-b)^2 - c$
0	0	1	-1
0	0	0	0
0	1	1	0
0	1	0	1
1	0	1	0
1	0	0	1
1	1	1	-1
1	1	0	0

Tab. 7.2: Tabla de verdad para $a \oplus b = c$ respecto a la fórmula $(a-b)^2 - c$

El Opcode `BlackBoxFunction::XOR` está definido por 2 witnesses v, w y la cantidad de bits que poseen (ej. 8, 16, 32), por lo tanto todo programa de Noir que tenga un `XOR` como opcode antes sí o sí deberá tener un `RangeCheck` para sus 2 operandos. Esto nos permite asumir al momento de procesar el opcode que los targets t_v y t_w asociados a v y w respectivamente en el `witness_target_map` fueron chequeados por su rango, por como Noir crea el código ACIR.

Nuevamente tenemos 2 estrategias para esta traducción.

XOR con descomposición en bits

La idea básica es realizar una descomposición en bits igual que como hicimos en la sección 7.3.4, pero ahora llevarlo un poco más lejos y operar con esos bits. Veamos cómo dados 2 targets binarios a y b podemos restringir un tercer target binario c para que se cumpla

$$c = a \oplus b$$

La ecuación que se deriva entonces es

$$(a-b)^2 - c = 0$$

$$a^2 + b^2 - 2ab - c = 0$$

La demostración se puede ver por extensión en la tabla 7.2. Vemos que

$$c = a \oplus b \Leftrightarrow (a-b)^2 - c = 0$$

En el caso del `AND` es más sencillo, ya que la operación es equivalente al producto de bits. Siempre que $a, b \in \{0, 1\}$ se cumple que

$$a \ \& \ b = a \cdot b$$

Ahora veamos la operación completa para los targets a, b y c de k bits. Con la operación en mente para un solo bit, podemos pensar que el range-check anterior nos da una descomposición de cada uno de los targets

$$a = a_0 + 2 \cdot a_1 + \ldots 2^{k-1} \cdot a_{k-1} \quad \text{y} \quad b = b_0 + 2 \cdot b_1 + \ldots 2^{k-1} \cdot b_{k-1}$$

a	b	$a \oplus b$
0	0	0
0	1	1
0	2	2
0	3	3
...	...	...
1	0	1
1	1	0
1	2	3
1	3	2
...	...	...
255	254	1
255	255	0

Tab. 7.3: Tabla de XOR para inputs de 8 bits

Tenemos que realizar la misma descomposición de c para "reconstruir" el resultado luego de la operación bit a bit, definiendo targets $c_0, \cdots c_{k-1}$ y la ecuación

$$c = c_0 + 2 \cdot c_1 + \ldots 2^{k-1} \cdot c_{k-1}$$

Finalmente el sistema de ecuaciones se completaría para el XOR con

$$(a_i - b_i)^2 - c_i = 0 \quad \forall\ i \in [0..k)$$

y para el AND con

$$a_i \cdot b_i - c_i = 0 \quad \forall\ i \in [0..k)$$

XOR con descomposición en limbs y lookup tables

Nuevamente podemos usar lookup tables y descomposición en limbs para resolver nuestro problema, en este caso usando las tablas para guardar resultados pre-computados del XOR. La idea básica va a ser tener una tabla con todos los resultados para inputs chicos, de hasta 8 bits. Una lookup table de esta pinta se puede ver en la tabla 7.3. En ella se calculan todas las combinaciones posibles de resultados para $a \oplus b$ donde a y b son valores de 8 bits. La longitud de esta tabla es de $2^8 \cdot 2^8 = 2^{16}$, el máximo permitido por Plonky2.

Sin embargo vimos que las lookup tables en Plonky2 se definen a partir de 2 columnas, pero esta tabla tiene 3. Para solucionar este problema vamos a codificar los inputs en una única columna como un único número $c = 256 \cdot a + b$. Notamos que esta codificación permite distinguir a los pares de inputs inequivocamente, siendo

$$a = \lfloor \frac{c}{256} \rfloor \quad \text{y} \quad b = c \,\%256$$

Como el valor máximo que pueden tomar a y b es 255, el máximo valor de c es $255 * 256 + 255 = 2^{16} - 1$ que entra dentro del rango de valores posibles de una lookup table de Plonky2.

Una vez definida esta Lookup Table, la forma de realizar la operación va a ser la siguiente. Veámoslo primero para una operación XOR de 8 bits para luego generalizar con

descomposición en limbs, al igual que hicimos con el caso de Range-Check. Para empezar, el opcode se define sobre 2 witnesses de input v, w, 1 witness de salida z y un valor constante, en este caso $num\_bits = 8$. Primero debemos obtener o crear los targets asociados a v, w y z; los nombramos a, b y out respectivamente. Lo siguiente que tenemos que hacer es construir el target que va a ser buscado en la primera columna de la lookup table, es decir, necesitamos un target c tal que $c = 256 \cdot a + b$. Este target puede ser obtenido a partir de operaciones básicas del `CircuitBuilder`, en particular

```
let c = builder.mul_add(256, a, b)
```

luego realizamos el lookup en la tabla definida con

```
let lookup_result = builder.add_lookup_from_index(c, table_index)
```

En este caso sí vamos a utilizar el resultado del lookup, ya que tenemos que restringirlo a que tome el mismo valor que out. Esto lo hacemos a través de la operación

```
builder.connect(lookup_result, out)
```

El caso general para 16 bits y 32 bits consiste en realizar la separación en limbs tal como vimos en la operación de Range-Check y realizar el XOR con la lookup table limb por limb. Veamos el caso para N limbs. Nos aseguramos con restricciones aritméticas básicas de que

$$a = \sum_{i=0}^{N-1} 2^{8i} \cdot limb_{a,i} \quad | \quad b = \sum_{i=0}^{N-1} 2^{8i} \cdot limb_{b,i} \quad | \quad out = \sum_{i=0}^{N-1} 2^{8i} \cdot limb_{out,i}$$

y con lookups nos aseguramos de que

$$limb_{a,i} \oplus limb_{b,i} = limb_{out,i} \quad \forall i \in [0, N)$$

y

$$limb_{a,i} \in [0, 2^8), \quad limb_{b,i} \in [0, 2^8), \quad limb_{c,i} \in [0, 2^8), \quad \forall i \in [0, N)$$

En este caso alcanza con definir una única tabla de 2^{16} filas para el XOR de $8, 16$ y 32 bits. Nuevamente incurrimos en el overhead que conlleva el procesar una tabla por parte del prover.

8. EVALUACIÓN DE LA HERRAMIENTA

8.1. Validación con tests funcionales

Una parte crucial del proceso de desarrollo es validar a través de tests que la herramienta generada está cumpliendo con los requerimientos, sin embargo, en el contexto de los SNARKs y en particular con Plonky2 esto puede llegar a ser un desafío. Supongamos el siguiente escenario: Noirky2 recibe un código ACIR y un conjunto de witnesses como input y a partir de ellos crea un circuito en Plonky2, crea una prueba de este circuito con los valores de witnesses provistos y la verifica. Todo parece correcto, sin embargo, ¿Qué pasaría si la traducción de Noirky2 consistiera en hacer un circuito vacío que siempre verifica? Automáticamente cualquier programa generado debería verificar correctamente, sin embargo sabemos que la implementación no existe.

Tratamos de solucionar este problema haciendo tests por la negativa, es decir, proveyendo datos de entrada a la generación de la prueba que no satisfacen el sistema de ecuaciones y viendo que la verificación falla. Sin embargo no fue la verificación la que falló, sino la generación de la prueba misma. La explicación de esto es que Plonky2 resuelve todos los valores de sus Targets (o variables del sistema de ecuaciones). Si un valor provisto como input para la prueba es distinto a un valor que el motor de Plonky2 resuelve, este va a fallar en tiempo de resolución de circuito, ya que se trató de asignar a una misma variable del sistema de ecuaciones 2 valores distintos en momentos diferentes. El hecho de que falle la generación de la prueba también es útil para validar que la herramienta está haciendo lo previsto, sin embargo esto deja de lado un problema de seguridad.

Plonky2 hace 2 cosas. Resuelve el valor de todos los Targets cuando genera la prueba y verifica que las restricciones polinomiales se cumplen cuando la verifica. Resolver el valor de los targets podemos pensarlo como "llenar la traza" (la matriz T) vista en la sección 2.5. Sin embargo, estamos asumiendo que luego de que Plonky2 "llena la traza" ningún atacante la modifica, lo cual es potencialmente incorrecto. Lo bueno es que si modificamos la traza, luego la verificación debería fallar, asumiendo que establecimos todas las restricciones correctamente. Ese es el centro de la cuestión, ¿cómo validamos que las restricciones están establecidas correctamente? Para hacer eso tendríamos que hacer tests por la negativa, pero no como los anteriores, sino modificando la traza luego de que esta sea generada por Plonky2 y viendo que es la verificación (y no la generación de la prueba) la que falla.

El problema con esto es que Plonky2 no va a generar una traza incorrecta. La resolución de operaciones de Plonky2 está pensada para corresponderse con las ecuaciones que se intentan verificar. Por ejemplo, si se establece una restricción de la forma $t_3 = t_1 + t_2$ la resolución de targets de Plonky2 le va a asignar a t_3 el valor de $t_1 + t_2$. Si proveyéramos con valores inconsistentes a t_1, t_2 y t_3 fallaría la generación de la prueba, pero nunca vamos a poder verificar que las restricciones están haciendo lo que nosotros queremos (o dicho con otras palabras, hacer que falle la verificación).

Al mismo tiempo, hacer tests por la negativa del primer tipo (que fallan en la generación de la prueba) no es tan malo como suena. El sistema de resolución de Targets de Plonky2 va a fallar si detecta una desconexión en el circuito (targets que no están resueltos), lo cual valida cierto grado de consistencia en la construcción del circuito. Si a eso le sumamos que

las herramientas que se están usando para construir el circuito son propias de la API de Plonky2 (`CircuitBuilder` y las Custom Gates preexistentes) en realidad lo que estamos diciendo es que la confianza en lo que estamos construyendo depende directamente de la confianza que tengamos en que las herramientas preexistentes validan correctamente las operaciones que hacen.

Lo anterior no quita que no se puedan hacer tests adicionales para asegurar que se están verificando las operaciones, sin embargo la cantidad de trabajo requerido para intervenir la generación de la prueba de Plonky2 descarta esta alternativa en pos de un modelo de confianza más débil.

Todos los tests usados para validar la funcionalidad pueden encontrarse en el repositorio del proyecto.

8.2. Profiling

En la sección 4 planteamos como último objetivo del trabajo hacer un reporte de performance de la herramienta creada. Sin embargo, decir que la performance está atada únicamente a Noirky2 sería incorrecto: recordemos que la herramienta no hace más que una adaptación de un lenguaje a otro. El procesamiento de cada prover (Plonky2 y Barretenberg) va a influir enormemente en la performance de los distintos comandos, así como en en el tamaño de los artefactos generados (prueba y verifying key).

Nos interesa evaluar ciertas familias de programas, cada una de ellas orientada a uno de los opcodes del código ACIR. Estas familias de programas son:

1. Orientada a AssertZero.
2. Orientada a operaciones de memoria.
3. Orientada a RangeCheck.
4. Orientadas a XOR.

Por cada una de ellas queremos variar el tamaño de los programas. Esto quiere decir que queremos generar códigos ACIR que posean **distintas cantidades de opcodes**. `N` es la meta-variable que vamos a usar en los templates de Noir para parametrizar estas cantidades. A su vez, para los programas orientados a RangeCheck y XOR también nos interesa variar la cantidad de bits de las operaciones, siendo las posibles variantes 8, 16 y 32 bits. Esta variación será parametrizada con la meta-variable `K`.

Programas orientados a AssertZero

El template de programa utilizado es el que se puede ver en el programa de Noir 2.

```
fn main(a: [Field; N]) -> pub Field {
    let mut acc: Field = 0;
    for i in 0..N {
        acc += a[i];
        acc *= a[i];
    }
    acc
}
```

Listing 2: Familia de programas orientada a AssertZero

Este programa recibe como input una lista de N elementos de cuerpo y luego toma una variable `acc` para acumular valores. La intención con este programa es generar muchas operaciones de tipo AssertZero, como veremos a continuación. Más en general, la idea es crear programas a partir del template con distintos valores de N. Al compilar esos programas de Noir, el código ACIR generado tendrá N opcodes AssertZero que usen términos lineales y cuadtráticos, de la forma

$$(1 \cdot w_0 \cdot w_1) + (1 \cdot w_2) + (-1 \cdot w_3) = 0$$

La forma del programa es relativamente arbitraria, su único objetivo es generar la cantidad adecuada de opcodes AssertZero con la forma deseada. A su vez, los inputs para cada programa serán N valores aleatorios en el rango $[0, p)$, donde usamos $p_{plonky2}$ para aquellos que serán procesados por Noirky2 y $p_{barretenberg}$ para los que serán procesados por Barretenberg.

Programas orientados a operaciones de memoria

El template de programa utilizado es el que se puede ver en el programa 3.

```
fn main(idx: Field, val: Field) -> pub Field {
    let mut arr: [Field; 100] = [0; 100];
    for _ in 0..N {
        arr[idx] = val;
        assert(arr[idx] == val);
    }
    arr[idx]
}
```

Listing 3: Familia de programas orientada a operaciones de memoria, variando cantidad de iteraciones

El codigo ACIR generado se compone de N bloques consecutivos como los que se pueden ver en la figura 8.1 (pero inicializando el bloque de memoria una única vez).

```
EXPR [ (-1, v) 0 ]
INIT (id: 0, len: 100)
MEM (id: 0, write w at: z)
MEM (id: 0, read at: w, value: x)
```

Fig. 8.1: Bloque de ACIR para la familia de programas orientada a memoria

Concretamente, el programa recibe un índice $idx < N$ y un $val \in [0, p)$. Se declara una lista de tamaño 100 llena de ceros y se realiza N veces la acción de asignar el valor en la posición. Tanto el valor como la posición tienen que ser inputs para que Nargo no resuelva la operación con AssertZeros y podamos ver operaciones de memoria en el ACIR. Por otro lado, la comparación de `assert(arr[idx] == val);` es necesaria porque sino Nargo simplifica el programa y tampoco genera opcodes de memoria.

En este caso resultó imposible aislar opcodes únicamente de memoria en el código ACIR y por lo tanto tenemos 1 AssertZero involucrado por bloque. Debido al alto costo estimado de las operaciones de memoria en relación a los AssertZero, vamos a asumir que estos últimos no van a afectar significativamente la performance ya que el cuello de botella se encuentra en las operaciones de memoria. Una forma de verificar esta hipótesis con resultados será comparar la performance de los programas orientados a AssertZero con la de los programas orientados a memoria.

También vamos a usar una familia de programas orientada a operaciones de memoria, donde a diferencia del caso anterior vamos a variar el tamaño la lista preservando la cantidad de iteraciones. El template de Noir utilizado se puede ver en el programa 4.

```
fn main(idx: Field, val: Field) -> pub Field {
    let mut arr: [Field; N] = [0; N];
    for _ in 0..100 {
        arr[idx] = val;
        assert(arr[idx] == val);
    }
    arr[idx]
}
```

Listing 4: Familia de programas orientada a operaciones de memoria, variando tamaño de la lista

Es importante notar también que al medir la performance o el tamaño de los artefactos generados, no afectan ni los valores de la lista o los índices que se leen.

Programas orientados a RangeCheck

El template de programa utilizado es el que se puede ver en el programa 5.

```
fn main(a: [uK; N]) {}
```

Listing 5: Familia de programas orientada a RangeCheck

En este caso la meta-variable K tomará los valores de $\{8, 16, 32\}$, generando programas donde se chequea el rango de $u8$, $u16$ y $u32$ respectivamente. También se usará la meta-variable N para generar programas que varíen en la cantidad de estas operaciones. Como se puede ver, el cuerpo del programa está vacío, alcanza con declarar a los inputs como $u8$, $u16$ o $u32$ para que el ACIR generado necesite chequear el rango de estas variables.

Programas orientados a XOR

El template de programa utilizado es el que se puede ver en el programa 6. Al igual que en el ejemplo anterior, la meta-variable K sirve para identificar el tamaño en bits de los operandos y N para determinar la cantidad de operaciones.

```
fn main(a: uK, b: uK) -> pub uK {
    let mut c: uK = a;
    for _ in 0..N {
        c = c ^ b;
    }
    c
}
```

Listing 6: Familia de programas orientada a XOR

En este caso, el programa va a realizar N operaciones de XOR que va a acumular en una variable. La forma del programa nuevamente es arbitraria, pero sirve para que el ACIR generado tenga N opcodes de tipo XOR.

8.2.1. Variantes de Noirky2

Para el análisis (tanto de tiempos como de tamaño de artefactos) vamos a tomar 2 posibles variaciones para Noirky2:

1. Propiedad de **Blinding**. Los circuitos de Plonky2 pueden ser configurados para procesarse con o sin blinding en sus polinomios. Una forma de blinding fue vista en la sección 2.5.4, sin embargo cada implementación de un protocolo puede tener sus propias variantes. Vamos a considerar 2 opciones: **con blinding** y **sin blinding**.

2. **Lookup Tables vs. operaciones de bits**. Para las operaciones de RangeCheck y de XOR consideramos 2 alternativas a la hora de traducirlas en las secciones 7.3.4 y 7.3.5 respectivamente. Vamos a querer estudiar ambas variantes en las operaciones pertinentes.

Con estas opciones en mente vemos que tenemos 4 variantes de Noirky2 (con blinding y operaciones de bits, con blinding y operaciones con lookup table, sin blinding y operaciones de bits, sin blinding y operaciones con lookup table). A la hora de estudiar las familias de programa de AssertZero y operaciones de memoria solo nos van a interesar la primera y la tercera, ya que no nos interesa variar cómo se resuelven los RangeCheck ni el XOR.

8.2.2. Blowup factor

El **blowup factor** (también conocido como "bits de seguridad") fue explicado en la sección 1.2.2. Tanto Plonky2 como Barretenberg cuentan con ciertos bits de seguridad para distintas partes del protocolo, y para realizar una comparación justa es necesario que estén equiparados en este sentido. En ambos casos, los bits de seguridad se encuentran entre 100 y 110, con lo cual podemos decir que son comparables en la seguridad que brindan.

8.2.3. Medición de tiempos

Las mediciones de tiempos de ejecución se realizaran en una PC con las siguientes características:

- OS: Ubuntu 22.04 LTS (64 bits)
- CPU: 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz × 8
- GPU: Mesa Intel® Xe Graphics (TGL GT2)
- RAM: 16GB

Hay 3 tiempos que nos interesa medir: la generación de la prueba, la generación de la verifying key y la verificación de la prueba. Cada una de estas operaciones se corresponde con la ejecución de los comandos **prove**, **write_vk** y **verify** respectivamente, descritos en la sección 5. En todos los casos caso se realizarán 20 mediciones bajo las mismas condiciones sobre un mismo experimento para normalizar el ruido del ambiente.

Plantear hipótesis que involucren comparaciones entre Plonky2 y Barretenberg está fuera del alcance de este trabajo debido a que para hacer eso deberíamos hacer un análisis profundo de la complejidad teórica de ambas herramientas (una de las cuales no se encuentra documentada). Si bien ambas son implementaciones del protocolo Turbo-PLONK, usan dos Polynomial Commitment Schemes distintos con implementaciones propias, las cuales tampoco se encuentran documentadas. Dicho esto, es de interés hacer una comparación de performance entre ambas en forma de reporte.

Por otro lado, RangeCheck y XOR son 2 opcodes que tienen 2 posibles implementaciones en Noirky2, es decir, con o sin lookup tables. En este caso sí vamos a plantear una hipótesis concreta que queremos verificar a través de la experimentación:

Las implementaciones tanto de RangeCheck como de XOR con lookup tables tienen un costo adicional relacionado con la creación de las tablas. Para programas donde se hagan pocas operaciones de estos tipos, el tiempo dedicado a la creación de la tabla es muy grande para que valga la pena crearlas. Sin embargo, cuando un programa contiene una cantidad suficiente de estas operaciones, la suma de los costos de las operaciones bit a bit será mayor que la combinación del costo de la creación de la tabla y todas las operaciones de búsqueda en ella.

Dicho en otras palabras, esperamos que exista un N a partir del cual es conveniente usar operaciones con lookup tables.

8.2.4. Medición de tamaño de artefactos

Los artefactos que nos interesa medir son la **Verifying Key** y la **prueba** de cada uno de los ejemplos, tanto de Barretenberg como de cada una de las variantes de Noirky2. Hay un par de hipótesis a validar en este aspecto.

1. Las pruebas de Barretenberg van a tener un menor tamaño respecto a las de Noirky2. Esto se debe a que el tamaño de la prueba depende principalmente del Ploynomial Comitment Scheme (FRI o KZG). En el caso de Barretenberg, el tamaño de la prueba debería ser constante, compuesto por algunos puntos de curva elíptica. En el caso de Plonky2, el PCS usado es FRI, que para obtener el mismo grado de seguridad que en KZG se obtiene eligiendo una curva adecuada, en FRI se consigue aumentando la cantidad de iteraciones y por ende el tamaño del transcript (ver sección 2.4).

En este caso alcanza con realizar una única iteración por experimento, ya que el tamaño de los artefactos no depende del azar o del ambiente.

Parte III

RESULTADOS Y DISCUSIÓN

A continuación vamos a ver un reporte y análisis de los resultados obtenidos en las mediciones de tiempos de los comandos **prove**, **write_vk** y **verify** y tamaños de los 2 artefactos generados en los comandos **prove** y **write_vk**. Los resultados se corresponden con la experimentación planteada en la sección 8.2.

Las distintas cantidades de operaciones usadas para cada familia de programas pueden verse en la tabla 8.1. Notamos que no en todos los casos la cantidad de operaciones elegida es la misma, y esto se debe a que hay operaciones que llevan más tiempo que otras. Hubo un proceso de estimación manual orientado a que la experimentación completa pueda correrse en menos de 24hs. Además, se pudo observar que hacer mediciones con una cantidad tan grande de operaciones no siempre aportaba información tan relevante.

AssertZero	100	1000	10000	100000	1000000
Memory	100	500	1000	3000	-
Memory Wide	100	500	1000	3000	-
Range u8	1000	2000	5000	10000	20000
Rango u16	1000	2000	5000	10000	20000
Rango u32	1000	2000	5000	10000	20000
Xor u8	1000	2000	5000	10000	20000
Xor u16	1000	2000	5000	10000	20000
Xor u32	1000	2000	5000	10000	20000

Tab. 8.1: Tamaños elegidos para cada familia de programas

8.3. Análisis de tiempos

Vamos a hacer algunas observaciones a grandes rasgos de cada uno de los comandos (prove, write_vk y verify) para luego pasar a un estudio más enfocado en cada operación individual, en los casos donde esto tenga sentido. En este análisis tenemos que tener en cuenta, como se dijo en la sección 5, que el comando `write_vk` realiza un subconjunto de las acciones del comando `prove`. Por esto mismo esperamos ver que los tiempos de ejecución del segundo sean estrictamente superiores que los del primero. Por esta razón también nos vamos a enfocar más en los tiempos de generación de la prueba, ya que representa el flujo más completo del sistema. Los gráficos completos pueden encontrarse en el apéndice B. En esta sección solo vamos a mostrar aquellos que sean de interés para conclusiones determinadas.

Comando **prove**

Comencemos viendo los resultados del comando **prove** en la figura B.10. Hay una generalidad que podemos observar en todos los casos. A grandes rasgos tenemos 2 variantes de Noirky2 para implementar las operaciones de Range y de XOR. Estas son bit a bit o usando lookup tables. Para cada una de estas también tenemos la opción de aplicar un blinding sobre la traza o no hacerlo. Podemos ver que en todos los casos, aplicar ese blinding incurre en un costo adicional, es decir, la versión que aplica este blinding siempre tiene un tiempo de ejecución mayor a la que no. Esto tiene sentido, ya que la fase de procesamiento de la traza en la generación de la prueba va a tener un costo mayor porque tienen que interpolarse polinomios de grado mayor. Podemos ver un representante de este fenómeno en la figura 8.2, con el caso del comando **XOR u8**.

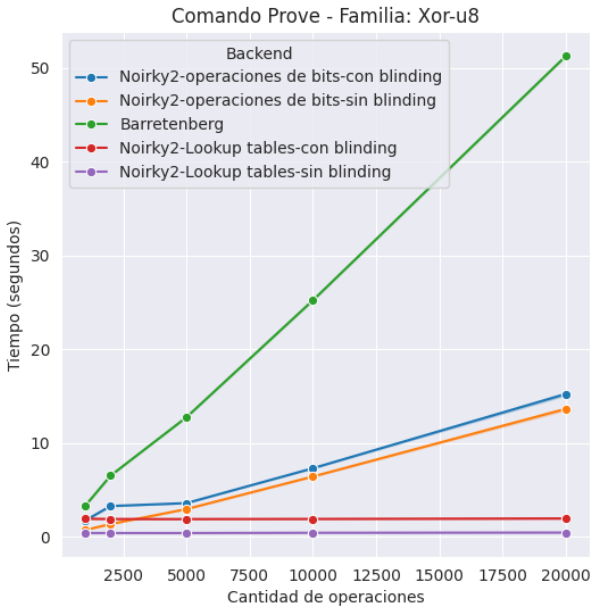


Fig. 8.2: Representante de los tiempos de ejecución del comando **Prove**

Comando **write_vk**

A continuación, veamos el caso de la operación **write_vk**. Los resultados pueden verse en la figura B.20. Retomamos en este caso la observación realizada en el comando prove y vemos que esta se mantiene, aunque de forma menos notable. Un representante de este fenómeno puede encontrarse en la figura 8.3.

prove vs. **write_vk**

En este punto vamos a hacer una comparación entre los tiempos de ejecución del comando **prove** y el comando **write_vk** para el caso de Noirky2 con blinding para el opcode `AssertZero`. Como podemos ver en la figura 8.4, los tiempos de ejecución del comando **prove** son mayores, y esto es esperable dado que **write_vk** realiza un subconjunto de las acciones que realiza **prove**, como ya habíamos adelantado. Esto se va a repetir en todos los opcodes, en todas las variantes de Noirky2.

Comando **verify**

Finalmente, vamos a ver los tiempos de la operación **verify**. Los resultados pueden verse en la figura B.30. Si bien en esta operación estamos viendo órdenes de magnitud mucho menores (del orden de $5ms$ a $25ms$ contra $1s$ a $50s$ en las operaciones de `prove` y `write_vk`), podemos notar en todos los casos que:

- Los tiempos de ejecución parecen estabilizarse en un tiempo constante, o por lo menos de crecimiento muy leve frente al aumento de cantidad de operaciones. Esto se ve especialmente en el caso del AssertZero, donde incluso ejecutando 1,000,000 de operaciones se observa un crecimiento muy leve en los tiempos de ejecución.

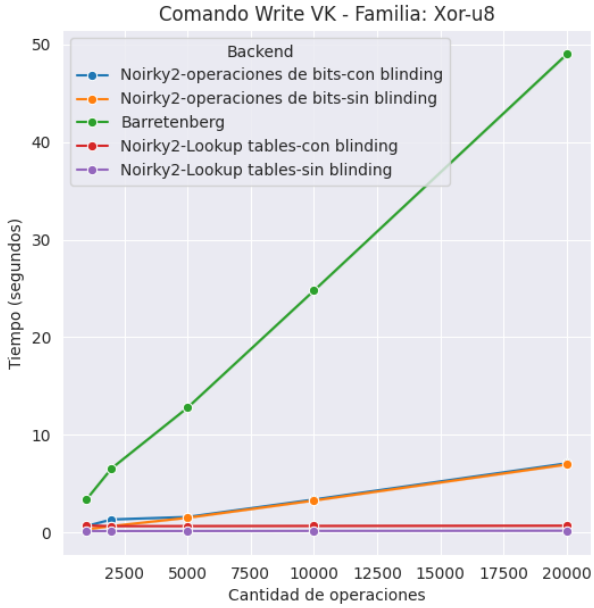


Fig. 8.3: Representante de los tiempos de ejecución del comando **write_vk**

Fig. 8.4: Representante de los tiempos de ejecución del comando **prove** vs. el comando **write_vk**

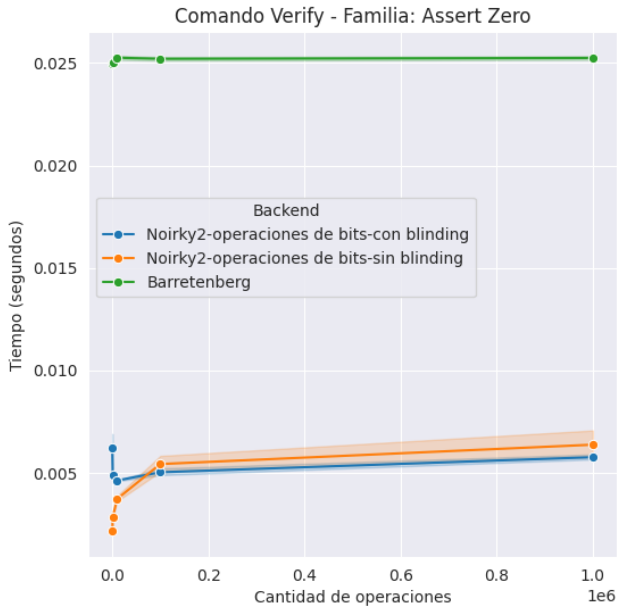


Fig. 8.5: Representante de los tiempos de ejecución del comando **Verify**

- Barretenberg (con KZG) tarda siempre un tiempo cercano a los $25ms$ mientras que todas las versiones de Noirky2 (con FRI) tardan alrededor de $5ms$. Esta comparación está más relacionada con la implementación de los provers en sí que con la implementación realizada en este trabajo.

Para dejar constancia de este comportamiento vamos a tomar como representante el caso del AssertZero, como podemos ver en la figura 8.5, pero teniendo en mente que todas las familias de programas se comportan de manera similar.

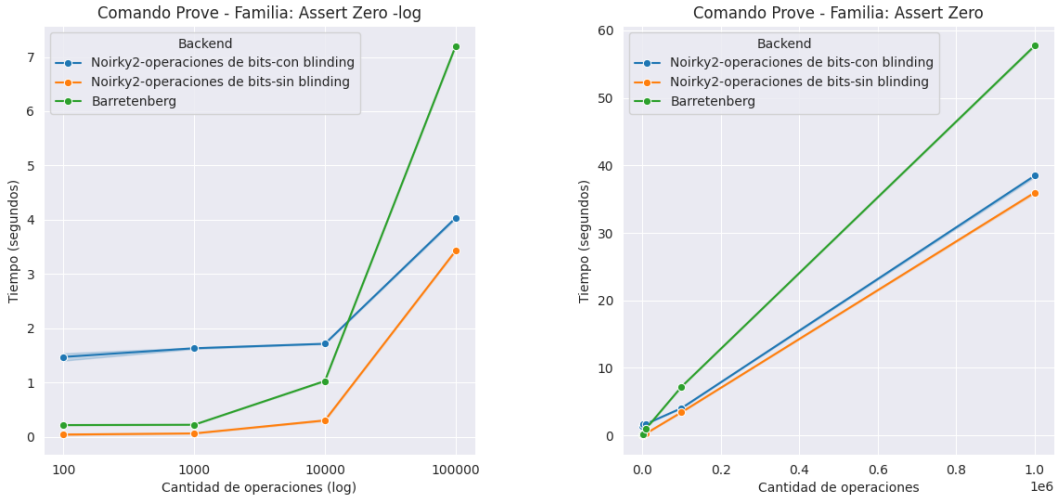
Estos provers (Barretenberg y Plonky2) están diseñados para depositar el costo de la ejecución en la generación de la prueba, permitiendo que la verificación sea rápida o `succint` (recordemos que ese es uno de los principios básicos de los SNARKs descritos en la sección 2.3). Este es un resultado esperable, y no vamos a prestar más atención a los tiempos de ejecución del comando **verify** en el análisis que viene.

8.3.1. Análisis particular de opcodes

A continuación vamos a hacer un un foco en los opcodes. En este caso también vamos a querer comparar las implementaciones entre si. Vamos a centrarnos en particular en el comando `prove` porque consideramos que es el más representativo, ya que ejecuta el flujo completo que incluye la traducción de Noir a Plonky2, la generación de la verifying key y la generación de la prueba. Vamos a dejar a la verificación de lado ya que como vimos, su análisis no presenta tanto interés.

AssertZero

Comencemos por el AssertZero. En este caso vamos a ver los tiempos de ejecución con muchas y con pocas operaciones, ya que puede llegar a ser de interés ver cómo se



(a) Hasta 100,000 operaciones, vista logarítmica *(b)* Hasta 1,000,000 de operaciones

Fig. 8.6: Comando **prove** del opcode AssertZero, tiempos de ejecución para pocas y muchas operaciones

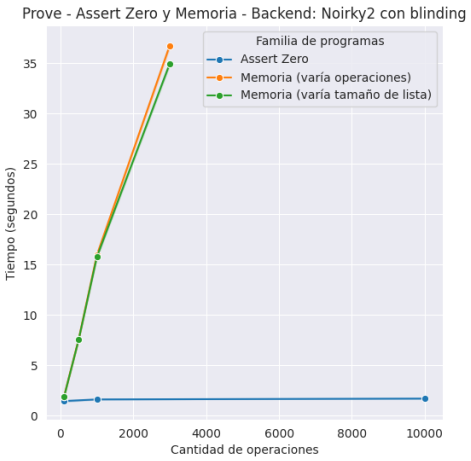
comporta esta herramienta para programas de Noir relativamente pequeños. Para ver el comportamiento con muchas operaciones (específicamente, 1000000) nos remitimos a la figura 8.6b. Podemos ver que a medida que la cantidad de operaciones crece, el tiempo de ejecución aparenta tener un crecimiento lineal, teniendo el backend de Barretenberg una pendiente más pronunciada que aquellas variantes de Noirky2. En particular, para 1000000 de operaciones, la ejecución de este comando tarda 20 segundos más con Barretenberg que con Noirky2, representando aproximadamente un 50 % más de tiempo de ejecución.

Para ver el comportamiento en casos más pequeños, veamos el gráfico de la figura 8.6a. En este caso, la escala del eje horizontal es logarítmica para facilitar la vista. Lo que es interesante de este gráfico es que existe una cantidad de operaciones k tal que si queremos generar una prueba de menos de $k \approx 20000$ opcodes de tipo AssertZero, opera más rápido el backend de Barretenberg. Sin embargo para valores mayores a k ya es conveniente usar Noirky2, en lo que respecta a tiempo de ejecución de la prueba. Por completitud aclaramos que si no fuera de interés aplicar un blinding sobre los datos, siempre es conveniente usar Noirky2 sin blinding, pero compararlos de forma directa no sería del todo justo ya que esta variante no posee el mismo grado de seguridad que las otras dos.

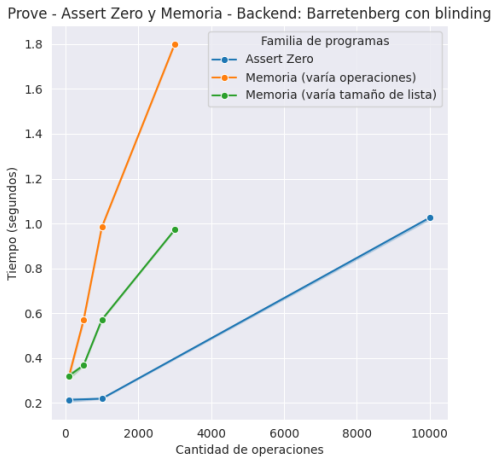
Operaciones de memoria

Avancemos entonces a las operaciones de memoria. Recordemos que tenemos 2 variantes en los programas: uno que aumenta la cantidad de lecturas y escrituras en una lista de tamaño 100; y otra que aumenta el tamaño de la lista pero siempre realiza 100 lecturas y escrituras.

Antes de comenzar vamos a validar una pequeña hipótesis que planteamos en la sección 8.2: una operación de AssertZero es despreciable en cuanto a tiempo de procesamiento en comparación con una operación de memoria. Tuvimos que plantear esta hipótesis porque no fue posible armar un programa que tuviera operaciones de memoria aisladas del Asser-



(a) Memoria vs. Assert Zero, Noirky2 con blinding



(b) Memoria vs. Assert Zero, Barretenberg

Fig. 8.7: Comparación de opcodes de memoria vs opcodes de AssertZero

tZero. Para validarla, vamos a tomar 2 Backends: Barretenberg y Noirky2 con blinding, y comparar directamente los tiempos de ejecución de las operaciones de memoria con los de AssertZero. La comparación puede verse en la figura 8.7.

Notamos que en el caso de Noirky2 esta diferencia es notable: el crecimiento de los tiempos de ejecución es marcadamente diferente (como se ve en la figura 8.7a). Sin embargo, cuando usamos Barretenberg esta diferencia no es tan marcada (como se ve en la figura 8.7b). A su vez, si prestamos atención a la escala de ambos gráficos, vemos que los tiempos de ejecución con Noirky2 para las operaciones de memoria son mucho mayores que los de Barretenberg. Para ver mejor este fenómeno podemos tomar como ejemplo el gráfico de 8.8, donde se muestra que cuando aumenta la cantidad de operaciones de memoria el backend de Barretenberg actúa de forma mucho más veloz.

La diferencia en los tiempos de ejecución de ambos backends (Noirky2 y Barretenberg) es tan marcada que nos hace pensar que las implementaciones de lectura y escritura hechas por Barretenberg se basan en restricciones polinomiales distintas a las de Noirky2. Esto puede dar lugar a una investigación más profunda sobre Barretenberg para ver cómo esta herramienta simula con restricciones polinomiales la lectura y escritura en un bloque de memoria.

Operaciones de Range

Finalmente llegamos a familias de programas donde tenemos variantes implementativas entre los backends de Noirky2. Con esto me refiero que tenemos por un lado implementaciones basadas en Lookup Tables y por otro lado implementaciones basadas en operaciones de bits. Vamos a dejar de lado para este análisis las implementaciones de Noirky2 sin blinding.

Si comparamos únicamente las implementaciones de Noirky2 con Lookup Tables y con Bits, podemos ver que parece cumplirse una de las hipótesis planteadas anteriormente: las Lookup Tables tienen mejor performance cuando crece la cantidad de operaciones pero a

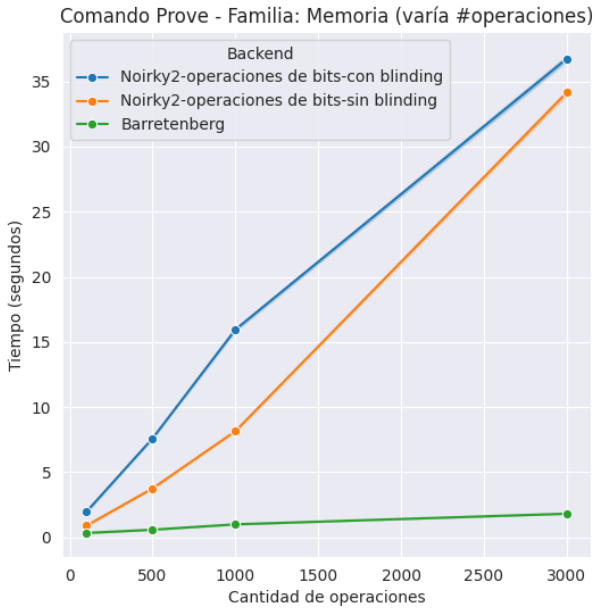


Fig. 8.8: Memoria (operaciones)

un costo de inicialización muy alto. Un ejemplo de esto lo podemos ver en la figura 8.9, que muestra el ejemplo de los opcodes de tipo **Range-u8**, para el comando **prove**.

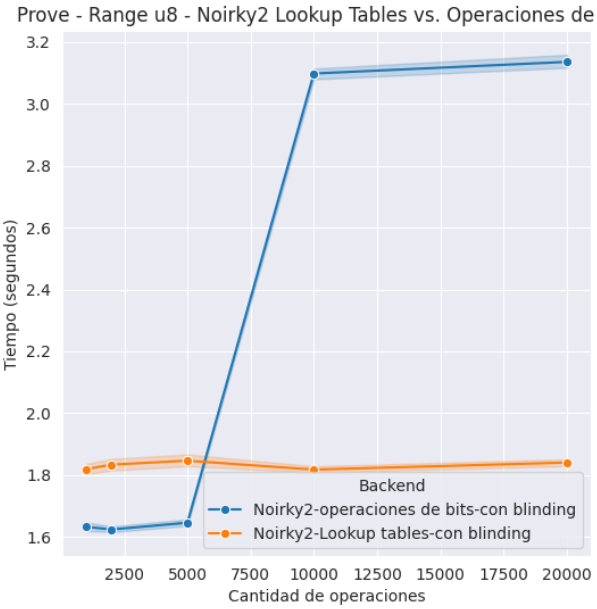


Fig. 8.9: Prove - Range-u8 - Noirky2: Lookup Tables vs. operaciones bit a bit

Por otro lado, nos interesa comparar la performance de Barretenberg con la de Noirky2.

Los gráficos completos pueden verse en las figuras B.4, B.5 y B.6. Estos resultados tienen sin embargo una particularidad que resulta anti-intuitiva para explicar: tanto la implementación de Noirky2 con operaciones de bits como Barretenberg presentan saltos discretos en sus tiempos de ejecución. Estas mediciones fueron repetidas en varias ocasiones para descartar un posible problema de ambiente. Estos saltos dificultan la comparación, y se necesitaría un análisis más exhaustivo de las implementaciones de las Custom Gates de Barretenberg y Plonky2 para explicarlos. Un representante de este comportamiento se puede ver en el gráfico 8.10.

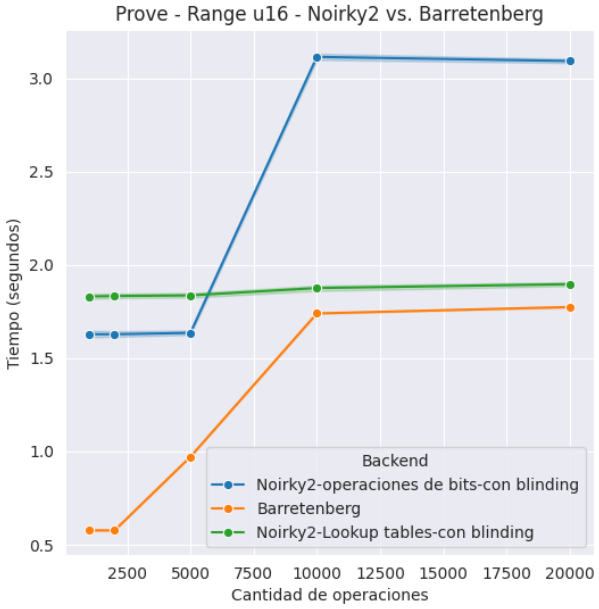


Fig. 8.10: Prove - Range-u16 - Noirky2 vs. Barretenberg

Operaciones de XOR

En el caso del XOR, tenemos nuevamente una implementación de Noirky2 basada en Lookup Tables y una basada en operaciones de bits, además de Barretenberg. Al igual que en el caso anterior, vamos a ignorar las variantes de Noirky2 sin blinding ya que se comportan de modo muy similar a sus correspondientes variantes con blinding, en lo que respecta a tiempos de ejecución.

Los gráficos para realizar este análisis fueron presentados en las figuras B.7 (XOR u8), B.8 (XOR u16) y B.9 (XOR u32). En ellos podemos ver claramente que todos se comportan de forma lineal, pero Barretenberg tiene la pendiente más pronunciada y por ende peores tiempos de ejecución. Por otro lado, comparando ambas implementaciones de Noirky2, notamos que la que usa Lookup Tables tiene una pendiente mucho más llana que aquella con operaciones de bits. En el caso de XOR u8 incluso parecería no haber variaciones entre la ejecución de 1000 y 20,000 operaciones, siendo el grueso del cómputo el overhead de la creación de las Lookup Tables. A su vez, notamos un crecimiento lineal en la cantidad de operaciones en el caso de Noirky2 con operaciones bit-a-bit. Un ejemplo de esto podemos verlo concretamente en el caso del opcode **XOR-u16** en la figura 8.11.

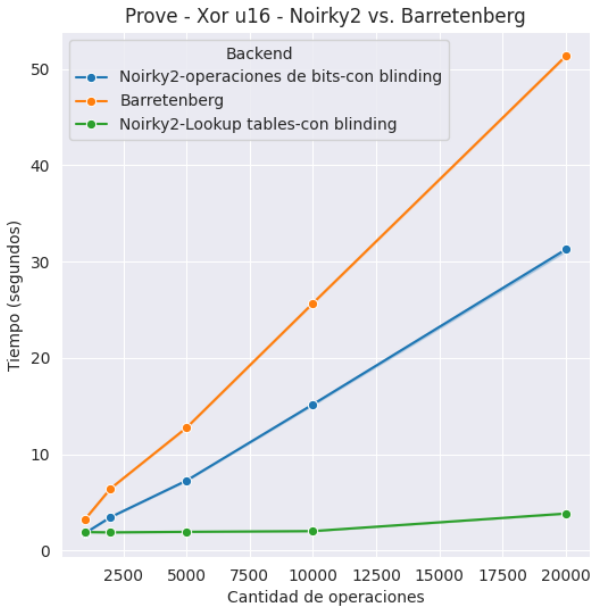


Fig. 8.11: Prove - XOR u16

Por otro lado, notamos que en las variantes con Noirky2, pasar de XOR u8 a XOR u16 y XOR u32 presenta pendientes cada vez más pronunciadas, mientras que Barretenberg tiene los mismos tiempos de ejecución para todas las operaciones de XOR, dada una cantidad de operaciones. Esto se ve especialmente en el caso Noirky2 con operaciones de bits, como podemos apreciar en la figura 8.12.

8.4. Análisis de tamaños de artefactos

En esta sección vamos a hacer un reporte de los tamaños de los artefactos generados en el flujo de trabajo. Estos artefactos son la **Verification Key** en el caso del comando **write_vk** y la **prueba** en el caso del comando **prove**.

8.4.1. Tamaño de la Verification Key

Comencemos por ver los tamaños que tiene cada verification key, segun la cantidad de opcodes de cada tipo. Los resultados completos pueden verse en la figura B.40. Algo que notamos en todos los gráficos es que tanto Barretenberg como Noirky2 con operaciones de bits tienen el mismo tamaño de Verification Key para todas las familias de programas (por una cuestión de escala, esto no se percibe en el gráfico de las operaciones de XOR, pero fue constatado con los datos). Además, aplicar blindings no afecta en ningún caso el tamaño de la Verification Key. Un representante de este fenómeno lo encontramos en el caso del AssertZero como se ve en la figura 8.13, sin embargo esto es así en todos los casos.

Las diferencias pueden empezar a verse cuando usamos Lookup Tables. Recordemos que estas permiten tener tablas pre-computadas en la Verification Key, permitiendo tiempos de ejecución mucho más rápidos como vimos en las secciones 8.3.1 y 8.3.1. Sin embargo,

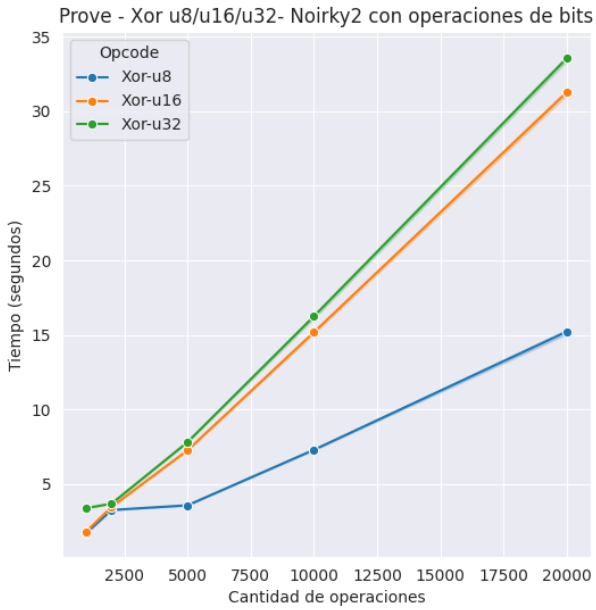


Fig. 8.12: Prove - XOR u8/u16/u32 - Noirky2 con operaciones de bits

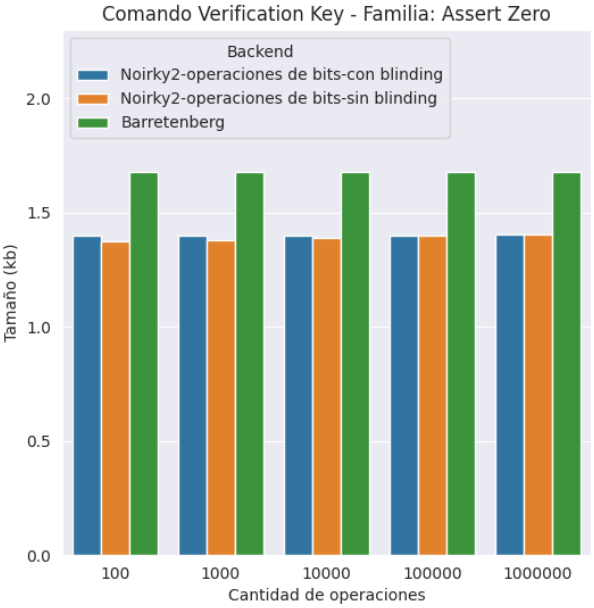


Fig. 8.13: Tamaño de la VK en Noirky2 con operaciones de bits y Barretenberg

acá podemos ver cómo esto trae desventajas en los tamaños de la verification key.

En el caso de las operaciones de Range con Lookup Tables, estas usan una tabla simple de 2^8 elementos de cuerpo, y eso se ve reflejado en el aumento visible en las figuras B.34, B.35 y B.36, que es igual en todos los casos. Por otro lado, cuando nos movemos al caso

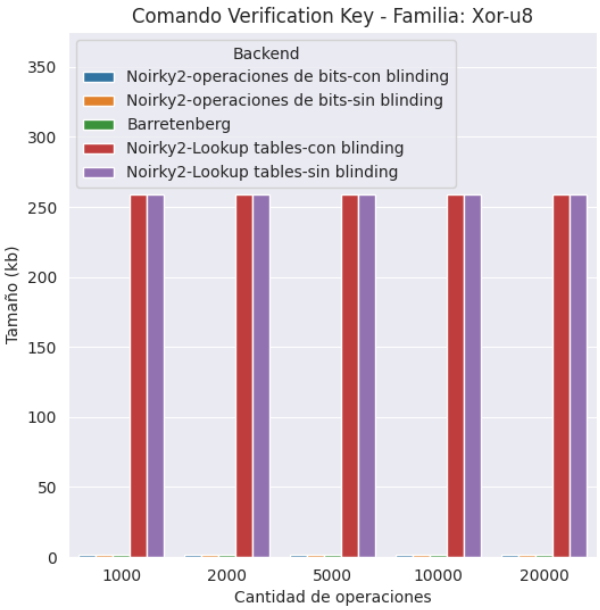


Fig. 8.14: Tamaño de la VK para operaciones de tipo Xor

del XOR, estamos usando tablas de 2^{16} elementos, por lo cual el grueso de la Verification Key se ve ocupado por esta tabla. Esto hace que se opaque casi en su totalidad a las otras implementaciones, como podemos ver en la figura 8.14, superando el tamaño de 250kb.

8.4.2. Tamaño de la prueba

Nos movemos ahora al tamaño de las pruebas generadas. Los resultados completos pueden verse en la figura B.50. En este caso, podemos ver que Barretenberg presenta una clara ventaja frente a Plonky2: los tamaños de sus pruebas son constantes. En todos los casos, la prueba pesa 2176 bytes o 2,125 kb. Esto se debe a que el Polynomial Commitment Scheme usado es KZG, explorado en la sección 3.1. Esto no es así en el caso de Plonky2, que usa FRI.

Notamos que en todas las variantes de Noirky2 la prueba es igual o más grande cuando se aplica un blinding, comparado con su respectiva variante. También notamos que esta diferencia tiende a igualarse a medida que aumenta la cantidad de operaciones. Estos fenómenos así como lo mencionado en el párrafo anterior puede verse representado en la figura 8.15, donde tomamos como ejemplo el caso del opcode **Range-u8**.

8.4.3. Comparación entre artefactos y el CRS

Para finalizar el análisis quiero hacer una comparación entre los tamaños de las pruebas generadas y el tamaño del CRS de KZG visto en la sección 3.1.1. La idea de este trabajo fue librarnos de tener que usar una herramienta que requiriera un CRS potencialmente muy pesado para generar pruebas. Es interesante ver en este contexto si esto efectivamente aporta algún valor. Por ejemplo, si el tamaño de las pruebas fuera mucho mayor a las del CRS necesario para un programa con un tamaño dado, usar FRI en lugar de KZG

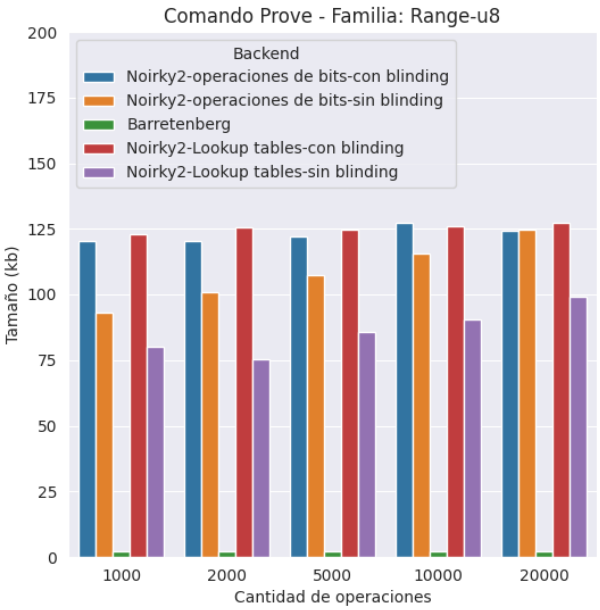


Fig. 8.15: Tamaño de prueba - Range-u8 - Todos los backends

no presentaría una ventaja tan grande ya que el cuello de botella no estaría en el CRS requerido por KZG.

Para hacer este análisis vamos a comparar el tamaño del CRS necesario para procesar una traza de largo N con el tamaño de los artefactos generados para esa misma traza (Prueba y Verification Key). Vamos a usar únicamente la familia de programas orientados a AssertZero para acotar este análisis. Recordemos de la sección 6 que el ancho de la traza es de 135 columnas y que Plonky2 empaqueta restricciones polinomiales del mismo tipo en una misma fila de la traza. Esto sumado a que una operación de AssertZero como las que usamos en el ejemplo ocupa 9 casillas de la traza (ya que son 3 términos matemáticos donde cada uno ocupa 3 casillas en la traza) nos permite saber un largo aproximado de la traza para una cantidad de operaciones dada. El resultado es

$$\frac{9 \cdot N}{135} = \frac{N}{15}$$

Entonces por ejemplo, si realizamos 10000 operaciones de tipo AssertZero, Noirky2 va a generar una traza de Plonky2 con un estimado de $10000/15 \approx 666$ filas. Volviéndonos a la tabla 3.1 el CRS tendría que tener un tamaño mínimo de $62kb$. Para esta cantidad de operaciones, la figura B.41 nos muestra que una prueba de Plonky2 para este programa pesa aproximadamente $125kb$, que es un tamaño mayor al del CRS. Sin embargo, si tomamos un programa con 1000000 de AssertZeros, la traza de Plonky2 tendría un largo de aproximadamente 66666 y por ende el CRS tendría un tamaño mínimo de $6200kb$, superando ampliamente a los aproximadamente $150kb$ de la prueba para este mismo tamaño.

¿Qué nos dice esto? Para programas chicos, en términos espaciales, usar Noirky2 en lugar de Barretenberg no presenta ventajas. Sin embargo, cuando los programas crecen, el crecimiento del CRS en función de la cantidad de operaciones es mucho más pronunciado que el del tamaño de las pruebas. Resultados más detallados pueden verse en la tabla 8.2.

N	100	1000	10000	10000	100000
Prueba	118kb	115kb	120kb	126kb	145kb
CRS	0.6kb	6.2kb	62kb	620kb	6201kb

Tab. 8.2: Comparación entre el tamaño del CRS y el tamaño de la prueba

Parte IV

CONCLUSIONES Y TRABAJO FUTURO

Conclusiones

Este trabajo trajo consigo muchos desafíos, tanto teóricos como de implementación, y es momento de hacer un pequeño resumen de las cosas aprendidas.

Respecto a la implementación en sí, se construyó una herramienta (Noirky2) que depende de otras dos (Plonky2 y Noir), lo cual fue especialmente complicado teniendo en cuenta que el repositorio de Noir estaba en una versión inestable y recibía muchos cambios todos los días (todavía lo hace). En cierto punto se decidió fijar una versión, pero para usar Noirky2 de forma productiva habría que actualizar la herramienta para que se adapte a la versión más nueva de Noir.

Respecto al uso de la herramienta en sí, pudimos ver que no hay una receta perfecta para la generación de pruebas de programas en Noir. El backend que tengamos que usar y su respectiva variante va a depender del tamaño del programa y del tipo de operaciones que predominen en él. Por ejemplo, para algunos programas muy chicos vimos que conviene usar Barretenberg, tanto temporal como espacialmente. Sin embargo, para programas más grandes, a menos que sea un programa con muchas operaciones de memoria, conviene usar Noirky2. El uso de Lookup Tables frente a operaciones de bits debe estar pautado por la cantidad de operaciones de tipo XOR o Range que haya en el programa, ya que el costo de inicializar esa tabla es muy alto. Por otro lado, si no es crítica la propiedad de **ZK** en la prueba, siempre conviene NO aplicar un blinding en Noirky2 en términos del tiempo de generación de la prueba.

Por último, también vimos que el costo del CRS crece de manera mucho más acelerada que las pruebas de Plonky2. Esto hace que para programas relativamente chicos tenga sentido usar Barretenberg, ya que el CRS tendrá un tamaño pequeño. Sin embargo para programas que pasen cierto tamaño, el CRS se transforma en un cuello de botella para la memoria del dispositivo que genera las pruebas, y por ende puede ser limitante para ellos.

Trabajo futuro

Me gustaría finalizar dejando en claro las cosas que no se llegaron a estudiar y posibles continuaciones de este trabajo.

- Hacer tests por la negativa para verificar que las restricciones están presentes. Esto fue desarrollado en la sección 8.1.
- Analizar en detalle el código de Barretenberg para explicar por qué son tanto más rápidas las operaciones de memoria. También se podría estudiar cómo Barretenberg implementa las operaciones de Range para comparar con la implementación de Noirky2.
- Analizar las implementaciones de las Custom Gates de Plonky2 y Barretenberg para explicar los saltos discretos que se dan en las mediciones de tiempos del opcode **Range**.
- Hacer un análisis más profundo de cómo crecen los tamaños de las pruebas en los casos donde esto tenga sentido.
- Explorar en más detalle la forma en la que cada opcode ocupa la traza de Plonky2 para hacer un modelo teórico de la traza en sí misma que permita explicar mejor los tiempos de ejecución.

- Se podrían crear Custom Gates de Plonky2 para operaciones como las de memoria, o AssertZero, pensando en llenar la traza de la forma más eficiente posible.
- Hoy en día existen más backends de Noir hechos con otros provers. Estos backends no existían al momento de comenzar este trabajo. Se podría hacer un estudio de performance sobre los mismos, analizando ventajas y desventajas.
- Se podrían aplicar modificaciones sobre Noir para que Noirky2 soporte el uso de enteros mayores a u32. Esto requeriría modificar la generación de restricciones por parte de Noir porque los enteros u64 tendrían que resolverse divididos en 2 partes (32 bits más significativos y 32 bits menos significativos).
- Podrían usarse técnicas de fuzzing para explorar el espacio de programas de Noir y ver si el ACIR generado cambia en algún caso dependiendo del valor de $\mathbb{F}_p$ utilizado.

Apéndice

A. FLUJO COMPLETO DE NOIR

1. Lo primero que tenemos que hacer instalar Nargo, el compilador de Noir.

2. Después tenemos que crear un nuevo proyecto con el comando `nargo new nombre_proyecto`. Esto nos generará un directorio con la siguiente estructura:

```
nombre_proyecto
├── src
│   └── main.nr
└── Nargo.toml
```

3. A continuación tenemos que escribir un programa en Noir en el archivo `main.nr`. Noir es un lenguaje estáticamente tipado (es decir, se tiene que conocer el tipo de las variables en tiempo de compilación). Ahora es el momento donde cabe preguntarnos ¿qué tipo de programas queremos escribir en Noir? Estamos hablando de un lenguaje de propósito general, pero lo que este lenguaje nos permite es ejecutar un programa con inputs tanto públicos como privados.

   - Los inputs públicos son conocidos tanto por el prover como por el verifier, potencialmente en un ejemplo del mundo real es el verifier el que provee estos inputs para delegar la ejecución del programa.
   - Los inputs privados que solo van a ser conocidos por el prover (quien genera la prueba de ejecución).

   Veamos un ejemplo trivial con inputs tanto públicos como privados:

   ```
   fn main(x: pub Field, y: Field) {
   assert(x == y*y);
   }
   ```

   ¿Qué es lo que hace este programa? `main` es el punto de entrada, recibe un parametro público y uno privado y no tiene un valor de retorno. Su propósito en lenguaje coloquial es generar una prueba de ejecución de que quien lo ejecuta conoce el valor de $\sqrt{x}$ en $\mathbb{F}_p$.

4. Lo siguiente que tenemos que hacer para generar la prueba de ejecución es proveer un listado de parámetros "testigo" para que este programa se instancie con valores concretos. Para esto vamos a ejecutar `nargo check`, comando que creará un archivo Prover.toml. En este archivo tenemos que completar los valores de los parámetros testigo: `x = 4 y = 2`

5. Una vez provistos estos parámetros, lo que vamos a querer es ejecutar el programa para generar un sistema de restricciones que lo respresente. Esto concretamente toma la forma de un **ACIR** (*Abstract Circuit Intermediate Representation*). Para ejecutarlo tenemos que correr el comando `nargo execute witness --print-acir`. El sistema de ecuaciones generado puede verse en la figura A.1.

```
private parameters indices : [0]
public parameters indices : [1]
return value indices : []
EXPR [ (-1, _1, _1) (1, _0) 0 ]
```

Fig. A.1: ACIR para el programa de ejemplo con un AssertZero

Vamos a desglosar lo anterior: el output de la ejecución del programa en formato ACIR se maneja con variables llamadas **Witnesses**. Los Witnesses son nombrados de forma secuencial (de 0 en adelante) según su orden de aparición en el código.

- **private parameters indices** es un listado de los inputs privados del circuito. En nuestro programa de alto nivel llamamos `x` al input privado, pero en el ACIR este input es representado por el witness 0. Esto se debe a que `x` es la primera variable que procesa el compilador.
- **public parameters indices** es un listado de los inputs públicos del circuito. En este caso, "y" es representado por el witness 1.
- **return value indices** es el listado de los witness de retorno. En este caso no hay ninguno, pero serían los siguientes en ser nombrados.
- **Circuito**: la última linea representa la única ecuación del sistema de ecuaciones del circuito. El Opcode que denota es el **AssertZero**, con lo cual esa linea se puede traducir a la ecuación:

  $$-1 \cdot w_1 \cdot w_1 + 1 \cdot w_0 + 0 = 0,$$

  que si la despejamos es equivalente a

  $$w_0 = w_1^2,$$

  que es lo que el programa busca demostrar. Estamos usando la notación w_i para denotar al i-esimo witness (lo que vimos como $\_i$). Vamos a desarrollar sobre el `AssertZero` en breve, pero antes sigamos con el flujo de generar una prueba de ejecución.

En este punto el directorio debería verse parecido a esto:

```
nombre_proyecto
├── Nargo.toml
├── Prover.toml
├── src
│   └── main.nr
└── target
    ├── witness.gz
    └── nombre_proyecto.json
```

Los 2 recursos que generó la ejecución anterior son:

- **ACIR**: en archivo ubicado en `target/nombre_proyecto.json`.
- **Valores generados de witness**: en `target/witness.gz`. En esta instancia Nargo ya resolvió todos los valores de los witnesses. Para hacerlo, tomó $\mathbb{F}_p$ con

  $$p = 2^{254} + 2^{224} + 2^{192} + 2^{96} - 1$$

6. El siguiente paso va a ser generar la prueba de ejecución de este circuito. Para esto vamos a usar un prover. El único prover que sabe interpretar código ACIR al momento es **Barretenberg**, por lo cual tendremos que instalarlo. Lo siguiente que tenemos que hacer es generar la prueba de ejecución de Barretenberg, con el comando:

   ```
   bb prove -c target/nombre_proyecto.json -w /target/witness -o proof
   ```

   Esto va a crear una prueba (π) de Barretenberg y la va a guardar en el archivo `nombre_proyecto/proof`.

   A continuación queremos verificar la prueba. Esto consta de 2 partes: generar la clave de verificación y verificarla. En un caso real querríamos que la verificación sea hecha por un agente distinto al que generó la prueba, por ende, tanto la clave de verificación como la prueba deberían ser pasadas al verifier. La clave de verificación también puede ser creada por el mismo verifier.

7. **Generación de la Verifying Key:**

   ```
   bb write_vk -b target/nombre_proyecto.json -o target/vk
   ```

   Este comando va a generar una **Verification Key** en la ruta indicada. Esta contiene toda la información necesaria para verificar la prueba: el programa codificado, los public inputs, etc. En lenguaje de PLONK, podríamos pensarlo como las matrices Q, V y PI.

8. **Verificación de la prueba:**

   ```
   bb verify -k target/vk -p proof
   ```

   Este comando va a realizar la verificación, valiéndose de la Verification Key y la prueba.

Es importante notar también que la semántica que da Noir a las operaciones hace que el comando `prove` y el comando `write_vk` repitan acciones. En particular, podemos decir que `write_vk` realiza un subconjunto de las acciones que realiza `prove` ya que este último tiene que generar una verifying key antes de poder generar la prueba. Una decisión de diseño distinta sería que el comando `prove` reciba como parámetro una proving key previamente generada, sin embargo no pretendemos en este trabajo modificar la semántica de las operaciones del programa sobre el cual deseamos construir.

B. GRÁFICOS DE RESULTADOS DE LA EXPERIMENTACIÓN

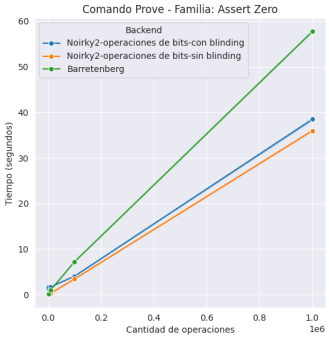


Fig. B.1: AssertZero

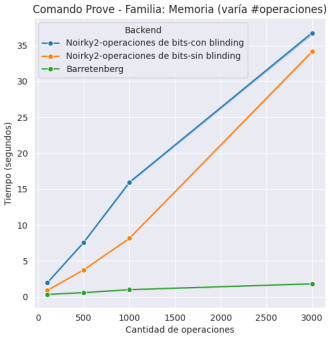


Fig. B.2: Memoria (operaciones)

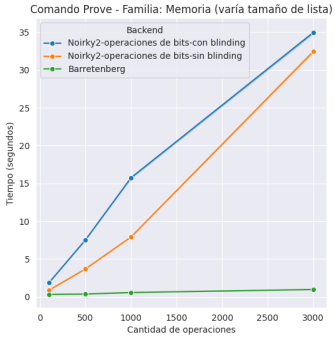


Fig. B.3: Memoria (tamaño)

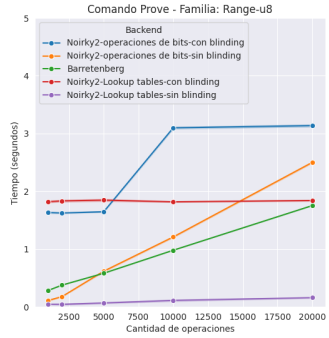


Fig. B.4: Range u8

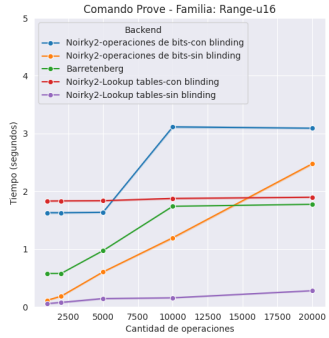


Fig. B.5: Range u16

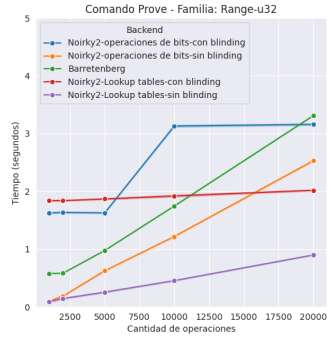


Fig. B.6: Range u32

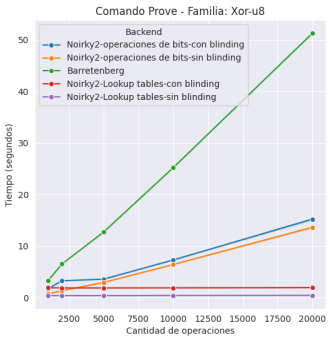


Fig. B.7: XOR u8

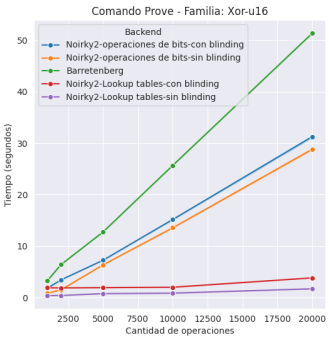


Fig. B.8: XOR u16

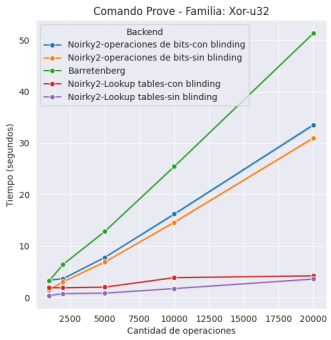


Fig. B.9: XOR u32

Fig. B.10: Mediciones de tiempos para el comando **prove** para todas las familias de programas de Noir

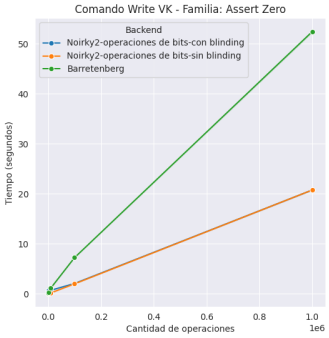


Fig. B.11: AssertZero

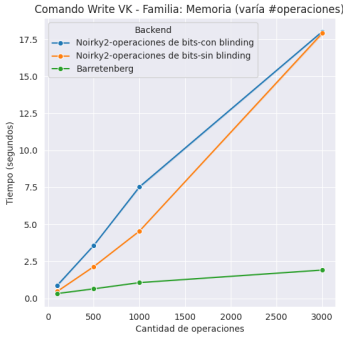


Fig. B.12: Memoria (op)

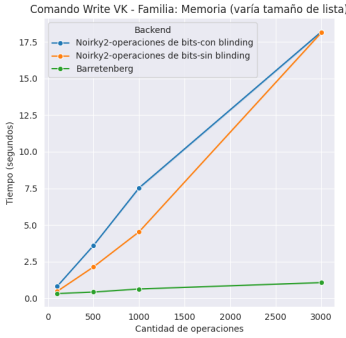


Fig. B.13: Memoria (tamaño)

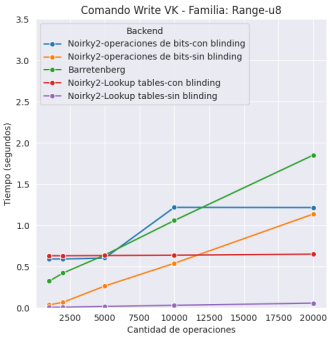


Fig. B.14: Range u8

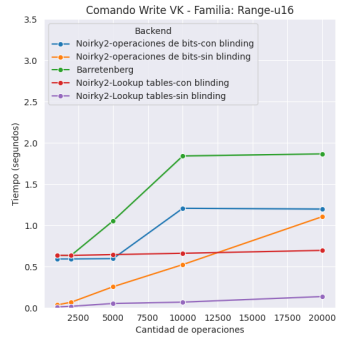


Fig. B.15: Range u16

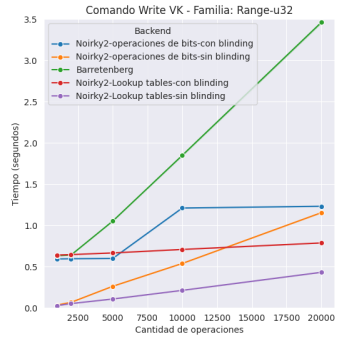


Fig. B.16: Range u32

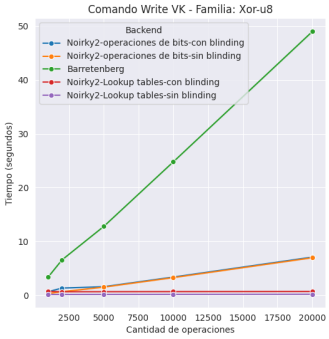


Fig. B.17: XOR u8

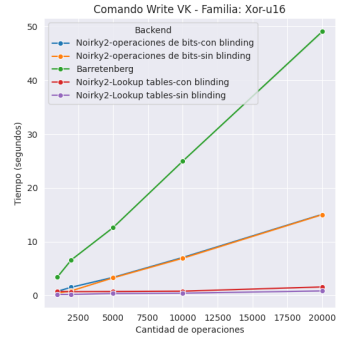


Fig. B.18: XOR u16

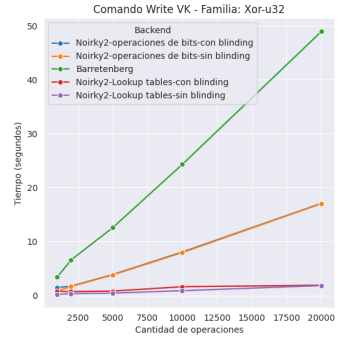


Fig. B.19: XOR u32

Fig. B.20: Mediciones de tiempos para el comando **write_vk** para todas las familias de programas de Noir

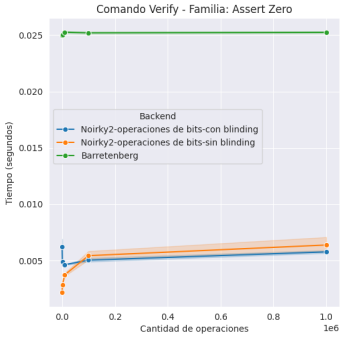


Fig. B.21: AssertZero

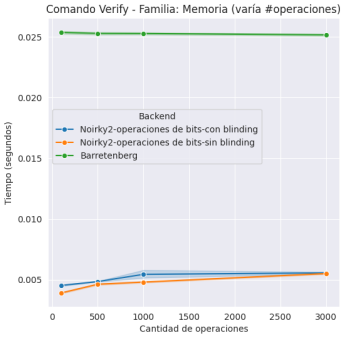


Fig. B.22: Memoria (op)

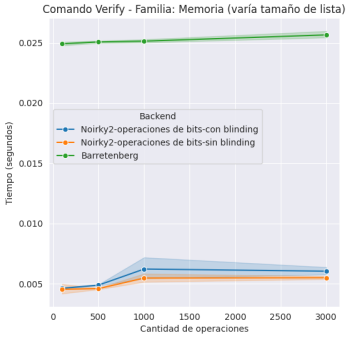


Fig. B.23: Memoria (tamaño)

Fig. B.24: Range u8

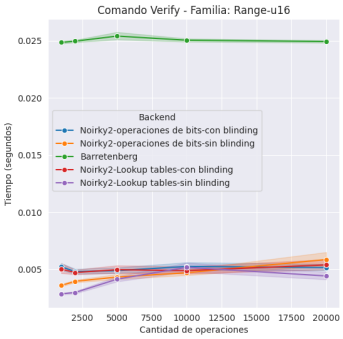


Fig. B.25: Range u16

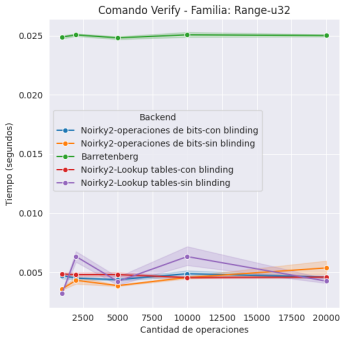


Fig. B.26: Range u32

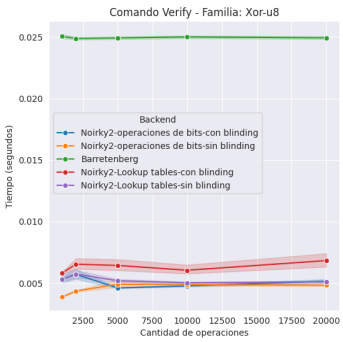


Fig. B.27: XOR u8

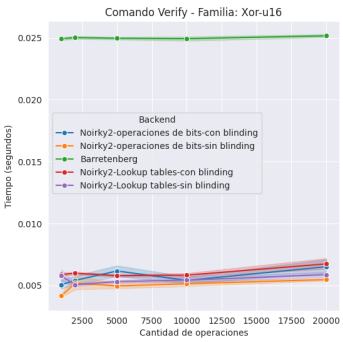


Fig. B.28: XOR u16

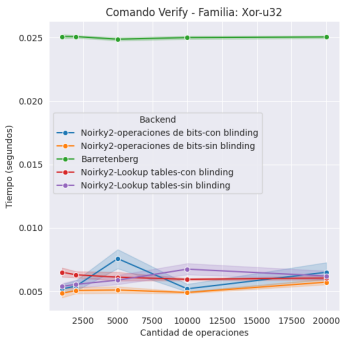


Fig. B.29: XOR u32

Fig. B.30: Mediciones de tiempos para el comando **verify** para todas las familias de programas de Noir

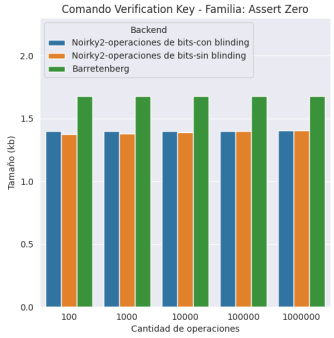


Fig. B.31: AssertZero

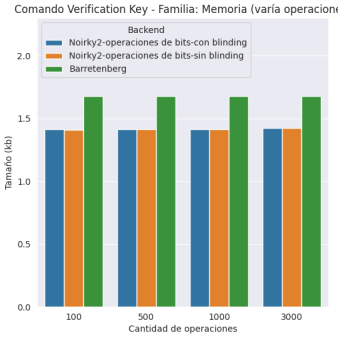


Fig. B.32: Memoria (op)

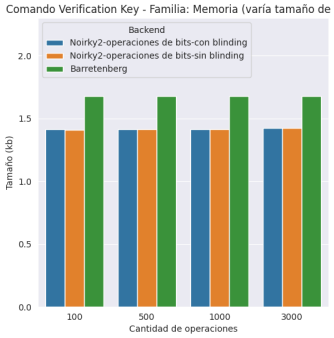


Fig. B.33: Memoria (tamaño)

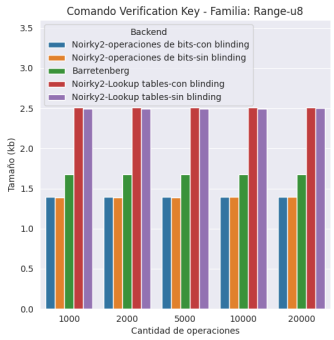


Fig. B.34: Range u8

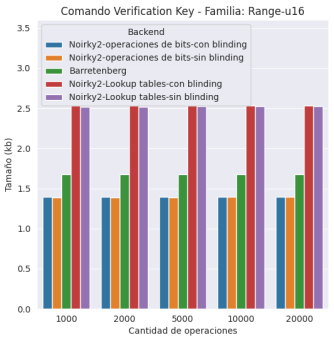


Fig. B.35: Range u16

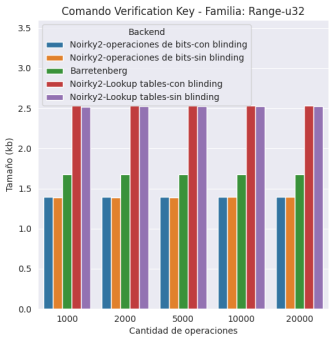


Fig. B.36: Range u32

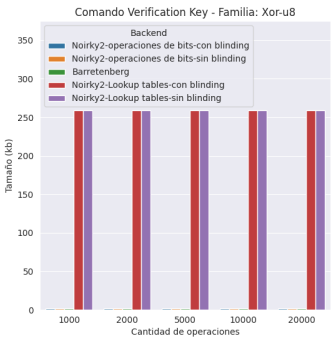


Fig. B.37: XOR u8

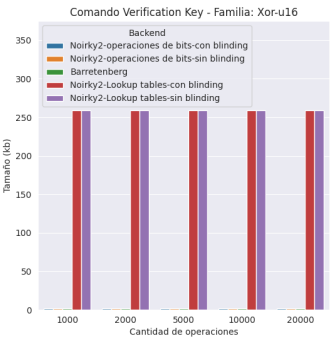


Fig. B.38: XOR u16

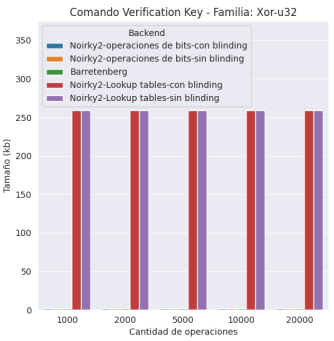


Fig. B.39: XOR u32

Fig. B.40: Tamaño de la Verification Key para todas las familias de programas de Noir

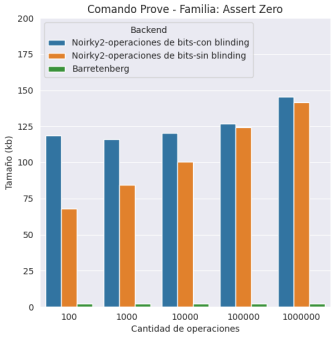


Fig. B.41: AssertZero

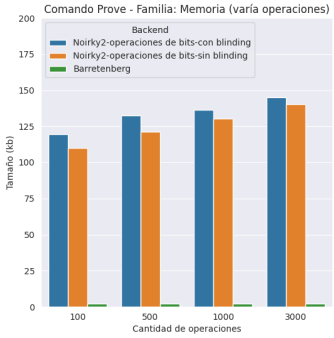


Fig. B.42: Memoria (op)

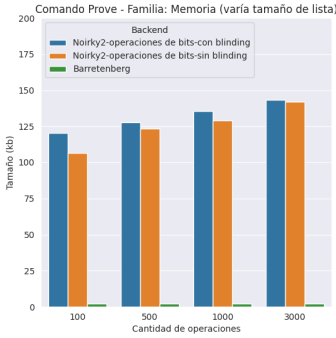


Fig. B.43: Memoria (tamaño)

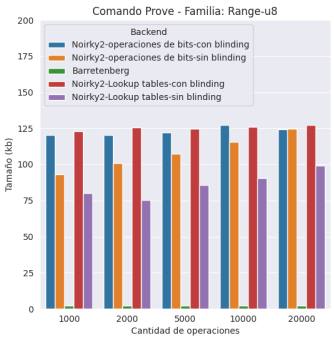


Fig. B.44: Range u8

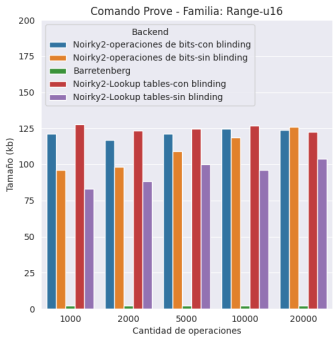


Fig. B.45: Range u16

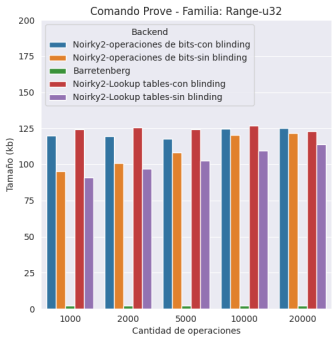


Fig. B.46: Range u32

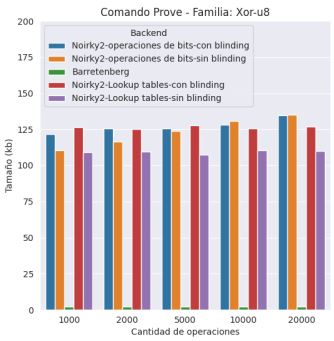


Fig. B.47: XOR u8

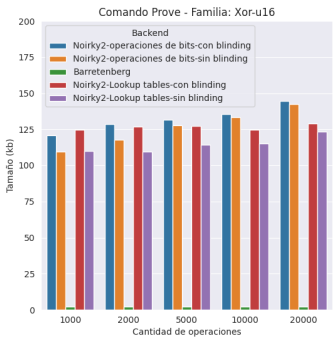


Fig. B.48: XOR u16

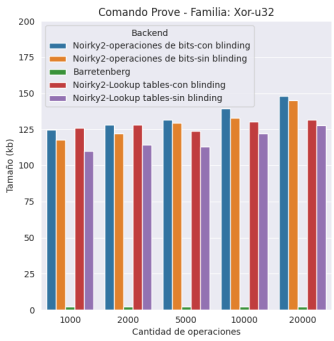


Fig. B.49: XOR u32

Fig. B.50: Tamaño de la prueba para todas las familias de programas de Noir

C. PROTOCOLO PARA LA MATRIZ V

Para completar el protocolo tenemos que hablar sobre cómo demostramos la segunda propiedad de validez de la matriz T respecto a un programa dado por Q y V, es decir

$$\forall i, j, k, l, \quad V_{i,j} = V_{k,l} \implies T_{i,j} = T_{k,l}$$

Para esto vamos a usar la idea de **permutación**. Una permutación es un ordenamiento de un conjunto, denotado con σ. Dado un conjunto $A = \{a_i\}_{0 \leq i < n}$, definimos a un ordenamiento como una función biyectiva $\sigma : A \to A$.

En nuestro caso particular, queremos una permutación sobre el conjunto de posiciones de la traza T, es decir, los pares

$$I = \{(i, j) \mid 0 \leq i < N \ \wedge \ 0 \leq j < 3\}$$

Llamamos $T_{i,j}$ al valor de la matriz T en la posición (i, j), y lo mismo para la matriz V. De la matriz V podemos obtener una permutación σ donde $\sigma((i, j))$ es igual al par de índices de la **siguiente** ocurrencia del valor $V_{i,j}$. Si en la posición (i, j) está la última ocurrencia de $V_{i,j}$ en V, $\sigma((i, j))$ es igual al par de índices de la primera aparición de $V_{i,j}$.

Veamos un ejemplo. La tabla V de la figura 2.8 induce una permutación tal que

- $\sigma((0, 0)) = (0, 0)$
- $\sigma((0, 1)) = (1, 1), \ \sigma((1, 1)) = (0, 1)$
- $\sigma((0, 2)) = (1, 0), \ \sigma((1, 0)) = (0, 2)$
- $\sigma((1, 2)) = (2, 0), \ \sigma((2, 0)) = (1, 2)$
- $\sigma((2, 2)) = (2, 2)$

En otras palabras, vemos que una permutación forma **ciclos** dentro de un conjunto.

La restricción que establecida por la matriz V es equivalente a decir que

$$T_{i,j} = T_{\sigma((i,j))} \quad \forall (i, j) \in I$$

y esto a su vez equivale a decir que los conjuntos

$$A = \{((i, j), T_{i,j}) \mid (i, j) \in I\}$$

$$B = \{(\sigma((i, j)), T_{i,j}) \mid (i, j) \in I\}$$

son iguales. En otras palabras, $T_{i,j} = T_{\sigma((i,j))} \Leftrightarrow A = B$. La demostración de esto es por absurdo en ambas direcciones. La pregunta entonces es ¿cómo podemos reducir la verificación $A = B$ a ecuaciones polinomiales?

Igualdad de conjuntos

Si quisiéramos demostrar que dos conjuntos arbitrarios $A = \{a_i\}_{0 \leq i < k}$ y $B = \{b_i\}_{0 \leq i < k}$ son iguales, podemos multiplicar todos sus elementos y ver si el producto de cada conjunto es el mismo. Sin embargo, vemos como esto se rompe para los conjuntos $\{3, 4\}$ y $\{2, 6\}$, donde el producto de los elementos de ambos es 12 y sin embargo no son iguales. Pero esto no ocurriría si se tratara de conjuntos de polinomios irreducibles, debido a su factorización única.

Entonces tomemos ahora los conjuntos $A' = \{x + a_i\}_{0 \leq i < k} \subset \mathbb{F}_p[x]$ y $B' = \{x + b_i\}_{0 \leq i < k} \subset \mathbb{F}_p[x]$ construidos a partir de A y B respectivamente. En el ejemplo anterior, pasaríamos a tener los conjuntos de polinomios $\{x+3, x+4\}$ y $\{x+2, x+6\}$ y los productos ya serían distintos. Con esta construcción, $A = B \Leftrightarrow A' = B'$.

El truco consiste en usar el lema de Schwartz-Zippel trabajado en la sección 1.2.2. Tomamos un γ aleatorio (en el contexto de un protocolo, sería un challenge por parte del verifier que el prover no puede conocer de antemano) y computamos el producto de los polinomios de A' y B' evaluados en γ. Formalmente

$$\prod_{0 \leq i < k} (\gamma + a_i) = \prod_{0 \leq i < k} (\gamma + b_i) \Leftrightarrow A = B$$

con probabilidad altísima. Ahora, ¿cómo pasamos de esto a ecuaciones polinomiales?

Sea $H = \{1, \omega, \omega^2, \cdots, \omega^{k-1}\}$ donde ω es una raíz de orden k, y sean f y g los polinomios que interpolan en el dominio H a los valores $(a_0 + \gamma, \cdots, a_{k-1} + \gamma)$ y $(b_0 + \gamma, \cdots, b_{k-1} + \gamma)$ respectivamente. Decimos que $\prod_{0 \leq i < k}(\gamma + a_i) = \prod_{0 \leq i < k}(\gamma + b_i)$ si solo si existe un polinomio Z tal que

$$Z(\omega^0) = 1 \quad \wedge \quad Z(h)f(h) = g(h)Z(\omega h) \;\; \forall h \in H$$

Si tomamos a Z como el polinomio que interpola en el dominio H a los valores

$$(1, \frac{a_0 + \gamma}{b_0 + \gamma}, \frac{(a_0 + \gamma)(a_1 + \gamma)}{(b_0 + \gamma)(b_1 + \gamma)}, \cdots, \prod_{0 \leq i < k-1} \frac{a_i + \gamma}{b_i + \gamma})$$

la propiedad se cumple en ambas direcciones.

D. FRI: LOW DEGREE PROOF

Vamos a contar resumidamente cómo se realiza una *Low Degree Proof* sobre un polinomio. La idea es demostrar que un polinomio $p \in \mathbb{F}_p[X]$ tiene un grado $deg(p) < N$. En particular, lo que va a demostrar este protocolo es que tiene un grado $deg(p) \in [\frac{N}{2}, N)$, para algún $N = 2^n$. La idea del protocolo va a ser reducir a la mitad (redondeando para abajo) el grado del polinomio sucesivas veces, mientras prover y verifier interactúan, para que el verifier se convenza de que una serie de puntos pertenecen a un polinomio de grado acotado.

Vamos a comenzar definiendo un *blowup factor* b, una medida del grado de seguridad del protocolo. A partir de este valor, vamos a calcular

$$m = n + b$$

y finalmente

$$M = 2^m = 2^{n+b}$$

El prover a tomar un generador ω de $\mathbb{F}_p$ de grado M para armar un dominio

$$D = [\omega^i : 0 \leq i < M] = [1, \omega, \omega^2, ..., \omega^{M-1}]$$

y va a evaluar p en todos los puntos del dominio, obteniendo así

$$P = [p(1), p(\omega), p(\omega^2), ..., p(\omega^{M-1})]$$

A continuación, va a armar un Merkle Tree con esos puntos y obtener el Merkle Root. Este va a ser pasado al verifier como Commit de la lista obtenida. Esto mismo va a repetirse en varias iteraciones, cada vez con un dominio de la mitad de tamaño y un polinomio de la mitad de tamaño. Para facilitar la explicación podemos pensar que el prover va a ir dándole oráculos de ciertos polinomios al verifier, aunque ahora sabemos que estos oráculos están implementados como merkle trees.

Lo siguiente que va a ocurrir es una reducción sucesiva del grado del polinomio: en cada iteración este va a reducirse a la mitad. La estrategia para hacer esto va a ser armar 2 polinomios de grado $\lfloor \frac{d}{2} \rfloor$ siendo d el grado actual del polinomio y realizar una combinación lineal entre ellos. Uno de estos polinomios va a ser armado con todos los términos impares del polinomio original, y el otro con todos los términos pares. Sea:

$$p_i(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_d \cdot x^d$$

vamos a armar 2 polinomios

$$p_{i,par}(x) = a_0 + a_2 \cdot x + \cdots + a_d \cdot x^{\frac{d}{2}}$$

y

$$p_{i,impar}(x) = a_1 + a_3 \cdot x + \cdots + a_{d-1} \cdot x^{\lfloor \frac{d-1}{2} \rfloor}$$

de manera tal que se cumple la igualdad

$$p_i(x) = p_{i,par}(x^2) + x \cdot p_{i,impar}(x^2)$$

Para obtener el siguiente polinomio de la siguiente iteración (de grado $\lfloor \frac{d}{2} \rfloor$) vamos a tomar

$$p_{i+1} = p_{i,par} + \gamma \cdot p_{i,impar}$$

donde γ es un *challenge* por parte del verifier que el prover no puede saber de antemano. El dominio D_{i+1} con el cual vamos a crear el nuevo Merkle Tree ahora está dado por D^2, es decir

$$[1, \omega^2, \omega^4, ..., \omega^{2(M-1)}]$$

Al elevar todos los elementos de D al cuadrado, obtendremos un dominio de la mitad de tamaño porque la mitad de los valores se repiten, dado que ω es una raíz de la unidad.

Notemos también la siguiente igualdad:

$$p_i(-x) = p_{i,par}(x^2) - x \cdot p_{i,impar}(x^2)$$

Esto hace que el verifier pueda obtener $p_{i,par}$ y $p_{i,impar}$ a partir de p con las ecuaciones

$$p_{par}(x^2) = \frac{p(x) + p(-x)}{2}$$

y

$$p_{impar}(x^2) = \frac{p(x) - p(-x)}{2x}$$

y con ellas verificar a través de los oráculos que el oráculo de la siguiente iteracción se corresponde con una reducción correcta. Si aplicamos n iteraciones, finalmente vamos a alcanzar una lista de 2^b elementos iguales, ya que se corresponden con evaluaciones de un polinomio constante. El verifier puede abrir esta lista en varios puntos aleatorios para convencerse probabilísticamente de que todos los elementos son iguales. Llegado este punto, la conclusión a la que llega el verifier es que el polinomio original tuvo que haber sido de grado a lo sumo $N - 1$.

Bibliografía

[1] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. 17 stoc, 1985.

[2] Chengpeng Huang, Rui Song, Shang Gao, Yu Guo, and Bin Xiao. Data availability and decentralization: New techniques for zk-rollups in layer 2 blockchain networks. *arXiv preprint arXiv:2403.10828*, 2024.

[3] Daniel Benarroch. Diving into the zk-snarks setup phase, 2019. Available at `http://www.urlverdadera.com`, recuperado el día xx/xx/xxxx.

[4] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[5] Aztec team. Introducing noir: The universal language of zero-knowledge, 2022.

[6] Aztec team. Abstract circuit intermediate representation specification, 2022.

[7] Ariel Gabizon and Zachary J Williamson. Proposal: The turbo-plonk program syntax for specifying snark programs, 2020.

[8] Polygon Zero Team. Plonky2: Fast recursive arguments with plonk and fri, 2022.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*. Springer Berlin Heidelberg, 2001.

[10] David Bernhard, Olivier Pereira, and Bogdan 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.

[11] Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive*, 2020.

[12] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*. Springer Berlin Heidelberg, 2010.

[13] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[14] https://github.com/AztecProtocol/barretenberg?tab=readme-ov fileultrahonk. Ultrahonk.

[15] https://www.rust lang.org/es. Rust.

[16] https://aztec.network/noir. Aztec.