



**FACULTAD
DE INGENIERIA**

Universidad de Buenos Aires

Trabajo práctico profesional

Distcom

Sistema de distribución de cómputo con
generación de pruebas de integridad

Marzo 2025

Alumno	Padrón	email
Cambiano Agustín	102.291	acambiano@fi.uba.ar

Docente
Ing. Leandro Ferrigno

Índice

1. Resumen	3
2. Palabras clave	3
3. Abstract	4
4. Keywords	4
5. Introducción	5
6. Fundamentos matemáticos	6
6.1. Aritmética modular	6
6.1.1. Operaciones inversas	6
6.2. Data commitment	7
6.3. Polinomios	8
6.3.1. Comparación de polinomios	8
6.3.2. Corroboración de grado de polinomios	9
6.3.3. Restricciones polinomiales	11
6.3.4. División de polinomios	11
7. Zero Knowledge proofs	13
7.1. Ejemplo con pelotas de colores	13
8. STARKS	14
8.1. Introducción	14
8.2. Traza ejemplificada	14
8.3. Restricciones sobre la traza	15
8.4. Generación de la prueba	18
9. ZKVM	21
9.1. Memoria	21
9.2. Código ejecutado	21
9.3. Componentes de la máquina virtual	22
9.4. Restricciones sobre la traza	23
9.5. Restricciones básicas de memoria	27
9.6. Restricciones de memoria pública	29
10. Proyecto realizado	30
10.1. Tecnologías utilizadas	30
10.1.1. Rust	30
10.1.2. Diesel	30
10.1.3. Docker	30
10.1.4. Makefile	31
10.2. Servidor	32
10.2.1. Modelos de base de datos	32
10.2.2. Almacenamiento en AWS	33
10.2.3. Flujo general de uso	33
10.3. Organizaciones	37
10.3.1. Código a desarrollar	37
10.3.2. Programa ejecutado	38
10.4. Verificación de pruebas	42
10.5. Donantes	44
10.5.1. Programa ejecutado	44

10.5.2. Generación de las pruebas	45
11.Ejemplos implementados	47
11.1. Prueba básica	47
11.2. Prueba probabilística de fermat	47
11.3. Miller-Rabin	48
11.4. Mezcla de tests de primalidad	49
12.Comparación de tiempos de ejecución	51
12.1. Mejoras a futuro en Risc Zero	52
13.Problemas de desarrollo	53
13.1. Uso de Rust	53
13.1.1. Actix-web	53
13.1.2. Diesel	53
13.1.3. Problemas varios	53
13.2. Cambio de Cairo a Risc Zero	54
14.Mejoras del programa a futuro	55
Referencias	56

1. Resumen

Este trabajo tiene como objetivo presentar una primera implementación de un sistema de donación de cómputo con pruebas de integridad. Se desarrollaron tres programas distintos para que organizaciones y donantes puedan interactuar con un servidor como intermediario, utilizando los donantes pruebas de conocimiento cero para generar sus resultados, y verificando las organizaciones que estas pruebas sean válidas. De esta forma los usuarios donantes podrán calcular de forma gratuita resultados que servirán a las organizaciones para utilizar en una buena causa.

Debido a que la teoría en la que estas pruebas se encuentran basadas no fue muy conocida hasta recientemente, una gran parte del informe fue destinada a las bases de conocimiento necesarias para poder entenderla. Se presenta así un informe de dos partes, una con fuerte carga matemática y teórica, y otra con descripciones sobre las distintas partes que componen el proyecto realizado.

2. Palabras clave

Pruebas de conocimiento cero, RISC0, RISC0 ZKVM, Docker, Rust, Donación de cómputo, S3, Integridad de cómputo, Validación de resultados.

3. Abstract

The objective of this work is to present a first version of a system of computing power donation with proofs of integrity. Three different programs were developed so that organizations and donors can interact with each other through a server as an intermediary, with the donors using zero knowledge proofs to generate their results, and the organizations verifying that those proofs are valid. This way, donor users will be able to calculate for free results that will be useful to the organizations to use for a good cause.

Because the theoretical foundation behind these proofs was not widely known until recently, a big portion of the report was destined to the knowledge bases necessary to understand it. Thus, the report is divided in two parts, one with a strong focus on mathematical and theoretical concepts, and another with descriptions of the different parts that compose the created project.

4. Keywords

Zero knowledge proof, RISC0, ZKVM, Docker, Rust, Computing power donation, S3, Computational integrity, Result validation

5. Introducción

Es muy común que tanto empresas como individuos realicen donaciones en forma de materiales o capital para asistir en causas humanitarias. Sin embargo, dado que actualmente las computadoras son extremadamente accesibles, se puede agregar a la lista de recursos donados el poder de cómputo.

Generalmente, los dueños de productos tecnológicos no aprovechan todo su poder de cómputo o, si lo hacen, no es constantemente. Debido a esto, tienen una gran cantidad de poder de cómputo por el cual pagaron, pero no suelen aprovechar (esto puede darse también en empresas que mantienen sus propios servidores si tienen períodos de actividad alta y períodos de actividad baja).

Para aprovechar este cómputo no utilizado, IBM creó el proyecto *World Community Grid*. Agrupaciones necesitadas de resultados que son obtenidos a partir de la ejecución de programas acuden a este para que su código sea distribuido y ejecutado por donantes de cómputo. De esta forma, individuos que generalmente usan sus computadoras para tareas de baja intensidad computacional (ver una película, responder emails, tener reuniones) pueden ejecutar los programas cargados por estas organizaciones necesitadas y así aportar a su causa.

Sin embargo, dicho proyecto presenta un gran problema, necesita confiar en el resultado que le fue donado o, en caso de no querer depender de la confianza, debe limitarse únicamente a solicitar resultados de problemas con soluciones fácilmente verificables. De otra forma, los donantes podrían generar resultados maliciosos o la verificación de los resultados tardaría tanto que el pedir la resolución del problema por parte de un tercero no se realizaría en un ahorro significativo de recursos.

Se podría tomar inspiración de las medidas de seguridad que otros proyectos basados en blockchain implementan, pero la ausencia de incentivos para realizar donaciones haría que la red generada sea susceptible a ataques del 51 %, además de que gran parte del cómputo de los donantes sería destinado a las medidas de seguridad tomadas por estas redes (como proof of work), y la réplica de cómputo limitaría la cantidad de resultados que podrían donarse.

A pesar de esto, hay otra área relacionada con la seguridad de las criptomonedas que se puede investigar y que será de utilidad: las pruebas de conocimiento cero (Zero Knowledge Proofs, o ZKP a partir de ahora). Este concepto será aquel sobre el que se centrará el marco teórico del trabajo, y será explicado en varias etapas de complejidad incremental a continuación.

6. Fundamentos matemáticos

Las pruebas Zk abarcan un amplio rango de conceptos matemáticos, cuyo entendimiento previo es necesario para poder seguir correctamente los distintos pasos que componen la generación de la prueba. Esta sección se encargará de explicar cada uno de los conceptos que se consideran vitales para entender el proceso de generación de una prueba STARKS.

6.1. Aritmética modular

Se trabaja con aritmética modular cuando el universo de números sobre los que se operan consiste en los números naturales (y el cero) $0, \dots, n-1$, siendo n el módulo sobre el cual se trabaja. El universo es cíclico, por lo que al llegar al valor n el resultado se convierte en cero. Dicho esto, se ve que todo número natural se trata como uno compuesto de la forma $p = k \cdot n + r$, siendo r el verdadero valor de p en nuestro universo.

Todo resultado de una operación que finalizaría en un valor mayor o igual a n , tendrá, como fue mencionado el resto, de la división entre ese resultado y n . Si por ejemplo se realiza la operación $3 + 4$ sobre módulo 6, entonces se obtendrá el resultado $4 + 4 \equiv 1 \pmod{6}$, ya que el resto de la división de 7 por 6 es 1.

6.1.1. Operaciones inversas

A pesar de que las operaciones directas son simples de comprender en la aritmética modular, las inversas requieren de una explicación más detallada. Resultados que normalmente son sencillos de obtener requieren un análisis que, a pesar de no ser extremadamente complejo, no es intuitivo. Cuando se quiere obtener el valor tal que al aplicarse una función se obtiene un valor específico, se debe tomar en cuenta la naturaleza cíclica del universo sobre el que se trabaja.

Al realizar por ejemplo una resta $a - b$, lo que realmente se intenta obtener es el valor tal que al sumarle b se obtiene a . Esto puede considerarse sencillo en los casos en los que el resultado quedaría dentro de los positivos, sin embargo, al probar una combinación numérica que terminaría en los negativos, se ve que el resultado diverge de la lógica común, ya que se debe volver al set de números válidos en el universo. Para llegar a un resultado válido en estas ocasiones, cualquiera sea la operación realizada, se debe obtener como resultado de la inversa aquel valor perteneciente al universo tal que al realizarle la operación se obtiene el resultado conocido.

Esto puede verse por ejemplo en la operación $4 - 5 \pmod{7}$. El resultado será, como en cualquier resta, aquel valor tal que al sumarle 5 se obtenga 4, con la diferencia de que este valor buscado pertenecerá al set $0, \dots, 6$. Se obtendrá entonces un valor dentro del rango mencionado tal que al sumarle 5 tenga una diferencia de 4 con 7, el cual resulta ser 6. Así, en módulo 7, al restarle 5 al 4 se obtiene sorprendentemente un valor mayor al número con el que se comenzó.

Esta misma lógica se aplica a operaciones como la división y la obtención de ceros de polinomios. Estas operaciones son de especial interés ya que son centrales para el algoritmo de integridad de cómputo que se explicará más adelante. En los reales, al hacer la operación $\frac{a}{b}$, siendo a y b enteros, si el resultado no es entero entonces puede ser expresado como un número con coma, tal que si se multiplica ese resultado por b entonces se obtendrá a . Este mismo concepto se aplica en la aritmética modular pero, al tener que estar el resultado dentro de un rango determinado por el módulo y al tener que ser entero, se deberá considerar como resultado aquel valor c entero tal que si $c = \frac{a}{b}$, entonces $c \cdot b \equiv a \pmod{n}$.

Si por ejemplo se trabajara en módulo 5, $\frac{3}{2}$ generaría como resultado un número tal que al multiplicarlo por 2 y quedarse con el resto de la división por 5 se obtiene 3 como resultado. Al calcular 2×4 se obtiene como resultado 8, que al dividirlo por 5 y quedarse con el resto se obtiene 3, por lo que, aunque parezca anti intuitivo, el resultado de la división resulta ser 4, que es mayor al numerador.

Esta lógica se aplica también con los ceros de los polinomios. Si en los reales un polinomio tendría un cero negativo, no perteneciente a los naturales, o fuera del rango de valores permitidos,

entonces se debería realizar un análisis similar al mostrado para ver en qué valor dentro del rango se obtiene un cero al evaluarlo en el polinomio.

6.2. Data commitment

Dado un set de datos guardados por A, y de los cuales B solo querrá algunos, se quiere utilizar algún método que permita a B corroborar que cada vez que solicita un dato a A este pertenece al set original y no fue inventado a conveniencia en el momento por este. Es por esto que se solicita a A que genere un hash (llamado commitment) que permita corroborar que los datos que está otorgando fueron generados antes de crear una situación en la que A pueda aprovecharse inmediatamente de falsificar un dato.

Dado un polinomio P, A puede generar un hash sobre una lista de evaluaciones de este polinomio usando un árbol de Merkle, el cual tiene la siguiente forma:

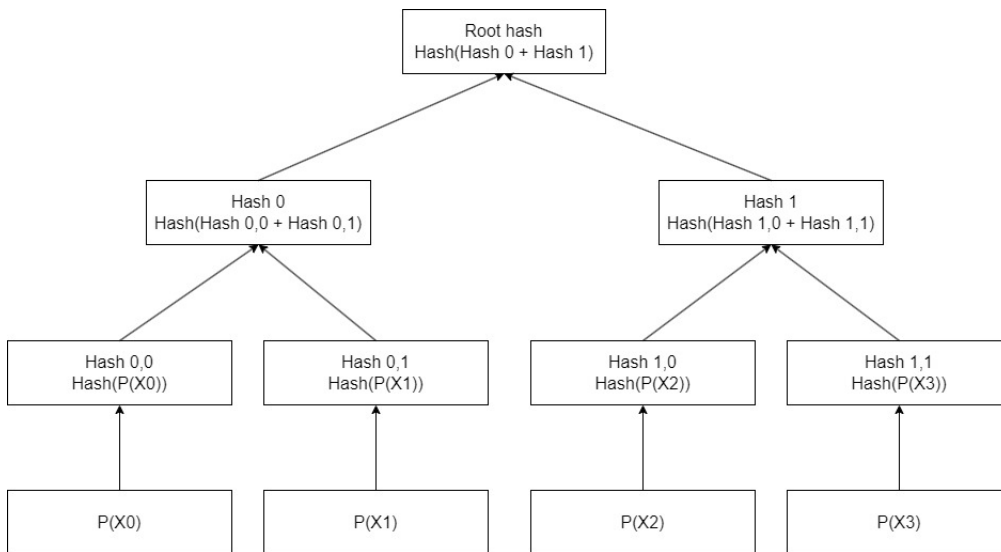


Figura 1: Representación de un merkle tree

Como se ve en la ilustración 1, un Merkle tree consiste en un árbol binario en el cual las hojas son los datos a hashear, y el resto de los nodos contienen el hash de la concatenación de los hashes de sus hijos, a excepción de los padres de los nodos hoja, que contienen el hash de su hijo, es decir, los datos a hashear. Se llama root hash al hash contenido por el nodo raíz del árbol, y es este dato el que será considerado el hash de los datos y, por ende el valor del commitment.

El valor del hash de los datos se denomina commitment ya que permite a B corroborar que todos los datos proporcionados por A luego de entregar el valor del hash (hacer commit) fueron generados previamente, y no en el momento en el que son pedidos, probablemente para intentar obtener alguna ventaja no correspondida en el proceso para el que estén siendo utilizados los datos.

Tomando como ejemplo el caso de la figura 1, si B le pide a A que entregue el valor de $P(X_0)$, entonces A deberá otorgar $P(X_0)$, el valor de $\text{Hash}_{0,1}$ y el valor de Hash_1 . Teniendo estos valores, B puede hashear $P(X_0)$ para obtener $\text{Hash}_{0,0}$, y utilizar este valor junto con el proporcionado $\text{Hash}_{0,1}$ para obtener Hash_0 , que será hasheado junto con el proporcionado Hash_1 para obtener así el root hash. Al generar el cambio de un único bit drásticos cambios en el resultado del hash, cualquier cambio en el valor de $P(X_0)$ o los hashes intermedios proporcionados por A generará inevitablemente un cambio que se propaga hasta la generación del root hash, haciendo que el hash final calculado sea distinto al proporcionado en el momento del commitment, identificando así el intento de engaño de A.

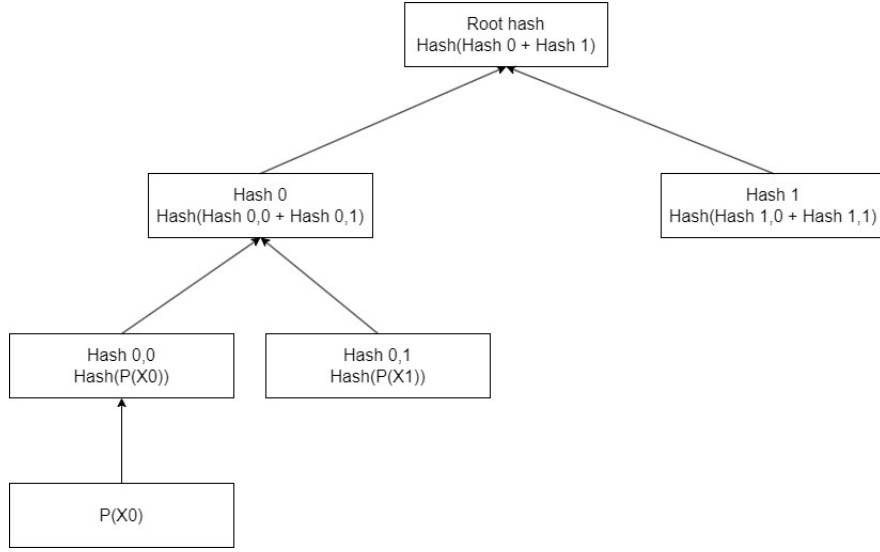


Figura 2: Cálculo del root hash con los datos proporcionados en las hojas

6.3. Polinomios

6.3.1. Comparación de polinomios

Dados A con el polinomio $A(x)$ y B con el polinomio $B(x)$, A quiere corroborar que $B(x)$ es igual a $A(x)$. Sin embargo, B no quiere revelar su polinomio en caso de que no se cumpla la igualdad, por lo que se busca una forma de que se cumplan ambas condiciones.

Recurriremos entonces a una de las propiedades más importantes de los polinomios, el Teorema fundamental del álgebra. Siendo $A(x)$ y $B(x)$ dos polinomios distintos, podemos obtener los puntos en los que estos coinciden en su imagen al resolver la ecuación $A(x) = B(x)$. Esto fácilmente puede despejarse de forma tal que se llega a la siguiente expresión:

$$A(X) - B(x) = 0$$

Al ser $A(x)$ y $B(x)$ polinomios, su suma o resta resultará también en un polinomio, siendo el máximo grado posible del resultado de esta operación igual al mayor de los grados de los polinomios involucrados (puede ser menor a este valor máximo ya que, en caso de que ambos polinomios sean del mismo grado, el término con el mayor exponente podría llegar a cancelarse), es decir

$$\max(\text{gr}(A(x) - B(x))) = \max(\text{gr}(A(x)), \text{gr}(B(x)))$$

También podemos notar que, al mirar la expresión resultante de la resolución de la ecuación de los puntos de igualdad entre $A(x)$ y $B(x)$, llegamos a lo que se ve como una búsqueda de los ceros del polinomio $A(x) - B(x)$, que por el mencionado Teorema fundamental del álgebra tendrá $\max(\text{gr}(A(x)), \text{gr}(B(x)))$ ceros, y por ende esa cantidad de puntos en los que coincidirán esos polinomios en caso de no ser iguales.

La prueba para corroborar si $A = B$ consistirá entonces en elegir un valor X_0 de forma aleatoria de una lista de p números. Este se lo pasaremos a B, quien nos devolverá $B(X_0)$. En caso de que $A(X_0)$ sea igual a $B(X_0)$ entonces puede ser que A sea igual a B, o que hayamos elegido uno de los n puntos en los que A puede coincidir con B, es decir, hay una probabilidad máxima de n/p de que los polinomios no sean iguales y se haya elegido en realidad uno de los puntos en los que estos coinciden.

Podemos hacer un cálculo de ejemplo con los valores $n = 10$ y $p = 10000$. En este caso, si coinciden los valores, la probabilidad de un falso positivo es del 0,1 %, prácticamente nula. Aún así, en caso de querer tener una seguridad mayor sobre el resultado, podría volver a repetirse

el experimento con otro valor aleatorio de X_0 , reduciendo así aún más la posibilidad de un falso positivo. Esto puede realizarse tantas veces como se considere necesario, pero vemos en los números provistos que podría incluso alcanzar con una sola prueba.

Este tipo de prueba no toma en cuenta ciertos aspectos, como la posibilidad de que B no tenga el grado esperado y sea mucho mayor, lo cual no está siendo corroborado y permitiría a B tener una probabilidad mucho mayor a la hora de generar un falso positivo ya que aumentaría el valor de n . Además de esto, no podemos saber si los valores que nos está proporcionando B realmente provienen del polinomio que posee o si son aleatorios o provienen de otra fuente.

6.3.2. Corroboración de grado de polinomios

Dado un polinomio de grado $d-1$, se necesitan d puntos para definirlo inequívocamente. Dicho esto, dado un set de $m > d$ puntos que se quiere corroborar si pertenecen a un polinomio cualquiera de grado máximo $d-1$, se necesita entonces utilizar al menos $d+1$ puntos para corroborar si cumplen con dicha propiedad.

Es evidente que, de querer verificar si la lista en su totalidad pertenece a un polinomio de grado $d-1$, se deberá probar para cada uno de los puntos de esta si pertenecen o no. Sin embargo, se pueden tomar ciertos compromisos con tal de obtener un mayor nivel de eficiencia en la corroboración. Si se asume que de la lista una proporción p de puntos no pertenece al polinomio, entonces con cada punto elegido para corroborar la pertenencia al polinomio de estos habrá una probabilidad p de descubrir el intento de engaño, y $1-p$ de tratarse de un falso positivo (en caso de que realmente la lista de puntos no cumpla con la pertenencia al polinomio). Con cada nuevo punto que se decida utilizar para la prueba de corroboración de grado, mejorará el nivel de certeza sobre la lista.

A pesar de que el muestreo ya otorga una forma de sacrificar certeza por eficiencia, es posible volver a aplicar esta táctica en otros aspectos de la verificación. Dados 2 polinomios A y B de grado $d-1$, el polinomio $C = A + B$ probablemente tendrá también grado $d-1$, siendo la excepción los casos en los que uno o más de los términos con los mayores exponentes se cancelen entre sí. Esto presenta un riesgo ya que, al querer corroborar que el grado del polinomio que representa a todos esos puntos no supera $d-1$, la cancelación de los términos de mayor exponente presenta una forma de que la información de testeo aparenta provenir de polinomios de grado menor o igual a $d-1$, cuando en realidad provienen de polinomios de grado superior. Sin embargo, la probabilidad de que 2 términos de los polinomios se cancelen es extremadamente baja, ya que los términos pueden ser cancelados por un único coeficiente igual al mismo valor pero con el signo invertido. Esto quiere decir que, dado el polinomio A, al sumarlo con el polinomio B, la probabilidad de que la suma de los polinomios tenga un grado menor al de los polinomios iniciales es $1/W$, siendo W la cantidad de elementos del set de valores que pueden tomar los coeficientes de los polinomios (que por tratarse de valores procesados por una computadora es limitado, aunque potencialmente gigantesco). Esto se repite con cada término que sigue al del coeficiente de mayor grado, decrementando así aún más la probabilidad de que polinomios de grado mucho mayor al buscado terminen pasando la prueba cuando deberían fallarla. Tomando en cuenta este pequeño riesgo, se considera altamente aceptable al compararlo con el enorme beneficio de probar alrededor de $d+1$ puntos en vez de 2 veces esa cantidad. Este procedimiento es de gran utilidad, ya que la interpolación de puntos para la creación de un polinomio es un proceso computacionalmente intenso.

Esto puede utilizarse para optimizar la corroboración del grado de un polinomio particular. Dado un polinomio Q_0 de grado $d-1$, este puede ser escrito como la suma de dos polinomios, siendo uno la suma de los términos de exponentes pares, y siendo el otro la suma de los términos de exponentes impares, sacando x como factor común.

$$Q_0(x) = a_0 + \sum_{i=1 \dots d-1} a_i X^i$$

$$Q_0(x) = pares_0(x) + x \cdot impares_0(x)$$

$$pares_0(x) = a_0 + a_2 X^2 + a_4 X^4 + a_6 X^6 + \dots$$

$$x.impares_0(x) = x(a_1 + a_3X^2 + a_5X^4 + a_7X^6 + \dots)$$

Se ve que de esta forma se obtienen dos polinomios cuyos factores son siempre pares, por lo que pueden definirse otros polinomios cuyos exponentes son la mitad de estos, de la siguiente manera:

$$pares_0(x) = P_0(X^2)$$

$$impares_0(x) = I_0(X^2)$$

$$P_0(x) = a_0 + a_2X^1 + a_4X^2 + a_6X^3 + \dots$$

$$I_0(x) = a_1 + a_3X^1 + a_5X^2 + a_7X^3 + \dots$$

Los polinomios I y P tienen el mismo grado, igual a $\lfloor \frac{\text{grado}(Q_0)}{2} \rfloor = \text{floor}(\frac{d-1}{2})$. Estos pueden sumarse para, siguiendo el procedimiento de verificación de grado de dos polinomios de un mismo grado, achicar a la mitad el grado del polinomio a verificar y por ende también reducir el tiempo de interpolación del polinomio y cantidad de puntos. En caso de verificar correctamente el grado de este nuevo polinomio, al ser la mitad del original y provenir de este, se estará verificando también su grado.

$$Q_1(x) = P_0(x) + I_0(x)$$

Sin embargo, se debe tomar en cuenta que no se poseen más puntos para probar en adición a los que se tenían originalmente, por lo que se utilizará la siguiente propiedad:

$$Q_0(x) = P_0(x^2) + x.I_0(x^2)$$

$$Q_0(-x) = P_0(x^2) - x.I_0(x^2)$$

$$P_0(x^2) = \frac{Q_0(x) + Q_0(-x)}{2}$$

$$I_0(x^2) = \frac{Q_0(x) - Q_0(-x)}{2x}$$

Dado que puede obtenerse $P_0(x^2)$ e $I_0(x^2)$, puede obtenerse también $Q_1(x^2)$. Como se vio previamente en esta sección, Q_1 tendrá probablemente el grado de los polinomios sumados, es decir $\text{floor}(\frac{\text{grado}(Q_0)}{2})$. De esta forma, al provenir Q_1 de Q_0 , si se prueba el grado del primero, para el cual se necesita la mitad de puntos, se probará también el grado del segundo. Este proceso puede repetirse cuantas veces se considere necesario, pudiendo incluso llegar hasta el polinomio constante luego de una cantidad de pasos de orden logarítmico.

Es importante notar que, a pesar de poder obtener valores de Q_i dado Q_{i-1} , no es posible obtener $Q_i(-x)$ dados $Q_{i-1}(x)$ y $Q_{i-1}(-x)$, por lo que esas evaluaciones deberán ser obtenidas de forma externa.

6.3.3. Restricciones polinomiales

Dado un polinomio $A(X)$, pueden corroborarse varias propiedades de este mediante la composición de polinomios. Dada una restricción B que queremos corroborar que A la cumple, esto puede ser realizado con una composición junto con el polinomio $B(X)$, que representa la restricción B , marcando esto con un valor cero cuando se cumple la restricción. Para verificar que se cumple lo buscado en los puntos esperados, se corrobora si el polinomio $B(A(X))$ tiene ceros en los puntos esperados.

Si por ejemplo se quiere corroborar que A vale 1, 2 o 3 en los puntos enteros 1 a 100, entonces el polinomio de restricción será $B(X) = (X-1)(X-2)(X-3)$. Si A cumple con la propiedad esperada, el polinomio $B(A(X))$ será cero en todos los puntos enteros de 1 a 100, ya que la evaluación de A en esos puntos dará como resultado 1, 2 o 3, que resultan en un valor nulo para B .

Este procedimiento puede utilizarse para representar una gran variedad de restricciones sobre polinomios, y será de mucha importancia en la prueba de integridad de cómputo.

6.3.4. División de polinomios

Dado un polinomio de grado n , este puede ser expresado en su forma canónica como

$$P(x) = a \cdot \prod_{i=1}^n (x - x_i)$$

siendo a la constante de escalamiento del polinomio, y cada x_i un cero del polinomio. Dado un polinomio $Q(x)$, si al calcular $\frac{P(x)}{Q(x)}$ no se cancelan todos los ceros de Q con los de P , se tendrá entonces como resultado una función que consiste en la división de 2 polinomios no divisibles, lo cual no es un polinomio ya que no puede ser expresado en la forma canónica.

Sin embargo, se debe tomar en cuenta que para cualquier serie de puntos existe un polinomio que los interpola, por lo que un muestreo de una función que no es un polinomio también podría verse como el muestreo de un polinomio de grado alto. Dada esta serie de puntos

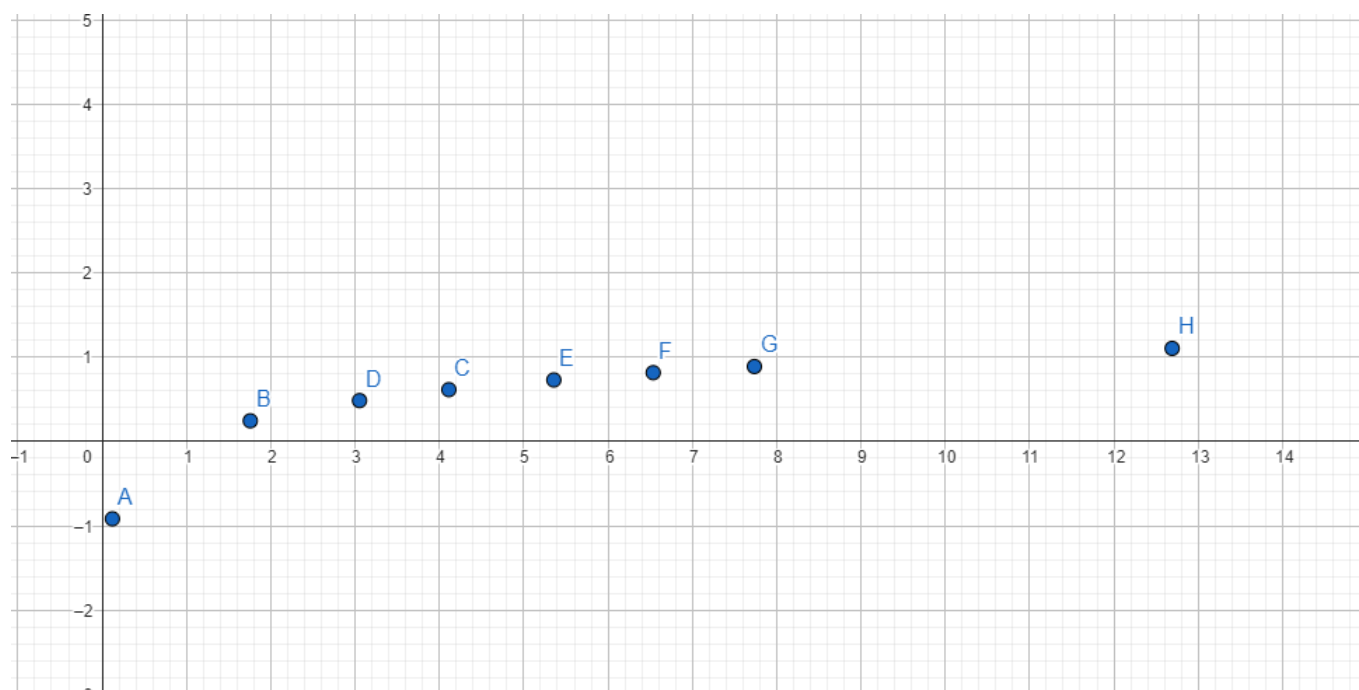


Figura 3: Lista de puntos

puede verse que pertenecen a $f(x) = \log_{10}(x)$, lo cual no es un polinomio.

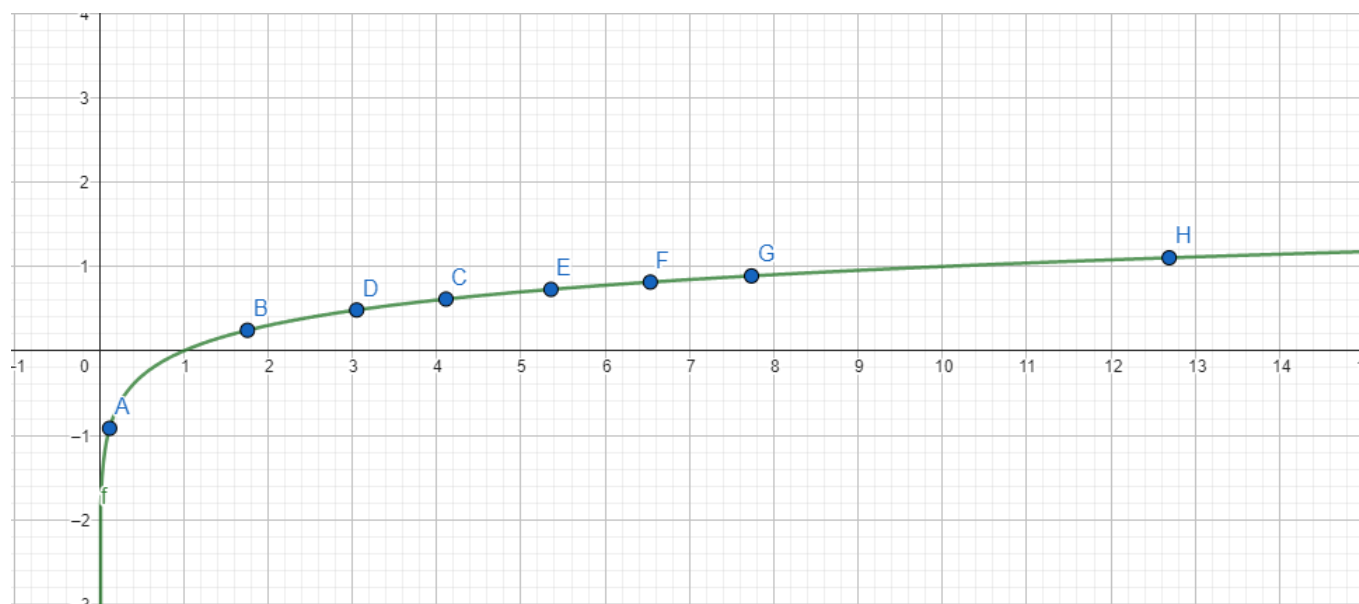


Figura 4: Puntos trazados en el logaritmo al que pertenecen

Sin embargo, puede también usarse esos puntos para generar un polinomio que pase por todos ellos

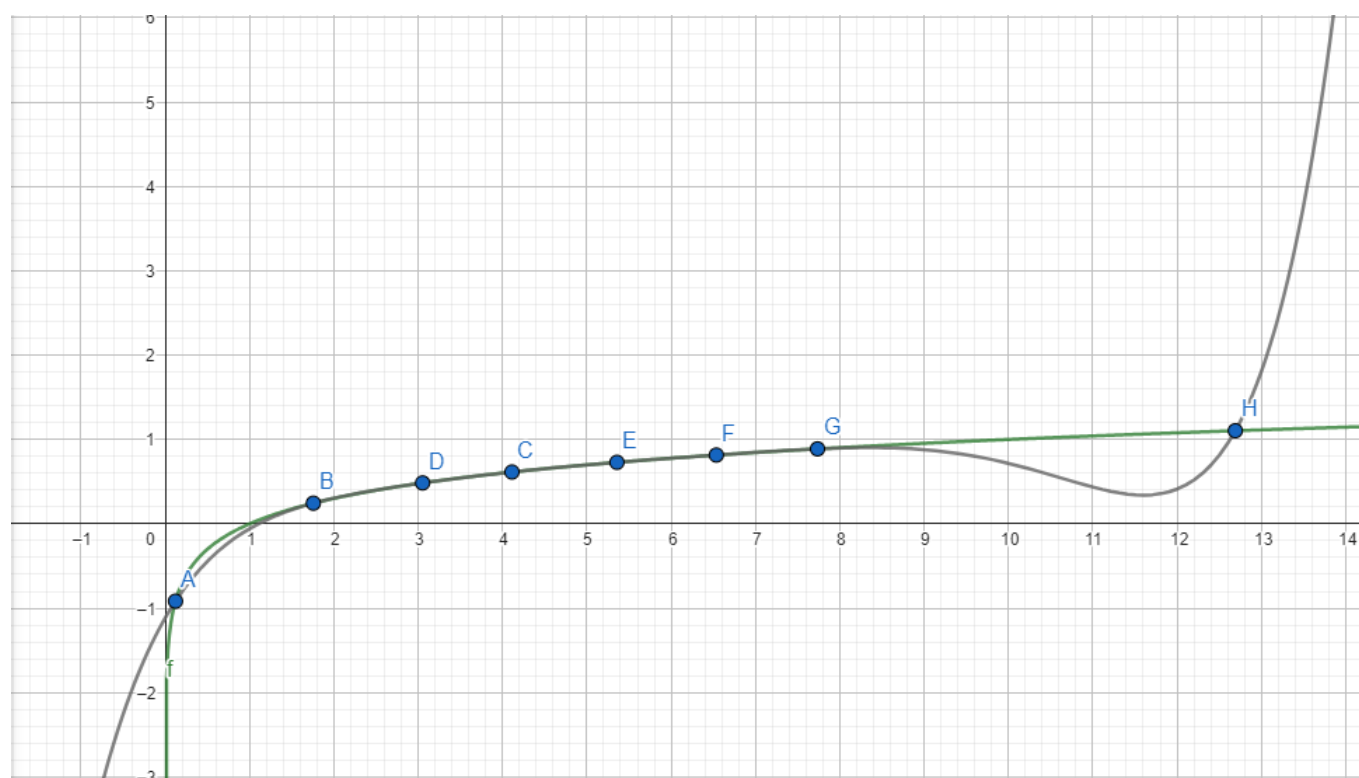


Figura 5: Logaritmo junto con el polinomio

es evidente que el polinomio difiere en una gran cantidad de puntos con f , pero sigue siendo un polinomio. Es debido a esto que si se quiere corroborar que una serie de puntos pertenece a un polinomio, no se debe corroborar únicamente si se cumple aquello, ya que toda serie de puntos puede cumplir con esa condición, sino que se debe corroborar que el grado del polinomio que pasa por todos esos puntos no sea extremadamente grande. Se verifica entonces que la lista de puntos obtenida pertenece a un polinomio de grado bajo (o del grado esperado), y no a uno de grado alto, que sería necesario generar para pasar por todos los puntos de la lista en caso de que estos no pertenezcan a un polinomio de grado bajo (es decir, pertenecen a una función que no es un polinomio o pertenecen a un polinomio de grado alto).

7. Zero Knowledge proofs

Una ZKP es un procedimiento por el cual un *prover* demuestra con probabilidad lo suficientemente alta una declaración (a partir de ahora *statement*) de forma tal que el *verifier* pueda comprobar que la declaración es verdadera sin obtener información innecesaria para hacerlo. En otras palabras, una ZKP es un procedimiento tal que dado un X se demuestra que

$$f(X) = Y$$

sin revelar X . Generalmente estas pruebas tienen una naturaleza iterativa y probabilística, en el que cada iteración se reduce la probabilidad de que un prover esté engañando a un verifier.

Este tipo de pruebas tiene una gran cantidad de usos, ya que en la vida cotidiana se suele revelar más información de la necesaria. Se podría considerar por ejemplo una situación en la que una persona solicita un préstamo a un banco, para lo cual debe demostrar que su sueldo se encuentra dentro de un rango determinado. Normalmente esto se comprobaría comunicando directamente el valor, pero una ZKP permitiría con probabilidad despreciable de engaño demostrar que este se encuentra dentro del rango esperado sin revelar específicamente cuánto es que quien pide el préstamo gana por mes.

7.1. Ejemplo con pelotas de colores

Uno de los ejemplos mas sencillos se trata de la corroboración del conocimiento de una forma de distinguir dos elementos que para otra entidad no son discernibles. Si se asume la existencia de un verifier A , que por alguna razón no puede distinguir el color rojo del azul, y un prover B , quien sí puede hacerlo, puede darse el caso en que A quiera poder corroborar que B realmente es capaz de hacerlo. Para no revelar su método de distinción, ni cuál pelota tiene cada color, B puede recurrir al uso de una ZKP.

Si A toma las 2 pelotas y se las muestra a B , puede luego esconderlas detrás de su espalda y decidir si intercambia sus posiciones o no. Una vez mantenidas o cambiadas de lugar las pelotas, A procederá a revelarlas nuevamente a B , quien dirá si fueron cambiadas o no de lugar. En caso de responder de forma errónea, se sabrá que B realmente no sabe diferenciarlas, y por lo tanto miente. Sin embargo, si responde correctamente, se deberá considerar que la posibilidad de que se diera este suceso sin saber distinguir las pelotas es del 50 %.

Este experimento puede repetirse todas las veces que A lo considere necesario ya que, al ser cada uno independiente del anterior, pueden multiplicarse las probabilidades entre sí, y por tratarse de un experimento con 2 eventos equiprobables se dividirá por 2 en cada .^acierto"de B la probabilidad de que simplemente haya tenido suerte en sus respuestas. Se ve entonces que la fórmula de la probabilidad de que la prueba sea confiable es

$$1 - \left(\frac{1}{2}\right)^n$$

siendo n la cantidad de veces que se realizó el experimento. Si se calcula la probabilidad de una ausencia de un falso positivo luego de 10 repeticiones, se verá que es del 99,9 %, lo cual es considerablemente aceptable y, en caso de no serlo, puede mejorarse con la adición de tantas repeticiones como A juzgue como necesarias.

8. STARKS

8.1. Introducción

El acrónimo STArK proviene de la frase Scalable Transparent Argument of Knowledge, es un tipo de zero knowledge proof que demuestra la correcta ejecución de un cómputo, es decir, que un resultado proviene de la ejecución de un algoritmo específico. Esto se hace mediante la generación de una lista de todos los pasos intermedios del código a ejecutar (llamada trace de ejecución), junto con restricciones para las transiciones entre esos pasos. En simples palabras, la prueba consiste en demostrar que el trace cumple con las restricciones impuestas sin revelar el trace en sí.

Este tipo de pruebas resulta de gran utilidad ya que, aunque la generación de una prueba es de orden $n \cdot \text{polylog}(n)$, siendo n la cantidad de pasos ejecutados y polylog una composición entre un polinomio y un logaritmo (por ejemplo de la forma $5\log^3(n) + 2\log(n) + 7$), por otro lado la verificación es de orden $\text{polylog}(n)$. Esto implica que, a cambio de tardar más tiempo en generar una prueba de ejecución, se obtiene una forma de corroborar el resultado tal que para un n lo suficientemente grande resulta más eficiente realizar una verificación de la prueba que ejecutar el programa probado en sí.

8.2. Traza ejemplificada

Esto puede verse en el siguiente ejemplo de un cálculo del precio final de una compra. Notar que para un caso tan pequeño la prueba no sería eficiente, pero es útil para seguir los pasos de generación de la prueba.

Item	Precio
Producto 1	1,15
Producto 2	2,36
Producto 3	9,64
Producto 4	2
Total	15,15

Figura 6: Ejemplo de factura de una compra

Para corroborar que el total es realmente el valor calculado finalmente, se puede recalcular la suma de los precios y comparar el resultado con el proporcionado. Sin embargo, esto no es lo que se quiere realizar ya que consistiría en replicar el cálculo, lo ideal sería corroborar de otra manera que el resultado es correcto, sin repetir la suma.

Para comenzar a generar la prueba se debe crear el trace de ejecución, este no es la lista provista ya que esta es únicamente una serie de items con su precio. Para obtener el resultado final de la suma de todos los elementos se deben ir sumando los precios, es decir, comenzar con un precio acumulado de 0, sumar el valor del producto 1 y guardar el resultado como valor acumulado, repitiendo este proceso para el resto de los productos hasta llegar al último. Esto puede verse en la siguiente ilustración:

Item	Precio	Total actual
Producto 1	1,15	0
Producto 2	2,36	1,15
Producto 3	9,64	3,51
Producto 4	2	13,15
Total	15,15	15,15

Figura 7: Trace de ejecución de la suma de la factura

Se ve entonces que el trace consiste en una matriz de 2 columnas y 5 filas, siendo la primera columna la de los precios de los productos y la segunda la del valor actual de la suma, es decir, una variable de acumulación. Es evidente que el valor de una celda de la suma acumulada consiste en la suma del precio del producto de la fila anterior y el valor anterior de la variable de acumulación, por lo que puede definirse la restricción, siendo $T_{a,b}$ la celda de la columna a, fila b.

$$T_{1,i} + T_{2,i} = T_{2,i+1}, 1 \leq i \leq 4$$

Las restricciones suelen expresarse igualadas a cero, ya que, como se vio en la sección **6.3.3**, se considera que una restricción se cumple cuando una evaluación de esta retorna cero. Queda entonces la siguiente expresión.

$$T_{1,i} + T_{2,i} - T_{2,i+1} = 0$$

La restricción expresada se denomina de transición, ya que corrobora que en el cambio de estado del programa se mantenga una relación específica, que en este caso sería la correcta ejecución de la suma. Se tiene también una restricción de límite, o boundary constraint, que es aquella que define valores específicos sobre celdas específicas, en este caso es necesario asegurarse de que el valor inicial de la variable de acumulación de la suma sea cero. Aunque viendo el trace parezca obvio, para el verifícar, que no tiene acceso a este, es una restricción necesaria.

$$T_{1,2} = 0$$

Por último, se tiene una restricción local, que es aquella que se aplica a una única fila, siendo esta la que se asegura de que el valor final de la suma reportada sea igual al del último valor de la variable de sumas acumuladas.

$$T_{5,1} - T_{5,2} = 0$$

Lo que se quiere corroborar con la prueba para verificar que el cómputo fue realizado correctamente es que las restricciones valgan 0 en todos los puntos en las que deben ser evaluadas. A continuación se realizará una explicación genérica sobre cómo es que se corrobora la validez de estas restricciones.

8.3. Restricciones sobre la traza

Se vio en la sección anterior que las celdas de una traza pueden ser expresadas con coordenadas sobre una matriz, sin embargo, resulta de gran comodidad y utilidad representar cada columna como evaluaciones de un polinomio. Como cada polinomio debe pasar exactamente por cada punto indicado por el trace, cada columna será representada entonces por una interpolación de puntos que pasan por los valores almacenados en la traza, que serán parte del conjunto de imagen, y se

posicionarán en el eje x sobre valores del tipo g^i , siendo g un generador de valores del espacio modular sobre el que se trabajará.

Las restricciones también se representarán con polinomios, aunque estos generalmente serán multivariados, ya que representarán relaciones necesarias entre valores de distintas columnas y filas de la traza. Una restricción será entonces de la forma

$$C(X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_m, \dots)$$

Siendo m la cantidad de columnas de la traza, representando así cada variable de una misma letra los valores de distintas columnas de una fila de la traza, y considerando que se cumple una restricción cuando su valor es 0. Al hacer una composición entre un polinomio de restricción y los polinomios de traza se obtendrá un polinomio univariado que representa la evaluación de la restricción sobre las columnas de la traza. A modo de ejemplo, una restricción pensada para relacionar 3 filas de la traza tendría la forma

$$C(X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_m, Z_1, Z_2, \dots, Z_m)$$

Si fue planeada para relacionar 3 filas consecutivas, al ejercer la restricción sobre la traza se obtendría entonces la siguiente composición

$$C(t_1(x), t_2(x), \dots, t_m(x), t_1(x.g), t_2(x.g), \dots, t_m(x.g), t_1(x.g^2), t_2(x.g^2), \dots, t_m(x.g^2))$$

Al ser las evaluaciones del polinomio en los puntos de la traza sobre g^i , el multiplicar a la variable x por g resulta en obtener el siguiente valor de la traza en el eje x respecto del actual. Esto puede repetirse a medida que sea necesario para obtener valores de otras filas distintas a la inmediatamente siguiente, como es el caso de g^2 , permitiendo incluso utilizar exponentes negativos. Notar también que todas las variables pasan a ser polinomios que dependen de la misma variable, por lo que se termina pasando de un polinomio multivariado a uno univariado. La restricción mostrada relaciona los valores de la traza para una fila con los valores de la traza para la fila siguiente (es decir, el siguiente estado de la ejecución) y la que sigue a esta.

Volviendo al ejemplo de la sección **8.2**, si se llama a la columna 1 $t_1(x)$ y a la 2 $t_2(x)$ las restricciones quedan de la siguiente manera:

- Restricción de transición:

$$C_0(X_1, X_2, Y_2) = X_1 + X_2 - Y_2$$

Evaluado en la traza queda entonces

$$C_0(t_1(x), t_2(x), t_2(x.g)) = t_1(x) + t_2(x) - t_2(x.g)$$

Expresando así que para los valores g , g^2 , g^3 y g^4 (previamente denotados como $1 \leq i \leq 4$) el resultado de la evaluación debe dar 0 si la ejecución fue honesta.

- Restricción de límite:

$$C_1(X_2) = X_2$$

Evaluado en la traza queda entonces

$$C_1(t_2(x)) = t_2(x)$$

Expresando así que para el valor g el resultado de la evaluación debe dar 0 si la ejecución fue honesta.

- Restricción local

$$C_2(X_1, X_2) = X_1 - X_2$$

Evaluado en la traza queda entonces

$$C_2(t_1(x), t_2(x)) = t_1(x) - t_2(x)$$

Expresando así que para el valor g^5 (previamente la fila 5 en la traza de ejemplo) el resultado de la evaluación debe dar 0 si la ejecución fue honesta.

Si además quiere expresarse en las funciones en qué puntos deben cumplirse estas restricciones en vez de tener que expresarlo de forma externa como se hizo previamente, se debe recordar lo dicho en la sección **6.3.4**, y cambiar la verificación de que las restricciones evaluadas en la traza son cero por la verificación de que las restricciones evaluadas en la traza, divididas por los puntos en los que estas restricciones se deben cumplir, son polinomios. Quedarían escritas entonces de la siguiente manera, siendo D_i la restricción reescrita de esta nueva forma, y Z_i el polinomio que representa los puntos en los que C_i (que por facilidad serán escritas como $C_i(x)$, en vez de volver a expresar la composición con los polinomios de la traza) debe cumplirse:

- Restricción de transición:

$$D_0(x) = \frac{C_0(x)}{Z_0(x)} = \frac{t_1(x) + t_2(x) - t_2(x.g)}{(x-g)(x-g^2)(x-g^3)(x-g^4)}$$

Expresando así que se cumple la restricción en los puntos g, g^2, g^3 y g^4 si D_0 es un polinomio.

- Restricción de límite:

$$D_1(x) = \frac{C_1(x)}{Z_1(x)} = \frac{t_2(x) - 0}{x - g}$$

Expresando así que se cumple la restricción en el punto g si D_1 es un polinomio.

- Restricción local

$$D_2(x) = \frac{C_2(x)}{Z_2(x)} = \frac{t_1(x) - t_2(x)}{x - g^5}$$

Expresando así se cumple la restricción en el punto g^5 si D_2 es un polinomio.

Se ve entonces que dada una traza que debe cumplir con n restricciones genéricas C_i , estas podrán ser corroboradas mediante la verificación de que las funciones D_i son polinomios de grado bajo como se indicó en la sección **6.3.4**. Si todos los D_i son polinomios, entonces la suma de estos resultará en un polinomio, si alguno de estos no es un polinomio o es un polinomio de alto grado, entonces la suma resultará en una función que es un polinomio de grado alto o no es un polinomio. Debido a esto, en vez de probar que cada D_i es un polinomio de grado lo suficientemente bajo, se puede probar que su suma lo es, pasando así de probar el grado de n polinomios a probar el grado de uno solo, que será llamado $H(x)$. De esta forma, operando sobre un único polinomio se prueban todas las restricciones sobre la traza. Notar que, al ser este la suma de los polinomios, probablemente tendrá el mismo grado que el mayor de estos y, al tener cada polinomio D_i en su numerador una composición de polinomios, si la restricción es un polinomio de grado 2, al evaluarlo sobre la traza resultará en un polinomio de grado igual al doble del proveniente de la traza, lo cual duplicará también el grado de H .

8.4. Generación de la prueba

El verifer debe comprobar que la traza cumple con las restricciones, pero como se indicó en la sección 8.1, el prover no quiere realmente revelar la traza, solo probar su validez. Se comentó en la sección anterior que puede demostrarse corroborando que el grado del polinomio $H(x)$ no es muy alto y, como se mostró en la sección 6.3.2, esto puede probarse utilizando un muestreo de puntos. Se comenzará entonces a explicar el proceso de generación de la prueba.

Dada una traza de ejecución de m columnas y r líneas, el prover la extenderá hasta que esta tenga una cantidad de líneas que sea potencia de 2, es decir 2^n . Esto puede realizarse de varias formas, una de ellas es definir algunas columnas como de control, las cuales indicarán a cuáles líneas deben aplicarse las restricciones.

Para cada columna de la traza, el prover realizará una interpolación de sus valores sobre ciertos puntos, cuyo patrón será explicado más adelante. Para poder entenderlo, se debe explicar primero qué es el *blowupfactor*. Este valor indica el multiplicador de la cantidad de puntos en los que el prover deberá evaluar los polinomios generados por su traza, si por ejemplo 6 puntos definen un polinomio de grado 5, un factor de 3 requeriría que el prover calcule los valores del polinomio para $6 \cdot 3 = 18$ puntos distintos. Dado que las trazas tienen largo 2^n , y el *blowupfactor* es de la forma $b = 2^l$, el prover deberá evaluar los polinomios resultantes de su traza en 2^{n+l} puntos. Mientras mayor sea este número, más fácil será corroborar si el polinomio tiene un grado mayor al esperado o no, ya que, como fue explicado en la sección 6.3.1, al aumentar la cantidad de puntos en los que se evalúan estos polinomios, mayor será la probabilidad de que, al consultar uno de estos, se obtenga un resultado que no coincida con el polinomio esperado (en caso de que el prover no sea honesto), y que por lo tanto al ser evaluado en las restricciones genere puntos que no pertenecen a un polinomio de grado bajo.

También es necesario entrar en detalle sobre qué constituyen las raíces unitarias primitivas. Dado el polinomio $X^n - 1$, este tendrá n raíces que, al ser elevadas a la n valdrán 1 y por lo tanto se consideran unitarias. Todo set de raíces unitarias tiene una raíz g que puede utilizarse como generadora del resto de las raíces unitarias, a este valor se lo denomina raíz unitaria primitiva. De esta forma, puede obtenerse todo el set de raíces mediante el cálculo $g^i, 0 \leq i < n$. Esto es de gran utilidad ya que si la interpolación de los polinomios se da sobre los puntos g^i , entonces la división de las restricciones por los puntos en los que deben aplicarse puede realizarse de forma más eficiente para aquellas restricciones que deban aplicarse sobre toda la traza ya que podrá expresarse como

$$\frac{C_j(x)}{X^n - 1}$$

y en casos en los que la restricción se aplica sobre una gran parte de la traza salvo excepciones, se puede dividir la expresión compacta del polinomio por los ceros que quieren ignorarse, de la siguiente manera, siendo X_i todo valor de x en los que no se deba cumplir la restricción

$$\frac{C_j(x)}{\frac{X^n - 1}{\prod (X - X_i)}}$$

en vez de realizar el producto de todos los $(x - g^i)$ del polinomio, lo cual requiere un poder de cómputo mucho mayor.

Dicho esto, el prover comienza definiendo el valor w como la raíz unitaria primitiva de grado 2^{n+l} , y $g = w^b$ (recordar que $b = 2^l$ es el blowup factor). Al ser el generador $g^i = (w^b)^i$, se ve que los valores generados por g son un muestreo de los generados por w , tomando uno de cada b elementos, de esta forma, g es la raíz unitaria primitiva de grado 2^n . Hecho esto, se procede a realizar una interpolación para cada columna de la traza sobre los puntos $g^i, 0 \leq i < 2^n$, generando así m polinomios de grado $2^n - 1$.

Luego, se procede a evaluar estos polinomios sobre $2^n \cdot b = 2^n \cdot 2^l = 2^{n+l}$ puntos. A simple vista puede parecer conveniente realizar las evaluaciones sobre w^i , sin embargo, como los polinomios de restricciones serán divididos por los ceros en los que deben cumplirse, y estos serán de la forma g^i ,

en caso de solicitar el valor de un polinomio en uno de estos puntos se estará intentando realizar una división por 0, además de estar el prover revelando valores específicos de la traza. Es por esto que en vez de evaluar sobre w^i , el prover evaluará los polinomios sobre un corrimiento de este set, es decir, sobre $h \cdot w^i$, $0 \leq i < 2^{n+l}$, siendo h un entero distinto de cualquier valor que pueda ser generado con w^i . Comienza entonces el prover el desarrollo de la prueba evaluando sus polinomios de interpolación sobre $h \cdot w^i$, $0 \leq i < 2^{n+l}$, y generando un commit sobre estas evaluaciones con un merkle tree, esto asegura al verifíer que los valores que solicite no fueron generados a conveniencia, sino en un momento en el que no se podía tomar ventaja de ningún pedido realizado.

Uno de los polinomios sobre los que el prover hará un commitment es el polinomio H de composición, conformado por la suma de los polinomios D_i , cada uno multiplicado por un valor aleatorio definido por el verifíer (de esta forma el prover no puede generar polinomios o un resultado tal que se cancelen los términos de mayor grado).

$$H(x) = \sum \alpha_i D_i$$

Dicho esto, se comenta que el verifíer hará commit sobre H además de los polinomios de las columnas de la traza. Sin embargo, si alguna de las restricciones definidas para la prueba es de grado 2, entonces el grado de H será aproximadamente el doble del grado de los polinomios generados por la interpolación de los valores de una columna de la traza, por lo que se recurre a parte de lo visto en la sección **6.3.2**, y en vez de realizar un commit sobre H , se hace sobre $H_1(x^2)$ y $H_2(x^2)$ en los mismos puntos que el resto de los polinomios de la traza.

$$H(x) = H_1(x^2) + x \cdot H_2(x^2)$$

El verifíer procede ahora a definir un valor fuera del dominio sobre el que se hizo el commit para los polinomios, es decir, un valor que no pueda ser generado por g^i o $h \cdot w^i$, que será denominado z , y le pedirá al prover que indique los valores de los polinomios de la traza para el cálculo de H en ese punto. Al ser un valor sobre el cual no hay commit, el prover podría mentir a conveniencia sobre el resultado en este punto, por lo que se crean restricciones de límite sobre el punto indicado y los otros relacionados a este para verificar que realmente los polinomios tienen ese valor en el punto solicitado. Se define entonces el polinomio, con escalares aleatorios provistos por el verifíer, siendo f_i el polinomio resultante de una interpolación de una columna de la traza

$$H'_0(x) = \beta \frac{H_1(x) - H_1(z^2)}{x - z^2} + \beta' \frac{H_2(x) - H_2(z^2)}{x - z^2} + \sum \beta_i \frac{f_i(x) - f_i(z \cdot g^s)}{x - z \cdot g^s}$$

siendo s una constante que depende de si el valor que se está corroborando presenta un desfase respecto de z o no, esto será determinado según si es necesario obtener el valor de una fila distinta respecto de la actual para una columna para calcular la validez de alguna restricción. Si por ejemplo se necesitara para calcular $H(z)$ los valores de z y de $z \cdot g$ para cada columna de la traza, entonces el polinomio sería el siguiente

$$\beta \frac{H_1(x) - H_1(z^2)}{x - z^2} + \beta' \frac{H_2(x) - H_2(z^2)}{x - z^2} + \sum \beta_i \frac{f_i(x) - f_i(z)}{x - z} + \beta'_i \frac{f_i(x) - f_i(z \cdot g)}{x - z \cdot g}$$

El polinomio $H'_0(x)$ se denomina polinomio DEEP (Domain Extension for Elimination of Pretenders), y se calcula para que haya mayor probabilidad de detectar un fallo de parte de un prover malicioso que logró crear un polinomio H que pasa la prueba en los puntos muestreados, y no fuera de estos. Al probar el bajo grado de $H'_0(x)$, se prueba que $H(x)$ es un polinomio de grado bajo (y por lo tanto que la prueba es válida), y lo mismo para todos los $f_i(x)$, además de probar que en el punto z y relacionados los polinomios tienen los valores indicados.

Para probar que $H'_0(x)$ es un polinomio de grado bajo, se utiliza el método descrito en **6.3.2** con ciertas modificaciones y conceptos a tomar en cuenta. A diferencia de lo explicado previamente, si $H'_i(x) = P_i(x^2) + x \cdot I_i(x^2)$, en la prueba se calcula el polinomio hijo de mitad de grado como

$$H'_{i+1}(x) = P_i(x) + \alpha \cdot I_i(x)$$

siendo α un valor aleatorio provisto por el verifíer para evitar que el prover cancele términos de alto grado en los polinomios sumados. Se debe tomar en cuenta también que, al trabajar con valores provenientes de un commit, otras consideraciones son necesarias. Se vio en la sección mencionada que, dados $H'_i(x)$ y $H'_i(-x)$ puede obtenerse $H'_{i+1}(x^2)$, sin embargo, no se puede obtener $H'_{i+1}(-x^2)$, por lo que el prover deberá prover en su commit, esto ya se da naturalmente por el espacio de evaluación elegido para el commitment. Se debe tomar en cuenta también que, con cada paso de la reducción de grado del polinomio, se reducirá a la mitad la cantidad de puntos sobre los que se evalúa a este (si w^i tarda 2^{n+l} puntos en terminar de generar todos los valores que puede brindar, entonces w^{2i} generará la mitad de puntos ya que llegará a $w^{2^{n+l}}$ en la mitad de pasos, y la multiplicación de todos los valores por una constante genera únicamente un corrimiento de estos). Por último, si $\text{grado}(H'_0) = d$, entonces luego de $\log_2(d)$ pasos se llegará a un polinomio constante, valor que el prover enviará al verifíer para una correcta verificación de la prueba. El prover entonces calcula y realiza un commitment sobre todos los polinomios intermedios $H'_{i+1}(x)$ hasta llegar al polinomio constante, cuyo valor indicará al verifíer antes de que este corrobore la validez de la prueba.

Definidos todos los polinomios mencionados por parte del prover, comienza el procedimiento de verificación de que $H'_0(x)$ es un polinomio de grado bajo. El verifíer conoce las restricciones que debe cumplir la traza y todos los valores que debe proveerle al prover, por lo que dados los valores de f_i para un punto en particular puede calcular H y H'_0 para ese punto sin problemas. El verifíer podrá comenzar ahora a iterativamente realizar tests de validez sobre los valores sobre los que el prover realizó un commitment, agregando bits de seguridad con cada test realizado.

Antes de realizar estas pruebas iterativas, se debe corroborar que los valores que provienen de los commits sobre H_1 y H_2 provienen de un cálculo honesto sobre los valores de la traza sobre los que hizo commit el prover. Para hacer esto se utilizarán las imágenes de los polinomios en el punto z fuera de dominio que se eligió para crear el polinomio DEEP H'_0 . Se corrobora entonces (tomando en cuenta que, al conocer el verifíer las restricciones que aplican sobre la traza y los puntos sobre los que deben aplicar, pueden calcularse los valores de $D_i(q)$) que

$$H_1(z^2) + z \cdot H_2(z^2) = \sum \alpha_i D_i(z)$$

En caso de que la verificación falle, se rechaza la prueba, sino se procede con el resto de los pasos. Esto permite corroborar correctamente que los valores de los commits de H_1 y H_2 son derivados de la suma de los D_i ya que, como se vio en la sección 6.3.1, si el grado del polinomio que se quiere verificar es conocido entonces se sabe que la probabilidad de que coincida la imagen de un punto aleatorio es de $\frac{g}{w}$, siendo g el grado del polinomio al que se está realizando el test (que será del mismo que el esperado en el caso ideal, y mayor a este si el prover es malicioso) y w la cantidad de elementos de los que se puede seleccionar el valor aleatorio, que será determinado por el tamaño del espacio sobre el que se realiza la aritmética modular (el cual generalmente es considerablemente grande, en algunos casos determinado por un número primo en un espacio de números de 256 bits). Dado un w lo suficientemente grande, y una verificación de un grado bajo para H'_0 , la probabilidad de que en un solo test sobre H se sufra un falso positivo en cuanto a coincidencia de imágenes será minúscula, por lo que la corroboración de una sola imagen bastará.

Pasando ahora a realizar los test iterativos, para un test de validez el verifíer elige aleatoriamente uno de los puntos sobre los que se realizó commit en los polinomios, el cual será llamado q . Solicitará entonces $H_1(q)$, $H_2(q)$ y $H_1(-q)$, $H_2(-q)$ y todos los $f_i(q)$ y $f_i(q \cdot g^s)$, y $f_i(-q)$ y $f_i(-q \cdot g^s)$ necesarios para calcular $H'_0(q)$ y $H'_0(-q)$. De esta forma se puede calcular $H'_1(q^2)$ como se indicó previamente, rechazando la prueba en caso de que el valor calculado difiera con el valor sobre el que hizo commit el prover para $H'_1(q^2)$. En caso de que coincida el resultado, entonces solicitará al prover el valor de $H'_1(-q^2)$ para poder calcular $H'_2(q^4)$, y verificar que el valor para el commit del prover de $H'_2(q^4)$ sea el mismo. Esto se repetirá hasta llegar al polinomio constante, punto en el cual el verifíer corroborará si llegó al mismo valor, rechazando la prueba en caso de tratarse de otro.

Esta prueba se repetirá con distintas elecciones de q hasta que el verifier esté satisfecho con la probabilidad de fallo. Se indicó previamente que el *blowupfactor* tenía la forma 2^l e influenciaba la probabilidad de encontrar puntos que elevaran el grado del polinomio analizado. Si se analiza la seguridad del protocolo en cuanto a bits de seguridad, este otorga por cada query realizada a H'_0 (es decir, por cada elección de un q , seguido por el proceso explicado) aproximadamente l bits de seguridad. Esto quiere decir que para un factor de 16, por cada query se agregan aproximadamente 4 bits de seguridad, por lo que con 64 queries se obtendría una seguridad de aproximadamente 264 bits, lo cual es perfectamente viable. Si se toma en cuenta que la prueba se genera en un momento, y no se está encriptando algo que puede ser descryptado en el futuro, se pueden usar menos bits de seguridad, ya que lo que es relevante es el poder de cómputo al momento de generar la prueba. En risc0 se utiliza un blowup factor de 4 y 50 queries, obteniendo así 100 bits de seguridad.

9. ZKVM

Hasta el momento fue explicada únicamente la teoría general de STARKS. Sin embargo, deben aplicarse sobre esta otras abstracciones para facilitar la ejecución de código, ya que con lo presentado se debe diseñar un set de restricciones y columnas de traza para cada cómputo distinto que se quiera realizar.

Para solucionar esto, actualmente se recurre al diseño de una máquina virtual cuya correcta ejecución será verificada. Esto quiere decir que, en vez de probar la correcta ejecución de un cómputo particular, se simulará la ejecución de una computadora, y lo que se probará es que esta simulación fue correcta, que el código que esta computadora ejecutó fue el esperado, y que las transiciones de estado en los distintos pasos del programa son consistentes con el código en ejecución. Se tomará como ejemplo la máquina virtual de Cairo, un lenguaje pensado para la generación de pruebas STARK para programas arbitrarios (es decir, cualquier programa que pueda ser escrito en el lenguaje de programación Cairo).

Se explicará parcialmente cómo se adapta la ejecución de la máquina virtual de Cairo para la generación de una prueba starks. No se entrará en detalles sobre su funcionamiento, ciertas características de la memoria, o pruebas matemáticas sobre estas.

9.1. Memoria

La memoria de la máquina virtual de Cairo es continua, por lo que si a_i tiene un valor entonces todos los a_j con $j < i$ tendrán que tener un valor asignado. Esta también puede ser asignada una única vez (es inmutable), por lo que si a_i fue asignado un valor, mantendrá ese valor toda la ejecución. Esto facilita la generación de la prueba, ya que se pueden establecer relaciones entre operaciones sobre direcciones de memoria, y estas serán válidas durante todo el programa. Si se quiere verificar para $a_i + a_j - > a_k$ que a_k realmente guarda la suma de los valores de a_i y a_j , entonces la verificación fallará si en el futuro a_k se reutiliza para guardar $a_i \cdot a_j$.

Habrán también restricciones sobre la memoria que permitirán al prover demostrar que una sección de memoria, denominada memoria pública, proviene realmente de la ejecución del código. Esta será considerada el output de la ejecución, y allí podrá ser incluido cualquier dato que desee el programa.

Existirá también un input para la ejecución que será definido por el prover, el cual será privado. Sin embargo, este puede ser expuesto en la memoria pública si el desarrollador del programa a probar lo desea. Si el prover no cumple con el requerimiento de exponer el input, entonces estará ejecutando un programa distinto al que espera el verifier y fallará la prueba.

9.2. Código ejecutado

Como se especificó previamente, en vez de probarse un cómputo específico, se probará la ejecución de la máquina virtual. Esto presenta la duda sobre cómo sabe el verifier que el código que ejecutó esta máquina es el suyo, y no otro arbitrario elegido maliciosamente por el prover. A

modo de solución, se carga el programa a ejecutar en memoria, pero adicionalmente se corre sobre este un algoritmo de hashing. El verifier entonces verificará que el hash del programa ejecutado sea el esperado, y de esta forma se asegura que el prover no lo engaña respecto a las instrucciones realizadas, y adicionalmente permite al prover generar pruebas de código sobre el cual el verifier conoce únicamente el hash, y no las instrucciones que lo componen.

9.3. Componentes de la máquina virtual

La máquina virtual posee los siguientes registros:

- Program counter (**pc**): contiene la dirección de memoria de la instrucción que se encuentra siendo ejecutada en el momento.
- Allocation pointer (**ap**): contiene la dirección de memoria de la siguiente celda no utilizada. Este valor es una convención, no es enforzado por la prueba.
- Frame pointer (**fp**): apunta al comienzo del marco del stack de la función que está siendo ejecutada en el momento. Al comenzar la ejecución de una función, **fp** tendrá el mismo valor que **ap**, luego **ap** irá incrementando a medida que se van asignando valores. Una vez que termina la ejecución de una función, **fp** toma su valor anterior, es decir, el comienzo del marco de ejecución que la función que llamó a la que recientemente terminó. Se ve entonces que **fp** se mantiene constante para todas las instrucciones de la función que se encuentra en ejecución.

Las instrucciones se encuentran conformadas por una o dos palabras de 64 bits cada una, tratándose del segundo caso cuando se está utilizando un valor inmediato (es decir, una constante escrita explícitamente en la instrucción). Para el caso de una sola palabra se presenta la siguiente estructura:

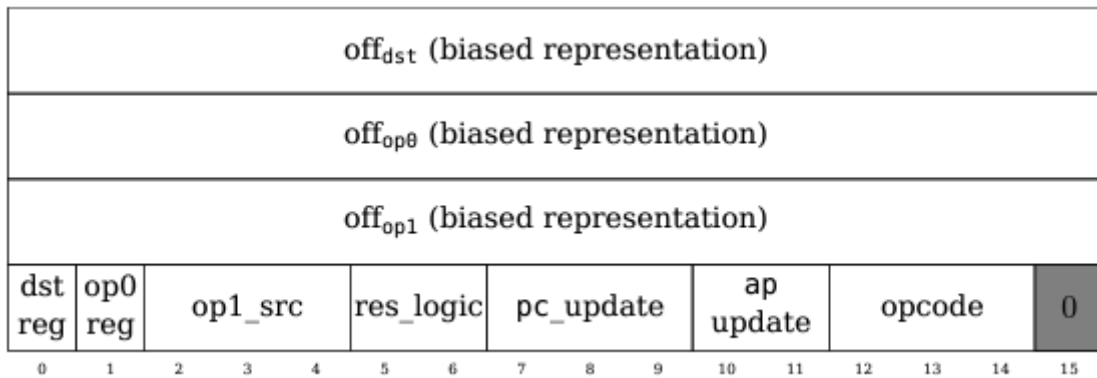


Figura 8: Estructura de instrucción de Cairo ¹

Los valores de **off_{dst}**, **off_{op0}** y **off_{op1}** son números de 16 bits que pertenecen al rango $[-2^{15}, 2^{15})$ y representan el offset en memoria de los distintos operadores de la instrucción actual. Los últimos 16 bits de la instrucción son destinados para flags y son los siguientes:

- dst_reg: 1 bit, define si el offset de la dirección de memoria del destino del resultado es respecto a **ap** o **fp**.
- op0_reg: 1 bit, define si el offset de la dirección de memoria de op0 es respecto a **ap** o **fp**.

¹Cairo – a Turing-complete STARK-friendly CPU architecture, Lior Goldberg, Shahar Papini, Michael Riabzev, August 2021, Figure 3

- `op1_src`: 3 bits, define respecto de qué valor se toma el offset de **op1**, puede ser respecto del valor inmediato que iría en la otra palabra, respecto de **ap**, de **fp** o del valor que guarda la dirección de **op0**. Solo uno de ellos puede valer 1 al mismo tiempo, por lo que puede tomar los valores 0, 1, 2 y 4.
- `res_logic`: 2 bits, define si se trata de una operación de suma, multiplicación o ninguna, puede tomar los valores 0, 1 y 2.
- `pc_update`: 3 bits, define los tipos de saltos que puede realizar el program counter (**pc**), estos pueden ser del tipo regular (pasa a la siguiente instrucción), con desfase absoluto o relativo, o de tipo condicional. Puede tomar los valores 0, 1, 2 y 4.
- `ap_update`: 2 bits, aporta en la definición del próximo valor del registro **ap**, puede tomar los valores 0 y 1.
- `op_code`: 3 bits, define si se está realizando un llamado a una función, un retorno de una función o una verificación de equidad. Puede tomar los valores 0, 1 y 2.
- bit 15: siempre en cero.

Si se llama f_i al bit i de los flags, entonces estos se encuentran compuestos de la siguiente forma:

$$\begin{aligned}
 \text{dst_reg} &= f_{\text{DST_REG}} \\
 \text{op0_reg} &= f_{\text{OP0_REG}} \\
 \text{op1_src} &= f_{\text{OP1_IMM}} + 2 \cdot f_{\text{OP1_FP}} + 4 \cdot f_{\text{OP1_AP}} \\
 \text{res_logic} &= f_{\text{RES_ADD}} + 2 \cdot f_{\text{RES_MUL}} \\
 \text{pc_update} &= f_{\text{PC_JUMP_ABS}} + 2 \cdot f_{\text{PC_JUMP_REL}} + 4 \cdot f_{\text{PC_JNZ}} \\
 \text{ap_update} &= f_{\text{AP_ADD}} + 2 \cdot f_{\text{AP_ADD1}} \\
 \text{opcode} &= f_{\text{OPCODE_CALL}} + 2 \cdot f_{\text{OPCODE_RET}} + 4 \cdot f_{\text{OPCODE_ASSERT_EQ}}
 \end{aligned}$$

Figura 9: Composición de los flags como combinación de los bits de la instrucción ²

Se denomina $\widetilde{\text{off}}_*$ a $\text{off}_* + 2^{15}$, pasando así del rango de valores $[-2^{15}, 2^{15})$ al rango $[0, 2^{16})$. Dicho esto, la instrucción se conforma con la siguiente operación (recordar que en binario multiplicar un número por 2^i genera un shift de i bits hacia los dígitos más significativos):

$$\text{inst} = \widetilde{\text{off}}_{\text{dst}} + 2^{16} \cdot \widetilde{\text{off}}_{\text{op0}} + 2^{16 \cdot 2} \cdot \widetilde{\text{off}}_{\text{op1}} + 2^{16 \cdot 3} \cdot \sum_{i=0}^{14} 2^i \cdot f_i$$

9.4. Restricciones sobre la traza

La traza de la máquina virtual es considerablemente compleja, contiene columnas, columnas virtuales, subcolumnas, etc, que serán representadas mediante variables. Esto suma dificultad que no es relevante para el nivel de entendimiento que se busca sobre la generación de una prueba de ejecución correcta de una VM. Debido a esto, se mostrarán únicamente las restricciones que se consideren necesarias para un entendimiento general sobre la forma de verificar una correcta transición entre estados de la máquina. Se listarán las ecuaciones de cada una de las restricciones

²Cairo – a Turing-complete STARK-friendly CPU architecture, Lior Goldberg, Shahar Papini, Michael Riabzev, August2021, Página 51

seleccionadas y se explicará cómo es que fuerzan el funcionamiento esperado. Se recuerda que para toda restricción funcional se deberá despejar todos los términos de las ecuaciones hacia un lado de la igualdad, para verificar su cumplimiento en los ceros.

$$\text{inst} = \widetilde{\text{off}}_{\text{dst}} + 2^{16} \cdot \widetilde{\text{off}}_{\text{op0}} + 2^{16 \cdot 2} \cdot \widetilde{\text{off}}_{\text{op1}} + 2^{16 \cdot 3} \cdot \sum_{i=0}^{14} 2^i \cdot f_i$$

Esta ecuación que ya fue vista previamente como la forma de calcular el número binario que representa una instrucción, sirve también como restricción para corroborar que toda instrucción de la traza esté compuesta por la suma correcta de los componentes apropiados de la traza.

$$(f_i - 1) \cdot f_i = 0, 0 \leq i < 15$$

Asegura que todos los bits de los flags de la instrucción realmente valgan 1 o 0. Aunque a simple vista esto parece ser innecesario, se debe recordar que starks corrobora restricciones sobre cualquier tipo de traza, que podría contener valores no binarios. Por esto se debe corroborar que las columnas de los bits de los flags de las instrucciones no contienen valores que no tienen sentido para una VM.

$$\text{dst_addr} = f_{\text{DST_REG}} \cdot \text{fp} + (1 - f_{\text{DST_REG}}) \cdot \text{ap} + \text{off}_{\text{dst}}$$

Corrobora que la dirección de destino de la instrucción actual sea consistente con la lógica de los flags, siendo $\text{fp} + \text{off}_{\text{dst}}$ si $f_{\text{DST_REG}} = 1$, y $\text{ap} + \text{off}_{\text{dst}}$ si $f_{\text{DST_REG}} = 0$.

$$\text{op0_addr} = f_{\text{OP0_REG}} \cdot \text{fp} + (1 - f_{\text{OP0_REG}}) \cdot \text{ap} + \text{off}_{\text{op0}}$$

Corrobora que la dirección de fuente del valor del operador 0 de la instrucción actual sea consistente con la lógica de los flags, siendo $\text{fp} + \text{off}_{\text{op0}}$ si $f_{\text{OP0_REG}} = 1$, y $\text{ap} + \text{off}_{\text{op0}}$ si $f_{\text{OP0_REG}} = 0$.

$$\text{op1_addr} = f_{\text{OP1_IMM}} \cdot \text{pc} + f_{\text{OP1_FP}} \cdot \text{fp} + f_{\text{OP1_AP}} \cdot \text{ap} +$$

$$(1 - f_{\text{OP1_IMM}} - f_{\text{OP1_FP}} - f_{\text{OP1_AP}}) \cdot \text{op0} + \text{off}_{\text{op1}}$$

Corrobora que la dirección de fuente del valor del operador 1 de la instrucción actual sea consistente con la lógica de los flags, siendo $\text{fp} + \text{off}_{\text{op1}}$ si $f_{\text{OP1_FP}} = 1$, $\text{ap} + \text{off}_{\text{op1}}$ si $f_{\text{OP1_AP}} = 1$, $\text{pc} + \text{off}_{\text{op1}}$ si $f_{\text{OP1_IMM}} = 1$ (recordar que, al ser una instrucción compuesta por 2 palabras cuando op1 utiliza un valor inmediato, el valor de la dirección de la palabra a la que apunta pc contendrá la base del offset), y $\text{op0} + \text{off}_{\text{op1}}$ si todos los bits del flag son cero (revisar la ecuación previa para obtener la dirección que almacenará el valor de op0).

$$\text{next_ap} = f_{\text{AP}} + f_{\text{AP_ADD}} \cdot \text{res} + f_{\text{AP_ADD1}} \cdot 1 + f_{\text{OPCODE_CALL}} \cdot 2$$

Corrobora que la asignación del nuevo valor de ap sea el apropiado para cada caso correspondiente. El resultado será $\text{ap} + 1$ si $f_{\text{AP_ADD1}} = 1$, $\text{ap} + \text{res}$ si $f_{\text{AP_ADD}} = 1$, $\text{ap} + 2$ si $f_{\text{OPCODE_CALL}} = 1$ (ya que al llamar a una función se deben guardar los valores previos de fp y pc para restaurarlos cuando se deba volver al contexto de ejecución de la función que llamó a la actual), y no se actualizará en caso de que ninguno de los flags valga 1.

$$\text{next_fp} = f_{\text{OPCPDE_RET}} \cdot \text{dst} + f_{\text{OPCPDE_CALL}} \cdot (\text{dst} + 2) + (1 - f_{\text{OPCPDE_RET}} - f_{\text{OPCODE_CALL}}) \cdot \text{fp}$$

Corroborar que la asignación del nuevo valor de fp sea el apropiado para cada caso correspondiente. Almacenará dst si se está retornando de una función (para dst con un offset de -2 respecto del fp actual), es decir, si $f_{\text{OPCODE_RET}} = 1$, dst + 2 (para dst con offset 0 respecto del fp actual) si se está llamando una función, es decir, si $f_{\text{OPCODE_CALL}} = 1$, y no cambiará de valor si ninguno de los flags mencionados está activado.

$$\text{instruction_size} = f_{\text{OP1_IMM}} + 1$$

Como se indicó previamente, el tamaño de la instrucción puede ser de 1 o 2 palabras de 64 bits cada una, siendo el segundo caso aquel en el que se utiliza un valor inmediato en la instrucción. Se ve entonces que si el flag $f_{\text{OP1_IMM}} = 1$ entonces instruction_size será 2, y sino será 1.

$$t_0 = f_{\text{PC_JNZ}} \cdot \text{dst}$$

$$t_1 = f_0 \cdot \text{res}$$

$$(t_1 - f_{\text{PC_JNZ}}) \cdot (\text{next_pc} - (\text{pc} + \text{instruction_size})) = 0$$

Como se indicó en la sección 8.4, se quiere que el mayor grado de las restricciones utilizadas sea 2, para únicamente duplicar el grado del polinomio de composición. Debido a esto se generan las ecuaciones de t_0 y t_1 , ya que la utilización de cambios de variables permite la reducción del grado de las restricciones. El flag $f_{\text{PC_JNZ}}$ indica si la instrucción actual se trata de un jump condicional, que realizará el salto si la dirección de memoria indicada contiene un valor distinto de cero. Al abstraerse de las necesidades de la traza, si se realizan los reemplazos apropiados sobre la ecuación se obtiene la siguiente expresión:

$$(f_{\text{PC_JNZ}} \cdot \text{dst} \cdot \text{res} - f_{\text{PC_JNZ}}) \cdot (\text{next_pc} - (\text{pc} + \text{instruction_size})) = 0$$

Si $f_{\text{PC_JNZ}} = 0$ se cumple la restricción ya que el término de la izquierda es 0 y cancela toda la ecuación, por lo que el resto de las variables quedan libres. Si $f_{\text{PC_JNZ}} = 1$ y $\text{dst} = 0$ (no se cumple la condición del jump condicional) entonces el término de la izquierda no valdrá 0 y deberá cancelarse el de la derecha, generando así la restricción $\text{next_pc} = (\text{pc} + \text{instruction_size})$, es decir, una actualización normal del program counter. Si $f_{\text{PC_JNZ}} = 1$ y $\text{dst} \neq 0$ entonces se setea $\text{res} = \text{dst}^{-1}$ (ya que no se utiliza realmente en esta instrucción, entonces queda libre para asistir en la restricción), causando así que se cancele el término izquierdo y queden las variables del derecho libres.

$$\begin{aligned} & t_0 \cdot (\text{next_pc} - (\text{pc} + \text{op1})) + (1 - f_{\text{PC_JNZ}}) \cdot \text{next_pc} - \\ & ((1 - f_{\text{PC_JUMP_ABS}} - f_{\text{PC_JUMP_REL}} - f_{\text{PC_JNZ}}) \cdot (\text{pc} + \text{instruction_size}) + \\ & f_{\text{PC_JUMP_ABS}} \cdot \text{res} + f_{\text{PC_JUMP_REL}} \cdot (\text{pc} + \text{res})) = 0 \end{aligned}$$

Para entender correctamente esta restricción, se debe tomar en cuenta en primer lugar que del grupo de flags $f_{\text{PC_JUMP_ABS}}$, $f_{\text{PC_JUMP_REL}}$ y $f_{\text{PC_JNZ}}$ siempre que haya un salto tendrá que valer 1 alguno de ellos, y no más de uno en simultáneo. De esta forma puede verse que si $f_{\text{PC_JNZ}} = 1$ entonces se obtiene la siguiente expresión:

$$dst \cdot (next_pc - (pc + op1)) - (f_{PC_JUMP_ABS} \cdot res + f_{PC_JUMP_REL} \cdot (pc + res)) = 0$$

Se debe recordar que como $f_{PC_JNZ} = 1$, se cancela el término de la derecha ya que un solo jump flag puede estar en 1 en simultáneo, por lo que queda únicamente el izquierdo. Si $dst \neq 0$ (es decir, si se da la condición de salto) entonces se deberá cancelar la expresión $next_pc - (pc + op1)$ para que se cumpla la restricción, realizando así el salto condicional a la dirección especificada por $pc + op1$. Si $dst = 0$ entonces se cancelan los 2 términos, y se obtiene $0 = 0$, es decir, no se refuerza ninguna condición. Si por otro lado se tiene $f_{PC_JNZ} = 0$, entonces la restricción original queda de la siguiente manera:

$$next_pc - ((1 - f_{PC_JUMP_ABS} - f_{PC_JUMP_REL}) \cdot (pc + instruction_size) + f_{PC_JUMP_ABS} \cdot res + f_{PC_JUMP_REL} \cdot (pc + res)) = 0$$

Si $f_{PC_JUMP_ABS} = 0$ y $f_{PC_JUMP_REL} = 0$ entonces se obtiene la restricción $next_pc - (pc + instruction_size) = 0$, es decir, una actualización normal del program counter, lo cual tiene sentido ya que no se realiza ningún tipo de salto. Si $f_{PC_JUMP_ABS} = 1$ entonces se obtiene la restricción $next_pc - res = 0$, es decir, un salto directo hacia la dirección de memoria que indica el resultado de la operación. Si $f_{PC_JUMP_REL} = 1$ entonces se obtiene la restricción $next_pc - (pc + res) = 0$, es decir, un salto hacia $pc + res$.

$$mul = op0 \cdot op1$$

$$(1 - f_{PC_JNZ}) \cdot res = f_{RES_ADD} \cdot (op0 + op1) + f_{RES_MUL} \cdot mul + (1 - f_{PC_JNZ} - f_{RES_ADD} - f_{RES_MUL}) \cdot op1$$

Nuevamente por razones de grado de restricciones se realiza un cambio de variable, creando así mul . Si $f_{PC_JNZ} = 0$ entonces se obtiene la siguiente restricción (notar que de los 3 flags utilizados en esta restricción, nuevamente solo uno de ellos podrá ser 1 en simultáneo):

$$res = f_{RES_ADD} \cdot (op0 + op1) + f_{RES_MUL} \cdot mul + (1 - f_{RES_ADD} - f_{RES_MUL}) \cdot op1$$

En este caso, si $f_{RES_ADD} = 0$ y $f_{RES_MUL} = 0$ entonces res guardará el valor de $op1$. Si $f_{RES_ADD} = 1$ se tiene la instrucción de suma, y con $f_{RES_MUL} = 1$ la de multiplicación. Si por otro lado $f_{PC_JNZ} = 1$ entonces se tiene la siguiente restricción:

$$0 = f_{RES_ADD} \cdot (op0 + op1) + f_{RES_MUL} \cdot mul - (f_{RES_ADD} + f_{RES_MUL}) \cdot op1$$

Esta se cumple únicamente si $f_{RES_ADD} = 0$ y $f_{RES_MUL} = 0$.

9.5. Restricciones básicas de memoria

En esta sección se explicarán algunas de las restricciones que se aplican sobre la memoria, principalmente las necesarias para exponer eficientemente como pública una sección de la memoria (generar un output). Para esto, primero se explicará una forma de probar que dos listas de valores tienen el mismo contenido en distinto orden.

Dadas dos listas que contienen valores del tipo (a_i, v_i) , donde a_i es una dirección de memoria y v_i es el valor asociado a esta, obteniendo un valor aleatorio α puede generarse el siguiente polinomio:

$$\Pi(z - (a_i + \alpha v_i))$$

Si el verificador elige un z aleatorio, entonces es poco probable que (dados los conceptos ya explicados en la sección de STARKS) una lista de valores distintos genere el mismo resultado. Podría agregarse a la traza una columna que almacene la acumulación del producto del polinomio definido, para agregar restricciones que permitan verificar que cada paso de la operación fue realizada correctamente, y que (si tenemos las listas de (a_i, v_i) y (a'_i, v'_i)) el resultado final sea el mismo para las dos listas. La columna guardaría valores del tipo

$$\Pi_{i=0}^j(z - (a_i + \alpha v_i))$$

siendo j el índice en la columna, por lo que se irá acumulando el producto (teniendo el valor de z ya definido previo a empezar a almacenar los resultados), llegando al resultado final en la última posición. Cairo realiza este chequeo de otra forma que será explicada a continuación.

Dadas las listas de acceso a memoria $L_1 = \{(a_i, v_i)\}_{i=0}^{n-1}$ y $L_2 = \{(a'_i, v'_i)\}_{i=0}^{n-1}$ (que terminarán siendo columnas de la traza) y la lista de valores $\{p_i\}_{i=0}^{n-1}$ tales que se cumplen las siguientes restricciones:

- Continuidad de L_2 :

$$(a'_{i+1} - a'_i)(a'_{i+1} - a'_i - 1) = 0, 0 \leq i < n - 1$$

Dada la dirección de memoria a'_i (la dirección en el índice i), la dirección a'_{i+1} (la dirección en el índice $i+1$) deberá contener la misma dirección de memoria que en i , o la siguiente. Esto asegura que las direcciones de memoria utilizadas en L_2 están ordenadas, y no hay saltos de direcciones en las que no se guarda ningún valor. Si a'_{i+1} y a'_i contienen el mismo valor (la misma dirección de memoria) entonces se cancelará el primer término, si en cambio son dos direcciones continuas tendrán una diferencia de 1 y se cancelará el segundo.

- Valor único de memoria de L_2 :

$$(v'_{i+1} - v'_i)(a'_{i+1} - a'_i - 1)$$

Si dos elementos contiguos de la lista almacenan un valor distinto, entonces el primer término no se cancelará y deberá hacerlo el segundo, por lo que se fuerza que para valores contiguos distintos las direcciones que apuntan a esos valores también deben ser contiguas (y por lo tanto no iguales entre sí). De esta forma se fuerza a que una misma dirección de memoria tendrá siempre el mismo valor una vez que se le asigna uno.

- Mismos valores con distinto orden: para demostrar que L_1 y L_2 guardan los mismos valores pero con distinto orden se deben cumplir 3 restricciones

$$(z - (a'_0 + \alpha v'_0)) \cdot p_0 = z - (a_0 + \alpha v_0)$$

$$p_{n-1} = 1$$

$$(z - (a'_i + \alpha v'_i)) \cdot p_i = (z - (a_i + \alpha v_i)) \cdot p_{i-1}, 1 \leq i < n$$

Si se cumple todo entonces L_1 , al igual que L_2 , será también continuo (es decir, aunque las direcciones de memoria en la lista no estén ordenadas como en L_2 , no habrá huecos de valores no asignados entre las direcciones almacenadas en la lista) y de valores inmutables, y L_1 contendrá los mismos valores que L_2 pero con distinto orden. La restricción que demuestra la propiedad del ordenamiento no se entiende con facilidad, por lo que se explicará a continuación.

Lo que se quiere demostrar al indicar que se tratan de los mismos valores en distinto orden es que se cumple la igualdad

$$\Pi_{i=0}^{n-1}(z - (a'_i + \alpha v'_i)) = \Pi_{i=0}^{n-1}(z - (a_i + \alpha v_i))$$

Se comienza probando por inducción que para todo $n', 0 \leq n' < n$ se cumple que

$$p_{n'} \cdot \Pi_{i=0}^{n'}(z - (a'_i + \alpha v'_i)) = \Pi_{i=0}^{n'}(z - (a_i + \alpha v_i))$$

Se comienza la prueba por inducción por el primer caso, $n' = 0$, para el cual si se limpia la expresión se obtiene la restricción de valor inicial

$$(z - (a'_i + \alpha v'_i)) \cdot p_i = (z - (a_i + \alpha v_i)) \cdot p_{i-1}, 1 \leq i < n$$

por lo que se cumple la propiedad para el caso base. Se debe probar ahora que si se cumple para $n' - 1$ se cumple también para n' . Dada la expresión

$$p_{n'-1} \cdot \Pi_{i=0}^{n'-1}(z - (a'_i + \alpha v'_i)) = \Pi_{i=0}^{n'-1}(z - (a_i + \alpha v_i))$$

si se multiplican ambos lados de la igualdad por $(z - (a_{n'} + \alpha v_{n'}))$ se obtiene la siguiente expresión

$$(z - (a_{n'} + \alpha v_{n'}))p_{n'-1} \cdot \Pi_{i=0}^{n'-1}(z - (a'_i + \alpha v'_i)) = (z - (a_{n'} + \alpha v_{n'}))\Pi_{i=0}^{n'-1}(z - (a_i + \alpha v_i))$$

mirando nuevamente las restricciones sobre el orden forzadas, puede reemplazarse $(z - (a_{n'} + \alpha v_{n'}))p_{n'-1}$ por $(z - (a'_{n'} + \alpha v'_{n'}))p_{n'}$, obteniendo así

$$(z - (a'_{n'} + \alpha v'_{n'}))p_{n'} \cdot \Pi_{i=0}^{n'-1}(z - (a'_i + \alpha v'_i)) = (z - (a_{n'} + \alpha v_{n'}))\Pi_{i=0}^{n'-1}(z - (a_i + \alpha v_i))$$

Si se incluyen los factores que están fuera del producto iterativo dentro de este, con $i = n'$ se obtiene la expresión

$$p_{n'} \cdot \Pi_{i=0}^{n'}(z - (a'_i + \alpha v'_i)) = \Pi_{i=0}^{n'}(z - (a_i + \alpha v_i))$$

Esta es la propiedad que se quiere probar que se cumple, por lo que de esta forma se probó que si se cumple para $n' - 1$ se cumple también para n' y, tomando en cuenta que se cumple también para $n' = 0$ se termina de probar por inducción que la propiedad se cumple para todo $0 \leq n$. Si, como se indica en la restricción, el último valor de la lista es 1 (recordar que $p_{n-1} = 1$ es forzado por las restricciones) entonces se obtiene

$$\Pi_{i=0}^{n-1}(z - (a'_i + \alpha v'_i)) = \Pi_{i=0}^{n-1}(z - (a_i + \alpha v_i))$$

que es lo que se buscaba desde el principio. Se debe tomar en cuenta que los valores p_i no son elegidos arbitrariamente, por lo que la propiedad $p_{n-1} = 1$ no es algo que puede decidirse, sino que se cumple únicamente si los valores comparados son los mismos pero en orden distinto.

9.6. Restricciones de memoria pública

Se indicó previamente que el output de la ejecución de una ZKVM se almacena en la memoria pública. Sin embargo, se debe corroborar que este output realmente pertenece a una sección de memoria generada por la ejecución del programa. Dados los valores de la memoria pública otorgados por el prover, podría agregarse una restricción de límite para cada dirección de memoria y cada valor asociado a cada una de estas, pero eso sería extremadamente ineficiente. Si se llama P^* a la lista de direcciones de memoria pública, y $m(x)$ una función de la memoria que retorna el valor almacenado en la memoria para la dirección x , entonces $\{(a, m(a))\}_{a \in P^*}$ es el set de dirección-valor para la memoria pública.

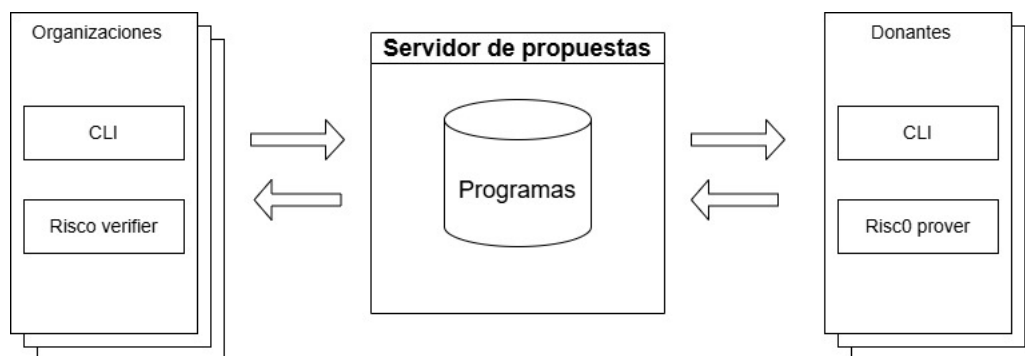
Dados nuevamente los mismos L_1 y L_2 mencionados previamente, si se fuerza a L_1 a que las celdas que guardarían los valores pertenecientes a la lista de valores de la memoria pública en realidad guarden $(0, 0)$, entonces para esos puntos el polinomio de la memoria se calculará como $z - (0 + \alpha 0)$, debido a esto, la restricción sobre p_{n-1} se debe cambiar por

$$p_{n-1} = \frac{z^{|P^*|}}{\prod_{a \in P^*} (z - (a + \alpha m(a)))}$$

ya que L_2 sí acumuló el producto de los valores asociados a la memoria pública en sus p_i mientras que L_1 no y por eso deben substraerse mediante división, y además al realizar el cálculo de L_1 se multiplicó por $z^{|P^*|}$ veces, lo cual no se hizo para L_2 y por eso se termina realizando en p_{n-1} . De esta forma, agregando restricciones de nulidad sobre las celdas de memoria pública de L_1 y cambiando la restricción sobre p_{n-1} se demuestra la validez de la memoria pública expuesta por la zkvm, en vez de realizando $2 \cdot |P^*|$ restricciones adicionales.

10. Proyecto realizado

Como se indicó en la introducción, se busca generar un sistema que haga uso de Zero Knowledge Proofs para donación de resultados asociados a pruebas que permiten verificar su integridad. Para esto se crearon tres programas de consola distintos, un servidor y dos clientes, programados en Rust.



El servidor se encarga de almacenar y distribuir las pruebas generadas por los clientes y los programas a ejecutar junto con los inputs con los que deben ser ejecutados. Los clientes son los que se encargan de conectarse al servidor, siendo uno utilizado por las organizaciones para subir los programas y descargar las pruebas, y otro por los donantes, quienes descargarán los archivos de los programas a los que quieren donar cómputo, y luego subirán las pruebas generadas.

10.1. Tecnologías utilizadas

10.1.1. Rust

Todo el código que genera los ejecutables del proyecto fue realizado en Rust. Esto se debe a que, aunque es un lenguaje relativamente nuevo, es altamente utilizado en el mundo de las ZKP. Las bibliotecas necesarias para utilizar Risc0 se encuentran implementadas en Rust.

10.1.2. Diesel

Para interactuar con la base de datos se utilizó la biblioteca de Rust llamada **Diesel**. Esta permitió definir un esquema fácilmente reproducible para la generación de la base de datos, aunque debido a ciertas limitaciones los índices deben ser agregados a mano en los archivos de creación de tablas generados por la biblioteca. Sin embargo, esto no es un motivo de preocupación para el usuario de la aplicación, ya que es un proceso que ya fue realizado, y deberá ser tomado en cuenta únicamente si se altera el esquema de las tablas.

10.1.3. Docker

Para facilitar el uso de la aplicación para los usuarios se utiliza docker. Al tener que instalar programas de terceros para utilizar correctamente Risc0, resulta de utilidad dockerizar un ambiente que lo instale. De esta forma se facilita la experiencia de usuario, ya que deberán instalar únicamente docker en vez de Rust y otros programas externos.

En adición a esto se debe tomar en cuenta que, como se dijo previamente, se realiza un hashing del código a ejecutar. Esta operación se realiza sobre el bytecode luego de realizar una compilación de un programa, por lo que variaciones en las bibliotecas utilizadas y hardware generarán un resultado distinto una vez terminada la operación. Esto causaría que una verificación de una prueba generada en un ambiente distinto a aquel en el que fue generada falle por la falta de capacidad para corroborar que se trata del mismo programa.

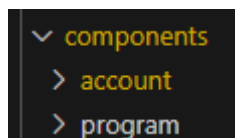
La dockerización permite generar un ambiente estandarizado de forma tal que el bytecode generado para cada programa es independiente de la computadora que lo escriba y, por ende, también lo es el hash.

10.1.4. Makefile

El uso de makefile permite facilitar aún más la experiencia de usuario. Permite juntar la ejecución de varios comandos con los que una persona no muy conocedora de la informática debería ejecutar, y con los que podría tener problemas, para pasar a poder utilizar un único comando mucho más simple para cada una de las acciones que se quiera realizar sobre el programa a ejecutar.

10.2. Servidor

Se trata del programa que se encargará de manejar la subida y descarga de código, inputs y pruebas. Consiste en un servidor HTTP implementado con el lenguaje de programación Rust. El código se encuentra en la carpeta **program_distributor**. El servidor cuenta con dos componentes principales para sus endpoints, **account** y **program**



Los endpoints pertenecientes a **account** son aquellos que se encargan de manejar las cuentas registradas en el server. Se encarga del registro, login y generación de tokens de validación para que las organizaciones puedan interactuar con él. Los endpoints pertenecientes a **program** son aquellos que se encargan de manejar la subida y descarga de programas, inputs de programas y pruebas de integridad de cómputo.

10.2.1. Modelos de base de datos

El servidor utiliza una base de datos para almacenar ciertos componentes que utiliza para su correcto funcionamiento, las tablas generadas son las siguientes:

- **account**: se encarga de almacenar las cuentas de las organizaciones junto con datos adicionales sobre estas. Las columnas que la componen son las siguientes:
 - organization_id, String: identificador aleatorio
 - name, String: nombre de la organización, decidido al momento de registrarse
 - description, String: explicación sobre los objetivos de la organización
 - username, String: usuario para login
 - password_hash, String: contraseña post-hashing
- **refresh_token**: se encarga de manejar los tokens con vencimiento que utilizan las organizaciones para validarse con el servidor luego de hacer un login. Las columnas que la componen son las siguientes:
 - token_id, String: identificador aleatorio
 - user_id, String: id de la organización representada por el token
- **program**: se encarga de almacenar datos descriptivos de los programas subidos por las organizaciones, no almacena los programas en sí. Las columnas que la componen son las siguientes:
 - organization_id, String: id de la organización que subió el programa
 - program_id, String: identificador aleatorio del programa
 - name, String: nombre del programa o proyecto
 - description, String: explicación sobre objetivos del proyecto o cualquier otra información adicional
 - input_lock_timeout, i64: cantidad de segundos por los que se reserva un grupo de inputs de un programa. Luego de reservar un input_group, este podrá ser reservado nuevamente una vez pasada esta cantidad de segundos.
- **program_input_group**: define un grupo de inputs para un programa específico. Las columnas que la componen son las siguientes:

- `input_group_id`, String: identificador aleatorio del grupo de inputs
 - `program_id`, String: id del programa al que esta asociado este grupo de inputs
 - `name`, String: nombre definido para el grupo de inputs
 - `last_reserved`, Nullable DateTime: timestamp del momento en el que este grupo de inputs fue reservado
 - `proven_datetime`, Nullable DateTime: timestamp del momento en el que se subió la prueba de integridad de cómputo
- **specific_program_input**: almacena inputs específicos relacionados a un grupo de inputs, varios inputs específicos pueden apuntar a un mismo grupo. Las columnas que la componen son las siguientes:
- `specific_input_id`, String: identificador aleatorio
 - `input_group_id`, String: id del grupo de inputs al que pertenece este input
 - `blob_data`, Vec<u8>: bytes que almacena este input específico. Un grupo de inputs está compuesto por varias entradas de 1024 bytes que deben usar las organizaciones para definir un protocolo de serialización de inputs
 - `order`, i32: orden en el que se deberá introducir como input del programa los bytes de esta entrada de la tabla

10.2.2. Almacenamiento en AWS

Podrá notarse que en la sección anterior no se presenta una tabla para guardar los programas ni las pruebas que maneja el servidor. Esto se debe a que estos archivos no son guardados en la base de datos local que acompaña al servidor, sino en el servicio S3 de Amazon Web Services. Se definió una interfaz en `program_distributor/src/services/file_storage` para que quien lo desee pueda seguirla y programar un handler distinto al definido en `program_distributor/src/services/aws_s3_handler` y así utilizar otro servicio o sistema para manejar la subida y descarga de archivos.

10.2.3. Flujo general de uso

El flujo general de uso del servidor es el siguiente:

1. Una organización sube un programa. Al hacerlo, este se registra en la base de datos y se envía también a aws para que los archivos sean almacenados.

89fd2d9f-fcb9-444d-a6fa-f89186d8f438/

Copy S3 URI

Objects

Properties

Objects (1) Info

Refresh

Copy S3 URI

Copy URL

Download

Open

Delete

Actions


Create folder

Upload























Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Find objects by prefix

< 1 > ⚙

	Name	Type	Last modified	Size	Storage class
	 program.tar	tar	November 3, 2024, 12:25:38 (UTC-03:00)	30.5 KB	Standard

2. La organización otorga una serie de inputs para el programa subido previamente. Estos serán registrados en una tabla como un grupo, y luego en otra se guardarán individualmente, ya que un grupo de inputs puede estar compuesto por múltiples inputs de 1024 bytes. Un ejemplo de un programa con varios grupos de inputs sería el siguiente:

		input_group_id	program_id	name	last_reserved	proven_datetime
	 Edit  Copy  Delete	40a736db-e1ee-4638-bcda-e6bfea94c615	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	charmichael_561_miller_rabin_big_endian	NULL	NULL
	 Edit  Copy  Delete	83b3b119-4141-4f9d-a4e7-2e9b5a80c5aa	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	7841_miller_rabin_big_endian	NULL	NULL
	 Edit  Copy  Delete	b4c7b5a4-14af-47b2-a59f-ba9451ac3bbf	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	2371_miller_rabin_big_endian	NULL	NULL
	 Edit  Copy  Delete	be9658ff-8864-4288-bb6a-7af6f1ee8a47	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	53_miller_rabin_big_endian	NULL	NULL
	 Edit  Copy  Delete	cee5f087-e65e-4e91-84a8-6f1844f110c6	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	2373_miller_rabin_big_endian	NULL	NULL

En este caso los grupos contienen un único elemento, pero si se tratara de grupos de más de uno entonces se almacenarían adicionalmente tuplas con valores mayores a 0, los inputs específicos guardados son los siguientes:

[illegible]

3. Donantes comienzan a reservar grupos de inputs para ejecutar los programas. Un input reservado se vería de la siguiente manera en la base de datos:

	input_group_id	program_id	name	last_reserved	proven_datetime
<input type="checkbox"/> Edit Copy Delete	40a736db-e1ee-4638-bcda-e6bfea94c615	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	charmichael_561_miller_rabin_big_endian	2024-11-05 00:49:20	NULL
<input type="checkbox"/> Edit Copy Delete	83b3b119-4141-4f9d-a4e7-2e9b5a80c5aa	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	7841_miller_rabin_big_endian	NULL	NULL
<input type="checkbox"/> Edit Copy Delete	b4c7b5a4-14af-47b2-a59f-ba9451ac3bbf	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	2371_miller_rabin_big_endian	NULL	NULL
<input type="checkbox"/> Edit Copy Delete	be9658ff-8864-4288-bb6a-7af6f1ee8a47	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	53_miller_rabin_big_endian	NULL	NULL
<input type="checkbox"/> Edit Copy Delete	cee5f087-e65e-4e91-84a8-6f1844f110c6	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	2373_miller_rabin_big_endian	NULL	NULL

Una vez probados todos los inputs del programa, la tabla se vería entonces así:

	input_group_id	program_id	name	last_reserved	proven_datetime
<input type="checkbox"/> Edit Copy Delete	cee5f087-e65e-4e91-84a8-6f1844f110c6	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	2373_miller_rabin_big_endian	2024-11-05 00:52:32	2024-11-05 00:53:15
<input type="checkbox"/> Edit Copy Delete	be9658ff-8864-4288-bb6a-7af6f1ee8a47	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	53_miller_rabin_big_endian	2024-11-05 00:52:07	2024-11-05 00:52:32
<input type="checkbox"/> Edit Copy Delete	b4c7b5a4-14af-47b2-a59f-ba9451ac3bbf	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	2371_miller_rabin_big_endian	2024-11-05 00:51:43	2024-11-05 00:52:07
<input type="checkbox"/> Edit Copy Delete	83b3b119-4141-4f9d-a4e7-2e9b5a80c5aa	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	7841_miller_rabin_big_endian	2024-11-05 00:51:19	2024-11-05 00:51:43
<input type="checkbox"/> Edit Copy Delete	40a736db-e1ee-4638-bcda-e6bfea94c615	89fd2d9f-fcb9-444d-a6fa-f89186d8f438	charmichael_561_miller_rabin_big_endian	2024-11-05 00:49:20	2024-11-05 00:51:19

Y el almacenamiento en aws se vería de la siguiente manera para el programa cuyos inputs fueron probados:

Objects (6) Info

Copy S3 URI

Copy URL

Download

Open

Delete

Actions

Create folder

Upload

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Find objects by prefix

1

<input type="checkbox"/>	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	47b2-a59f-ba9451ac3bbf.bin	bin	November 4, 2024, 21:52:10 (UTC-03:00)	233.2 KB	Standard
<input type="checkbox"/>	<div><div></div><div>be9658ff-8864-4288-bb6a-7af6f1ee8a47.bin</div></div>	bin	November 4, 2024, 21:52:35 (UTC-03:00)	233.2 KB	Standard
<input type="checkbox"/>	<div><div></div><div>cee5f087-e65e-4e91-84a8-6f1844f110c6.bin</div></div>	bin	November 4, 2024, 21:53:18 (UTC-03:00)	244.6 KB	Standard
<input type="checkbox"/>	<div><div></div><div>program.tar</div></div>	tar	November 3, 2024, 12:25:38 (UTC-03:00)	30.5 KB	Standard

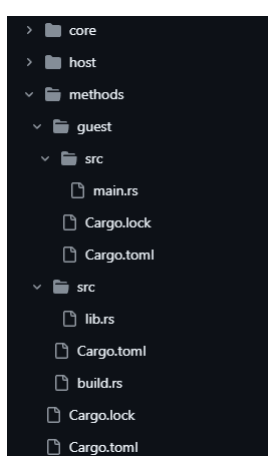
Una vez hecho esto, las organizaciones podrán contactar nuevamente el servidor para verificar las pruebas almacenadas, borrando así las entradas de la tabla y los archivos de las pruebas de AWS.

10.3. Organizaciones

Se trata del programa a usar por quienes desean solicitar que otras personas donen cómputo verificable para el cálculo de su programa. Para esto deberán generar un proyecto de Risc0, programar aquello que quieren que sea ejecutado por terceros, y luego subir el contenido de la carpeta **methods** al servidor, subiendo así un programa a este, junto con una serie de inputs que se utilizarán al distribuirse ejecuciones del mismo. El código para este cliente se encuentra en la carpeta **uploader_client**.

10.3.1. Código a desarrollar

Al tener el cliente que subir la carpeta **methods** de un proyecto de risc zero, se deberá tomar en cuenta que estos tienen la siguiente estructura



donde **core** contiene código compartido por **host** y **guest**, **host** es donde se genera la prueba de ejecución, y **guest** es donde se encuentra el código a probar. Al subir únicamente la carpeta **methods**, para utilizar **distcom** no resulta de utilidad la carpeta **core**, y la carpeta **host** es definida por el cliente en sí, por lo que, al ser lo único que subirá al servidor, a las organizaciones solo les interesará programar la carpeta **methods**, tomando en cuenta que en el archivo **guest/cargo.toml** se deberá establecer como nombre de la librería el string **"downloaded_guest"**. El archivo **guest/src/main.rs** deberá tener el siguiente formato:

```

1
2 #![no_main]
3 // If you want to try std support, also update the guest Cargo.toml file
4 #![no_std] // std support is experimental
5
6 use risc0_zkvm::guest::env;
7 risc0_zkvm::guest::entry!(main);
8
9 // use basic_prime_test_core;
10
11 // Your code inside main
12 fn main() {
13     // Read your inputs:
14     // let input: Vec<u8> = env::read();
15
16
17     // Validate necessary_properties:
18     // assert!(number_to_test % 2 != 0);
19
20     // Commit to outputs:
21     // env::commit(&outputs);
22 }

```

como indica el texto, el código a probar deberá ser llamado en la función `main`. El tipo de variable sobre el que se deberá hacer commit en el output es `string`, se recomienda retornar un output en forma de json tras conversión a texto. Al tener que definir los inputs de la ejecución como una serie de arrays de 1024 bytes, el código cada vez que obtenga inputs de `env` tendrá que hacerlo en forma de `Vec<u8>`, tantas veces como sea necesario.

10.3.2. Programa ejecutado

Para comenzar a utilizar el programa de consola que deberán utilizar las organizaciones, se deberá ejecutar en la consola el comando **make** dentro de la carpeta `uploader_client` del repositorio. Allí el usuario podrá elegir entre registrarse o hacer login, implementado para prevenir la posibilidad de que alguien suba código impersonando una organización específica. Una vez pasada esta validación, se verá la siguiente pantalla:

```
Please execute a command:
|
```

En caso de no saber qué comandos pueden ser ejecutados, se puede ejecutar el comando **help**:

```
Please execute a command:
help
Usage: <COMMAND>

Commands:
  upload      Upload of the methods folder of the code the user wants to upload
  template    Download of an example of the methods folder for implementation reference
  my-programs Display of the information of the user's uploaded programs, moves the execution to another commands set
  proven-programs Display of the user's programs that have at least one input group with a proven execution, moves the execution to another commands set
  logout      Log out of the program and exit
  exit        Exit the program
  help        Print this message or the help of the given subcommand(s)

Options:
  -h, --help    Print help
  -V, --version Print version
```

Este comando mostrará todos los comandos que pueden ser ejecutados en el nivel actual de la aplicación, y se encuentra disponible no importa en qué nivel se encuentre el usuario. Para cambiar de nivel se debe ejecutar algún comando que requiera más pasos, lo cual es indicado en la descripción de cada uno. En caso de querer más información sobre un comando específico, se puede ejecutar **help <comando>**, como se ve en la siguiente imagen para el caso **upload**:

```
Please execute a command:
help upload
Upload of the methods folder of the code the user wants to upload

Usage: upload --path <FOLDER_NAME> --name <NAME> --description <DESCRIPTION> --timeout <EXECUTION_TIMEOUT>

Options:
  -p, --path <FOLDER_NAME>    Name of the methods folder in the uploads directory
  -n, --name <NAME>            Name for the program what will be displayed for the provers
  -d, --description <DESCRIPTION> Explanation of the objectives of the program or any additional information considered necessary
  -t, --timeout <EXECUTION_TIMEOUT> How many seconds an input group of this program will be blocked before another prover can request it for execution
  -h, --help                  Print help
```

Una nueva organización comenzará subiendo un programa, ejecutando por ejemplo el comando `upload -path miller_rabin_methods -name "Miller Rabin test" -description "My`

description for the test upload” --timeout 3600 para subir la carpeta methods que contiene el código de la prueba de primalidad Miller rabin. Hecho esto, al ejecutar el comando **my-programs** se verá lo siguiente

```
Please execute a command:
upload --path miller_rabin_methods --name "Miller Rabin test" --description "My description for the test upload" --timeout 3600

Please execute a command:
my-programs

Program 0:
  organization_id: 2b1989c1-363b-422b-beb1-97f0e9c3181b
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  name: Miller Rabin test
  description: My description for the test upload
  input_lock_timeout: 3600
```

Al ejecutarlo, cambiamos de nivel en la aplicación (podrán verse los nuevos comandos disponibles con **help**), y se podrá ahora subir al servidor una serie de inputs con los que se verá que los donantes ejecuten el código. Para esto se ejecutará el comando **post-input 0 miller_rabin_inputs**

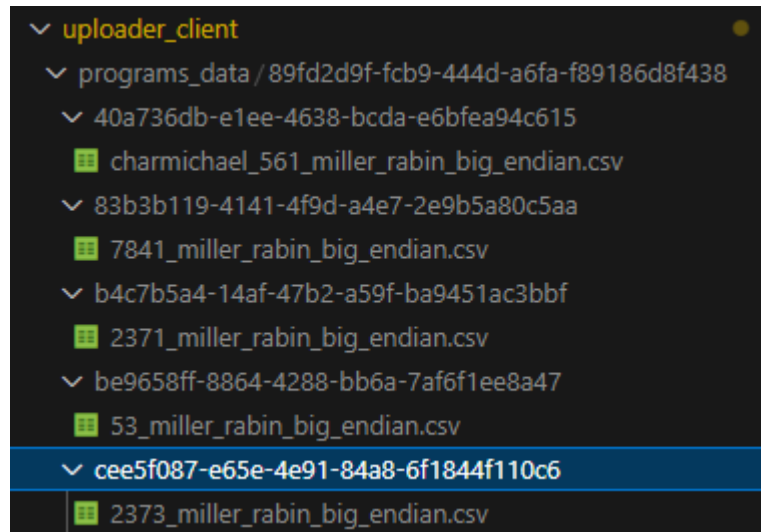
```
Please execute a command:
post-input 0 miller_rabin_inputs
Uploaded input group with path: ./uploads/miller_rabin_inputs/2371_miller_rabin_big_endian.csv
Uploaded input group with path: ./uploads/miller_rabin_inputs/2373_miller_rabin_big_endian.csv
Uploaded input group with path: ./uploads/miller_rabin_inputs/53_miller_rabin_big_endian.csv
Uploaded input group with path: ./uploads/miller_rabin_inputs/7841_miller_rabin_big_endian.csv
Uploaded input group with path: ./uploads/miller_rabin_inputs/charmichael_561_miller_rabin_big_endian.csv
```

Estos inputs pueden verse ejecutando el comando **get-inputs 0**.

```
get-inputs 0

Program input group 0:
  input_group_id: 40a736db-e1ee-4638-bcda-e6bfea94c615
  name: charmichael_561_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
Program input group 1:
  input_group_id: 83b3b119-4141-4f9d-a4e7-2e9b5a80c5aa
  name: 7841_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
Program input group 2:
  input_group_id: b4c7b5a4-14af-47b2-a59f-ba9451ac3bbf
  name: 2371_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
Program input group 3:
  input_group_id: be9658ff-8864-4288-bb6a-7af6f1ee8a47
  name: 53_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
Program input group 4:
  input_group_id: cee5f087-e65e-4e91-84a8-6f1844f110c6
  name: 2373_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
```

Se puede ver también que se generaron carpetas correspondientes a cada grupo de inputs subido al servidor, conteniendo cada una copias del archivo utilizado para la subida.



Una vez hecho esto, queda solo esperar a que donantes soliciten el código e inputs para ejecutar. Puede verse que actualmente no hay ejecuciones probadas en el servidor si se utiliza el comando **back** hasta llegar al primer nivel y se ejecuta el comando **proven-programs**:

```
Please execute a command:
proven-programs

No programs remaining
```

Subidas pruebas al servidor por parte de donantes, el comando mostrará la siguiente pantalla

```
Please execute a command:
proven-programs

Program 0:
  organization_id: 2b1989c1-363b-422b-beb1-97f0e9c3181b
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  name: Miller Rabin test
  description: My description for the test upload
  input_lock_timeout: 3600
```

Para verificar las pruebas de ejecución de los inputs del programa, se deberá elegir este para ir a la pantalla de verificación, ejecutando así el comando **verify 0**

```

Please execute a command:
verify 0

Program input group 0:
  input_group_id: 40a736db-e1ee-4638-bcda-e6bfea94c615
  name: charmichael_561_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  last_reserved: 2024-11-05 00:49:20
  proven_datetime: 2024-11-05 00:51:19
Program input group 1:
  input_group_id: 83b3b119-4141-4f9d-a4e7-2e9b5a80c5aa
  name: 7841_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  last_reserved: 2024-11-05 00:51:19
  proven_datetime: 2024-11-05 00:51:43
Program input group 2:
  input_group_id: b4c7b5a4-14af-47b2-a59f-ba9451ac3bbf
  name: 2371_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  last_reserved: 2024-11-05 00:51:43
  proven_datetime: 2024-11-05 00:52:07
Program input group 3:
  input_group_id: be9658ff-8864-4288-bb6a-7af6f1ee8a47
  name: 53_miller_rabin_big_endian
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  last_reserved: 2024-11-05 00:52:07
  proven_datetime: 2024-11-05 00:52:32
Program input group 4:
  input_group_id: cee5f087-e65e-4e91-84a8-6f1844f110c6

```

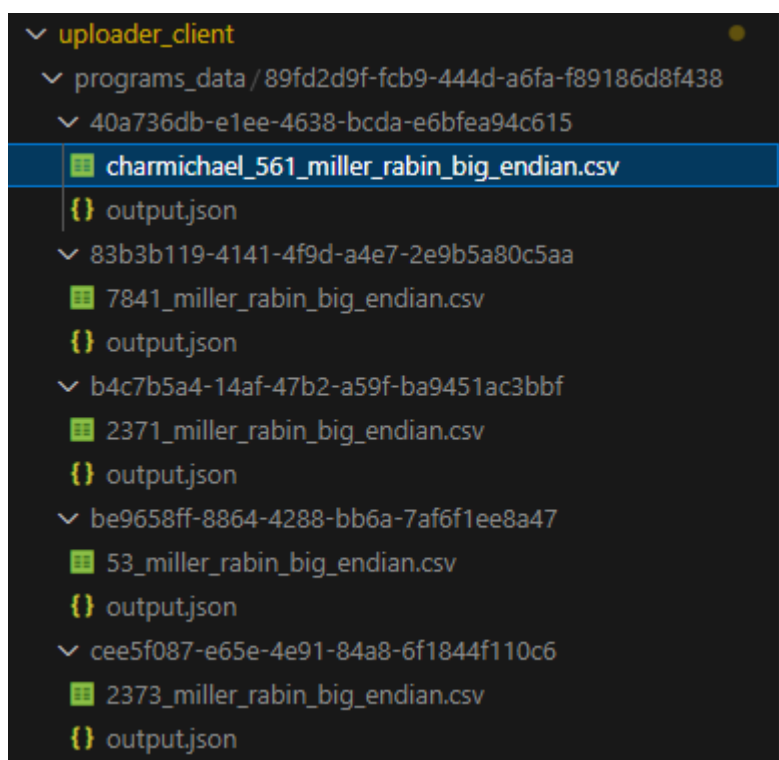
Finalmente, para verificar las pruebas subidas y obtener los resultados de las ejecuciones a las que están asociadas, se ejecuta el comando **verify-all**

```

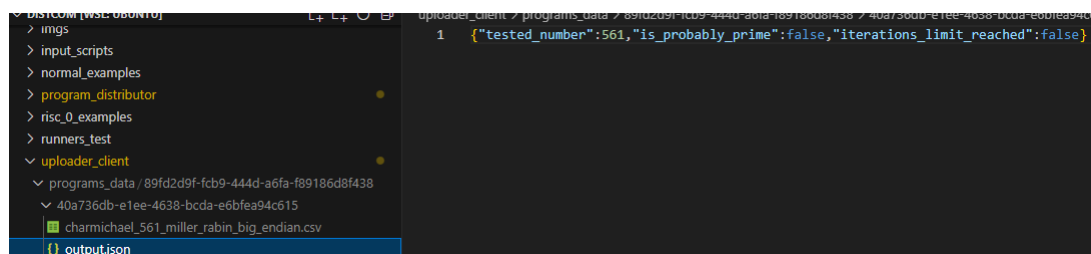
Please execute a command:
verify-all
Starting proof verification of program with id "89fd2d9f-fcb9-444d-a6fa-f89186d8f438" with input group with id "40a736db-e1ee-4638-bcda-e6bfea94c615"

```

A medida que se finalizan las verificaciones, se guardarán los resultados en la carpeta **programs_data** en el path correspondiente, obteniendo la siguiente configuración de archivos una vez verificada la última prueba:



De esta forma termina el flujo de ejecución de obtención de los resultados del cómputo donado, permitiendo a las organizaciones hacer uso de los resultados procesando los archivos de tipo json, como el siguiente:



10.4. Verificación de pruebas

La sección de código más relevante para la verificación de pruebas es el siguiente extracto del archivo `src/runner/verifier/src/main.rs`:

```

1
2 let receipt: risc0_zkvm::Receipt = bincode::deserialize(&proof_data).expect("Error
  in proof deserialization");
3 receipt
4   .verify(DOWNLOADED_GUEST_ID)
5   .expect("Proof verification failed");
6
7 let output: String = receipt.journal.decode().unwrap();

```

En este se deserializa el archivo de la prueba descargado del servidor para construir el struct de `risczero` de una prueba generada. Se utiliza la nueva variable para llamar al método de verificación de `risc0`, que recibe el hash del código ejecutado, el cual es obtenido de la compilación del código a verificar descargado. Una vez realizada la verificación correctamente, se obtiene el output de

la ejecución (aquello sobre lo que se realizó commit en el archivo main de la carpeta guest), que deberá ser un string, y luego se guardará el resultado en un archivo.

10.5. Donantes

Es el programa utilizado por las personas que tienen poder de cómputo no utilizado y desean donarlo para una causa. Determinarán la cantidad de núcleos de CPU que podrá usar el programa como límite, y podrán continuar con sus tareas usuales sin disturbios. El código para este cliente se encuentra en la carpeta **executor_client**.

10.5.1. Programa ejecutado

Para comenzar a utilizar el programa de consola que deberán utilizar las organizaciones, se deberá ejecutar en la consola el comando **make** dentro de la carpeta **executor_client** del repositorio. En caso de no saber qué comandos pueden ser ejecutados, se puede ejecutar el comando **help**:

```
Please execute a command:
help
Usage: <COMMAND>

Commands:
  organizations  Displays a list with the information of the organizations stored in the program distributor, moves the
                  execution to another commands set
  all-programs   Displays a list with the information of the programs stored in the program distributor without taking t
                  he uploader into account, moves the execution to another commands set
  exit           Exits the program
  help           Print this message or the help of the given subcommand(s)

Options:
  -h, --help      Print help
  -V, --version   Print version
```

El flujo de uso de la aplicación comienza generalmente eligiendo una organización que le interese al usuario para poder así ver los proyectos que ofrece. Para esto se ejecuta el comando **organizations**

```
Please execute a command:
organizations

Organization 0:
  organization_id: 2b1989c1-363b-422b-beb1-97f0e9c3181b
  name: Mi organizacion
  description: Una organizacion re fachera
```

El usuario deberá ahora elegir una organización que sea de su interés, para esto podría ejecutar el comando **choose 0**

```
Please execute a command:
choose 0

Program 0:
  organization_id: 2b1989c1-363b-422b-beb1-97f0e9c3181b
  program_id: 89fd2d9f-fcb9-444d-a6fa-f89186d8f438
  name: Miller Rabin test
  description: My description for the test upload
  input_lock_timeout: 3600
```

Hecho esto, se elegirá uno de los programas provistos por la organización, ejecutando en este caso el comando **run-all** para generar pruebas para todos los inputs disponibles de todos los programas subidos por la organización elegida

```
Please execute a command:
run-all
Running program "Miller Rabin test" from organization with id "2b1989c1-363b-422b-beb1-97f0e9c3181b"
downloaded_guest: Starting build for riscv32im-risc0-zkvm-elf
downloaded_guest: Downloading crates ...
downloaded_guest: Downloaded getrandom v0.2.12
downloaded_guest: Downloaded autocfg v1.1.0
downloaded_guest: Downloaded num-bigint v0.4.4
downloaded_guest: Downloaded either v1.10.0
downloaded_guest: Downloaded downcast-rs v1.2.0
downloaded_guest: Downloaded semver v1.0.22
downloaded_guest: Downloaded serde_derive v1.0.197
downloaded_guest: Downloaded serde v1.0.197
```

Este comando ejecutará un programa con todos los inputs que pueda reservar, y al terminar con estos pasará al siguiente programa para repetir el proceso. Debido a esto la primera ejecución de un programa tardará más que el resto, ya que deberá además compilarse el código descargado

```
downloaded_guest: warning: 'downloaded_guest' (bin "downloaded_guest") generated 1 warning
downloaded_guest: Finished 'release' profile [optimized] target(s) in 57.49s

Proof generated successfully.
Proof was uploaded, total seconds passed: 116
```

Una vez se pasa a las siguientes ejecuciones del mismo programa, se tardará considerablemente menos en generar la prueba

```
Running program "Miller Rabin test" from organization with id "2b1989c1-363b-422b-beb1-97f0e9c3181b"
downloaded_guest: Starting build for riscv32im-risc0-zkvm-elf
downloaded_guest: Compiling downloaded_guest v0.1.0 (/app/src/runner/methods/guest)
downloaded_guest: warning: unused variable: 'cast_base'
downloaded_guest: --> src/main.rs:28:9
downloaded_guest: |
downloaded_guest: 28 | let cast_base = base as u64;
downloaded_guest: |      ^^^^^^^^^ help: if this is intentional, prefix it with an underscore: '_cast_base'
downloaded_guest: |
downloaded_guest: = note: '#[warn(unused_variables)]' on by default
downloaded_guest: warning: 'downloaded_guest' (bin "downloaded_guest") generated 1 warning
downloaded_guest: Finished 'release' profile [optimized] target(s) in 0.42s

Proof generated successfully.
Proof was uploaded, total seconds passed: 21
```

Para cada una de las ejecuciones realizadas, se subirá un archivo al servidor conteniendo la prueba serializada, que será luego utilizado por la organización como se indicó en la sección anterior.

10.5.2. Generación de las pruebas

La sección de código más relevante para la generación de pruebas es el siguiente extracto del archivo `src/runner/prover/src/main.rs`:

```
1
2 let file = File::open(program_input_path).expect("Error while reading file");
3 let mut input_reader = csv::ReaderBuilder::new().has_headers(false).from_reader(
  (file);
```

```
4
5   for line in input_reader.records() {
6       let line_ok = line.expect("Error in line reading");
7       let line_iterator = line_ok.into_iter();
8       let mut counter = 0;
9
10      for value in line_iterator {
11          let bytes_vector = base64::decode(value).expect("Failed to decode
base64");
12          env_bulder_ref = env_bulder_ref.write(&bytes_vector).unwrap();
13          counter += 1;
14      }
15      assert!(counter == 1, "There is more than one element per line");
16  }
17
18  let executor_env = env_bulder_ref.build().unwrap();
19
20  let prover = default_prover();
21  let prove_info = prover
22      .prove(executor_env, DOWNLOADED_GUEST_ELF)
23      .unwrap();
24
25  let receipt = &prove_info.receipt;
26  let serialized_proof = bincode::serialize(&receipt).expect("Error in proof
serialization");
```

Este código recibe la serie de inputs que deberá utilizar para ejecutar el código y los introduce en el constructor de la prueba. Una vez hecho esto comienza a ejecutar el código descargado y a generar la prueba, para luego (en una sección de código que no se encuentra en lo mostrado) poder almacenarla en un archivo y enviarla al servidor.

11. Ejemplos implementados

Parte del proyecto implementado consistió también en implementar ejemplos de programas que podrían ser útiles para subir al servidor. Se decidió desarrollar una serie de pequeños tests de primalidad, los cuales pueden ser de utilidad ya que estos pueden tardar una gran cantidad de tiempo en ser corroborados normalmente, y utilizando STARKS podría verificarse que el número probado es primo en vez de simplemente creerle a una entidad o volver a ejecutar el test, lo cual puede tardar una gran cantidad de tiempo.

11.1. Prueba básica

La forma más fácil de probar definitivamente que un número realmente es es primo consiste en dividirlo por todos los números impares que lo preceden (podría dividirse por los primos conocidos, que además siempre eventualmente se terminará, pero por simplicidad se omitió la creación de la lista). El código de este programa se encuentra en `risc_0_examples/refactored_basic_prime_test/methods/guest/src/main` y es el siguiente:

```

1  let input: Vec<u8> = env::read();
2
3
4  let first_four_bytes = &input[0..4];
5  let number_to_test = u32::from_be_bytes(first_four_bytes.try_into().expect("
Error transforming into number from bytes"));
6
7  assert!(number_to_test % 2 != 0);
8  let mut divisor = 3;
9  let mut may_be_prime = true;
10
11 while (may_be_prime && (divisor < number_to_test)) {
12     may_be_prime = number_to_test % divisor != 0;
13     divisor += 2;
14 }
15
16 let outputs: Outputs = Outputs {
17     tested_number: number_to_test,
18     is_prime: may_be_prime && (divisor == number_to_test),
19 };
20
21 let serialized_outputs = to_string(&outputs).expect("Error in struct
serialization");
22 env::commit(&serialized_outputs);

```

Este programa recibe como input un único array de 1024 bytes, del cual toma únicamente los primeros 4 bytes encodados en big endian para obtener el número sobre el cual se aplica el test de primalidad. En el output se retorna el número que se utilizó para el test, y si es primo o no.

11.2. Prueba probabilística de fermat

La prueba de primalidad de fermat para el número b consiste en buscar un valor a tal que la ecuación $a^{b-1}b \equiv 1 \pmod{b}$ se cumpla. Si esta no se cumple para algún a , entonces se sabe con completa certeza que b no es primo. Si por otro lado se cumple, entonces pueden darse uno de 2 casos:

- No existe algún a tal que no se cumple la igualdad: b puede o no ser primo, es decir, hay casos en los que aunque b no sea primo no hay a tal que la cuenta no genere como resultado 1.
- Existe algún a tal que no se cumple la igualdad: si b no es primo y no pertenece al caso anterior, entonces la mayor cantidad de valores de a que podrían resultar en que la cuenta genere un 1 es la mitad.

La prueba implementada asume que se trata del segundo caso. Esto quiere decir que por cada valor de a que resulte en un 1, la probabilidad de que el número no sea primo se divide por 2 (comenzando en un 50 % si se prueba con un único valor de a). De esta forma, al probar con 20 números aleatorios, si siempre la operación resulta en 1 entonces la probabilidad de que b no sea primo es de 2^{-20} , la cual a fines prácticos puede considerarse 0.

El código para esta prueba puede encontrarse en `risc_0/examples/modular_probabilistic_fermat_test/methods/guest/src/main` y su parte más relevante es la siguiente:

```

1  let input: Vec<u8> = env::read();
2
3
4  let first_four_bytes = &input[0..4];
5  let number_to_test = u32::from_be_bytes(first_four_bytes.try_into().expect("
6  Error transforming into number from bytes"));
7  let mut may_be_prime = true;
8  let mut iteration_counter: u32 = 0;
9
10 while may_be_prime && iteration_counter < 20 { // At 20 iterations, if not all
11 tests are fools, then the chance of not being prime is really low
12     let current_byte_index = (4*(iteration_counter + 1)) as usize;
13     let last_number_byte_index_bound = (current_byte_index + 4) as usize;
14     let currently_analyzed_bytes: &[u8] = &input[current_byte_index..
15 last_number_byte_index_bound];
16     let random_input = u32::from_be_bytes(currently_analyzed_bytes.try_into().
17 expect("Error transforming into number from bytes"));
18     let divisor = modular_exponentiation(random_input, number_to_test - 1,
19 number_to_test);
20     may_be_prime = divisor % number_to_test == 1;
21     iteration_counter += 1;
22 }
23
24 let outputs: Outputs = Outputs {
25     tested_number: number_to_test,
26     is_probably_prime: may_be_prime,
27 };
28 let serialized_outputs = to_string(&outputs).expect("Error in struct
29 serialization");
30 env::commit(&serialized_outputs);

```

Nuevamente el input consiste en un único array de 1024 bytes, con la diferencia de que en este caso en adición al número sobre el que se realiza el test de primalidad, se guarda en este también un máximo de 20 números adicionales, que son los valores de a que se utilizarán para el test de primalidad. Esto se debe a que `risc_0` no permite el uso de funciones de generación de números aleatorios, por lo que deben ser obtenidos de forma externa.

Para realizar el test se itera entonces por la lista de valores aleatorios provistos en el input. Si alguno genera un resultado distinto de 1 entonces inmediatamente se termina la ejecución y se indica que el valor no es primo, sino se termina de iterar y se indica que sí lo es.

11.3. Miller-Rabin

El test de primalidad denominado Miller-Rabin consiste en la repetición de un algoritmo que puede o no converger, indicando nuevamente que un número no es primo, o que podría ser primo.

Dado un número n sobre el cual se quiere realizar la prueba, se define el valor $n - 1 = 2^k \cdot m$, es decir, se busca el valor m que al multiplicarlo por la potencia de 2 más grande que no supera a $n - 1$ resulta en $n - 1$ (recordar que, al ser n un candidato a primo, $n - 1$ será par y por lo tanto divisible por 2 al menos una vez). Se procede luego a elegir un valor arbitrario a tal que $1 < a < n - 1$. Una vez hecho esto, se obtiene $b_0 \equiv a^m \pmod{n}$, y se define $b_i \equiv b_{i-1}^2 \pmod{n}$. Se calcula b_0 y luego tantos b_i como sea necesario hasta obtener como resultado de la operación un 1 o un -1. En caso de obtener un 1, entonces se concluye que n definitivamente no es un número primo. Por otro lado, si el resultado es -1, entonces n es probablemente primo. También podría

darse el caso de una ausencia de convergencia en estos valores, caso en el que se concluye que n no es primo, pero se trata de una propiedad difícil de probar.

El código para esta prueba puede encontrarse en `risc_0_/examples/miller_rabin_test/methods/guest/src/main` y su parte más relevante es la siguiente:

```

1  let input: Vec<u8> = env::read();
2
3
4  let first_four_bytes = &input[0..4];
5  let number_to_test = u32::from_be_bytes(first_four_bytes.try_into().expect("
6  Error transforming into number from bytes"));
7  let second_four_bytes = &input[4..8];
8  let iterations_limit = u32::from_be_bytes(second_four_bytes.try_into().expect("
9  Error transforming into number from bytes"));
10 let mut iteration_counter = 1;
11 let mut was_result_obtained = false;
12 let mut is_probably_prime = false;
13
14 let initial_value = modular_exponentiation(2, get_base_2_multiplier(
15 number_to_test), number_to_test);
16
17 // Modulo == +-1
18 if (initial_value == 1) || (initial_value == (number_to_test - 1)) {
19     was_result_obtained = true;
20     is_probably_prime = true;
21 }
22 let mut current_value = initial_value;
23
24 while !was_result_obtained && iteration_counter < iterations_limit {
25     current_value = (current_value * current_value) % number_to_test;
26
27     // Modulo == 1
28     if current_value == 1 {
29         was_result_obtained = true;
30         is_probably_prime = false;
31     }
32
33     // Modulo == -1
34     if current_value == (number_to_test - 1) {
35         was_result_obtained = true;
36         is_probably_prime = true;
37     }
38     iteration_counter += 1;
39 }
40
41 let outputs: Outputs = Outputs {
42     tested_number: number_to_test,
43     is_probably_prime,
44     iterations_limit_reached: iteration_counter == iterations_limit,
45 };
46 let serialized_outputs = to_string(&outputs).expect("Error in struct
47 serialization");
48 env::commit(&serialized_outputs);

```

El input consiste en un único array de 1024 bytes, en el que se recibe el número que se va a probar si es primo y la cantidad máxima de iteraciones del algoritmo que se realizarán antes de concluir que no converge. Se elige siempre el número 2 como valor de a , y el output del algoritmo es el número probado, si es o no probablemente primo, y si se llegó al límite indicado.

11.4. Mezcla de tests de primalidad

Por último, se implementó un test de primalidad que junta Miller-Rabin y el test probabilístico de Fermat. Al no tener estos métodos los mismos falsos positivos, se sabe que aunque uno de estos métodos indique que el número es primo, en caso de que se trate de un número compuesto podría fallar el siguiente algoritmo ejecutado.

El código para esta prueba puede encontrarse en `risc_0_/examples/mixed_test/methods/guest/src/main` y su parte más relevante es la siguiente:

```
1
2   let input1: Vec<u8> = env::read();
3   let probabilistic_fermat_result: FermatOutputs = probabilistic_fermat(input1);
4
5   if !probabilistic_fermat_result.is_probably_prime {
6       let returned_output = MillerRabinOutputs {
7           tested_number: probabilistic_fermat_result.tested_number,
8           is_probably_prime: probabilistic_fermat_result.is_probably_prime,
9           iterations_limit_reached: false,
10      };
11      let serialized_outputs = to_string(&returned_output).expect("Error in
12      fermat result struct serialization");
13      env::commit(&serialized_outputs);
14  } else {
15      let input2: Vec<u8> = env::read();
16      let miller_rabin_result: MillerRabinOutputs = miller_rabin(input2);
17      let serialized_outputs = to_string(&miller_rabin_result).expect("Error in
18      miller rabin result struct serialization");
19      env::commit(&serialized_outputs);
20  }
```

Se ve que en caso de que Fermat indique que no es primo se finaliza el algoritmo, y si indica que lo es entonces se retorna el resultado de Miller-Rabin, ya que si indica que es compuesto entonces definitivamente lo es, y si indica que es primo entonces es considerablemente posible ya que fue indicado por ambos tests en simultáneo. En este caso el input consiste en dos arrays de 1024 bytes, ya que aunque los inputs entraran en uno solo resultaba de utilidad reutilizar el código de las pruebas anteriores, además de que de esta forma se tiene un ejemplo de uso de un caso con más de un input en un grupo.

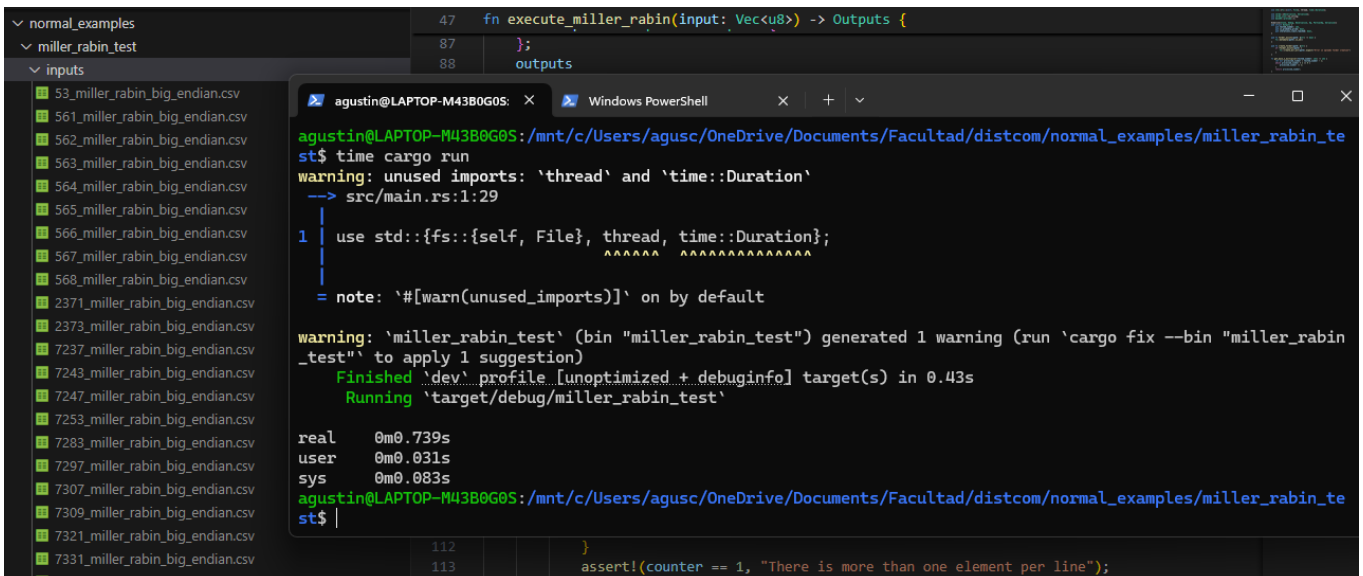
12. Comparación de tiempos de ejecución

Hasta el momento fue explicada la teoría de generación y verificación de la prueba y los objetivos del sistema de distribución de cómputo implementado. Sin embargo, se debe verificar si realmente tiene sentido utilizarlo, para lo cual se debe comparar el tiempo de verificación de una prueba con el tiempo del cómputo del resultado sin generar la prueba. Es decir, se debe ver si en el tiempo que la organización ejecuta una verificación podría haber directamente calculado el resultado por su cuenta, o incluso realizarlo varias veces.

Un ejemplo de tiempo de verificación del resultado de un test de primalidad de Miller-Rabin es el siguiente:

```
Starting proof verification of program with id "bd0c650d-1043-4541-9ec8-c90e17d452b9" with input group with id "f21267a0-b9ac-49cd-8570-8909c8a1f7f1"
downloaded_guest: Starting build for riscv32im-risc0-zkvm-elf
downloaded_guest: Compiling downloaded_guest v0.1.0 (/app/src/runner/methods/guest)
downloaded_guest: warning: unused variable: 'cast_base'
downloaded_guest: --> src/main.rs:28:9
downloaded_guest: |
downloaded_guest: 28 |     let cast_base = base as u64;
downloaded_guest: |         ^^^^^^^^^ help: if this is intentional, prefix it with an underscore: '_cast_base'
downloaded_guest: |
downloaded_guest: = note: '#[warn(unused_variables)]' on by default
downloaded_guest: warning: 'downloaded_guest' (bin "downloaded_guest") generated 1 warning
downloaded_guest: Finished 'release' profile [optimized] target(s) in 0.41s
Proof was verified, total seconds passed: 3
```

Sin embargo, este tiempo puede compararse con una prueba extrema realizada para la prueba Miller-Rabin sin el uso de ZKP. Se calculó cuánto tiempo se tarda en obtener el resultado de 29 inputs seguidos, obteniendo la siguiente medición:



```
agustin@LAPTOP-M43B0G0S: /mnt/c/Users/agusc/OneDrive/Documents/Facultad/distcom/normal_examples/miller_rabin_test$ time cargo run
warning: unused imports: 'thread' and 'time::Duration'
--> src/main.rs:1:29
1 | use std::{fs::{self, File}, thread, time::Duration};
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
= note: '#[warn(unused_imports)]' on by default

warning: 'miller_rabin_test' (bin "miller_rabin_test") generated 1 warning (run 'cargo fix --bin "miller_rabin_test"' to apply 1 suggestion)
Finished 'dev' profile [unoptimized + debuginfo] target(s) in 0.43s
Running 'target/debug/miller_rabin_test'

real    0m0.739s
user    0m0.031s
sys     0m0.083s
agustin@LAPTOP-M43B0G0S: /mnt/c/Users/agusc/OneDrive/Documents/Facultad/distcom/normal_examples/miller_rabin_test$
```

Puede verse que el tiempo total transcurrido para el usuario desde que se comenzó a ejecutar el programa hasta que terminó es de 739 milisegundos, lo cual es aproximadamente un tercio del tiempo en el que se tarda en verificar la prueba de un único resultado. Se ve entonces que finalmente no resulta rentable para las organizaciones utilizar el sistema, sino que es preferible que generen sus propios resultados, ya que se tiene un tiempo de ejecución de aproximadamente 25 milisegundos por input sin verificación, lo cual indica un incremento en el tiempo de ejecución de 120 veces más en este caso.

12.1. Mejoras a futuro en Risc Zero

A pesar de que actualmente no resulta viable para los programas ejecutados, se debe tomar en cuenta que hay fuertes mejoras de performance pendientes para Risc Zero ^[10], lo cual reducirá considerablemente los tiempos de ejecución de las verificaciones. Esto implica que, al tomar en cuenta que el tiempo de verificación de una prueba escala en orden polilogarítmico, se reducirá el tiempo mínimo de ejecución de un programa a partir del cual resulta rentable realizar una verificación de una prueba en comparación con la ejecución del programa en sí.

13. Problemas de desarrollo

A lo largo del desarrollo del proyecto surgieron variadas dificultades que se considera necesario aclarar por motivos didácticos.

13.1. Uso de Rust

A pesar de que Rust es un lenguaje de programación bastante amado entre sus usuarios, fue creado hace demasiado poco tiempo. Mientras que los lenguajes más conocidos de la industria fueron creados generalmente hace más de 20 años, Rust actualmente no sobrepasa los 10. Esto implica que no se trata de un lenguaje altamente usado, por lo que la documentación no suele ser clara ni abundante, no tiene una cantidad de bibliotecas comparable con otros gigantes de la industria, no hay una gran cantidad de interacción entre usuarios respecto a Rust, y el compilador no pasó por años de fuertes niveles de prueba.

13.1.1. Actix-web

Se trata de una biblioteca de rust considerada la estándar para el desarrollo de un server HTTP. Sin embargo, a pesar de esto no hay una gran cantidad de material disponible a la hora de definir el formato de los endpoints, sus inputs, cómo recibir archivos o cómo devolverlos en una respuesta del servidor. Una de las funcionalidades que más tiempo consumió descubrir cómo funcionaba correctamente fue el uso de middlewares, un concepto muy simple en otros lenguajes como Typescript con Express, pero excesivamente complicado y pobremente documentado en Actix-Web. El conjunto de todos estos pequeños problemas encontrados, que en otro lenguaje serían rápidamente solucionados, terminó sumando una gran cantidad de tiempo de desarrollo. Sin embargo, se debe tomar en cuenta también que, al haber ya pasado por estos problemas, no deberían presentarse en el futuro.

13.1.2. Diesel

Se trata de la biblioteca de Rust considerada el estándar para la interacción con bases de datos. Sin embargo, al igual que con Actix-Web, la documentación es considerablemente pobre y por lo tanto se dificulta su uso. En adición a esto, el uso de una subselección de columnas en una query a la base de datos causó un error en el compilador de Rust, el cual alternaba de forma aparentemente aleatoria entre compilar y fallar al inicio, y cuando no fallaba inmediatamente generaba un proceso que continuamente reservaba más memoria hasta generar fallos al nivel sistema operativo. Errores como este y la falta de claros usos de ejemplo por parte de los desarrolladores y usuarios (debido al poco uso de Rust en general) también consumieron una gran cantidad de tiempo adicional que normalmente hubiese ocupado unos pocos momentos.

13.1.3. Problemas varios

Otras fuentes de problemas que agregaron al tiempo de desarrollo son:

- S3: el uso de S3 de AWS en Rust no es muy común, por lo que aprender a utilizarlo también tomó más tiempo de lo esperado al compararlo con otros lenguajes.
- Problemas de detección de cambios en el código: al realizar los clientes de las organizaciones y los donantes una descarga y compilación de código, era necesario que detectaran los cambios cuando se cambiaba de programa a ejecutar o verificar. Sin embargo, estos cambios no eran detectados correctamente por el sistema, por lo que luego de investigar sobre las formas que tiene Rust de detectar dinámicamente estos cambios se terminó optando por indicar manualmente mediante el uso del comando de linux **touch** que se quería recompilar el código de la carpeta en la que se descargan los programas del sistema. Al tratarse de un problema relacionado con el compilador de Rust en sí, no es muy común y por lo tanto consumió más tiempo del esperado, tanto en su detección como en la búsqueda de la solución.

13.2. Cambio de Cairo a Risc Zero

Inicialmente se planeaba utilizar la máquina virtual de Cairo implementada por la empresa LambdaClass. Sin embargo, ese proyecto se encontraba en desarrollo, y eventualmente fue discontinuado, lo cual implicó que se debió buscar una alternativa. A esta altura ya se había dedicado una considerable cantidad de tiempo a la investigación del uso del prover y verifier de Cairo, y al lenguaje de programación en sí, el cual no terminó aportando al desarrollo del proyecto, salvo en lo relacionado a la teoría sobre la generación de pruebas de código genérico.

Luego de unos días de investigación se cambió a Risc Zero, el cual debió ser descargado, probado y utilizado, siendo todos estos pasos cuya documentación podría mejorarse considerablemente, ya que ocupó una enorme cantidad de tiempo.

Este cambio causó adicionalmente el abandono de un feature que estaba planeado implementarse previamente. Mientras que previamente se quería implementar la posibilidad de permitir a un donante simplemente ejecutar el código y subir al servidor la traza generada, para que otro donante genere la prueba sobre esta, esto no pudo realizarse ya que actualmente Risc Zero no permite generar una división entre la ejecución y la generación de la prueba.

14. Mejoras del programa a futuro

A pesar de que el sistema implementado se encuentra actualmente en un nivel aceptable para ser utilizado, podrían realizarse mejoras en el futuro en caso de que alguien decida continuar el desarrollo.

- Desarrollo de UI más amigable para los usuarios en los clientes de donantes y organizaciones.
- Mejoramiento de manejo de cuentas del servidor, para permitir utilizarlas en más de una computadora.
- Proporcionar opción de verificación de pruebas sin descargar y compilar el código, utilizando únicamente el hash del programa.
- Proporcionar opción de ejecución del programa descargando el bytecode en vez del programa para compilar. Esto permitiría a las organizaciones esconder hasta cierto grado su implementación, aunque tal vez ahuyentaría a algunos donantes.
- Implementación de interfaces para manejo de archivos con otros sistemas distintos a AWS.
- Implementación de proof-batching. El orden polilogarítmico de la verificación implica que, mientras más instrucciones sean ejecutadas para una misma prueba, más cómputo se ahorrará a la hora de realizar la verificación. Una gran mejora en eficiencia podría ser obtenida si se logra concatenar la generación de pruebas.

Referencias

- [1] LambdaClass. (n.d.). *STARKs Prover: Section 4 (Subsections 4.1 to 4.4.2)*. GitHub Pages. <https://lambdaclass.github.io/lambdaworks/starks/starks.html>
- [2] Risc Zero. (n.d.). *Stark by hand. Risc Zero Developer Documentation*. <https://dev.risczero.com/proof-system/stark-by-hand>
- [3] Alessandro Chiesa, Lior Goldberg. Starkware. (2019, Marzo 28). *Low Degree Testing: The Secret Sauce of Succinctness*. <https://medium.com/starkware/low-degree-testing-f7614f5172db>
- [4] Michael Riabzev, Eli Ben-Sasson. Starkware. (2019, Febrero 4). *STARK Math: The Journey Begins. Part 1* <https://medium.com/starkware/stark-math-the-journey-begins-51bd2b063c71>
- [5] Kineret Segal, Shir Peled. Starkware. (2019, Febrero 20). *Arithmetization I* <https://medium.com/starkware/arithmetization-i-15c046390862>
- [6] Shir Peled. Starkware. (2019, Marzo 14). *Arithmetization II: "We Need To Go Deeper"* <https://medium.com/starkware/arithmetization-ii-403c3b3f4355>
- [7] Alessandro Chiesa, Gideon Kaempfer. Starkware. (2019, Abril 17). *A Framework for Efficient STARKs: Combining probabilistic proofs and hash functions* <https://medium.com/starkware/a-framework-for-efficient-starks-19608ba06fbc>
- [8] Kineret Segal, Gideon Kaempfer. Starkware. (2019, Julio 16). *StarkDEX Deep Dive: the STARK Core Engine, Part 3 of 4* <https://medium.com/starkware/starkdex-deep-dive-the-stark-core-engine-497942d0f0ab>
- [9] Lior Goldberg, Shahar Papini, Michael Riabzev. (2021, Agosto). *Cairo – a Turing-complete STARK-friendly CPU architecture* <https://eprint.iacr.org/2021/1063.pdf>
- [10] Santiago Campos-Araoz. (2021, Junio 19). *zkVM Performance Upgrades Roadmap - Q3 2024* <https://risczero.com/blog/zkvm-performance-upgrades-roadmap---q3-2024>
- [11] Nesso Academy. (2022). *Testing for Primality (Miller-Rabin Test)* https://www.youtube.com/watch?v=8i0UnX7Snkc&ab_channel=NesoAcademy
- [12] Khan Academy Labs. (2014). *Fermat primality test* https://www.youtube.com/watch?v=oUMotDWLpw&ab_channel=KhanAcademyLabs
- [13] Mathologer. (2018). *Fermat's HUGE little theorem, pseudoprimes and Futurama* https://www.youtube.com/watch?v=_9fbBSxhkuA&ab_channel=Mathologer