Fundamentos de una Wallet MPC desde Scratch

Enmanuel Cabrera
cenmanuel257@gmail.com

4 de octubre de 2024

Resumen

La Computación Multipartita Segura (MPC) permite realizar cálculos conjuntos sin comprometer la privacidad de los datos involucrados. Este artículo explora su integración con blockchain, enfocándose en esquemas de firma umbral (TSS) para mejorar la seguridad en la gestión de claves privadas. Además, se presenta una implementación práctica de una billetera MPC en Rust, que distribuye las claves entre varias partes para evitar puntos únicos de falla. Finalmente, se discuten las implicaciones de tecnologías como pruebas de conocimiento cero (ZK) y cifrado homomórfico completo (FHE) en la privacidad y seguridad criptográfica.

Índice

1. Introducción

La computación multipartita segura (SMPC/MPC) permite que varios participantes calculen de manera colaborativa una función acordada previamente sin la necesidad de un tercero de confianza. Este concepto se originó a partir del problema del millonario del profesor Andrew Yao y su solución mediante criptografía en 1982. El esquema, es una interacción entre dos personas, podía descubrir quién era más rico sin revelar su riqueza real. Permite que los usuarios colaboren en los cálculos entre sí sin revelar ninguna información confidencial. Desde entonces MPC ha evolucionado hasta convertirse en una rama importante de la criptografía moderna.

1.1. Ejemplo motivador

Imagina que un grupo de 3 investigadores está trabajando en un proyecto conjunto y cada uno ha desarrollado un algoritmo innovador con resultados distintos. Los investigadores quieren colaborar para descubrir cuál de los algoritmos es el más eficiente, es decir, cuál tiene el menor tiempo de ejecución. Sin embargo, cada investigador considera que los detalles de su propio trabajo son confidenciales y no quiere revelar sus resultados exactos a los demás. Para solucionar esto, necesitan determinar cuál es el algoritmo más rápido sin exponer los tiempos de ejecución de cada uno.

El problema puede ser representado por la función:

$$F(t_1, t_2, t_3) = min(t_1, t_2, t_3)$$

donde t_1, t_2, t_3 son los tiempos de ejecución de cada uno de los algoritmos de los investigadores.

Una solución sencilla sería que todos revelaran sus tiempos de ejecución a un tercero que pudiera calcular el valor mínimo. Sin embargo, como no confían en ninguna entidad externa para manejar estos datos confidenciales, necesitan una forma de encontrar el resultado sin revelar sus tiempos a nadie más.

Usando un protocolo de computación multipartidaria (MPC), los investigadores podrían colaborar para determinar cuál es el algoritmo más rápido. El protocolo garantizaría que cada uno contribuya con su tiempo de ejecución a la función, pero sin que nadie pueda inferir el tiempo de los demás, más allá de saber cuál fue el más rápido. Así, logran comparar sus algoritmos sin comprometer la confidencialidad de sus investigaciones.

Las dos propiedades principales de MPC son la corrección y la privacidad:

- Corrección: el resultado producido por un algoritmo es correcto (como se esperaba).
- Privacidad: los datos de entrada secretos que una parte posee no se filtrarán a las otras partes.

1.2. MPC en Blockchain

Para mejorar la seguridad de los activos digitales en la cadena de bloques, se ha introducido MPC en el escenario de firma múltiple. se utilizan múltiples fragmentos de clave para calcular la firma final utilizando protocolos MPC durante el proceso de firma. Esta firma se puede verificar utilizando la clave pública única correspondiente. Esta técnica, conocida como firma múltiple MPC, proporciona una forma altamente segura y eficiente de proteger los activos digitales en la cadena de bloques.

Usaremos MPC para calcular una firma digital de forma distribuida. Veamos cómo se pueden aplicar las propiedades anteriores a las firmas. Recordemos que, para las firmas, tenemos tres pasos:

1. **Generación de claves**: el primer paso es también el más complejo. Necesitamos generar una clave que será pública y se usará para verificar firmas futuras. Pero también necesitamos generar un secreto individual para cada parte, al que llamaremos secreto compartido. En términos de corrección y privacidad, decimos que la función generará la misma clave pública para todas las partes y un secreto compartido diferente para cada una de ellas, de modo que se garantice: (1) privacidad: no se filtren datos de los secretos compartidos entre las partes, y (2) corrección: la clave pública es una función de los secretos compartidos.
2. **Firma**: este paso implica una función de generación de firma. La entrada de cada parte será su parte secreta, creada como salida del paso anterior (generación de clave distribuida). También hay una entrada pública conocida por todos, que es el mensaje que se va a firmar. La salida será una firma digital, y la propiedad de privacidad garantiza que no se produzcan fugas de partes secretas durante el cálculo.
3. **Verificación**: el algoritmo de verificación permanece como en la configuración clásica. Para ser compatible con firmas de clave única, todos los que conozcan la clave pública deberían poder verificar y validar las firmas. Esto es exactamente lo que hacen los nodosde validación de la cadena de bloques.

Esquema de firma de umbral (TSS, Threshold Signatures Scheme) es el nombre que le damos a esta composición de generación de clave distribuida (DKG) y firma distribuida de un esquema de firma de umbral.

1.3. TSS en Blockchain

Podemos crear una nueva dirección generando una clave privada y luego calculando la clave pública a partir de la clave privada. Finalmente, la dirección de la cadena de bloques se deriva de la clave pública. Utilizando TSS, tendríamos un conjunto de n partes que calculan conjuntamente la clave pública, cada una de las cuales posee una parte secreta de la clave privada (las partes individuales no se revelan a las otras partes). A partir de la clave pública, podemos derivar la dirección de la misma manera que en el sistema tradicional, lo que hace que la cadena de bloques sea independiente de cómo se genera la dirección. La ventaja es que la clave privada ya no es un único punto de falla porque cada parte posee solo una parte de ella.

La generación de claves distribuidas se puede realizar de una manera que permita diferentes tipos de estructuras de acceso: la configuración general "t de n" podrá soportar hasta t fallas arbitrarias en operaciones relacionadas con la clave privada, sin comprometer la seguridad.

- $\{t-n\}$ significa que el umbral es t y el número de participantes es n. Se requieren al menos t participantes para recuperar la clave privada y firmar un mensaje.
- $\{n-n\}$ significa que el umbral es n y el número de participantes es n.

2. Implementando Wallet MPC en Rust

Una billetera MPC utiliza la tecnología de cálculo multiparte con el objetivo de mejorar la seguridad de tus criptomonedas y otros activos digitales. Divide la clave privada de una cartera entre varias partes para aumentar la privacidad y reducir los riesgos de hackeos, brechas y pérdidas. Cada una de las partes crea una llave privada de forma independiente como parte de la clave privada de la wallet MPC para firmar mensajes. Las diferentes claves nunca se encuentran entre sí, ninguna parte tiene acceso a la clave privada completa, eliminando así los puntos únicos de fallo. La naturaleza interactiva de los procesos de configuración de la wallet MPC requiere que todas las partes estén presentes durante la acción. Analizaremos una sencilla implementación de este proceso en el lenguaje de programación Rust con el fin de ilustrar los conceptos de forma educativa. Revisamos paso por paso nuestra función principal Este es un ejemplo de código en `fn main()`. y apoyándonos en los módulos secundarios que también fueron implementados. La implementación completa esta disponible aquí wallet-mpc-zk-paillier

2.1. MPC y TSS utilizando ECC

La criptografía de curva elíptica (ECC) es una familia moderna de criptosistemas de clave pública. Este tipo de criptografía se basa en las estructuras algebraicas de curvas elípticas sobre cuerpos finitos ($\mathbb{F}_p$) donde p es un número primo grande, p. ej. 256 bits, y en la dificultad del problema del logaritmo discreto en curvas elípticas (ECDLP). ECC implementa todas las capacidades principales de los criptosistemas asimétricos: cifrado, firmas e intercambio de claves. Para esta demostración, que es solo para fines educativos, trabajaremos con una curva elíptica definida sobre el campo finito $\mathbb{F}_{17}$ por la ecuación en forma de Weierstrass. Cuando definimos una curva sobre un cuerpo finito restringimos la pertenencia de las coordenadas a esta estructura algebraica.

$$E(\mathbb{F}_{17}) : y^2 = x^3 - 2 * x + 7 \quad \text{mód } (17)$$

```
    let new_ec: EcWei = EcWei::new(
      BigInt::from_i64(-2).unwrap(),
      BigInt::from_i64(7).unwrap(),
      BigInt::from_i64(17).unwrap(),
    );
```

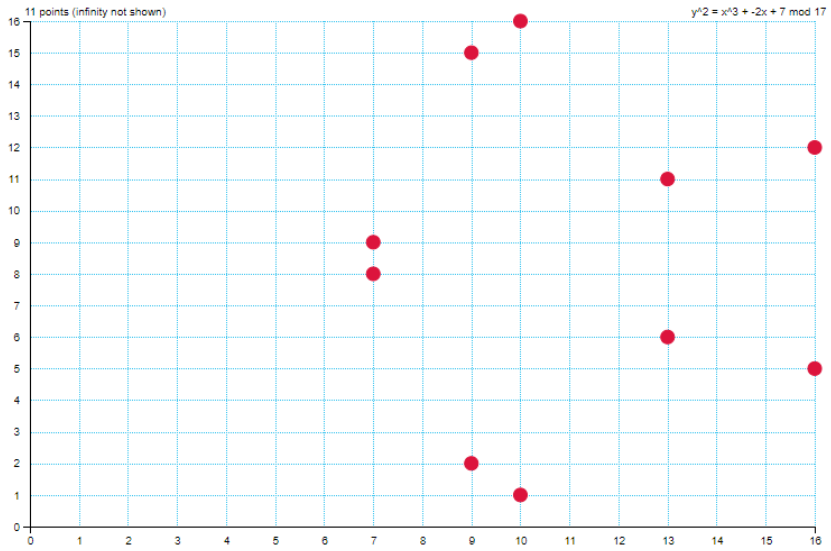
En resumen una curva elíptica sobre el campo finito $\mathbb{F}_p$ consta de:

- un conjunto de coordenadas enteras $\{x, y\}$, tales que $0 \leq x, y < p$
- permaneciendo en la curva elíptica: $y^2 \equiv x^3 + ax + b \mod (p)$

Puedes chequear si algún punto pertenece a la curva definida:

```
    println!(
      "{:?}",
      new_ec.is_point(&Point::new(
        BigInt::from_i64(4).unwrap(),
        BigInt::from_i64(7).unwrap()
      ))
```

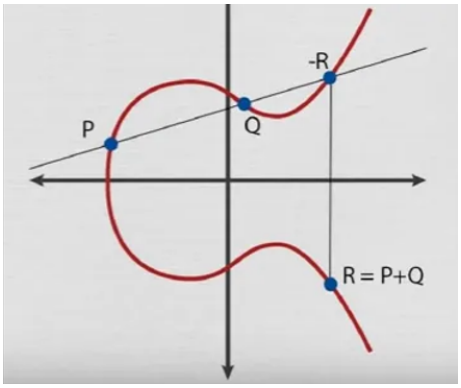
```
    ); //false
    println!(
      "{:?}",
      new_ec.is_point(&Point::new(
        BigInt::from_i64(4).unwrap(),
        BigInt::from_i64(8).unwrap()
      ))
    ); //true
```



Esta curva elíptica, tiene propiedades que la hacen ideal para operaciones criptográficas. Al estar definida sobre un campo finito, el conjunto de soluciones de la curva (los puntos) forma un grupo abeliano finito bajo la operación de suma de puntos en la curva. La curva que estamos utilizando tiene un grupo de orden 11, es decir, hay 11 puntos en total que forman este grupo, incluyendo el punto en el infinito, que actúa como el elemento neutro en la operación de suma. Te invito a crear curvas elípticas nuevas de tipo Weierstrass pasando los nuevos valores enteros para a, b y p en ese orden al siguiente método:

```
    let new_ec = EcWei::new(a, b, p);
```

Es hermoso observar la posibilidad de expresar una operación binaria en un conjunto finito mediante una tabla. En este conjunto que esta formado por todos los puntos que satisfacen la ecuación de la curva de Weierstrass, una de las propiedades que cumple es que podemos sumar cualquiera dos puntos y obtener un punto que pertenece al mismo conjunto. Implementamos la suma de dos puntos de la siguiente forma:



Given an Elliptic Curve E: $y^2 = x^3 + AX + B$
Also, being $P1$ and $P2$ points on the curve E. It applies that:

(a) If $P_1 = (0, 0)$, then $P_1 + P_2 = P_2$.
(b) Otherwise, if $P_2 = (0, 0)$, then $P_1 + P_2 = P_1$.
(c) Otherwise, write $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$.
(d) If $x_1 = x_2$ and $y_1 = -y_2$, then $P_1 + P_2 = (0, 0)$.
(e) If $x_1 = x_2$ and $y_1 \neq y_2$, then $P_1 + P_2 = (0, 0)$.
(f) Otherwise:

- (e1) if $P_1 \neq P_2$, then: $\lambda = (y_2 - y_1)/(x_2 - x_1)$
- (e2) if $P_1 = P_2$, then: $\lambda = (3 * x_1^2 + A)/(2 * y_1)$

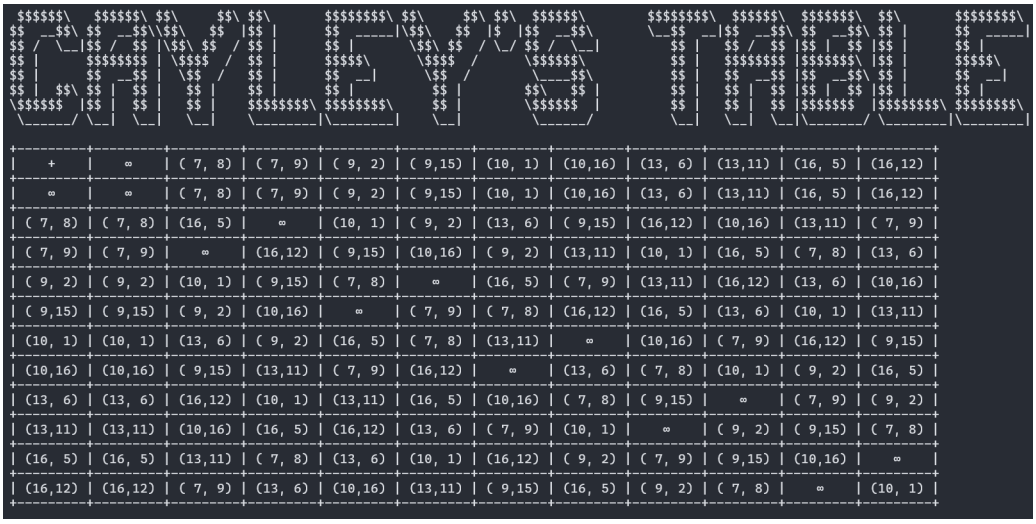
(g) $x_3 = \lambda^2 - x_1 - x_2$, $y_3 = \lambda(x_1 - x_3) - y_1$
(h) $P_1 + P_2 = (x_3, y_3)$

```
    println!(
      "{:?}",
      new_ec.point_add(
        &Point::new(BigInt::from_i64(7).unwrap(), BigInt::from_i64(9).unwrap()),
        &Point::new(BigInt::from_i64(7).unwrap(), BigInt::from_i64(9).unwrap())
```

```
        )
    ); // (16,12)
    println!(
        "{:?}",
        new_ec.point_add(
            &Point::new(BigInt::from_i64(7).unwrap(), BigInt::from_i64(9).unwrap()),
            &Point::new(BigInt::from_i64(10).unwrap(), BigInt::from_i64(16).unwrap())
        )
    ); // (13,11)
```

Siempre que este conjunto cumpla con las propiedades de una estructura algebraica de grupo, podemos construir una tabla de Cayley. Puedes observar la tabla de Cayley para el grupo que forma la curva elíptica creada, utilizando el siguiente método:

```
    let group_add = new_ec.group_points();
    new_ec.cayley_table(&group_add);
```



```
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
|    +    |    ∞    | ( 7, 8) | ( 7, 9) | ( 9, 2) | ( 9,15) | (10, 1) | (10,16) | (13, 6) | (13,11) | (16, 5) | (16,12) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
|    ∞    |    ∞    | ( 7, 8) | ( 7, 9) | ( 9, 2) | ( 9,15) | (10, 1) | (10,16) | (13, 6) | (13,11) | (16, 5) | (16,12) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| ( 7, 8) | ( 7, 8) | (16, 5) |    ∞    | (10, 1) | ( 9, 2) | (13, 6) | ( 9,15) | (16,12) | (10,16) | (13,11) | ( 7, 9) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| ( 7, 9) | ( 7, 9) |    ∞    | (16,12) | ( 9,15) | (10,16) | ( 9, 2) | (13,11) | (10, 1) | (16, 5) | ( 7, 8) | (13, 6) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| ( 9, 2) | ( 9, 2) | (10, 1) | ( 9,15) | ( 7, 8) |    ∞    | (16, 5) | ( 7, 9) | (13,11) | (16,12) | (13, 6) | (10,16) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| ( 9,15) | ( 9,15) | ( 9, 2) | (10,16) |    ∞    | ( 7, 9) | ( 7, 8) | (16,12) | (16, 5) | (13, 6) | (10, 1) | (13,11) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| (10, 1) | (10, 1) | (13, 6) | ( 9, 2) | (16, 5) | ( 7, 8) | (13,11) |    ∞    | (10,16) | ( 7, 9) | (16,12) | ( 9,15) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| (10,16) | (10,16) | ( 9,15) | (13,11) | ( 7, 9) | (16,12) |    ∞    | (13, 6) | ( 7, 8) | (10, 1) | ( 9, 2) | (16, 5) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| (13, 6) | (13, 6) | (16,12) | (10, 1) | (13,11) | (16, 5) | (10,16) | ( 7, 8) | ( 9,15) |    ∞    | ( 7, 9) | ( 9, 2) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| (13,11) | (13,11) | (10,16) | (16, 5) | (16,12) | (13, 6) | ( 7, 9) | (10, 1) |    ∞    | ( 9, 2) | ( 9,15) | ( 7, 8) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| (16, 5) | (16, 5) | (13,11) | ( 7, 8) | (13, 6) | (10, 1) | (16,12) | ( 9, 2) | ( 7, 9) | ( 9,15) | (10,16) |    ∞    |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
| (16,12) | (16,12) | ( 7, 9) | (13, 6) | (10,16) | (13,11) | ( 9,15) | (16, 5) | ( 9, 2) | ( 7, 8) |    ∞    | (10, 1) |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
```

La suma del punto (10, 16) con el punto (9, 2) es: (7, 9)

Note la belleza de esta tabla con respecto a la diagonal en el que se observa que los elementos son simétricos. Esto se debe a que la suma de dos puntos es conmutativa. $A + B = B + A$. Si esta propiedad se cumple para todos los puntos, como muestra la tabla, decimos que el grupo es abeliano.

2.2. Punto Generador G

Una de las propiedades más relevantes de esta curva es que satisface una condición especial: todos los puntos del grupo pueden actuar como generadores del grupo. Notaremos que esta propiedad es fundamental para las operaciones aritméticas que se realizarán en el campo escalar que define este grupo, es decir $\mathbb{F}_{11}$. Un generador es un punto G tal que cualquier otro punto de la curva puede obtenerse como una combinación del punto G sumado a sí mismo, $G \cdot n$ (es decir, mediante multiplicación escalar). Podemos seleccionar cualquier punto en la curva

$$E(17) : y^2 = x^3 - 2x + 7 \mod (17),$$

como el generador G del grupo. Esto es posible gracias a la propiedad de que el orden de G es la cantidad de elementos del grupo. Nos referimos con orden de G a la cantidad de veces que debe sumarse este punto para generar el punto al infinito (∞), que en este caso es el elemento identidad del grupo.

```
    // set of all enerator point
    let points_g = new_ec.get_base_points(&group_add);
    for point in points_g.iter() { println!("Generator {:?}", point) };
```

Note que si instanciamos otra curva de Weierstrass por ejemplo:

$$E(17) : y^2 = x^3 - 3x + 4 \mod (17)$$

hay puntos que no generan a todos los demás elementos del grupo y estos no serán tomados como puntos generadores para el grupo de la curva:

```
    let other_ec = EcWei::new(
      BigInt::from_i64(-3).unwrap(),
      BigInt::from_i64(4).unwrap(),
      BigInt::from_i64(17).unwrap(),
    );
    println!(
      " G * 2 = {:?}",
      other_ec.scalar_mul(
        &Point::new(BigInt::from_i64(6).unwrap(), BigInt::from_i64(10).unwrap()),
        &BigInt::from(2)
      )
    ); // (6,7)
    println!(
      " G * 3 = {:?}",
      other_ec.scalar_mul(
        &Point::new(BigInt::from_i64(6).unwrap(), BigInt::from_i64(10).unwrap()),
        &BigInt::from(3)
      )
    ); // (0,0) => infinity point
    println!(
      " G * 4 = {:?}",
      other_ec.scalar_mul(
        &Point::new(BigInt::from_i64(6).unwrap(), BigInt::from_i64(10).unwrap()),
        &BigInt::from(4)
      )
    ); // (6,10)
    println!(
      " G * 5 = {:?}",
      other_ec.scalar_mul(
        &Point::new(BigInt::from_i64(6).unwrap(), BigInt::from_i64(10).unwrap()),
        &BigInt::from(5)
      )
    ); // (6,7)
```

Podemos decir que el punto (6,10) genera solo tres puntos en el grupo dado por la curva $E(\mathbb{F}_{17})$: $y^2 = x^3 - 3x + 4 \mod (17)$, a lo que es lo mismo que (6,10) es de orden 3.

2.3. ECDSA

Las claves privadas en el ECC son números enteros (en el rango del tamaño del campo de la curva, normalmente números enteros de 256 bits). La generación de claves en la criptografía ECC es tan sencilla como generar de forma segura un número entero aleatorio en un rango determinado, por lo que es extremadamente rápida. Cualquier número dentro del rango es una clave privada ECC válida. Las claves públicas en el ECC son puntos EC - pares de coordenadas enteras (x, y), que se encuentran en la curva. Uno de los usos más comunes de las curvas elípticas en criptografía es el Algoritmo de Firma Digital basado en Curvas Elípticas (ECDSA). En este algoritmo, la seguridad está basada en la dificultad de resolver el problema del logaritmo discreto en el grupo de puntos de la curva. Es decir dada una clave pública generada es computacionalmente difícil encontrar la clave privada. Generemos algunos pares de claves ECDSA a partir de un punto generador podemos generar claves públicas para claves privadas. Con fines educativos trabajamos en la curva $E(\mathbb{F}_{17}) : y^2 = x^3 - 2x + 7 \mod (17)$

2.3.1. Generación de clave: ECDSA

1. Seleccione un punto G generador (que pertenezca a $E(\mathbb{F}_{17})$) de orden n. Para esta curva todos los puntos son de orden $n = 11$, la cantidad de elementos del grupo.
2. Seleccione aleatoriamente un número d en el intervalo $[1,\ n-1]$.
3. Calcule $Q = dG$.
4. d será la llave privada. Note que en el punto 2 si $d > n - 1$, otra clave privada $d_1 < d$ genera también Q y esto no es deseado.
5. Q será la llave pública.

```
    // TSS setup with ECDSA: For two parties
    let generators = new_ec.get_base_points(&group_add);
    let point_g = generators[0].clone();
    let key_pair_1 = new_ec.gen_key_pair(&point_g);
```

El principal inconveniente de utilizar una configuración tradicional para generar claves de ECDSA en una billetera MPC para firmar y verificar firmas ECDSA, es que crea un único punto de falla.

Si cada usuario posee una copia completa de la clave privada, cualquier usuario podría comprometer la seguridad del protocolo. Al generar la clave privada en fragmentos y distribuir estos fragmentos entre los participantes del MPC, se elimina el riesgo de que un solo individuo represente un punto de vulnerabilidad para el sistema. Por lo tanto, es necesario modificar el protocolo para evitar este punto único de falla. En lugar de que cada usuario tenga acceso directo a la clave privada completa, se debe implementar un esquema de generación y distribución de fragmentos de clave. De esta manera, el control se descentraliza, y el protocolo se vuelve más seguro.

2.3.2. Configuración de TSS con ECDSA: Para dos parte

A diferencia de otros esquemas como RSA, firmas Schnorr y otros, es particularmente difícil construir protocolos de firma de umbral eficientes para ECDSA. Como resultado, los protocolos más conocidos hoy en día para ECDSA distribuido seguro requieren ejecutar pruebas de conocimiento cero pesadas y calcular muchas exponenciaciones de módulo grande para cada operación de firma. A continuación, consideramos el caso específico de dos partes (y, por lo tanto, sin mayoría honesta). [1]

1. La Parte 1 elige un número al azar d_1 donde $0 < d_1 < n$. Calcula $Q_1 = d_1 G$.
2. La Parte 2 elige un número al azar d_1 donde $0 < d_2 < n$. Calcula $Q_2 = d_2 G$.

```
    let key_pair_2 = new_ec.gen_key_pair(&point_g);
```

3. Cada parte enviará Q_1 o Q_2 entre sí de manera que la Parte 2 calcule $Q = Q_1 d_2$ y la Parte 1 computará $Q = Q_2 d_1$. Ambas partes deben llegar al mismo punto Q donde $Q = d_1 d_2 G$. Este protocolo criptográfico que permite a dos partes que no tienen conocimiento previo entre sí, establecer un secreto compartido sobre un canal público es conocido como el intercambio de claves Diffie-Hellman (DH).

```
    // Diffie-Hellman
    let part_1_dh = new_ec.scalar_mul(
              &key_pair_2.as_ref().unwrap().pk,
              &key_pair_1.as_ref().unwrap().sk
            );
    let part_2_dh = new_ec.scalar_mul(
              &key_pair_1.as_ref().unwrap().pk,
              &key_pair_2.as_ref().unwrap().sk
            );
```

4. La Parte 1 también generará un par de claves Paillier con un módulo N y enviará una versión cifrada de su secreto d_1,$c_{key} = Enc(d_1)$, con la clave pública de Paillier para la Parte 2. Para el protocolo Paillier fijamos algunos valores con fines educativos, por lo general para lograr niveles de seguridad requeridos se trabajan con números grandes por ejemplo de 256 bits.
   - **Generación de claves Paillier:**
     a) Seleccionamos 2 números primos cualesquiera ($p = 11$ y $q = 3$). Calculamos $N = pq = 33$.
     b) $\lambda = lcm(p-1, q-1) = 10$, donde lcm es el mínimo común múltiplo
     c) Seleccionamos aleatoriamente un entero g que pertenece a $\mathbb{Z}^*_{N^2} = \mathbb{Z}^*_{33^2} = \{1, 2, 3, ..., 1087, 1088\}$
     d) Aseguramos que Ndivida a g chequeando el siguiente inverso multiplicativo μ:

$$\mu = \left[L(g^{\lambda} \mod N^2)\right]^{-1} \mod (N),$$

donde L es la función $L(x) = \dfrac{x-1}{N}$ $\mu = \left[L(g^{\lambda} \mod 1089)\right]^{-1} \mod (33)$

     e) Entonces clave privada: (λ, μ) y clave pública: (g, N)

```
let paillier_key_p1 = gen_key_paillier(
              BigInt::from_i64(11).unwrap(),
              BigInt::from_i64(3).unwrap()
            );
```

- **Cifrar secreto d_1 con Paillier:**
  a) Seleccionamos aleatoriamente un entero r que pertenece $\mathbb{Z}_N^* = \mathbb{Z}_{33}^*$ y que su máximo común divisor con N sea igual a 1. Es decir $gcd(r, N) = 1$. Note que si $gcd(r, N) \neq 1$ se puede calcular la clave privada correctamente con poca probabilidad.
  b) Cifrado:

$$c_{key} = Enc(d_1) = g^{d_1} \cdot r^N \mod (N^2) = g^{d_1} \cdot r^{33} \mod (1089)$$

```
let chiper_p1 = cipher_paillier(
          &paillier_key_p1.public_key,
          &key_pair_1.as_ref().unwrap().sk
        );
```

- **Descifrado de Paillier**:

$$d_1 = Dec(c_{key}) = \left[L(c_{key}^{\lambda} \mod N^2) \cdot \mu\right] \mod (N) = \left[L(c_{key}^{\lambda} \mod 1089) \cdot \mu\right] \mod (33)$$

```
let dechiper_p1 = decipher_paillier(
    &paillier_key_p1.private_key,
    chiper_p1.as_ref().unwrap().clone(),
    &paillier_key_p1.public_key
  );
```

5. Al final, la Parte 1 tiene un par de claves, una privada y una pública: (d_1, Q) y la Parte 2 tiene un par de claves públicas y una privada (d_2, Q, c_{key}).

2.3.3. Homomorfismo de Anillo

¿Por qué un par de claves Paillier? El cifrado Paillier se utiliza para que se puedan realizar operaciones en la clave privada de otra parte sin revelar la clave privada. Paillier presenta propiedades homomórficas tanto aditivas como multiplicativas. [2]

El descifrado de multiplicar dos textos cifrados juntos resulta en $m_1 + m_2$. Donde m_1 y m_2 son los textos claros para los textos cifrados. Esta propiedad es lo que llamamos homomorfismo aditivo:

$$Dec(Enc(m_1, r_1) \cdot Enc(m_2, r_2)) = m_1 + m_2$$

El descifrado de elevar un texto cifrado a la potencia de otro texto claro resultará en $m_1 \cdot m_2$ donde m_1 y m_2 son los textos claros para los textos cifrados. Esta propiedad es lo que llamamos homomorfismo multiplicativo:

$$Dec(Enc(m_1, r_1)^{m_2}) = m_1 \cdot m_2$$

Para lograr esta propiedad es necesario que ambas partes tengan la misma clave pública (g, N)***. Es un supuesto criptográfico el problema computacional de factorizar*** N ***para obtener la clave privada de Pailler. En teoría este problema es difícil para la Parte 2, o cualquier atacante.***

Anillo $\mathbb{Z}_n$

Un anillo $(A, +, \cdot)$ es un conjunto A junto con dos operaciones binarias (leyes internas). Usualmente $+$ y $\cdot$, que llamamos suma y multiplicación respectivamente, bien definidas en $\mathbb{A}$ tales que se satisfacen los siguientes axiomas:

φ_1 El grupo aditivo $(A, +)$ es un grupo abeliano. El elemento neutro lo denotaremos por 0.

φ_2 La multiplicación es asociativa, o sea $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

φ_3 Para todos elementos $a, b, c \in A$, se cumple la ley distributiva izquierda $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$ y la ley distributiva derecha $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$.

Ejemplo: Probemos que la estructura algebraica $(\mathbb{Z}_n, +_n, \cdot_n)$ es un anillo unitario y conmutativo, para todo $n > 0$, donde $\mathbb{Z}_n = \{0, 1, ..., n-1\}$

1. *Probaremos que* $(\mathbb{Z}_n, +_n)$ *es un grupo abeliano*
   a) (Ley interna): Definamos $a +_n b = r$ para todo $a, b \in \mathbb{Z}_n$ tal que $a + b = nq + r$, donde $0 \leq r < n$. Verifiquemos que el resultado de esta suma pertenece siempre a $\mathbb{Z}_n$:
   - Si $a + b < n \Rightarrow a +_n b \in \mathbb{Z}_n$
   - Si $a + b \geq n$ (aplicamos def.) $\Rightarrow a +_n b = r$, donde $r \in \mathbb{Z}_n$

   $$\therefore (\mathbb{Z}_n, +_n) \text{ es cerrado}$$

   b) (Neutro): El cero es el neutro aditivo en $\mathbb{Z}_n$. Pues para todo $a \in \mathbb{Z}_n$:

   $$a + 0 = 0 + a$$

   c) (Inverso): Sea $a \in \mathbb{Z}_n$, $n - a$ es el inverso de a en $\mathbb{Z}_n$:

   $$a +_n (n - a) = n +_n (a - a) = n = 0, \text{ pues } n = n +_n 0 = 0 \text{ por definición}$$

   Los inversos no pueden ser negativos porque no hay negativos en $\mathbb{Z}_n$***.***

   d) (Asociatividad): Sean $a, b, c \in \mathbb{Z}_n$ los elementos $a +_n (b +_n c)$ $(a +_n b) +_n c$ son residuos de la dividir $a + b + c$ entre n y por el algoritmo de la división el resto es único:

   $$\therefore a +_n (b +_n c) = (a +_n b) +_n c$$

   e) (Conmutatividad): Sean $a, b \in \mathbb{Z}_n$. La operación $a +_n b = b +_n a$ pues $a + b \equiv b + a$. Es decir tienen los mismos restos al dividir entre n.

   $$\therefore (\mathbb{Z}_n, +_n) \text{ es un grupo abeliano}$$

2. *Probemos que* $(\mathbb{Z}_n, \cdot_n)$ *cumple con la asociatividad*
   a) (Ley interna): *Seana*, $b \in \mathbb{Z}_n$. Definimos $a \cdot_n b = r$ tal que $a \cdot b = nq + r$:
   - Si $a \cdot b < n \Rightarrow a \cdot_n b = a \cdot b \in \mathbb{Z}_n$
   - Si $a \cdot b \geq n$ aplicando definición $\Rightarrow a \cdot_n b = r, r \in \mathbb{Z}_n$

   $$\therefore (\mathbb{Z}_n, \cdot_n) \text{ es cerrado}$$

   b) (Asociatividad): Similar a la asociatividad de la suma $(+_n)$. Entonces $\mathbb{Z}_n$ con la operación es asociativo.
3. *Probemos si se cumple la ley distributiva*
   a) (Distributividad): Sean $a, b, c \in \mathbb{Z}_n$, podemos decir que $a \cdot_n (b +_n c) = (a \cdot_n b) +_n (a \cdot_n c)$, puesto que $a \cdot (b + c) \equiv (a \cdot b) + (a \cdot c)$, es decir tienen los mismos resto al dividir entre n. Igualmente:

   $$(b +_n c) \cdot_n a = (b \cdot_n a) +_n (c \cdot_n a)$$

   $$\boxed{(\mathbb{Z}_n, +_n, \cdot_n) \text{ es un Anillo}}$$

   Además este anillo es unitario y conmutativo:
4. *a*) (Unitario): El 1 es neutro en $\mathbb{Z}_n$ para $(\cdot_n)$ siempre y cuando $n > 1$ porque si n = 1 entonces $\mathbb{Z}_n = \{0\}$. Pues para todo $a \in \mathbb{Z}_n$:

   $$a \cdot_n 1 = 1 \cdot_n a = a$$

   b) (Conmutatividad): Sean $a, b \in \mathbb{Z}_n$.

   $$a \cdot_n b = b \cdot_n a,$$

   pues $a \cdot b \equiv b \cdot a$, es decir ambos miembros tienen el mismo resto al dividirlos entre n

   $$\boxed{(\mathbb{Z}_n, +_n, \cdot_n) \text{ es un Anillo Unitario Conmutativo}}$$

Homomorfismo de Paillier $f : \mathbb{Z}_N \to \mathbb{Z}_{N^2}$

Comprobemos ahora que la función $f : \mathbb{Z}_N \to \mathbb{Z}_{N^2}$, del cifrado de Paillier donde $f(m) = g^m r^N$ mód N^2 es un homomorfismo de anillos, de $(\mathbb{Z}_N, +_N, \cdot_N)$ a $(\mathbb{Z}_{N^2}, +_{N^2}, \cdot_{N^2})$. Se debe cumplir que:

- Para todo $a, b \in \mathbb{Z}_N$, $f(a +_N b) = f(a) +_{N^2} f(b)$, y
- para todo $a, b \in \mathbb{Z}_N$, $f(a \cdot_N b) = f(a) \cdot_{N^2} f(b)$

Ejemplo: Para la configuración de Paillier que realizamos definimos $p = 11 yq = 3$, para de esta forma obtener $N = 33$ y entonces $N^2 = 1089$. Asi los textos claros pertenecen al dominio $\mathbb{Z}_{33}$ y los textos cifrados al codominio $\mathbb{Z}_{1089}$. Chequeemos el homomorfismo con elementos arbitrarios:

2.3.4. Firma Digital utilizando MPC: (TSS $\{2-2\}$)

Cada vez que un usuario desea iniciar una transacción, se crea una firma para que todos los demás usuarios del protocolo MPC la verifiquen y la aprueben. Si la firma es válida, se aprueba la instrucción. De lo contrario, si la firma no es válida, la instrucción no se aprueba y hay algo sospechoso en el usuario.

Por lo general, una billetera MPC utiliza ECDSA para las firmas. A partir de los procesos de generación y distribución de claves de SSSS (Shamir Secret Sharing Scheme), la clave privada se mantiene en secreto y un atacante no puede recuperarla. La firma ECDSA se publica para todas las partes en el protocolo MPC, y todos pueden verificarla.

1. Seleccione un punto G generador (que pertenezca a E) de orden n.
2. Seleccione aleatoriamente un número k en el intervalo $[1,\ n-1]$.
3. Calculamos r como x-coordenada de $P = kG$.
4. Calculamos $s = k^{-1} \cdot (H(M) + r \cdot d)$ donde d es la clave privada y $H(M)$ es el hash del mensaje.
5. Obtenemos la firma como (r, s).

Cualquier persona con la clave pública con respecto d puede utilizar los valores de r y s para validar la firma del mensaje. Note que r y s son enteros tal que $0 < r < n$ y $0 < s < n$.

En el caso de la firma ECDSA, en una configuración de umbral, los parámetros se modifican un poco. En un esquema de firma ECDSA de dos partes, ambas partes tienen el mensaje M que se va a firmar y el punto generador G público en la curva elíptica. La clave pública Q conjunta se crea a partir de un intercambio de claves de curva elíptica con Diffie-Hellman (ECDH) en el que ambas partes calculan sus propios secretos y obtienen un punto compartido. Este proceso lo realizamos en el punto 3

La cantidad de bit del hash del mensaje no puede ser mayor al orden de la curva.

- **El protocolo de firma es el siguiente:**

1. La Parte 1 elige un k_1 aleatorio, calcula $R_1 = k_1 \cdot G$, y publica una prueba ZK sobre el conocimiento de k_1 junto con R_1 a la Parte 2.

```
let mut rng = rand::thread_rng();
let k1 = BigInt::from(rng.gen_range(1..group_add.len()));
```

2. La Parte 2 elige un k_2 aleatorio, calcula $R_2 = k_2 \cdot G$, y publica una prueba ZK sobre el conocimiento de k_2 junto con R_2 a la Parte 1.

```
let k2 = BigInt::from(rng.gen_range(2..group_add.len()));
```

   Es necesario que la x-coordenada de R_1 y R_2 sean diferentes de 0, módulo orden de la curva, sino vuelva al punto uno. Se requiere una prueba ZK para garantizar que los puntos R se calculen a partir de un múltiplo del generador G y no de un punto maliciosamente creado.

3. Suponiendo que cada prueba sea verificada por la parte correspondiente, la Parte 2 calcula $R = k_2 \cdot R_1 = k_1 \cdot k_2 \cdot G$ donde r es la x-coordenada de R.
4. Por la propiedad de homomorfismo del cifrado de Paillier, la Parte 2, dada c_{key}, calcula una serie de sumas y multiplicaciones de Paillier.
   a) La Parte 2 genera un número aleatorio $0 \leq \rho < n^2$, donde n es el orden de la curva. Además aseguramos que $gcd(\rho, N) = 1$ y usa la clave pública para calcular

$$c_1 = Enc(\rho \cdot n + k_2^{-1} \cdot H(M)) = Enc(\rho \cdot 11 + k_2^{-1} \cdot H(M))$$

```
let mut hash_message_p2 = Sha256::new();
hash_message_p2.update(message.as_bytes());
let hash_result_p2 = hash_message_p2.finalize();
let hash_message_p2_to_ec = BigInt::from_bytes_be(
    num_bigint::Sign::Plus,
    &hash_result_p2) %
    BigInt::from(2_u32.pow((&group_add.len() + 1) as u32)
);

let mut rho = paillier_key_p1.public_key.1.clone();
while gcd(&rho, &paillier_key_p1.public_key.1) != BigInt::one() {
  rho = BigInt::from(rng.gen_range(0..(group_add.len() + 1).pow(2)));
}
let inv_k2 = inv_mod(&k2, &BigInt::from(group_add.len() + 1));
let c1 = cipher_paillier(
  &paillier_key_p1.public_key,
  &((&rho * &BigInt::from(group_add.len() + 1)
  + inv_k2.as_ref().unwrap() * &hash_message_p2_to_ec)
  % BigInt::from(group_add.len() + 1)),
);
```

b) La Parte 2 calcula $v = k_2^{-1} * r * d_2$ y lo multiplica por c_{key} (versión cifrada de d_1) para obtener c_2.

```
let v = (inv_k2.as_ref().unwrap() * &new_ec.scalar_mul(
  &point_r1,
  &k2).x * &key_pair_2.as_ref().unwrap().sk)
  % BigInt::fro (group_add.len() + 1
);
let c2 = ckey.as_ref().unwrap().modpow(
  &v,
  &paillier_key_p1.public_key.1.pow(2)
);
```

c) La Parte 2 calcula un $c_3 = c_1 + c_2$.

```
let c3 = (c1.as_ref().unwrap() * &c2).modpow(&BigInt::one(),
  &paillier_key_p1.public_key.1.pow(2)
);
```

d) La Parte 2 envía c_3 a la Parte 1.

e) La Parte 1 calcula $R = k_1 \cdot R_2$ y extrae r donde r es la x-coordenada de R. Esto debería dar como resultado el mismo R que en el Paso 3. La Parte 1 descifra c_3 para obtener $s^{'}$ y calcula $s = k_1^{-1} \cdot s^{'}$. Luego, la firma se publica en la forma (r, s). Si s es igual a 0 vuelva al punto 1.

```
let mut s = decipher_paillier(
  &paillier_key_p1.private_key,
  c3,
  &paillier_key_p1.public_key
);
let r = new_ec.scalar_mul(&point_r2, &k1).x
      % BigInt::from(group_add.len() + 1);

s = (s * inv_mod(&k1, &BigInt::from(group_add.len() + 1)).unwrap())
      % BigInt::from(group_add.len() + 1);
```

2.3.5. Verificar la firma

Todas las partes en el protocolo MPC tienen acceso a la clave pública, publicada por el usuario para la firma. El proceso de verificación varía según el algoritmo de firma, pero cada usuario puede

verificar la firma de forma individual utilizando la clave pública, publicada. Para ECDSA, siempre que la firma y el mensaje se publiquen, cualquiera con los valores públicos puede verificar la validez de la firma. Tanto en el protocolo ECDSA normal como en el de dos partes, el algoritmo de verificación es el mismo. El algoritmo es el siguiente:

1. Calcular $H(M)$ con la misma función hash del algoritmo de firma.
2. Calcular $u_1 = H(m) * s^{-1}$.
3. Calcular $u_2 = r * s^{-1}$.
4. Si r es la x-coordenada de $G * u_1 + Q * u_2 = r$, entonces la firma está verificada. De lo contrario, la firma es inválida. Aquí, $Q = G * d$ donde d es la clave privada utilizada para firmar el mensaje M.

```
let mut hash_message_verifier = Sha256::new();
hash_message_verifier.update(message.as_bytes());
let hash_result_verifier = hash_message_verifier.finalize();
let hash_message_verifier_to_ec = BigInt::from_bytes_be(
    num_bigint::Sign::Plus,
    &hash_result_verifier)
    % BigInt::from(2_u32.pow((&group_add.len() + 1) as u32)
);

let u1 = (hash_message_verifier_to_ec * inv_mod(&s, &BigInt::from(group_add.len()
       + 1)).unwrap()) % BigInt::from(group_add.len() + 1);
let u2 = (&r * inv_mod(&s, &BigInt::from(group_add.len() + 1)).unwrap())
       % BigInt::from(group_add.len() + 1);
let x = new_ec.point_add(
      &new_ec.scalar_mul(&points_g[0], &u1),
      &new_ec.scalar_mul(&point_g, &u2)
).x;

if &r == &(x % BigInt::from(group_add.len() + 1)) {
  println!("    The signature ({:?}, {:?}) is correct...", &r, &s)
} else {
  println!("    The signature ({:?}, {:?}) is incorrect...", &r, &s)
}
```

Referencias

[1] Yehuda Lindell. Fast secure two-party ecdsa signing. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part II 37*, pages 613–644. Springer, 2017.

[2] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.