UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE UNA RED DE MEZCLA UTILIZANDO ENCRIPTACIÓN DE PAILLIER PARA VOTACIONES ELECTRÓNICAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

PABLO VICENTE ASTUDILLO QUINTERO

PROFESOR GUÍA:
TOMÁS BARROS ARANCIBIA

MIEMBROS DE LA COMISIÓN:
ÉRIC TANTER
EDUARDO RIVEROS ROCA

SANTIAGO DE CHILE
2023

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: PABLO VICENTE ASTUDILLO QUINTERO
FECHA: 2023
PROF. GUÍA: TOMÁS BARROS ARANCIBIA

IMPLEMENTACIÓN DE UNA RED DE MEZCLA UTILIZANDO ENCRIPTACIÓN DE PAILLIER PARA VOTACIONES ELECTRÓNICAS

Una votación electrónica es un sistema de votación que utiliza medios electrónicos para registrar y contar los votos. A menudo se usa como una alternativa a los métodos tradicionales de votación en papel, que pueden llevar mucho tiempo y ser susceptibles a errores. El voto electrónico se ha vuelto cada vez más popular en los últimos años, y muchos países y organizaciones lo utilizan para realizar elecciones y otros tipos de votaciones. Si bien se ha demostrado que la votación electrónica es rápida y eficiente, también ha sido objeto de controversia debido a dudas respecto a la seguridad de los sistemas.

El objetivo del trabajo realizado para esta memoria es la implementación de un sistema complementario para una votación electrónica que utiliza un sistema de encriptación de Paillier, el cual permite realizar una mezcla verificada matemáticamente de los votos con el fin de contar los votos de manera análoga a como se realizan en las votaciones tradicionales.

Para ello se implementó una red de mezcla empleando la infraestructura de *Amazon Web Services* con una arquitectura "sin servidores", mediante contenedores de cómputo *Lambda* organizados por una máquina de estados.

El rendimiento de la red fue probado con un número variable de votos y su funcionamiento fue validada mediante pruebas unitarias, monitoreo de los datos y pruebas con implementaciones inválidas de la mezcla para demostrar la robustez de la verificación.

Luego de obtener los resultados del rendimiento, se propuso e implementó una versión distribuida de la red que permitió escalar la red mediante la paralelización del trabajo.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La votación electrónica se entiende como una votación o referéndum donde se utilizan medios electrónicos para la emisión y/o conteo de los votos. Existen diversas variantes de votaciones electrónicas como presencial, remota, mixta, remota-presencial, con respaldo en papel, tarjetas perforadas, etc. En el contexto de esta memoria nos referiremos como "votación electrónica" específicamente a aquella en que el voto se emite electrónicamente y se envía por Internet a uno o varios servidores donde se realiza electrónicamente el escrutinio y conteo una vez finalizada la votación.

Un aspecto central de las votaciones electrónicas, al igual que las en papel, es garantizar el secreto del voto, ya que este permite al votante votar libremente, sin coerción ni colusión. Se distinguen en la actualidad dos técnicas principales para garantizar el secreto del voto en las votaciones electrónicas:

- *Encriptación homomórfica:* Se refiere a un tipo de encriptación que permite computar sobre valores encriptados, sin necesidad de desencriptar individualmente cada valor. En el caso de una votación, esto significa que es posible computar el resultado total de la votación sin necesidad de desencriptar cada voto por separado.

- *Red de mezcla (mix-net):* Se basa en la operación de baraja re-encriptada o *re-encrypted shuffle*, cuyo resultado sobre una lista de valores encriptados es una permutación aleatoria de esta, pero cada valor es re-encriptado de forma que no es posible asociar los valores antiguos con los nuevos. Es una red, ya que esta operación se repite múltiples veces en una cadena de nodos o servidores independientes conectados en forma de lista enlazada, cada salida pasa a ser una entrada del siguiente nodo. Para verificar que cada salida corresponde efectivamente a una permutación de la entrada, cada operación también incluye una prueba matemática en su salida, la cual es validada por el siguiente nodo en la red.

Hay sistemas que usan encriptación homomórfica (como EVoting [1]), otras redes de mezcla (votaciones públicas en Estonia [2]) y otros que usan ambos (Helios Voting [3]).

EVoting es una empresa que ofrece un servicio de votación electrónica a múltiples clientes usando encriptación homomórfica de Paillier. El algoritmo de Paillier es público, por lo que cualquier tercera parte podría verificar la suma homomórfica repitiendo la misma técnica y la desencriptación final se realiza generando las pruebas (*zero knowledge proof* o ZKP) de que fue ejecutada correctamente, por lo que es prueba suficiente de la correctitud. Sin embargo, ya sea por costumbre o legislaciones locales, algunos de sus clientes han pedido la posibilidad de

realizar un re conteo uno a uno, es decir, desencriptando cada uno de los votos y sumando sus valores en texto no cifrado. Esto requiere para mantener el secreto poder desligar cada voto del votante antes de la desencriptación individual, lo que hace necesaria la implementación de una red de mezcla para el sistema de EVoting.

1.1. Objetivo General

El objetivo general corresponde a aumentar la funcionalidad de una votación electrónica que utilice criptografía de Paillier para poder desasociar los votos de sus votantes, de manera que se puedan contar uno a uno sin quebrar la confidencialidad del voto.

1.1.1. Objetivos Específicos

Los objetivos específicos para la memoria fueron:

1. Lograr la mezcla de los votos de manera secreta.
2. Evaluar el rendimiento del algoritmo de mezcla a utilizar.
3. Validar el correcto funcionamiento de la mezcla de los votos.
4. Verificar la seguridad de la red de mezcla.
5. Implementar una red flexible (que permita ajustar su estructura con facilidad) y escalable de acuerdo al número de votos.

Capítulo 2

Antecedentes

En este capítulo, se explicarán los conceptos necesarios para la comprensión del problema y solución desarrollada en la memoria.

2.1. Definiciones y conceptos matemáticos

2.1.1. Permutación

Definición 2.1 *Dado un conjunto S, una permutación π de S corresponde a una función biyectiva de S sobre sí mismo, lo que resulta en un ordenamiento o secuencia lineal de los elementos en el conjunto S. La inversa de una permutación π corresponde a π^{-1}, la cual satisface $\pi^{-1}(\pi(i)) = i$, para todo $i \in S$.*

Definición 2.2 *Una matriz (A_{ij}) es una matriz de permutación si satisface lo siguiente para una permutación π:*

$$A_{ij} = \begin{cases} 0, & si\ \pi(i) = j \\ 1, & si\ no \end{cases}$$

2.1.2. Conjuntos

Definición 2.3 *Dado N un número natural, el conjunto de los enteros módulos N se define como $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$.*

Definición 2.4 *Dado N un número natural y gcd(a, b) la función del máximo común divisor entre a y b, el conjunto multiplicativo de enteros módulo N se define como $\mathbb{Z}_N^* = \{x \in \mathbb{Z}_N \mid gcd(x, N) = 1\}$, esto es, el conjunto de todos los números de $\mathbb{Z}_N$ coprimos de N.*

2.2. Criptografía

La criptografía es la práctica y estudios de técnicas que intentan mantener la confidencialidad de mensajes sensitivos frente a adversarios maliciosos. En el ámbito de las votaciones electrónicas, la criptografía toma un rol crucial para asegurar la privacidad y correctitud de los votos.

A continuación se explican algunos de los conceptos claves de la criptografía:

2.2.1. Encriptación y desencriptación

La encriptación se refiere a una técnica por la cual un texto plano o *plaintext* (en el contexto de esta memoria, un texto plano será uno o varios datos representados por un número entero grande) es codificado por lo general utilizando una llave secreta, para mantenerlo oculto en lo que se denomina un texto cifrado o *ciphertext*. La desencriptación es el proceso contrario, transforma el texto cifrado de vuelta a texto plano. Se referirá a la encriptación con la función E y a la desencriptación con la función D, de esta manera se cumple la igualdad $D(E(m)) = m$.

También se introduce la operación de reencriptación denotada con la letra R, esta toma de entrada un texto cifrado y emite otro texto cifrado distinto pero que desencripta al mismo texto plano del inicial, esto es $E(m) \neq R(E(m))$ y $D(E(m)) = D(R(E(m))) = m$.

2.2.1.1. Encriptación simétrica

La encriptación simétrica se refiere a los algoritmos de encriptación que utilizan una misma llave tanto para encriptar como para desencriptar un mensaje, como se puede observar en la figura 2.1.

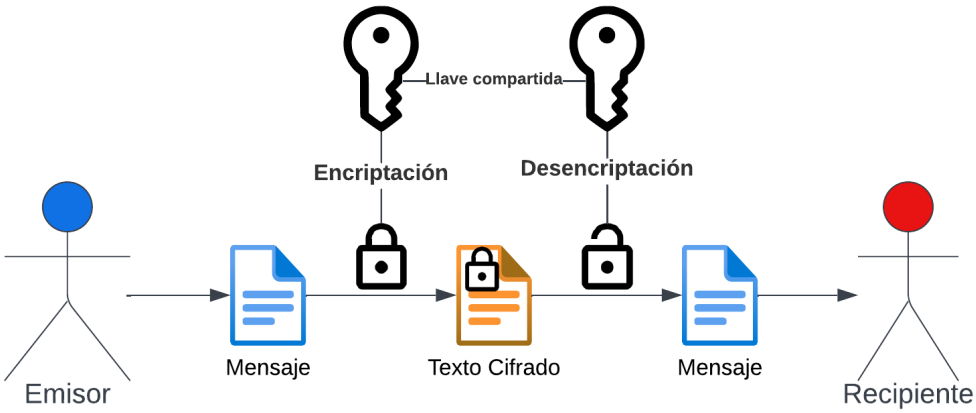

Figura 2.1: Ejemplo de un protocolo de encriptación simétrica

2.2.1.2. Encriptación asimétrica

La encriptación asimétrica se refiere a los algoritmos de encriptación que utilizan llaves para encriptar y para desencriptar, la llave para encriptar es pública de manera que cualquier entidad puede encriptar, pero la llave para desencriptar es privada y conocida solo por el recipiente del mensaje.

Figura 2.2: Ejemplo de un protocolo de encriptación asimétrica

2.2.1.3. Encriptación homomórfica

Los algoritmos de encriptación homomórfica permiten realizar operaciones sobre los textos cifrados sin necesidad de desencriptarlos, por ejemplo un algoritmo de encriptación homomórfico en la suma toma 2 textos cifrados y calcula un tercer texto cifrado mediante la operación homomórfica que al ser desencriptada, equivale a la suma, esto es:

$$x = E(m_1) \oplus E(m_2)$$
$$D(x) = m_1 + m_2$$

La operación homomórfica dependerá del algoritmo utilizado para encriptar los textos planos.

2.2.1.4. Desencriptación parcial

Algunos algoritmos generan un número de llaves privadas, cada una utilizada para desencriptar un texto cifrado de forma parcial, luego estas se combinan para obtener el texto plano original. Es posible configurar el algoritmo de manera que solo se requiera un subconjunto de las llaves privadas para desencriptar.

2.2.2. Mezcla verificable

Una mezcla es un procedimiento que toma como entrada una lista de n textos cifrados $T = (t_1, t_2, \ldots, t_n)$ y emite como salida otra lista de textos cifrados $T' = (t'_1, t'_2, \ldots, t'_n)$ de manera que:

- Existe una permutación π tal que $D(t'_i) = D(t_{\pi^{-1}(i)}), \forall i = 1 \ldots n$, donde D corresponde al algoritmo de desencriptación.
- No es posible relacionar T con T' sin conocimiento de la permutación π o del algoritmo de desencriptación.

Una mezcla es verificable si un probador P puede demostrar a un verificador V que T' es una mezcla de T sin revelar información sobre la permutación.

Existen varios protocolos de mezclas verificables que utilizan ElGamal como algoritmo de encriptación; Sako et al. [4], Abe [5] y Furukawa et al. [6]. Entre ellos, el más eficiente es Furukawa, el cual toma $18n$ exponenciaciones en comparación a $642n$ (Sako) y $22n \log n$ (Abe).

2.2.3. Red de mezcla o mixnet

Se refiere a una arquitectura criptográfica en la cual un número de servidores llamados mezcladores (*mixers*) se conectan en forma de lista enlazada, de manera que los cifrados de entrada de un mezclador corresponden a una mezcla verificable del anterior. Todas las comunicaciones son públicas y transparentes entre los servidores para evitar falsificación de los datos.

El verificador puede ser otro servidor o en conjunto todos los mezcladores pueden servir como verificadores, en cuyo caso se valida por la mayoría. Si el verificador no acepta una mezcla, luego se ignora la mezcla del mezclador corrupto y continua con la última mezcla válida.

Un ejemplo de red de mezcla utilizada en producción es Verificatum Mixnet [7], la cual implementa una red de mezcla distribuida, pero utiliza ElGamal como algoritmo de encriptación.

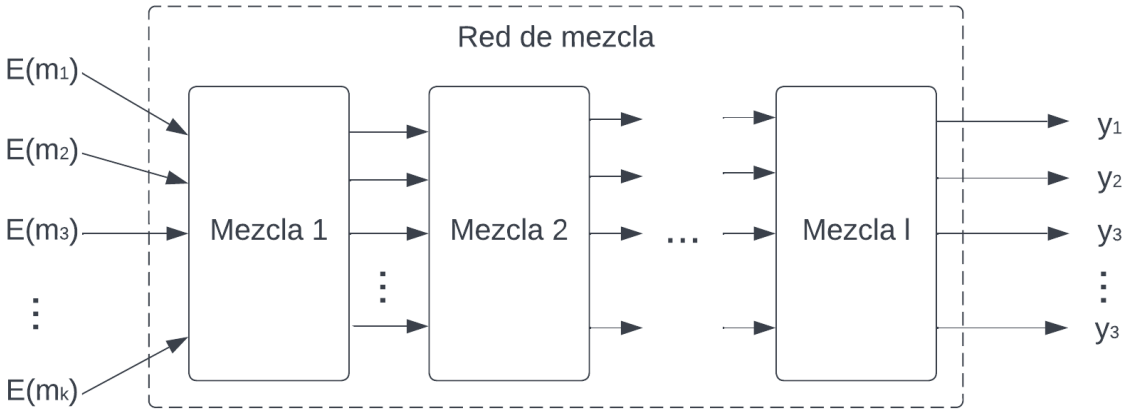

Figura 2.3: Una forma general de red de mezcla con k entradas y l mezclas

La ventaja de una red de mezcla por sobre un servidor que realiza una sola mezcla es que permite descentralizar el proceso de mezcla entre múltiples entidades, cada una encargada de realizar una mezcla y de verificar cada paso, de esta manera una sola entidad maliciosa no sería capaz de quebrar el anonimato guardando la información de permutación de forma secreta por ejemplo. Siempre y cuando la mayoría de las entidades actúen de forma correcta, la mezcla será válida, y no importa si alguna entidad guarda la permutación utilizada en su mezcla ya que necesitaría la información de todas las otras permutaciones para poder reconstruir el orden original de los votos.

2.3. Algoritmos criptográficos

2.3.1. Algoritmo de Rijndael

El cifrado Rijndael es un algoritmo de encriptación simétrica utilizado por AES (estándar de cifrado avanzado o *advanced encryption standard*). "Rijndael" es un acrónimo de los nombres de Joan Daemen y Vincent Rijmen, los inventores del algoritmo.

Rijndael utiliza un principio de diseño conocido como red de permutación de sustitución, o red-SP, para cifrar datos de texto sin formato. Tiene una descripción algebraica relativamente simple y se ejecuta de manera muy eficiente tanto en implementaciones de hardware como de software.

2.3.2. Algoritmo de ElGamal

ElGamal [8] es un algoritmo de encriptación asimétrica homomórfico en la multiplicación; es posible combinar valores encriptados en otro que corresponde al producto de estos. En el ámbito de las votaciones electrónicas, interesa la suma de votos, no el producto, por tanto, se modifica para que tome los votos al exponente, luego al desencriptar se calcula el logaritmo para obtener la suma de los votos. En caso de que el número de votos sea muy grande, esta última operación puede ser costosa.

Existen sistemas de votos electrónicos que utilizan ElGamal en la encriptación de votos, por ejemplo Helios Voting [3] de código abierto. Este también soporta redes de mezcla utilizadas en votación por ranking en la que es necesaria desencriptar los votos individuales para calcular el resultado de la votación.

2.3.3. Algoritmo de Paillier

Basado en el supuesto de Residuidad Compuesta, el algoritmo de Paillier [9] es un algoritmo de encriptación asimétrica homomórfico en la adición, lo que lo hace ideal para el conteo de votos porque, a diferencia de ElGamal, no es necesario hacer el cálculo extra del logaritmo. Se define de la siguiente manera:

- *Generación de las claves*: Sea $N = pq$, donde p y q son números primos grandes, y $\lambda = lcm(p-1, q-1)$. La llave pública es $pk = N$ y la llave privada es $sk = \lambda$.
- *Encriptación*: Sea el texto plano $m \in \mathbb{Z}_N$ y $r \leftarrow \mathbb{Z}_N^*$, luego el texto cifrado de m es $g = r^N(1+mN) \pmod{N^2}$.
- *Reencriptación*: Un texto cifrado g puede ser reencriptado como $g' = r'^N g \pmod{N^2}$ donde $r' \leftarrow \mathbb{Z}_N^*$.
- *Desencriptación*: Un texto cifrado g puede ser desencriptado como $m = L(g^\lambda \pmod{N^2})/\lambda \pmod{N}$, donde L toma su entrada desde el conjunto $\{u \in \mathbb{Z}_{N^2} \mid u = 1 \pmod{N}\}$ y se define como $L(u) = (u-1)/N$.

Como se mencionó, es homomórfico en la suma y para calcularlo se hace a través del producto de los textos cifrados:

$$D(E(m_1) \cdot E(m_2) \pmod{N^2}) = m_1 + m_2 \pmod{N}$$

2.3.4. Algoritmo de Furukawa

El algoritmo de Furukawa [6] es un algoritmo de mezcla verificable que se basa en la representación de la permutación como matriz definida en 2.2. Dada esta representación, luego deriva el siguiente teorema:

Teorema 2.1 *Una matriz (A_{ij}) es una matriz de permutación si y solo si para todo i, j y k, se cumple:*

$$\sum_{h=1}^{n} A_{hi}A_{hj} = \begin{cases} 1, & si\ i = j \\ 0, & si\ i \neq j \end{cases}$$

$$\sum_{h=1}^{n} A_{hi}A_{hj}A_{hk} = \begin{cases} 1, & si\ i = j = k \\ 0, & si\ no \end{cases}$$

La idea principal de la verificación radica en el cálculo de valores por parte del probador basados en la matriz (A_{ij}) y los cifrados T, T' y $C = (c_1, c_2, \ldots, c_n)$, este último correspondiente a un desafío aleatorio que el verificador le pasa al probador. Estos valores luego son utilizados por el verificador, el cual verifica que se cumplan una serie de ecuaciones basadas en el teorema 2.1. Para evitar filtrar información respecto de (A_{ij}) al verificador, se introducen números aleatorios en las ecuaciones que ofuscan información que pueda revelar la permutación.

El protocolo es completo en el sentido de que si P conoce la permutación, luego V siempre acepta la verificación. En caso de que P conozca información sobre los cifrados E desencriptados, entonces es posible que pueda calcular cifrados que cumplan las ecuaciones, pero que no correspondan a una permutación de E, es por esto que se introduce una lista de cifrados aleatorios $\tilde{T} = (\tilde{t_1}, \tilde{t_2}, \ldots, \tilde{t_n})$, luego P realiza el mismo procedimiento sobre $\tilde{T}$ con la misma matriz (A_{ij}), de esta manera la probabilidad de que V acepte una prueba que no corresponda a una permutación es despreciable.

Todo en conjunto se traduce en el siguiente protocolo (simplificado):

Entrada: N (llave pública), T (cifrados originales), T' (cifrados permutados), $\tilde{T}$ (cifrados aleatorios).

1. P genera valores aleatorios.
2. P calcula un conjunto de valores basados en los anteriores conocidos como el compromiso o *commitment*.
3. $P \rightarrow V$: P le entrega el compromiso a V.
4. $P \leftarrow V$: V le entrega un desafío aleatorio o *challenge* a P.
5. $P \rightarrow V$: P calcula y le entrega a V un conjunto de valores basados en los anteriores (incluido el desafío) conocidos como la respuesta o *response*.
6. V verifica que la respuesta junto con el compromiso cumplen una serie de ecuaciones.

2.3.5. Algoritmo de Nguyen

Nguyen et al. 2004 [10] utiliza ideas de Furukawa para formar un protocolo de mezcla verificable de 3 rondas que utiliza Paillier como algoritmo de encriptación.

El sistema completo de verificación de [10] es:

Definición 2.5 *Sea P el probador, V el verificador y n el número de mensajes a mezclar. P conoce una permutación π y $r_1, \ldots, r_n \in \mathbb{Z}_N^*$ tal que $t'_i = r_i^N t_{\pi^{-1}(i)}$. La entrada del sistema es N (llave pública), $\{t_i\}$ (cifrados originales), $\{t'_i\}$ (cifrados mezclados) y $\{\tilde{t}_i\}$ (cifrados aleatorios), $i = 1, \ldots, n$. La permutación es representada por una matriz de permutación $(A_{ij})_{n\times n}$. (P, V) forman un sistema de verificación:*

1. *P genera: $\alpha_i \leftarrow \mathbb{Z}_N$ y $\alpha, \tilde{r}_i, \tilde{\alpha}, \delta_i, \rho, \rho_i, \tau, \tau_i \leftarrow \mathbb{Z}_N^*, i = 1, \ldots, n$.*

2. *P calcula el compromiso:*

$$\tilde{t}'_i = \tilde{r}_i^N \prod_{j=1}^{n} \tilde{t}_j^{A_{ji}}; \quad \tilde{t}' = \tilde{\alpha}^N \prod_{j=1}^{n} \tilde{t}_j^{\alpha_j}; \quad t' = \alpha^N \prod_{j=1}^{n} g_j^{\alpha_j}$$

$$\dot{t}_i = \delta_i^N (1 + N \sum_{j=1}^{N} 3\alpha_j A_{ji}); \quad \dot{v}_i = \rho_i^N (1 + N \sum_{j=1}^{N} 3\alpha_j^2 A_{ji});$$

$$\dot{w}_i = \tau_i^N (1 + N \sum_{j=1}^{n} 2\alpha_j A_{ji}); \quad \dot{v} = \rho^N (1 + N \sum_{j=1}^{n} \alpha_j^3);$$

$$\dot{w} = \tau^N (1 + N \sum_{j=1}^{n} \alpha_j^2), \quad i = 1, \ldots, n$$

3. *$P \rightarrow V$: $\{\tilde{t}'_i\}, \tilde{t}', t', \{\dot{t}_i\}, \{\dot{v}_i\}, \dot{v}, \{\dot{w}_i\}, \dot{w}$.*

4. *$P \leftarrow V$: Desafío $\{c_i\}, c_i \leftarrow \mathbb{Z}_N$.*

5. *$P \rightarrow V$: Respuesta*

$$s_i = \sum_{j=1}^{n} A_{ij} c_j + \alpha_i \pmod N; \quad \tilde{s} = \tilde{\alpha} \prod_{i=1}^{n} \tilde{r}_i^{c_i} \tilde{t}_i^{d_i} \pmod N$$

$$s = \alpha \prod_{i=1}^{n} r_i^{c_i} e_i^{d_i} \pmod N; \quad u = \rho \prod_{i=1}^{n} \rho_i^{c_i} \delta_i^{c_i^2} \pmod N; \quad v = \tau \prod_{i=1}^{n} \tau_i^{c_i} \pmod N$$

$$d_i = (\sum_{j=1}^{n} A_{ij} c_j + \alpha_i - s_i)/N, \quad i = 1, \ldots, n$$

6. *V verifica:*

$$\tilde{s}^N \prod_{j=1}^{n} \tilde{t}_j^{s_j} = \tilde{t}' \prod_{j=1}^{n} \tilde{t}_j'^{c_j} \pmod{N^2}$$

$$s^N \prod_{j=1}^{n} g_j^{s_j} = t' \prod_{j=1}^{n} t_j'^{c_j} \pmod{N^2}$$

$$u^N (1 + N \sum_{j=1}^{n} (s_j^3 - c_j^3)) = \dot{v} \prod_{j=1}^{n} \dot{v}_j^{c_j} \dot{t}_j^{c_j^2} \pmod{N^2}$$

$$v^N (1 + N \sum_{j=1}^{n} (s_j^2 - c_j^2)) = \dot{w} \prod_{j=1}^{n} \dot{w}_j^{c_j} \pmod{N^2}$$

2.4. Implementación de votaciones electrónicas en EVoting

EVoting utiliza una modificación del algoritmo de Paillier basado en DamgardJ01 [11] para la encriptación de sus votos, que admite la desencriptación parcial de los textos cifrados, de manera que previo al comienzo de la votación, múltiples llaves privadas son generadas y distribuidas a un comité regulador y son reunidas una vez que se finaliza la votación para hacer el conteo de los votos.

Definiendo n como el número de candidatos y b como el rango en bits de cada voto (número máximo de votos que acepta un candidato) luego un voto es representado como un número de $(n+2)b$ bits, donde cada bloque de b bits representa una opción, los primeros 2 bloques son reservados para la opción de nulo y blanco y el resto corresponde a los n candidatos.

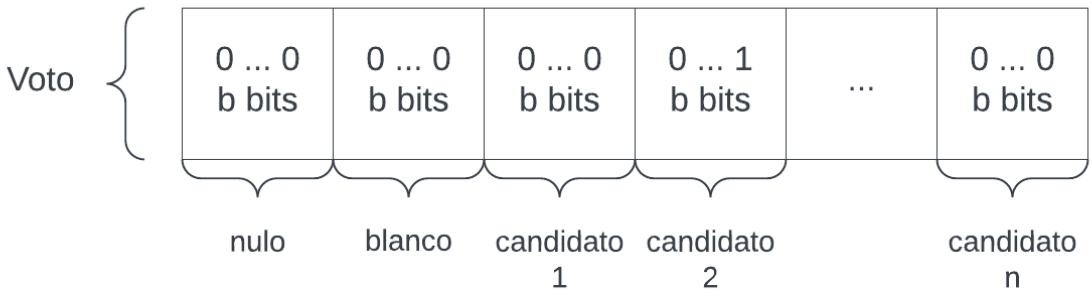

Figura 2.4: Ejemplo de un voto por el segundo candidato en una votación con n candidatos con rango de b bits por opción

Con esta representación se tiene la restricción de que solamente una opción o bloque debe tener el bit menos significativo encendido, correspondiente a la opción del votante. Luego el voto es encriptado con el algoritmo de Paillier junto a una prueba matemática conocida como *prueba de cero conocimiento* o *zero-knowledge proof* que demuestra que el voto emitido es válido (exactamente 1 voto por 1 opción). El texto cifrado es guardado en la base de datos de EVoting hasta el momento en que sea combinado homomórficamente para el conteo de los votos.

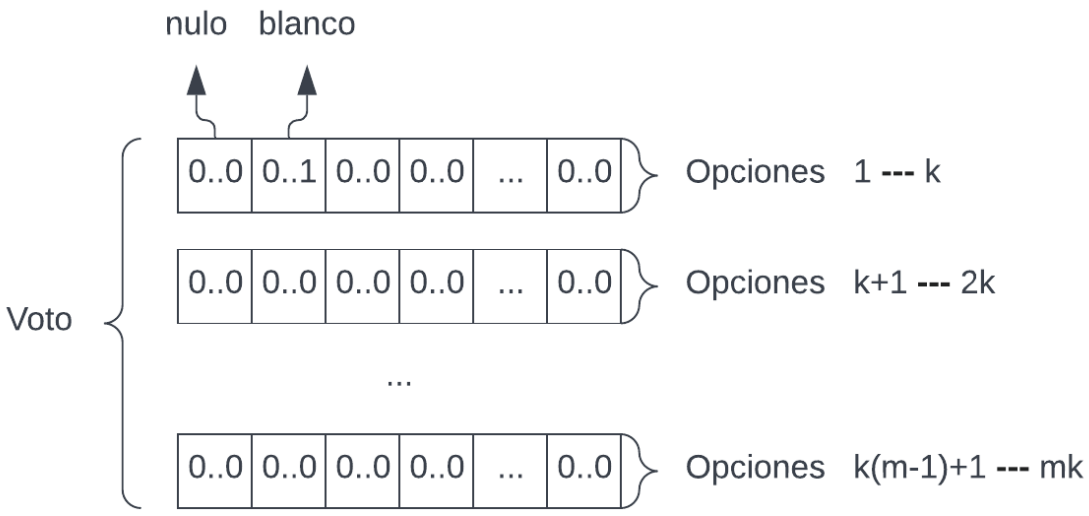

Figura 2.5: Ejemplo de un voto blanco en una votación con mk opciones

En caso de que sean muchos candidatos, el voto es dividido en múltiples textos cifrados, cada uno representando un número de opciones, como se puede observar en la figura 2.5 donde k es el número de opciones por parte del voto y m el número total de partes.

Al momento de hacer la combinación homomórfica de los votos, cada bloque en el nuevo texto cifrado corresponderá al número total de votos por esa opción, siempre y cuando el número de votos por la opción no sobrepase el rango admitido.

2.5. Arquitectura de nube por Amazon Web Services

Amazon Web Services (AWS) es la plataforma en la nube que ofrece Amazon, la cual ofrece diversos productos y servicios de cómputo por la nube. La computación en la nube es la entrega bajo demanda de poder de cómputo, base de datos, almacenamiento, aplicaciones y otros recursos de TI a través de una plataforma de servicios en la nube a través de Internet con un modelo de pago donde solo se paga por el uso de los servicios en tiempo real (número de transacciones en una base de datos por ejemplo).

La arquitectura "sin servidores" o *serverless* se refiere a un modelo de aplicaciones o servicios en el que el manejo de los servidores e infraestructura detrás es administrado por un tercero, en este caso Amazon. Los servidores son manejados de acuerdo a su uso y demanda, con la posibilidad de escalar automáticamente de acuerdo a las necesidades.

A continuación se explican algunos de los servicios de AWS utilizados en la solución.

2.5.1. Amazon Lambda

En AWS, una función *lambda* se refiere a la unidad básica de cómputo de una arquitectura serverless, es un servicio que permite ejecutar código a través de un contenedor con los recursos especificados para esa lambda, el cual es inicializado al momento en que se ejecuta la lambda en lo que se llama "inicialización fría" o *cold-start* (este depende del lenguaje

utilizado, para Scala es de alrededor de unos 20 s), luego de la ejecución de la lambda el contenedor es descartado o reutilizado por otra lambda (AWS se encarga de la administración de los servidores).

Una lambda es definida por:

- Código de programación.
- Ambiente de programación (C++, nodeJS, Java, Python, .NET, Go, Ruby).
- Memoria RAM disponible (máximo 3GB o 10GB para clientes especiales).
- Tiempo de ejecución (máximo 15 minutos).
- Eventos que la gatillan (solicitudes por HTTP, evento programado cada cierto tiempo, inserción a una tabla de una base de datos, etc.).

2.5.2. Amazon Simple Storage Service

Simple Storage Service o *S3* es un servicio que proporciona almacenamiento de objetos, diseñado para almacenar y recuperar cualquier cantidad de información o datos desde cualquier lugar a través de internet. Los objetos se guardan en baldes o *buckets* que funcionan como carpetas en un directorio, cada bucket tiene un nombre universalmente único y los objetos se obtienen mediante su nombre o llave, similar a una base de datos NoSQL.

2.5.3. Amazon Step Functions

Step Functions es un servicio de orquestación que integra Amazon Lambda junto con otros servicios de AWS por medio de una máquina de estados, cuyas tareas (*tasks*) y transiciones son definidas por un archivo JSON que sigue la estructura de *Amazon States Language*, a través de step functions se podría definir por ejemplo un flujo de trabajo para el entrenamiento de una red neuronal o la extracción y carga de datos desde una fuente externa.

2.5.4. Amazon Serverless Application Model

El modelo de aplicación sin servidor de AWS (SAM) es un framework de código abierto para crear aplicaciones sin servidor. Proporciona una sintaxis abreviada para expresar funciones, APIs y bases de datos, entre otros. Con solo unas pocas líneas por recurso, se puede definir la aplicación que desea y modelarla usando YAML o JSON. Durante la implementación, AWS SAM transforma y expande la sintaxis de SAM en la sintaxis de AWS CloudFormation, el servicio de modelación e implementación de los recursos en AWS.

Capítulo 3

Descripción e implementación de la solución

3.1. Descripción del problema

Como se mencionó en la sección anterior, EVoting utiliza la estrategia de encriptación homomórfica usando encriptación de Paillier para garantizar el secreto de los votos en sus votaciones; mediante la combinación homomórfica, los votos no necesitan ser desencriptados uno a uno para obtener el resultado de la votación total, pero esta modalidad no es compatible con los requerimientos de algunos de sus clientes que requieren realizar el conteo de los votos como se hace tradicionalmente con las votaciones de papel.

Técnicamente, EVoting puede usar las llaves privadas reunidas al momento de hacer el escrutinio y desencriptar cada voto por separado, pero esto quebraría la confidencialidad del voto, alguien podría argumentar de que EVoting tendría la capacidad de asociar la identidad del votante junto con el resultado del voto.

3.2. Descripción general de la solución

Para solucionar el problema, se requiere pasar los votos a través de una red de mezcla para desasociar la identidad los votantes de sus votos.

La solución se dividió en 3 etapas:

1. Investigación e implementación de un algoritmo de mezcla verificable compatible con la encriptación de Paillier utilizada por EVoting.
2. Implementación de una red de mezcla utilizando el algoritmo implementado en la etapa anterior.
3. Implementación de una aplicación web que ejecute la red de mezcla y visualice la verificación de cada paso.

3.3. Requisitos de la solución

- La infraestructura de la red debe estar contenida en AWS.
- Implementación de la mezcla verificable debe hacerse en Scala, el mismo lenguaje en el que está implementado la versión del algoritmo de Paillier de EVoting.

3.4. Mezcla verificable

3.4.1. Investigación y evaluación de algoritmos

En el capítulo 2 se explicó el algoritmo de mezcla verificable de Nguyen, aparte de este se consideraron otros 2 algoritmos de mezcla de encriptación de Paillier; el algoritmo de Onodera [12] y el algoritmo de Nguyen con Paillier modificado [13].

Para dar contexto, Nguyen et al. 2004 [10] y Onodera et al. 2005 [12] independientemente utilizan ideas de Furukawa [6] para formar un protocolo de mezcla verificable de 3 rondas que utiliza Paillier como algoritmo de encriptación. Onodera hace la comparación y estima que su protocolo es más eficiente que el postulado por Nguyen. En 2005 Nguyen et al. [13] publican otro protocolo más eficiente que el que publicaron previamente utilizando una variación de Paillier, el cual permite optimizar el cálculo de las exponenciaciones.

Se decidió proceder con el algoritmo original de Nguyen publicado en 2004 ya que este no requería modificación del algoritmo de Paillier como los otros 2, lo cual tomaría más trabajo en la implementación al sistema de EVoting. En caso de serlo necesario, se usaría el algoritmo de Nguyen con Paillier modificado si la ejecución no sea lo suficientemente rápida.

3.4.2. Implementación

El código se hizo sobre la implementación de Paillier de EVoting, escrito en Scala, utilizando la librería *Cats Effect* para el manejo de código asíncrono y efectos secundarios y la librería *ScalaTest* para las pruebas unitarias. Algunas de las clases relevantes de la implementación de Paillier de EVoting son:

- *PlainMessage*: Representa un texto plano, es de tipo *BigInt*.
- *EncryptedMessage*: Representa un texto cifrado, es de tipo *BigInt*.
  - *Ciphertext*: Extiende a *EncryptedMessage*, corresponde al dato de texto cifrado.
  - *CiphertextWithRandomness*: Extiende a *EncryptedMessage*, incluye el valor aleatorio usado para generarlo de tipo *BigInt*.
- *KeyGenerator*: Objeto que genera la llave pública y llaves privadas.
  - *PublicKey*: Representa una llave primaria, se construye a partir de un número de tipo *BigInt*.
  - *PrivateThresholdKey*: Representa una parte de llave privada, se construye a partir de un número de tipo *BigInt*.
- *PublicKey*: Representa una llave pública, se construye a partir de un número de tipo *BigInt*.
- *PaillierLike*: *trait* de las clases que implementan el algoritmo de Paillier, incluye los métodos *encrypt* (encripta un texto plano), *add* (combina homomórficamente 2 textos cifrados) y *combine* (toma las partes de llaves privadas junto con el texto cifrado y retorna su desencriptación).
  - *Paillier*: Implementa el *trait PaillierLike*, es la versión donde un voto es contenido en un solo texto cifrado como en la figura 2.4.

– *PaillierArbitrary*: Implementa el *trait PaillierLike*, es la versión donde un voto es separado entre varios textos cifrados como en la figura 2.5.

Antes de continuar con las clases implementadas, primero se debe aclarar un detalle importante de las particularidades de la mezcla, o sea qué ocurre con la representación del voto como lista de textos cifrados, ya que el algoritmo de mezcla utiliza solo una lista de cifrados como entrada, pero en este caso se quiere mezclar una lista de listas de cifrados.

Para arreglar esto, en vez de hacer todo en una sola mezcla, se divide el trabajo en múltiples mezclas utilizando la misma permutación, cada una correspondiente a una franja de candidatos por voto (la primera mezcla las primeras k opciones de todos los votos, la segunda las siguientes k, etc.) como se puede observar en la figura 3.1.

Ya que todas estas mezclas usan la misma permutación, la integridad de los votos permanece intacta; cada voto resultante de la mezcla corresponderá a un voto de entrada, la lista interna de los votos no es mezclada.

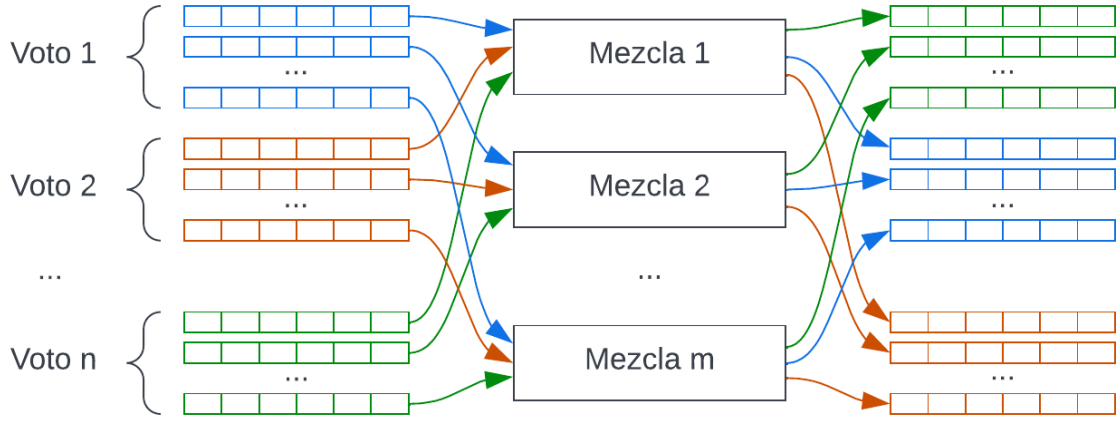

Figura 3.1: Mezclas con la representación de los votos como listas de cifrados

Por cada mezcla, hay una prueba matemática distinta de su validación, por tanto, se define que la mezcla en total es válida si y solo sí cada mezcla individual es válida.

Casi todas las clases implementadas tienen una versión "*arbitrary*" que indica que son usadas en el caso cuando los votos contienen múltiples cifrados, como en el caso de *PaillierArbitrary*. Las clases implementadas para la mezcla verificable son:

- *Permutation*: Corresponde a una permutación, representada por un vector de *Int*. Incluye el atributo *invertedPermutation* que corresponde a la inversa de la permutación utilizada en el protocolo de mezcla, su constructor por defecto genera una permutación al azar usando la clase criptográficamente segura *SecureRandom*.
- *ChallengeLike*: *trait* que representa al desafío aleatorio dado por el verificador V.
  - *Challenge*: Implementa el *trait ChallengeLike*, representado por un vector de *BigInt*, para ser usado con la implementación de la clase *Paillier*.
  - *ChallengeArbitrary*: Implementa el *trait ChallengeLike*, representado por un vector de *Challenge*, para ser usado con *PaillierArbitrary*.

- *VerifiableShuffle*: Clase que realiza la mezcla reencriptada de los votos, tiene 2 métodos: *shuffle* (toma un vector de textos cifrados y la permutación, para ser usado con la clase *Paillier*) y *shuffleArbitrary* (toma un vector de vectores de textos cifrados y una permutación, para ser usado con la clase *PaillierArbitrary*). La reencriptación se calcula sumando homomórficamente un 0 encriptado, esto es:

$$suma(E_1(m), E(0)) = E(m + 0) = E_2(m)$$

  La suma homomórfica de Paillier genera un texto cifrado distinto al original, pero que desencripta al mismo texto plano ($E_1(m) \neq E_2(m)$, $D(E_1(m)) = D(E_2(m))$).

- *ProverParams*: Clase que genera las variables aleatorias iniciales del probador P.

- *ShuffleProverLike*: *trait* que representa al probador en el protocolo de verificación de la mezcla, incluye los métodos *calculateCommitment* (calcula los valores del compromiso en la verificación, corresponde al primer paso de esta) y *respondChallenge* (toma el desafío generado por el verificador y calcula la respuesta, corresponde al tercer paso en la verificación).

  - *ShuffleProver*: Implementa el *trait ShuffleProverLike*, para ser usado con la clase *Paillier*.
  - *ShuffleProverArbitrary*: Implementa el *trait ShuffleProverLike*, para ser usado con la clase *PaillierArbitrary*, internamente utiliza una lista de *ShuffleProver* para manejar las mezclas.

- *ShuffleVerifierLike*: *trait* que representa al probador en el protocolo de verificación de la mezcla.

  - *ShuffleVerifier*: Implementa el *trait ShuffleVerifierLike*, para ser usado con la clase *Paillier*.
  - *ShuffleVerifierArbitrary*: Implementa el *trait ShuffleVerifierLike*, para ser usado con la clase *PaillierArbitrary*, internamente utiliza una lista de *ShuffleVerifier* para manejar las mezclas.

La interacción entre las clases en una mezcla completa se presentan en los fragmentos de código 3.1 para la mezcla simple y 3.2 para la mezcla arbitraria de votos.

Código 3.1: Ejecución de una mezcla verificable usando la clase *Paillier*.

```
...
val paillierSystem          = new Paillier(publicKey)
val permutation             = new Permutation(encryptedMessages.length)
val reencryptedMessages     = new VerifiableShuffle(paillierSystem)
                                .shuffle(encryptedMessages, permutation)
val proverParams            = new ProverParams(publicKey, encryptedMessages.length)

val proverSystem            = new ShuffleProver(publicKey,
                                                encryptedMessages,
                                                reencryptedMessages,
                                                publicCiphertexts,
```

```
                                        permutation,
                                        proverParams)

val verifierSystem          = new ShuffleVerifier (publicKey,
                                        encryptedMessages,
                                        reencryptedMessages,
                                        publicCiphertexts)

val commitment              = proverSystem.calculateCommitment()
val challenge               = verifierSystem.generateChallenge(commitment)
val response                = proverSystem.respondChallenge(challenge)
verifierSystem.verifyResponse(challenge, commitment, response) // true
```

Código 3.2: Ejecución de una mezcla verificable usando la clase *PaillierArbitrary*.

```
...
val paillierSystem          = new PaillierArbitrary(publicKey)
val permutation             = new Permutation(encryptedMessages.length)
val reencryptedMessages     = new VerifiableShuffle(paillierSystem)
                                .shuffleArbitrary (encryptedMessages, permutation)
val proverParams            = new ProverParams(publicKey, encryptedMessages.length)

val proverSystem            = new ShuffleProverArbitrary(publicKey,
                                        encryptedMessages,
                                        reencryptedMessages,
                                        publicCiphertexts,
                                        permutation,
                                        proverParams)

val verifierSystem          = new ShuffleVerifierArbitrary (publicKey,
                                        encryptedMessages,
                                        reencryptedMessages,
                                        publicCiphertexts)

val commitment              = proverSystem.calculateCommitment()
val challenge               = verifierSystem.generateChallenge(commitment)
val response                = proverSystem.respondChallenge(challenge)
verifierSystem.verifyResponse(challenge, commitment, response) // true
```

La ventaja de tener la lógica del probador P y el verificador V en clases separadas es que permite ejecutar las implementaciones en servidores distintos, aparte de evidenciar que efectivamente el verificador no contiene información sobre la permutación usada en la mezcla.

3.4.3. Optimización

Si bien la definición del protocolo utiliza la matriz de permutación $(A_{ij})_{n\times n}$ en los cálculos, no es necesario computar cada celda de la matriz (lo que haría que el protocolo sea de orden $O(n^2)$). Como cada fila y columna contiene un solo 1 y el resto 0, luego se puede optimizar aplicando las siguientes identidades:

$$a_i = \prod_{j=1}^{n} b_j^{A_{ij}} \implies a_i = b_{\pi(i)}$$
$$c_i = \prod_{j=1}^{n} d_j^{A_{ji}} \implies c_i = d_{\pi^{-1}(i)}$$
$$e_i = \sum_{j=1}^{n} f_j A_{ij} \implies e_i = f_{\pi(i)}$$
$$g_i = \sum_{j=1}^{n} h_j A_{ji} \implies g_i = h_{\pi^{-1}(i)}$$

3.4.4. Pruebas unitarias

Para probar el correcto funcionamiento de la mezcla, se crearon las siguientes pruebas unitarias usando *ScalaTest*:

- *Permutation* corresponde a una permutación válida (una permutación de largo n contiene todos los números de 0 a $n-1$).
- La reencriptación desencripta al mismo texto plano que la encriptación
- La mezcla generada por *VerifiableShuffle* corresponde a una permutación reencriptada de los textos cifrados de entrada.
- La mezcla generada por *VerifiableShuffle* es verificable mediante el algoritmo de Nguyen.
- La verificación de la mezcla falla al modificar un texto cifrado de entrada.
- La verificación de la mezcla falla al cambiar la permutación utilizada en la mezcla.

3.5. Red de Mezcla

3.5.1. Evaluación de soluciones

Como se mencionó en los requerimientos, la infraestructura de la red debe estar contenida en AWS, de aquí salen 2 posibles formas de abarcar el problema:

1. Tener una serie de servidores disponible 24/7 corriendo en máquinas virtuales del servicio de Amazon Compute Cloud (EC2), cada servidor tomaría un rol de probador y verificador, en cada ronda el primer servidor hace una mezcla y el resto de los servidores la verifica, en caso de que la mezcla pase la verificación los votos mezclados pasan al siguiente servidor, si no entonces se pasan los votos de entrada. Se necesitaría un servidor que coordine los servidores y que exponga una API para darle acceso a la red desde una aplicación web.
2. Manejar las distintas partes de la red a través de contenedores lambda, cada lambda ejecuta una función de la verificación. La coordinación de la red estaría manejada por Step Functions, básicamente una máquina de estados, la cual es manejada por AWS sin necesidad de tener un servidor.

Es difícil realizar una comparación exacta de los costos entre los 2 servicios ya que depende de múltiples factores como el número de invocaciones, tiempo por invocación y tipo de máquina. El costo de EC2 no es constante ya que toma en consideración la utilización de CPU por hora, mientras que Lambda hace cargos por invocación y cómputo por milisegundos.

Se optó por la segunda opción, ya que es más flexible si se desea agregar o eliminar más probadores o verificadores y es menos costosa porque solo se paga cuando es usada a diferencia de una máquina virtual. Dado que se usa lambda, entran en consideraciones sus limitaciones, en particular:

1. El tiempo límite de ejecución de 15 minutos.
2. La cantidad máxima de entrada y salida es de 6 MB.
3. El total de RAM disponible por lambda es de 3 GB.

Para lo primero, es difícil de estimar, ya que deben tomarse en cuenta el tiempo de *cold start* y la velocidad de la máquina prestada por AWS, cuya velocidad depende del valor de memoria, en caso de que el algoritmo de mezcla no sea lo suficientemente rápido, luego se optaría por la primera opción de utilizar máquinas virtuales

Para la segunda limitación sí existe una alternativa, en caso de ser necesario, que es pasar todas las variables a través de S3 (cada lambda descargaría las variables desde S3 al inicializarse y las actualizaría a S3 al término). Si bien el límite de entrada es 6MB, cada lambda tiene mucho más espacio de memoria RAM así que no hay problema por esa parte.

3.5.2. Comparación entre arquitectura serverless y tradicional

Si bien el uso de AWS fue requisito por EVoting para utilizar los recursos a su disposición, es importante reconocer las ventajas y desventajas entre una arquitectura sin servidores vs una tradicional con servidores:

Principales ventajas de serverless:

- Costos reducidos debido a que solo se paga por cómputo utilizado.
- Escalabilidad automática de recursos de acuerdo al uso.
- Mantenimiento de los servidores es manejado por Amazon.

Principales desventajas de serverless:

- Control limitado de los recursos ya que internamente son manejados por Amazon.
- Rendimiento no es constante, en particular en el caso de Lambda se tiene el problema de *cold start*.

En el caso de la red, el aspecto efímero de Lambda cuyos contenedores tienen un tiempo de vida corto es una ventaja en comparación con un servidor tradicional bajo el punto de vista de seguridad, el cual podría ser vulnerado si no se realiza el mantenimiento adecuado; da más control pero aumentan las responsabilidades del desarrollador. Todo esto asumiendo un nivel de confianza en la seguridad en los servidores de Amazon.

En términos de costo, dado que se requiere que cada entidad actúe de forma independiente, esto requeriría contratar 1 servidor por cada paso de mezcla en el caso del sistema tradicional, lo cual hace que la red sea poco flexible y costosa, en particular ya que el uso de la red es solo de forma esporádica en las votaciones que se requiera hacer la mezcla, luego se pagaría principalmente por el tiempo de inactividad de los servidores.

3.5.3. Serverless Application Model

Para la definición y carga de los recursos de AWS (lambdas, step functions, bucket de S3, permisos, etc.) se utilizó *Serverless Application Model* o SAM, un framework desarrollado por AWS en el cual cada recurso es definido en una plantilla de YAML o JSON siguiendo el formato de SAM, luego se puede crear los recursos usando el programa de comando SAM CLI con el comando *sam deploy*.

Un ejemplo de un recurso de lambda definido en SAM se encuentra en el código 3.3.

Código 3.3: Ejemplo de la definición de una lambda en SAM, con el ambiente de java8 y permisos para leer y escribir en un bucket de S3.

```
CommitmentFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: Commitment
      Runtime: java8
      Handler: lambda.commitment.CommitmentLambda::handle
      CodeUri: target/scala-2.13/scala.jar
      MemorySize: 3008
      Timeout: 900
      Policies :
         S3ReadPolicy:
            BucketName: !Ref ShuffleS3Bucket
         S3WritePolicy:
            BucketName: !Ref ShuffleS3Bucket
```

3.5.4. Step Functions

Step Functions es el servicio que permite orquestrar un sistema serverless, además de inicializar lambdas con los parámetros especificados, también puede modificar variables (sumar enteros) y hacer una decisión dependiendo del valor de una variable, esto permite hacer un ciclo o *loop* de la mezcla por medio de un iterador. Esto posibilita hacer una red de mezcla configurable, la cual toma como entrada:

- Ubicación de los votos encriptados en S3 (la carga de los votos es manejada por la API que se verá más adelante).
- Ubicación de los textos cifrados públicos en S3.
- Llave pública.
- Arreglo que indica las lambdas utilizadas en cada paso, estas serán:
  - Generar el compromiso (1.er paso).
  - Generador el desafío (2.º paso).
  - Responder al desafío (3.er paso).
- Listado de lambdas que validan la mezcla (4.º paso).

Esta flexibilidad de configuración de lambdas permite componer la red en *runtime*, con la posibilidad de tener múltiples implementaciones para cado paso en la mezcla verificable, las cuales podrían ser proveídas por distintas organizaciones, en vez de depender en la correctitud de una sola implementación.

Las lambdas implementadas para Step Functions fueron:

- *Commitment*: Realiza la mezcla reencriptada y calcula el compromiso por medio de *ShuffleProverArbitrary*.
- *Challenge*: Calcula el desafío por medio de *ShuffleVerifierArbitrary*.
- *Response*: Calcula la respuesta por medio de *ShuffleProverArbitrary*.
- *Verify*: Verifica la respuesta por medio de *ShuffleVerifierArbitrary*.
- *AggregateVerifications*: Junta todas las verificaciones en el 4.º paso de la mezcla, la verificación es decidida por mayoría.

Además de estas, se implementaron versiones de "adversario" para probar los casos en que intencionalmente un tercero malicioso quiera forzar una mezcla inválida por la red:

- *AdversaryReplaceCommitment*: Implementación del 1.er paso, intenta reemplazar el cifrado del voto con un voto fijo en el paso de cálculo de compromiso.
- *AdversaryShuffleCommitment*: Implementación del 1.er paso, intenta pasar una permutación distinta a la utilizada en la mezcla para el cálculo del compromiso.
- *AdversaryChallengeResponse*: Implementación del 3.er paso, intenta calcular la respuesta para un desafío distinto del pasado por el verificador.
- *AdversaryVerify*: Implementación del 4.er paso, entrega el opuesto a un verificador normal.

Un diagrama de la máquina de estado se puede ver en la figura 3.2 y la definición completa se encuentra en el anexo A.

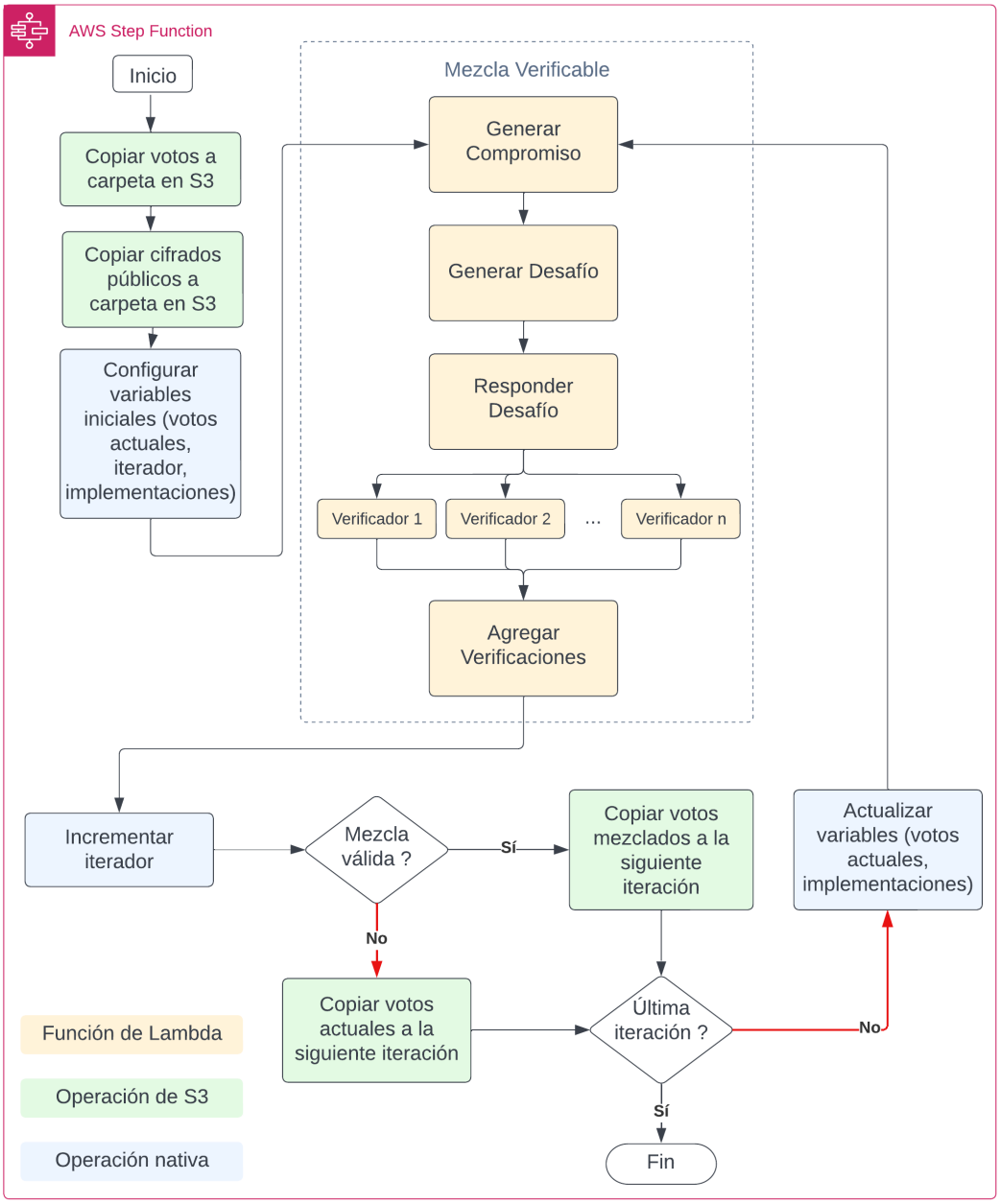

Figura 3.2: Máquina de estados de la red de mezcla

Como se mencionó, las variables son descargadas / cargadas desde S3 en la ejecución de cada lambda, pero un detalle de la implementación es que de alguna forma se debe pasar la variable de permutación entre la primera lambda *Commitment* y la tercera lambda *Response* para que esta pueda calcular la respuesta al desafío, pero guardar la permutación en S3 quiebra la seguridad de la mezcla, es por esto que se usa el algoritmo de AES de encriptación simétrica para guardar la permutación encriptada en S3, luego la tercera lambda la descarga, desencripta y posteriormente la elimina de S3.

Las variables se guardan en un bucket de S3, en un directorio correspondiente al identificador único de la mezcla, el cual es generado por Step Function. Por cada iteración se guardan las variables: desafío, compromiso, votos encriptados de entrada, paramétros iniciales, votos reencriptados, respuesta y verificación, todos en formato JSON. La estructura se puede observar en el diagrama 3.3.

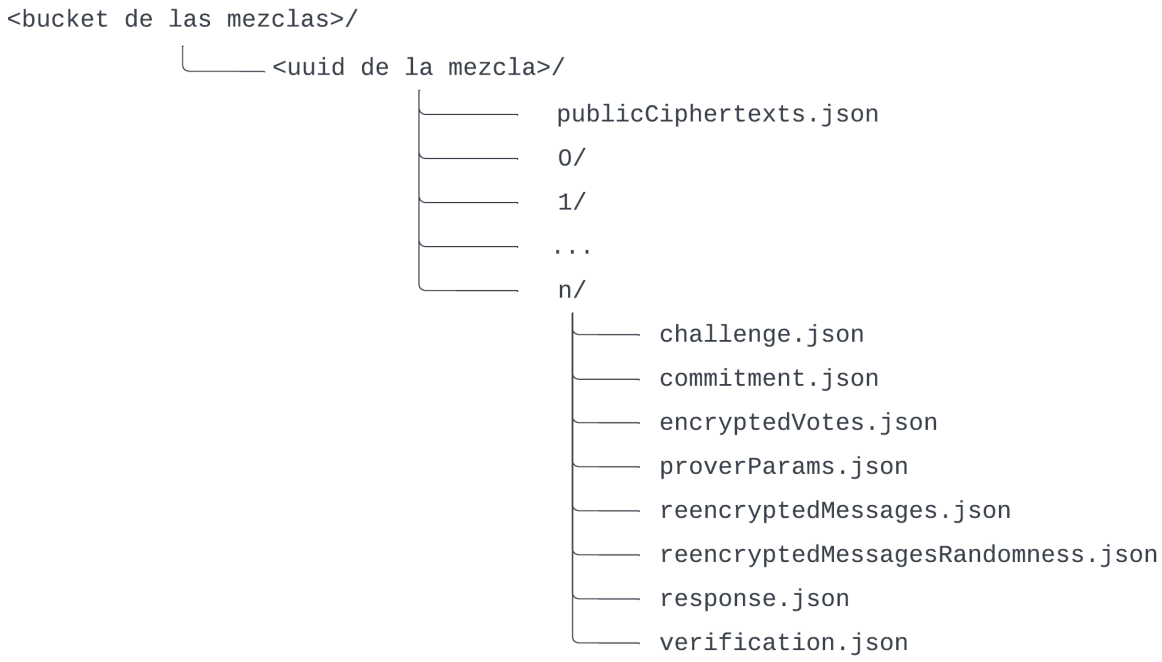

Figura 3.3: Estructura del bucket de S3

3.5.5. API

Para conectar la aplicación web a la red, se necesita una interfaz entre estas para iniciar la red y obtener el estado actual. Esta API solo requiere de 4 funciones o *endpoints*:

1. Iniciar la máquina de estado.
2. Obtener una carga a S3 prefirmada (esto es, una url asociada a un objeto en un bucket de S3 hacia donde cargar los votos / cifrados públicos).
3. Obtener el estado actual de la red.
4. Desencriptar los votos.

Fue implementada con lambdas programadas en Python (excepto la lambda que desencripta los votos, ya que utiliza la implementación en Scala), debido a que a diferencia de las lambdas en Scala no tiene un *cold-start* tan grande. Para conectar a los servicios de AWS se utilizó el SDK de AWS para Python, *boto3*. Las lambdas que conforman la API son:

- *InitiateStateMachine*: Inicia una ejecución de la red de mezcla.

- Tipo: POST.
- Cuerpo de entrada:
  - Ubicación en S3 de votos encriptados (*string*).
  - Ubicación en S3 de cifrados públicos (*string*).
  - Llave pública (*int*).
  - Nombres de las lambdas en cada paso de la mezcla (*array[object]*).
  - Nombres de las lambdas verificadoras (*array[string]*).
- Salida:
  - Identificador único de la mezcla (*string*).

Código 3.4: Ejemplo de entrada para la ejecución de una red con 2 mezclas y 3 verificadores

```
{
  "encryptedVotes": "presigned/54bO4AvGadm4jOY6AoiOoJv93x6nsiuQ.json",
  "publicCiphertexts": "presigned/tKuZLmPDBsXgszmZkugESyEPrEpIIkv8.json",
  "publicKey": "834397844726097161129608671165l...",
  "steps": [
    {
      "generateCommitment": "Commitment",
      "generateChallenge": "Challenge",
      "respondChallenge": "Response"
    },
    {
      "generateCommitment": "AdversaryReplaceCommitment",
      "generateChallenge": "Challenge",
      "respondChallenge": "Response"
    }
  ],
  "verifiers": [
    "Verify",
    "Verify",
    "AdversaryVerify"
  ]
}
```

- *GetPresignedPost*: Entrega una url para cargar un archivo en el bucket de S3, autorizada por 1 hora, debe ser usada para la carga de los votos y los cifrados públicos, ya que estos archivos son requeridos para iniciar la red. Los archivos quedan guardados en el directorio *presigned/* del bucket.
  - Tipo: GET.
  - Entrada: Vacía.
  - Salida:
    - Url autorizada para subir un archivo a S3 (*string*).
- *QueryStateMachine*: Entrega el estado actual de la máquina de estado, el cual lo obtiene desde el bucket de S3 en el cual se van guardando las verificaciones de cada mezcla.

  - Tipo: GET.
  - Parámetros de entrada:
    - Id único de ejecución de la red (*string*).
    - Marca que indica si debe generar una url a los votos reencriptados en S3 (*boolean*).
  - Salida:
    - Arreglo de los resultados de cada mezcla, incluye las verificaciones y la fecha y hora del término. opcionalmente, contiene una url a los votos mezclados (*array[object]*).
    - Estado de la máquina: corriendo, falla, terminado, cancelada (*string*).
    - Fecha de inicio (*string*).
    - Fecha de término, en caso de que la máquina haya terminado (*string*).

- *DecryptVotes*: Desencripta los votos, los guarda en un bucket en S3 y genera una url temporal de acceso a estos.
  - Tipo: POST
  - Cuerpo de entrada:
    - Id único de ejecución de la red (*string*).
    - Número de la iteración de la red (*string*).
    - Llaves privadas (*array[int]*).
    - Llave pública (*int*).

3.6. Aplicación Web

Finalmente, una simple aplicación web que consiste de 2 páginas: un formulario para ingresar los datos necesarios para iniciar la red y una segunda que muestra el estado actual de la red.

La aplicación fue creada con el framework React, usando el paquete *axios* para la conexión a la API y el framework *Ant Design* para la apariencia. Para el alojamiento, se usó el servicio de AWS Amplify.

La interacción entre la aplicación y la API sigue de la siguiente manera:

1. El usuario sube los archivos de votos y cifrados públicos, los cuales son cargados a S3 por medio de la url generada por *GetPresignedPost*.
2. El usuario llena el resto del formulario y envía la solicitud a *InitiateStateMachine*.
3. La aplicación envía solicitudes cada 30 s a *QueryStateMachine* para obtener el estado actual de la red.
4. Finalmente el usuario envía una solicitud junto con las llaves privadas para desencriptar los votos usando *DecryptVotes*.

En el anexo B se encuentra la interfaz de la aplicación, en esta se muestra la configuración y los resultados para una red de mezcla de 3 iteraciones (primera utiliza la implementación correcta de la mezcla y la segunda la versión del adversario) y 3 verificadores.

Toda la arquitectura en conjunto es representada en el diagrama dea la figura 3.4 y la interacción entre cada parte del sistema se encuentra en el diagrama de secuencias de la figura 3.5.

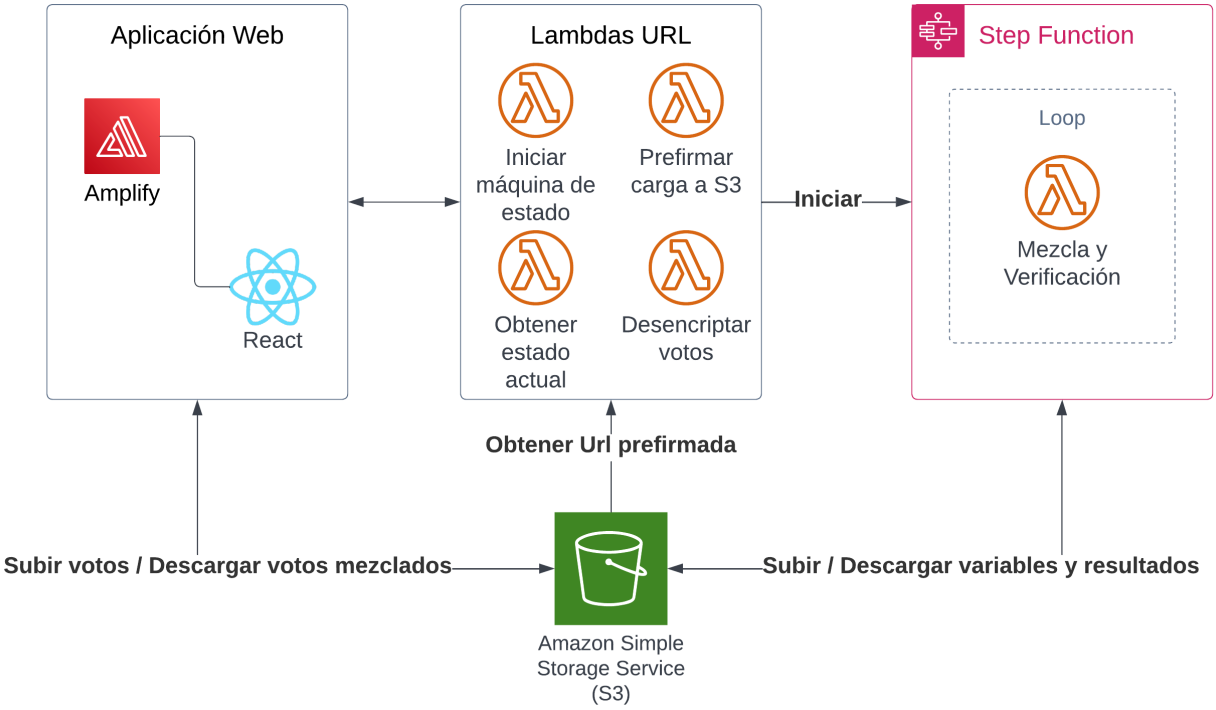

Figura 3.4: Diagrama de arquitectura del sistema

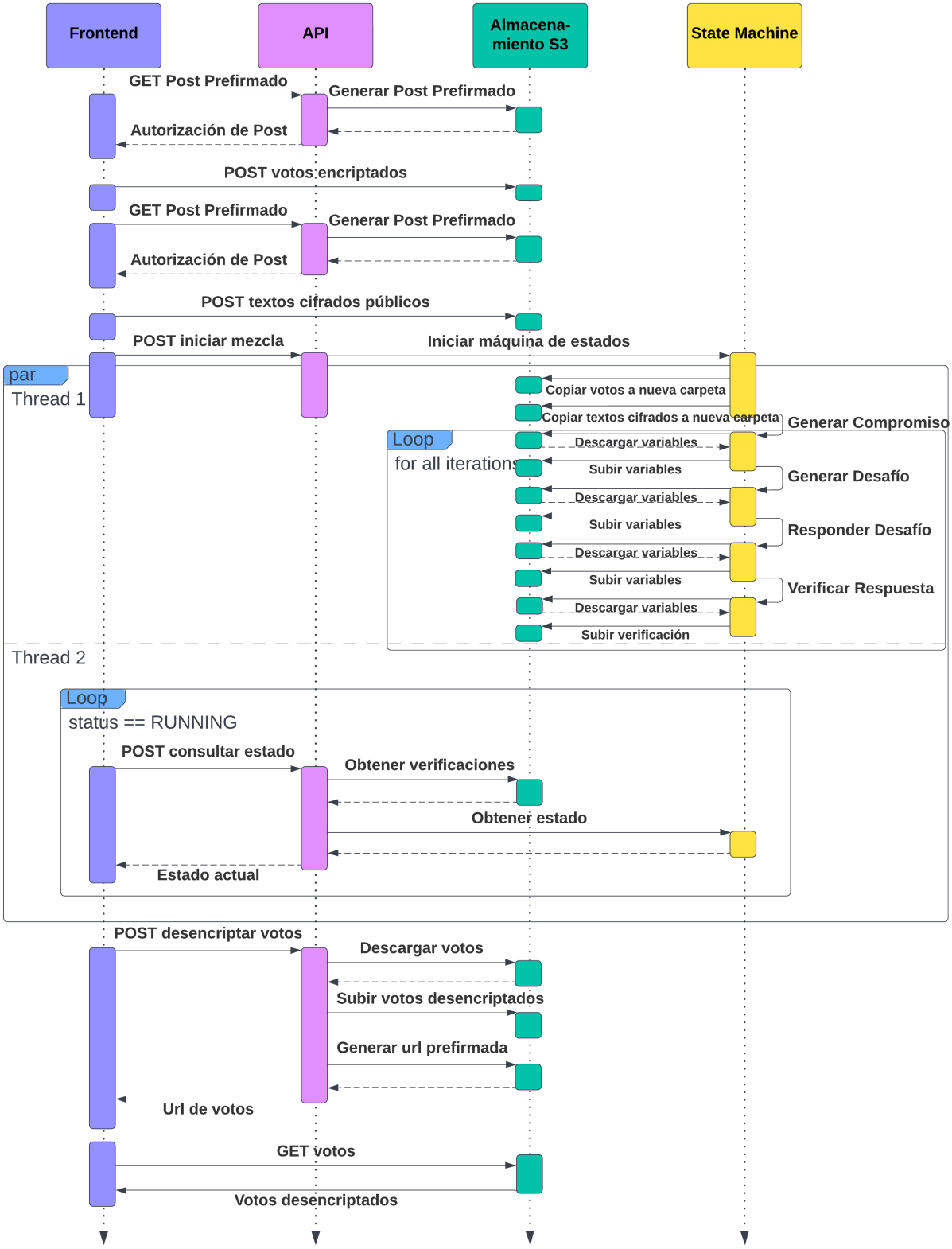

Figura 3.5: Diagrama de secuencias del sistema completo

3.7. Validación y rendimiento

La correctitud de la implementación de la mezcla verificable se validó mediante las pruebas unitarias hechas en ScalaTest, en estas se confirmó que la mezcla verificable efectivamente corresponde a una reencriptación de una permutación de los textos cifrados originales y la prueba de 3 rondas falla en caso que se modifique algún valor en el proceso (cambiando el valor de un texto cifrado o respondiendo con un desafío distinto).

La red de mezcla se probó utilizando cada combinación de implementaciones de la red (implementaciones válidas e implementaciones inválidas), los resultados se encuentran en la tabla 3.1.

Tabla 3.1: Pruebas de la red de mezcla con todas las combinaciones de implementaciones.

Generar Compromiso	Generar Respuesta	Resultado
Compromiso correcto	Respuesta correcta	Válido
Reemplazar voto	Respuesta correcta	Inválido
Cambiar permutación	Respuesta correcta	Inválido
Compromiso correcto	Cambiar desafío	Inválido
Reemplazar voto	Cambiar desafío	Inválido
Cambiar permutación	Cambiar desafío	Inválido

Adicionalmente, se revisó que la mezcla de los votos solo pasara al siguiente paso de la red en los casos en que la verificación fuera válida.

La limitante de las pruebas es que no aseguran que la red funciona en todas las posibles entradas, esta depende de la correctitud del algoritmo de Nguyen y en su correcta implementación, esta última también depende de las pruebas unitarias implementadas, entonces en estricto rigor la implementación no es 100 % confiable, lo cual no es irrelevante para un proceso tan crítico como lo podría ser una votación. Un posible camino para abordar este problema sería mediante código verificado formalmente, que aseguraría que el código funciona correctamente dada las especificaciones.

Para probar el rendimiento de la red se realizaron 2 conjuntos de pruebas: una utilizando una representación de voto encriptado como 1 cifrado de 2048 bits y el otro basados en una votación real, donde cada voto es representado por 4 cifrados de 4096 bits cada uno. La red consistió de una mezcla y un verificador, se realizaron pruebas cambiando la variable de número de votos. Una de las ventajas de separar cada paso en distintas lambdas es que permite monitorear el rendimiento de cada paso de la mezcla verificable. Los resultados se encuentran en la tabla 3.2 y figuras 3.6 y 3.7.

Tabla 3.2: Pruebas de velocidad de la red cuyos votos utilizan 4 cifrados de 4096.

Nº Votos	Generar Compromiso	Generar Desafío	Generar Respuesta	Validar Respuesta	Total
50	32s	7s	1s	32s	1m 12s
100	51s	2s	10s	53s	1m 56s
250	2m 25s	3s	25s	2m 12s	5m 5s
500	4m 43s	4s	48s	4m 23s	9m 58s
1000	14m 30s	15s	2m	8m 45s	25m 30s

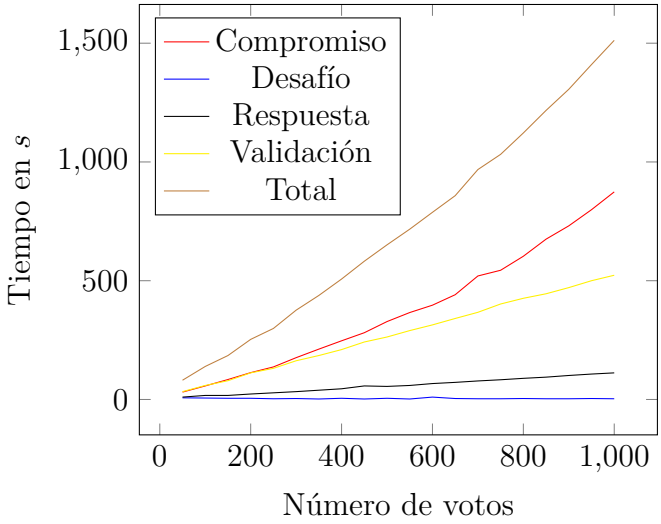

Figura 3.6: Rendimiento de la red con representación de voto como 4 cifrados de 4096 bits.

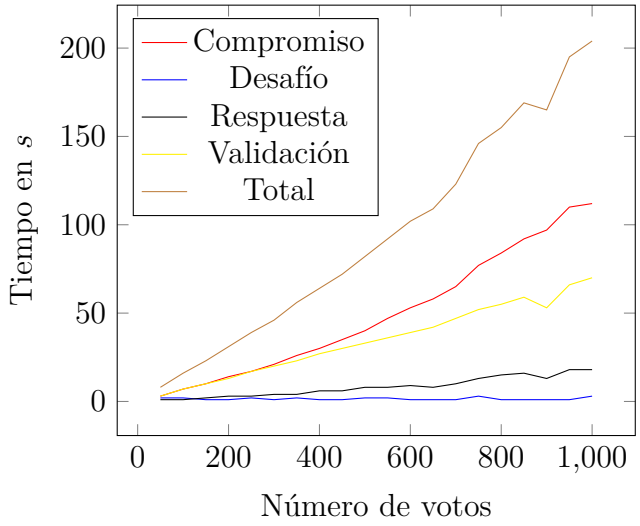

Figura 3.7: Rendimiento de la red con representación de voto como 1 cifrado de 2048 bits.

Las 2 operaciones más costosas corresponden a la generación del compromiso y la validación de la respuesta, la primera en particular ya que limita la cantidad a total a procesar a 1000 votos debido a la limitación de 15 minutos en la ejecución de una lambda en el caso de los votos de 4 cifrados de 4096 bits. No existe mayor diferencia relativa entre las operaciones al comparar con las 2 representaciones de los votos.

Adicionalmente, ya que las variables generadas son guardadas en S3, también se puede comparar la utilización de memoria en la tabla 3.3.

Tabla 3.3: Memoria utilizada en cada prueba.

Nº Votos	Peso de entrada	Peso Total
50	0.3 MB	2 MB
100	0.6 MB	4 MB
250	1.5 MB	10.1 MB
500	3 MB	20 MB
1000	6 MB	42.3 MB

De lo anterior se puede observar que el tamaño en memoria crece en forma lineal con una relación de alrededor de 7 a 1 con respecto a la entrada.

3.8. Mezcla distribuida

A partir de las pruebas, se observan 2 limitantes de la red: el tiempo y el tamaño total utilizado; el tiempo es la limitación dominante pero incluso si este no lo fuera, eventualmente el tamaño también lo sería debido al máximo de 3 GB de RAM por lambda, esto significa 2 cosas:

- Aunque se pruebe cambiando la implementación de la mezcla verificable con las alternativas más eficientes, dado que utilizan prácticamente la misma técnica en la mezcla verificable, se esperaría que tengan las mismas restricciones (osea no hay muchas posibilidades de escalabilidad).
- La alternativa de utilizar un servidor permanente en vez de lambda también tendría problemas; por ejemplo con 1000000 de votos, el peso crecería a unos 40 GB, asumiendo que el tiempo no sería problema si es que se aumenta la velocidad del procesador, lo cual es improbable.

Básicamente, el algoritmo no es eficiente si se desea utilizar con un gran número de votos, por lo tanto se evaluaron e implementaron 2 posibles mejoras:

- Primero fue utilizar la librería de colecciones paralelas de Scala, que reemplaza las implementaciones de colecciones normales por unas que procesan los datos en paralelo, esto en un intento de mejorar el rendimiento de los cálculos del compromiso y la verificación. Lamentablemente esto no resultó en una gran diferencia ya que los contenedores de lambda solo proveen 2 procesadores para el tamaño de 3 GB, por tanto no se benefician de programación paralela.

- La segunda posibilidad es cambiar el funcionamiento de la mezcla; en vez de mezclar todos los votos en un solo proceso, se dividen los votos en partes y se mezcla cada parte, de esta forma se puede distribuir el trabajo en múltiples mezclas que trabajen en paralelo, a esto se referirá como "mezcla distribuida".

La mezcla distribuida no corresponde a una mezcla total de los votos así que en ese sentido no se tiene ofuscación total de la relación de los votos, pero en términos prácticos para una votación, una mezcla distribuida si mantiene las propiedades deseadas de una mezcla completa, asumiendo una cantidad apropiada por mezcla.

Por ejemplo, dado 10000 votos de entrada para mezclar, estos se podrían dividir en 10 partes de 1000 votos, luego se realiza el proceso de mezcla para cada parte en paralelo. Finalmente se vuelven a combinar todas las partes, como resultado los primeros 1000 votos corresponderán a una mezcla de los primeros 1000 votos originales, luego los siguientes 1000 serán una mezcla de los votos 1001 a 2000 de los originales y así en adelante, es por esto que no corresponde a una mezcla total.

Esta se implementó sobre la misma red, modificando la máquina de estados con un paso previo que divide los votos en partes llamado *Map* de Step Functions el cual permite paralelización, luego se alimenta cada parte a la red de mezcla original de forma paralela. Un detalle es que Step Functions tiene un límite de 50 procesos que se pueden ejecutar en paralelo en una sola máquina. Como resultado se logró procesar una mezcla de 50000 votos (alrededor de 300 MB de entrada y 2.1 GB de datos generados), la prueba se realizó tomando 1000 votos por mezcla, en total tardó 24m 56s en completarse.

Un diagrama generado por AWS de de la red distribuida se encuentra en el anexo A.1.

Capítulo 4

Conclusiones

Se considera que se cumplió parcialmente con los objetivos planteados, puesto que:

- Se logró la implementación de una red de mezcla en la cual se mantiene secreta la permutación de la mezcla, ya que los datos sobre que la podrían revelar no son guardados o transmitidos hacia un tercero.
- Se evaluó el rendimiento de la red tanto en tiempo como en memoria utilizada para distintas cantidades de votos.
- Se verificó el correcto funcionamiento de la red mediante pruebas unitarias para la mezcla verificable y de la red en sí monitoreando el flujo de los datos.
- Se probó la seguridad de la red a través de implementaciones de adversario, las cuales intentan pasar datos incorrectos por la red.
- La red es flexible en el sentido de que permite configurar su estructura al momento de ejecución (cantidad de pasos de mezcla, implementación a utilizar en cada paso, número de verificadores). También se probó una versión más escalable con una arquitectura distribuida que permite ejecutar múltiples mezclas en paralelo.

4.1. Trabajo futuro

Si bien la versión de la red con mezcla distribuida es más escalable que la versión de la mezcla total, la implementación utilizada tiene limitantes, en particular el máximo de procesos concurrentes de Step Function. Es posible modificarla creando otro paso de paralelismo dentro del anterior, por lo que sí es posible escalarla aún más, pero eventualmente se llega al límite total de transiciones por ejecución, así que no es arbitrariamente escalable. Una alternativa sería implementar un servidor que actúe como administrador de la red, aunque en ese caso se perdería las ventajas de Step Function; se ejecuta sin servidor y permite transparentar las entradas de cada paso y el flujo de los datos de la red.

Otro punto a considerar es que si bien la red es flexible ya que permite usar implementaciones distintas en cada paso de la red, para EVoting solo existe una implementación por lo que no tiene sentido realizar múltiples mezclas ya que es el mismo código el que se ejecutaría múltiples veces. En el caso que un tercero desee involucrarse con su propia implementación de mezcla y verificación, tendría que integrar su código a la cuenta de AWS de EVoting para

que la red pueda aceptar su implementación. Para eliminar esta dependencia, una posibilidad sería crear un estándar de APIs, de esta manera cada paso de la red no estaría centralizada en lambda o AWS, aunque en este caso entraría nuevamente el tema del rendimiento.

Bibliografía

[1] EVoting, "Evoting, líder en democracia electrónica.", https://evoting.com/.

[2] Mulholland, P., "Estonia leads world in making digital voting a reality," 2021, https://www.ft.com/content/b4425338-6207-49a0-bbfb-6ae5460fc1c1.

[3] "Helios voting.", https://vote.heliosvoting.org/.

[4] Sako, K. y Kilian, J., "Receipt-free mix-type voting scheme - A practical solution to the implementation of a voting booth," en Advances in Cryptology - EUROCRYPT '95, International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France, May 21-25, 1995, Proceeding (Guillou, L. C. y Quisquater, J., eds.), vol. 921 de Lecture Notes in Computer Science, pp. 393–403, Springer, 1995, doi:10.1007/3-540-49264-X_32.

[5] Abe, M., "Mix-networks on permutation networks," en Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18, 1999, Proceedings (Lam, K., Okamoto, E., y Xing, C., eds.), vol. 1716 de Lecture Notes in Computer Science, pp. 258–273, Springer, 1999, doi:10.1007/978-3-540-48000-6_21.

[6] Furukawa, J. y Sako, K., "An efficient scheme for proving a shuffle," en Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings (Kilian, J., ed.), vol. 2139 de Lecture Notes in Computer Science, pp. 368–387, Springer, 2001, doi:10.1007/3-540-44647-8_22.

[7] "Verificatum mix-net.", https://www.verificatum.org/html/downloads.html/.

[8] Gamal, T. E., "A public key cryptosystem and a signature scheme based on discrete logarithms," en Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings (Blakley, G. R. y Chaum, D., eds.), vol. 196 de Lecture Notes in Computer Science, pp. 10–18, Springer, 1984, doi:10.1007/3-540-39568-7_2.

[9] Paillier, P., "Public-key cryptosystems based on composite degree residuosity classes," en Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding (Stern, J., ed.), vol. 1592 de Lecture Notes in Computer Science, pp. 223–238, Springer, 1999, doi:10.1007/3-540-48910-X_16.

[10] Nguyen, L., Safavi-Naini, R., y Kurosawa, K., "Verifiable shuffles: A formal model and a paillier-based efficient construction with provable security," en Applied Cryptography and Network Security, Second International Conference, ACNS 2004, Yellow Mountain, China, June 8-11, 2004, Proceedings (Jakobsson, M., Yung, M., y Zhou, J.,

eds.), vol. 3089 de Lecture Notes in Computer Science, pp. 61–75, Springer, 2004, doi:10.1007/978-3-540-24852-1_5.

[11] Damgård, I. y Jurik, M., "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," en Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings (Kim, K., ed.), vol. 1992 de Lecture Notes in Computer Science, pp. 119–136, Springer, 2001, doi:10.1007/3-540-44586-2_9.

[12] Onodera, T. y Tanaka, K., "Shufle for paillier's encryption scheme," IEICE Trans. Fundam. Electron. Commun. Comput. Sci., vol. 88-A, no. 5, pp. 1241–1248, 2005, doi:10.1093/ietfec/e88-a.5.1241.

[13] Nguyen, L., Safavi-Naini, R., y Kurosawa, K., "A provably secure and efficient verifiable shuffle based on a variant of the paillier cryptosystem," J. Univers. Comput. Sci., vol. 11, no. 6, pp. 986–1010, 2005, doi:10.3217/jucs-011-06-0986.

Anexos

Anexo A. Step Function

Código A.1: Definición de Step Function de la red de mezcla

```
{
  "Comment": "Vote shuffle state machine.",
  "StartAt": "CopyS3Votes",
  "States": {
    "CopyS3Votes": {
      "Type": "Task",
      "Parameters": {
        "CopySource.$": "States.Format('${ShuffleS3BucketName}/{}', $.encryptedVotes)",
        "Bucket": "${ShuffleS3BucketName}",
        "Key.$": "States.Format('{}/0/encryptedVotes.json', $$.Execution.Name)",
        "ContentType": "application/json"
      },
      "ResultSelector": {
        "bucketName": "${ShuffleS3BucketName}",
        "key.$": "States.Format('{}/0/encryptedVotes.json', $$.Execution.Name)"
      },
      "ResultPath": "$.encryptedVotes",
      "Resource": "arn:aws:states:::aws-sdk:s3:copyObject",
      "Next": "CopyS3PublicCiphertexts"
    },
    "CopyS3PublicCiphertexts": {
      "Type": "Task",
      "Parameters": {
        "CopySource.$": "States.Format('${ShuffleS3BucketName}/{}', $.publicCiphertexts)",
        "Bucket": "${ShuffleS3BucketName}",
        "Key.$": "States.Format('{}/publicCiphertexts.json', $$.Execution.Name)",
        "ContentType": "application/json"
      },
      "ResultSelector": {
        "bucketName": "${ShuffleS3BucketName}",
        "key.$": "States.Format('{}/publicCiphertexts.json', $$.Execution.Name)"
      },
      "ResultPath": "$.publicCiphertexts",
      "Resource": "arn:aws:states:::aws-sdk:s3:copyObject",
      "Next": "SetupStatus"
    },
    "SetupStatus": {
```

```
    "Type": "Pass",
    "Parameters": {
      "currentEncryptedVotes.$": "$.encryptedVotes",
      "currentIteration": 0,
      "currentImplementations.$": "$.steps[0]"
    },
    "ResultPath": "$.status",
    "Next": "GenerateCommitment"
  },
  "Success": {
    "Type": "Succeed"
  },
  "GenerateCommitment": {
    "Type": "Task",
    "Parameters": {
      "FunctionName.$": "$.status.currentImplementations.generateCommitment",
      "Payload": {
        "encryptedVotes.$": "$.status.currentEncryptedVotes",
        "publicKey.$": "$.publicKey",
        "publicCiphertexts.$": "$.publicCiphertexts",
        "executionName.$": "$$.Execution.Name",
        "currentIteration.$": "$.status.currentIteration"
      }
    },
    "Resource": "arn:aws:states:::lambda:invoke",
    "ResultPath": "$.lambdaResult.GenerateCommitment",
    "Next": "GenerateChallenge"
  },
  "GenerateChallenge": {
    "Type": "Task",
    "Parameters": {
      "FunctionName.$": "$.status.currentImplementations.generateChallenge",
      "Payload": {
        "encryptedVotes.$": "$.status.currentEncryptedVotes",
        "publicKey.$": "$.publicKey",
        "publicCiphertexts.$": "$.publicCiphertexts",
        "commitment.$": "$.lambdaResult.GenerateCommitment.Payload.commitment",
        "reencryptedMessages.$": "$.lambdaResult.GenerateCommitment.Payload.
↪ reencryptedMessages",
        "executionName.$": "$$.Execution.Name",
        "currentIteration.$": "$.status.currentIteration"
      }
    },
    "Resource": "arn:aws:states:::lambda:invoke",
    "ResultPath": "$.lambdaResult.GenerateChallenge",
    "Next": "RespondChallenge"
  },
  "RespondChallenge": {
    "Type": "Task",
    "Parameters": {
      "FunctionName.$": "$.status.currentImplementations.respondChallenge",
      "Payload": {
```

```
        "encryptedVotes.$": "$.status.currentEncryptedVotes",
        "publicKey.$": "$.publicKey",
        "publicCiphertexts.$": "$.publicCiphertexts",
        "reencryptedMessages.$": "$.lambdaResult.GenerateCommitment.Payload.
    ↪ reencryptedMessages",
        "reencryptedMessagesRandomness.$": "$.lambdaResult.GenerateCommitment.Payload
    ↪ .reencryptedMessagesRandomness",
        "permutation.$": "$.lambdaResult.GenerateCommitment.Payload.permutation",
        "proverParams.$": "$.lambdaResult.GenerateCommitment.Payload.proverParams",
        "challenge.$": "$.lambdaResult.GenerateChallenge.Payload.challenge",
        "executionName.$": "$$.Execution.Name",
        "currentIteration.$": "$.status.currentIteration",
        "permutationCrypto.$": "$.lambdaResult.GenerateCommitment.Payload.
    ↪ permutationCrypto"
      }
     },
     "Resource": "arn:aws:states:::lambda:invoke",
     "ResultPath": "$.lambdaResult.RespondChallenge",
     "Next": "VerifyAll"
    },
    "VerifyAll": {
     "Type": "Map",
     "ItemsPath": "$.verifiers",
     "ResultPath": "$.lambdaResult.VerifyAll",
     "Parameters": {
      "FunctionName.$": "$$.Map.Item.Value",
      "Payload": {
       "encryptedVotes.$": "$.status.currentEncryptedVotes",
       "publicKey.$": "$.publicKey",
       "publicCiphertexts.$": "$.publicCiphertexts",
       "challenge.$": "$.lambdaResult.GenerateChallenge.Payload.challenge",
       "commitment.$": "$.lambdaResult.GenerateCommitment.Payload.commitment",
       "reencryptedMessages.$": "$.lambdaResult.GenerateCommitment.Payload.
    ↪ reencryptedMessages",
       "response.$": "$.lambdaResult.RespondChallenge.Payload.response",
       "executionName.$": "$$.Execution.Name",
       "currentIteration.$": "$.status.currentIteration"
      }
     },
     "Iterator": {
      "StartAt": "VerifyResponse",
      "States": {
       "VerifyResponse": {
        "Type": "Task",
        "Resource": "arn:aws:states:::lambda:invoke",
        "Parameters": {
         "FunctionName.$": "$.FunctionName",
         "Payload.$": "$.Payload"
        },
        "End": true
       }
      }
```

```
    },
    "Next": "AggregateVerifications"
  },
  "AggregateVerifications": {
    "Type": "Task",
    "Parameters": {
      "verifications.$": "$.lambdaResult.VerifyAll",
      "executionName.$": "$$.Execution.Name",
      "iteration.$": "$.status.currentIteration"
    },
    "Resource": "${AggregateVerificationsFunctionArn}",
    "ResultPath": "$.lambdaResult.AggregateVerifications",
    "Next": "IncrementIterator"
  },
  "IncrementIterator": {
    "Type": "Pass",
    "Next": "ValidationChoice",
    "Parameters": {
      "currentIteration.$": "States.MathAdd($.status.currentIteration, 1)",
      "currentEncryptedVotes.$": "$.status.currentEncryptedVotes",
      "iterations.$": "States.ArrayLength($.steps)"
    },
    "ResultPath": "$.status"
  },
  "ValidationChoice": {
    "Type": "Choice",
    "Choices": [
      {
        "Variable": "$.lambdaResult.AggregateVerifications",
        "BooleanEquals": true,
        "Next": "CopyS3ShuffledVotes"
      }
    ],
    "Default": "CopyS3PreviousVotes"
  },
  "CopyS3PreviousVotes": {
    "Type": "Task",
    "Parameters": {
      "CopySource.$": "States.Format('{}/{}', $.status.currentEncryptedVotes.bucketName, $
 ↪ .status.currentEncryptedVotes.key)",
      "Bucket": "${ShuffleS3BucketName}",
      "Key.$": "States.Format('{}/{}/encryptedVotes.json', $$.Execution.Name, $.status.
 ↪ currentIteration)",
      "ContentType": "application/json"
    },
    "Resource": "arn:aws:states:::aws-sdk:s3:copyObject",
    "ResultPath": null,
    "Next": "CheckIteration"
  },
  "CopyS3ShuffledVotes": {
    "Type": "Task",
    "Parameters": {
```

```
      "CopySource.$": "States.Format('{}/{}', $.lambdaResult.GenerateCommitment.Payload
    ↪ .reencryptedMessages.bucketName, $.lambdaResult.GenerateCommitment.Payload.
    ↪ reencryptedMessages.key)",
      "Bucket": "${ShuffleS3BucketName}",
      "Key.$": "States.Format('{}/{}/encryptedVotes.json', $$.Execution.Name, $.status.
    ↪ currentIteration)",
      "ContentType": "application/json"
     },
     "Resource": "arn:aws:states:::aws-sdk:s3:copyObject",
     "ResultPath": null,
     "Next": "CheckIteration"
    },
    "CheckIteration": {
     "Type": "Choice",
     "Choices": [
      {
       "Variable": "$.status.currentIteration",
       "NumericEqualsPath": "$.status.iterations",
       "Next": "Success"
      }
     ],
     "Default": "UpdateCurrentStatus"
    },
    "UpdateCurrentStatus": {
     "Type": "Pass",
     "Next": "GenerateCommitment",
     "Parameters": {
      "currentIteration.$": "$.status.currentIteration",
      "currentEncryptedVotes": {
       "bucketName": "${ShuffleS3BucketName}",
       "key.$": "States.Format('{}/{}/encryptedVotes.json', $$.Execution.Name, $.status.
    ↪ currentIteration)"
      },
      "currentImplementations.$": "States.ArrayGetItem($.steps, $.status.currentIteration)"
     },
     "ResultPath": "$.status"
    }
   }
}
```

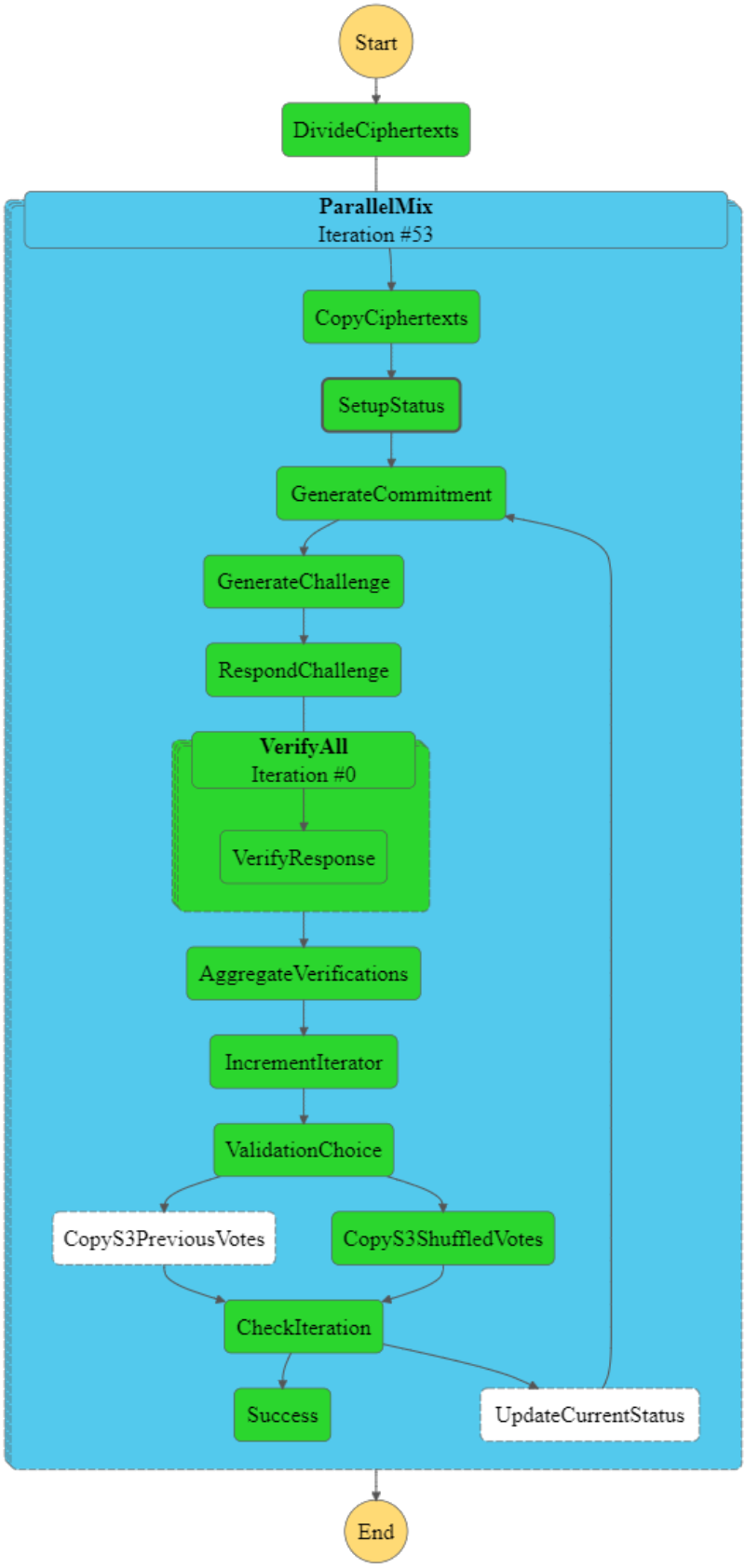

Figura A.1: Diagrama obtenido de AWS de la red utilizando mezcla distribuida.

Anexo B. Interfaz de aplicación web

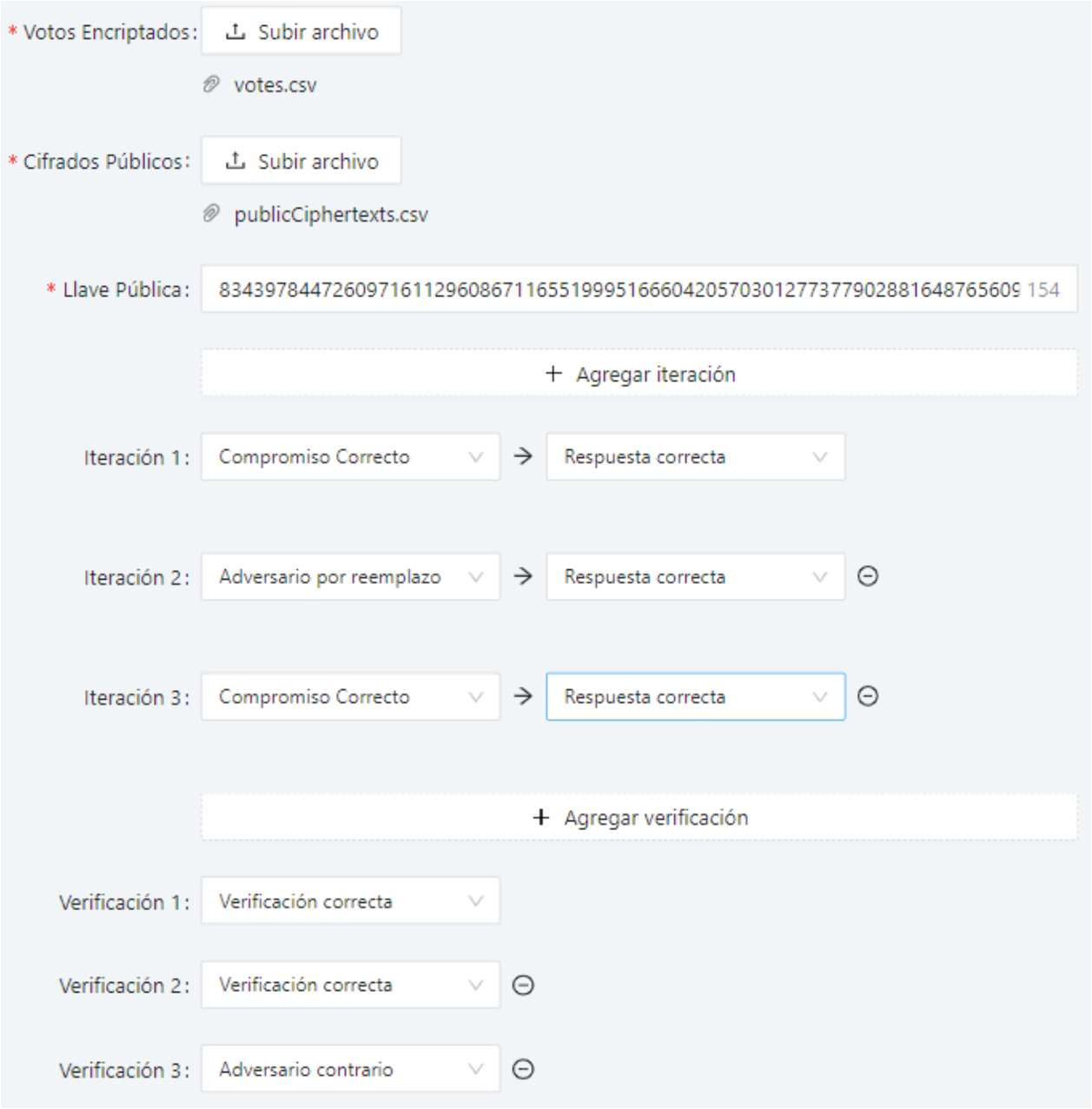

Figura B.1: Formulario para iniciar red de mezcla

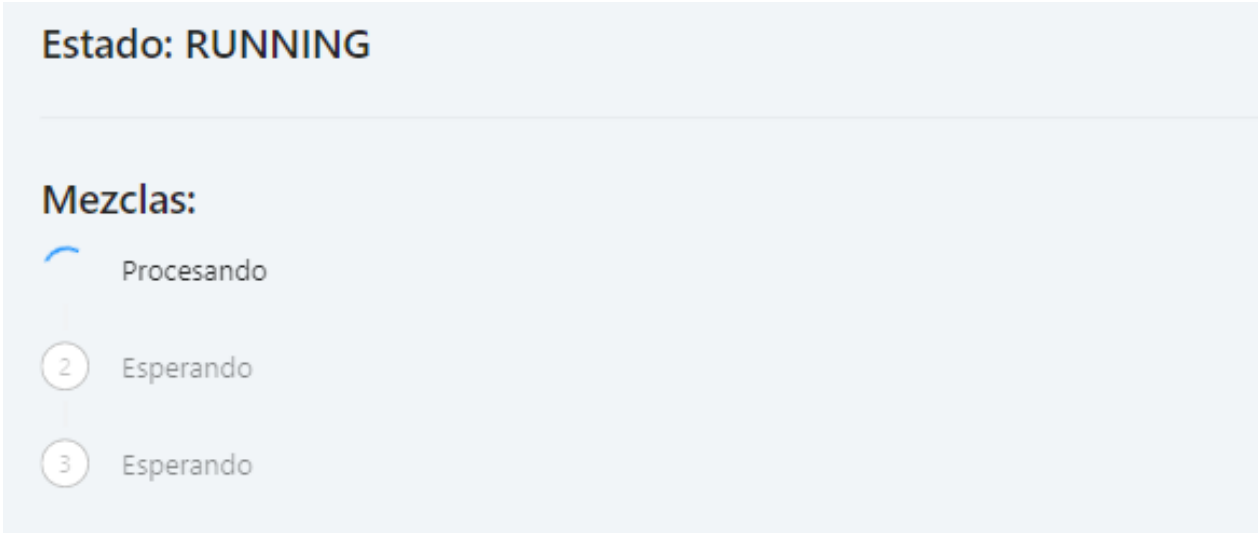

Figura B.2: Pantalla en espera de los resultados de las mezclas

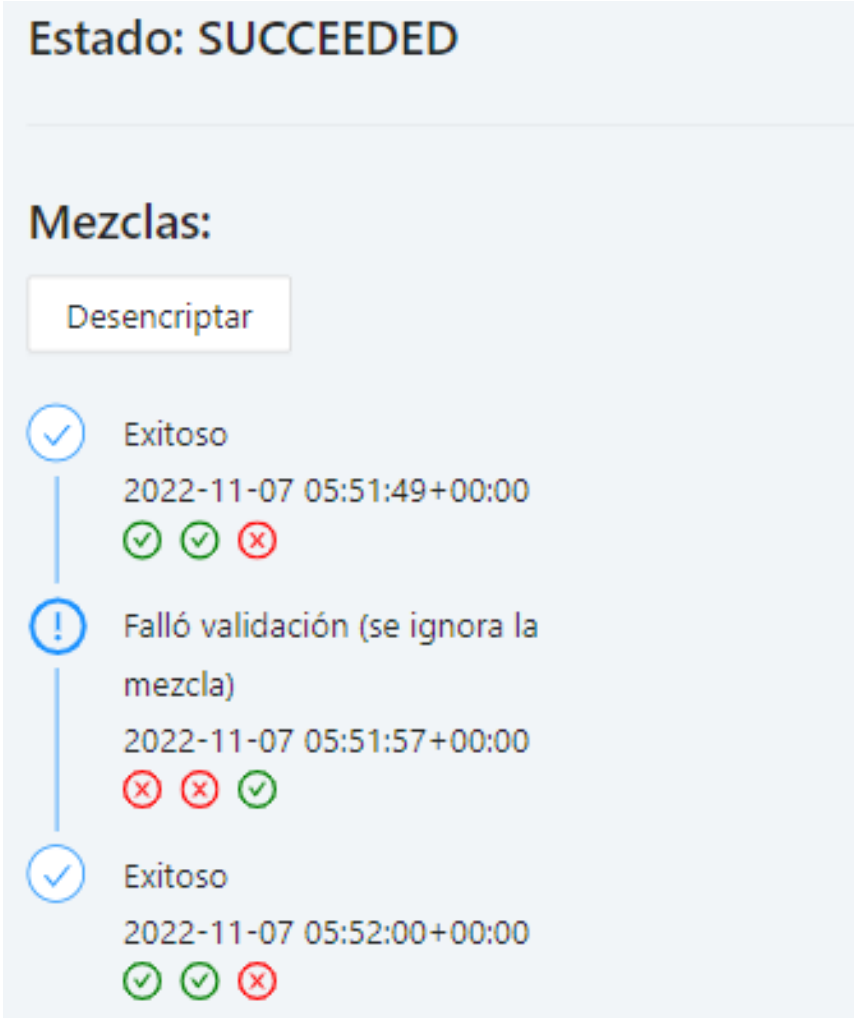

Figura B.3: Resultados de las mezclas finalizadas

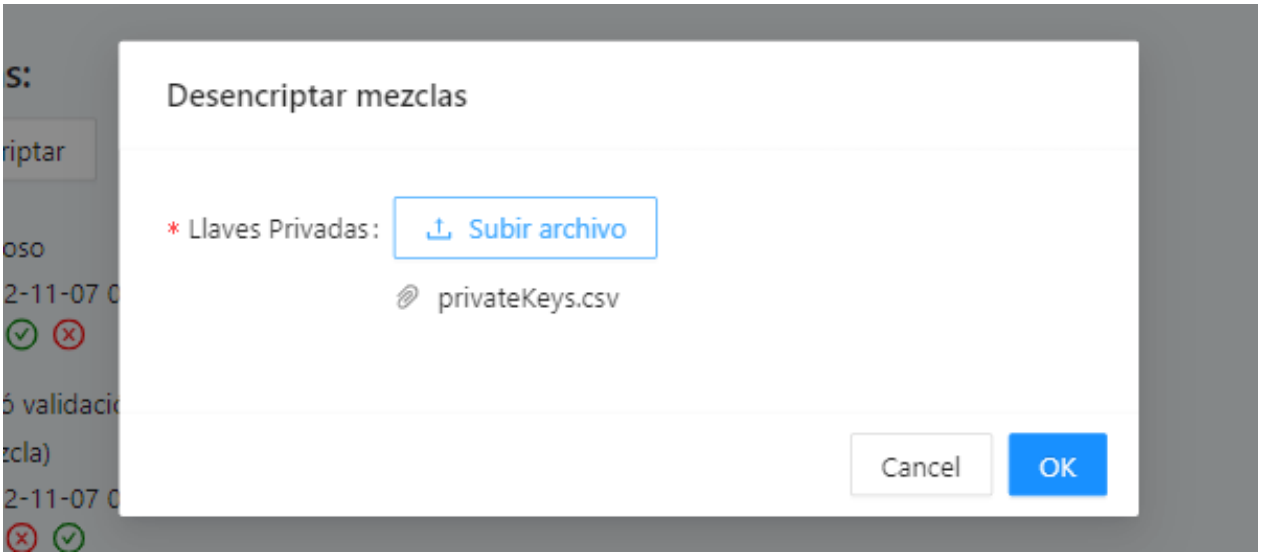

Figura B.4: Formulario de desencriptación de mezclas

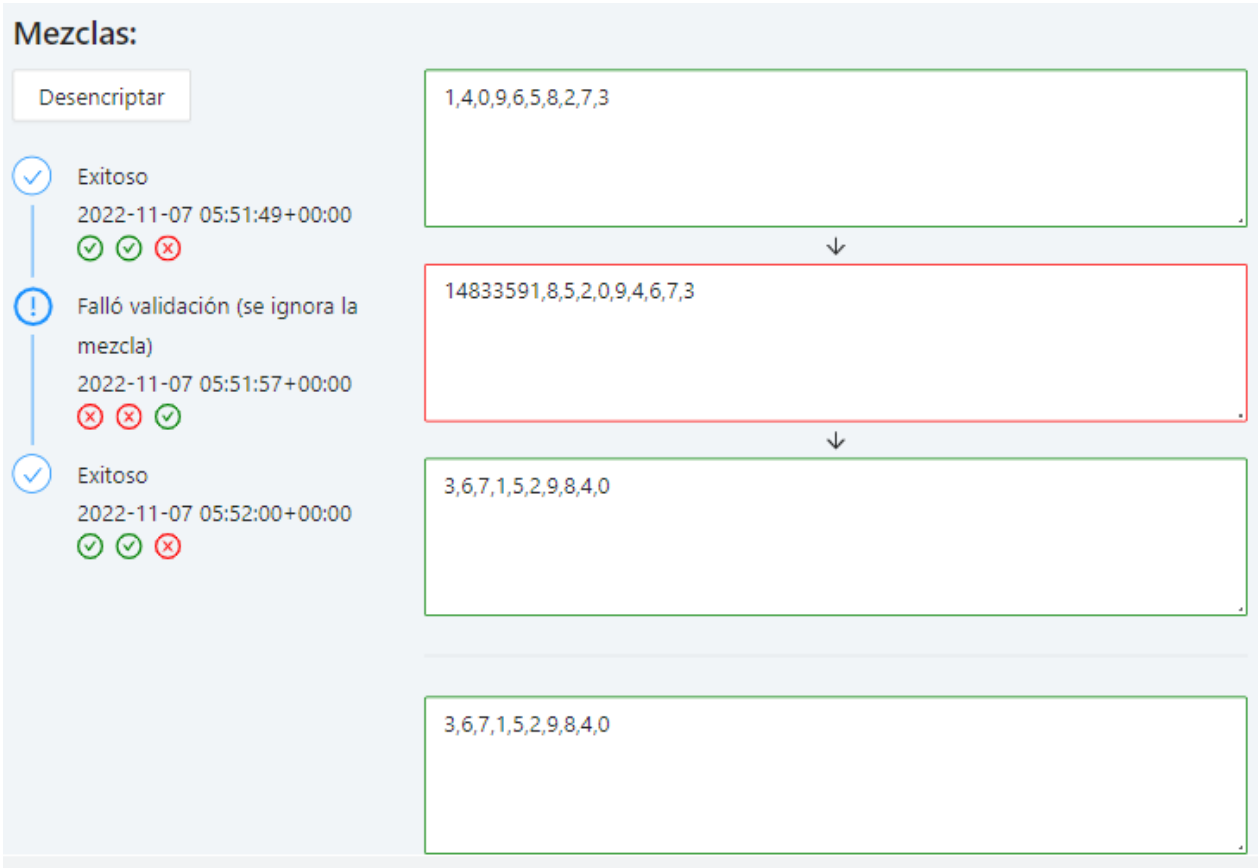

Figura B.5: Mezclas desencriptadas de cada iteración de la red