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

ENCRIPTACIÓN DE VOTOS USANDO EL ALGORITMO DE ENCRYPTACIÓN DE PAILLIER EN EL ORIGEN (NAVEGADOR WEB)

MEMORIA PARA OPTAR AL TITULO DE INGENIERA CIVIL EN COMPUTACIÓN

VALENTINA PIEDAD SEPULVEDA SILVA

PROFESOR GUÍA:
TOMÁS BARROS ARANCIBIA

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
CAMILO GÓMEZ NÚÑEZ

SANTIAGO DE CHILE
2023

RESUMEN DE LA MEMORIA PARA OPTAR
AL TITULO DE INGENIERA CIVIL EN COMPUTACIÓN
POR: VALENTINA PIEDAD SEPULVEDA SILVA
FECHA: 2023
PROF. GUÍA: TOMÁS BARROS ARANCIBIA

ENCRIPTACIÓN DE VOTOS USANDO EL ALGORITMO DE ENCRYPTACIÓN DE PAILLIER EN EL ORIGEN (NAVEGADOR WEB)

Este trabajo se enmarca en el sistema de votación por Internet provisto por la empresa EVoting. En este sistema, cada vez que un votante marca sus preferencias, el voto es encriptado usando el algoritmo de encriptación propuesto por Damgård y Jurik como generalización del algoritmo de Paillier . Una vez finalizada la votación, todos los votos son sumados homomórficamente y se procede a desencriptar solamente la suma usando desencriptaciones parciales que se mezclan para obtener el resultado final. Todo el motor criptográfico de EVoting está implementado en Scala.

Actualmente, los votos son encriptados en un servidor especializado. Este trabajo analiza e implementa el algoritmo de encriptación de Paillier para poder encriptar votos electrónicos en el extremo del cliente, es decir, en cualquier navegador moderno e integrarse al resto de la plataforma. Debido al cambio donde se realiza la encriptación desde un servidor especializado a cualquier navegador, se pierde el control de esta, por lo que se hace necesario complementarlo con pruebas de Zero Knowledge que permitan garantizar que el voto encriptado está correcto (es un voto válido), sin revelar su contenido.

Este trabajo analiza distintas alternativas de implementación y decide implementar el algoritmo en Javascript usando la biblioteca Scala-Js. Para las pruebas de Zero Knowledge se implementa una propuesta que permita probar que el voto pertenece al conjunto de todos los votos válidos.

Una vez implementado, se hicieron varios experimentos con distintos navegadores y parámetros de la votación (principalmente el número de candidatos y cantidad de preferencias a marcar) y se concluye sobre la factibilidad de su uso real en votaciones.

Finalmente, se plantean algunas dudas sin resolver para ser analizadas en un trabajo futuro.

A mi familia y amigos por el apoyo que me han entregado durante el proceso.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

Las votaciones son uno de los procesos más importante dentro de las sociedades democráticas. Estas suceden no solo en el contexto de elección de autoridades políticas (parlamentarios, presidentes, alcaldes, etc.), sino que en un amplio rango de la vida cotidiana, como lo son votaciones de gremios, asociaciones deportivas, sindicatos, asociaciones de funcionarios públicos, colegios profesionales, centro de alumnos, comités paritarios, etc.

Salvo contadas excepciones en que las votaciones no son secretas, en la gran mayoría es vital garantizar, entre otros, cuatro aspectos primordiales: 1) el secreto del voto (lo que garantiza la libertad al votar), 2) que el voto esté bien emitido (es decir, manifiesta claramente la voluntad del votante, 3) asegurar su integridad, es decir, asegurar que no haya sido manipulado y 4) que los votos hayan sido contabilizados (sumados) correctamente [1].

En el contexto de voto tradicional en Chile, estos factores son organizados y supervisados por el SERVEL (Servicio Electoral) quien designa aleatoriamente vocales de mesa y colegios escrutadores encargados de velar por la seguridad de la votación [2].

Por otra parte, EVoting es una empresa dedicada a entregar una plataforma de votaciones electrónicas remotas seguras. Fue fundada por ingenieros de la Universidad de Chile y funciona en Chile desde el 2013. Actualmente, da servicios también en otros países de la región como Perú, Costa Rica o México. El 2013 realizó una votación simbólica (no oficial) durante la votación presidencial para chilenos en el extranjero, recibiendo votos de más de 100 países [3].

EVoting utiliza el algoritmo de encriptación de Paillier [4], el cual permite realizar la suma homomórfica de los votos encriptados, es decir, permite calcular la suma total de los votos sin necesidad de desencriptar uno a uno, obteniendo la suma encriptada y por consecuencia, solo es necesario desencriptar la suma para obtener los resultados de la votación. En otras palabras, para dos papeletas distintas, la suma de dos papeletas encriptadas es igual a la encriptación de la suma de las papeletas [5].

En la solución actual de EVoting, la encriptación del voto se realiza en un servidor especializado, aislado en una red propia, de forma completamente anónima y sin generar ningún tipo de bitácora para garantizar el secreto del voto. Esta técnica tiene la ventaja de que se puede votar desde cualquier dispositivo conectado a Internet (sin ningún requerimiento de memoria,

CPU, sistema operativo u otros) y asegura que el voto ha sido encriptado correctamente.

Sin embargo, hay clientes y legislaciones locales para votaciones oficiales que exigen que el voto sea encriptado en el extremo del cliente, lo que a su vez también permite cumplir el concepto de *end-to-end verifiability*, que refiere a que el cliente pueda verificar que su voto no haya sido manipulado; que haya sido emitido correctamente y por último, poder ver que todos los votos han sido contados [6].

Esta memoria propone implementar la encriptación del voto en el extremo del cliente para las votaciones de EVoting, es decir, en el navegador del cliente. Debido a que la encriptación se realiza fuera del control del servidor de encriptación, se requiere implementar una prueba o verificación de que el voto está correctamente encriptado (es decir, contiene en su interior una o varias opciones válidas de la papeleta), pero sin dar ningún elemento que permita saber exactamente las opciones marcadas, lo que se implementará con una prueba conocida como *Zero Knowlwdge Proof* o ZKP [7].

1.1. Objetivo General

Implementar la encriptación verificable del voto en el lado del cliente para las votaciones de E-Voting de forma eficiente, usando la propuesta del algoritmo de Paillier de Damgård y Jurik.

1.2. Objetivos Específicos

1. Implementar el algoritmo de encriptación en el lado del cliente.
2. Comparar resultados en cliente respecto a los obtenidos con el servidor.
3. Investigar sobre trabajos en la academia que proponen ZKP y definir una prueba.
4. Implementar la prueba de ZKP y verificar que esta cumple con los requerimientos.
5. Realizar pruebas de rendimiento y de manejo de recursos.
6. Definir en base a estos resultados las limitantes de la solución (ver cuando se puede o no usar en un escenario real)

Capítulo 2

Antecedentes

2.1. Encriptación y ZKP

El algoritmo de Paillier corresponde a un esquema de encriptación asimétrico. En los sistemas de encriptación asimétricos se generan dos llaves: una pública y una privada. La llave pública, como su nombre lo indica, es conocida y se utiliza para encriptar. La llave privada se mantiene en secreto y se utiliza para desencriptar [8].

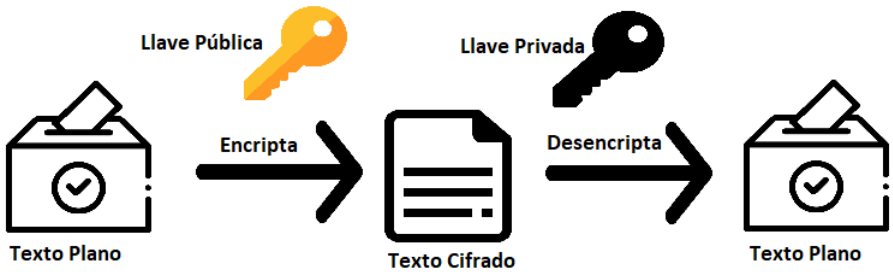

Figura 2.1: Esquema desencriptación de llave pública

Zero Knowledge Proof (ZKP), es una técnica que como sugiere su nombre, permite demostrar una condición sin entregar ninguna información adicional, por ejemplo, demostrar que una persona es mayor de edad sin revelar su edad [9].

En el contexto de los objetivos de este trabajo, usaremos una prueba de ZKP para demostrar que un voto encriptado corresponde efectivamente a una de las opciones válidas de la votación, sin revelar ninguna información que permita concluir por cuál opción se está votando. Esto es muy relevante, ya que los votos serán sumados mientras estén encriptados, es decir, no se podrá revisar su contenido, y se desea evitar situaciones, como por ejemplo alguien le asigne muchos votos a un candidato (o votos negativos). Para demostrar que ningún voto haya sido manipulado, se necesita que cada voto vaya acompañado de su prueba ZKP.

2.2. Paillier

Paillier corresponde a un esquema de encriptación asimétrico. Al igual que todos los esquemas asimétricos tiene tres etapas, generación de llaves, encriptación, y desencriptación.

En particular, en EVoting usan la Generalización de Pailler con Umbral propuesta por Damgård y Jurik en [10].

2.2.1. Congruencia

Sea $n > 0$. Se dice que a y b son congruentes (mod n) si $a - b$ es múltiplo de n [11]. Esto se escribe de la forma:

$$a \equiv b \ mod \ n \tag{2.1}$$

2.2.2. Grupo Cíclico

Se define G como un grupo cíclico, si es generado por un solo elemento [12]. Para los módulos, los grupos cíclicos se generan por medio de $mod\ n$. En caso que n sea primo, este corresponde a todos los números entre 0 hasta $n - 1$.

2.2.3. Generación de llave

Para generar la llave, el algoritmo de Damgård y Jurik eligen al azar dos números primos p y q, tal que $p = 2p' + 1$ y $q = 2q' + 1$, con p' y q' números primos diferentes de p y q.

Se define $n = pq$ y $m = p'q'$. Se escoge un $s \in \mathbb{N} > 0$, y se define el espacio de texto plano como Z_{n^s}. Este grupo cíclico corresponde a todos los restos co-primos de n^s, es decir, corresponde a $k \in \{0, 1, 2, ..., n^s - 1\}$, tal que $gcd(k, n^s) = 1$.

Escogemos un d tal que $d \equiv 0 \ mod \ m$ y $d \equiv 1 \ mod \ n^s$. Definimos el polinomio $f(X) = \sum_{i=0}^{k-1} a_i X^i \ mod \ n^s m$, escogiendo a_i como un número aleatorio entre $\{0, ..., n^s m - 1\}$ para $k > i > 0$ y $a_0 = d$.

El *secret share* i-ésimo se define como $s_i = f(i)$ para todo $1 \leq i \leq l$, para cada una de las l personas de confianza, o servidores privados con su propia llave privada. Este *secret share* corresponde a la desencriptacion parcial de la persona o servidor i-ésimo.

El valor n es la llave pública.

Para la verificación, se necesita que cada servidor posea una llave de verificación $v_i = v^{\Delta s_i} mod \ n^{s+1}$ donde $\Delta = l!$.

2.2.4. Encriptación

Para encriptar un mensaje M (representado como un entero), se elige al azar un número $r \in Z^*_{n^{s+1}}$. Se calcula el mensaje cifrado como:

$$c = g^M r^{n^s} \ mod \ n^{s+1} \tag{2.2}$$

Con (n, g) los valores de la llave pública.

2.2.5. Desencriptación parcial

Cada poseedor de un secreto s_i calculará $c_i \equiv c^{2\Delta s_i}$ donde c corresponde al texto cifrado. La prueba ZKP se define como $log(c_i^2)_{c^4} = log(v_i)_v$. Esta prueba convence de que se ha elevado c_i a su exponente secreto s_i. Notar que esta prueba ZKP permite verificar que se realizó la desencriptación parcial correctamente sin revelar s_i. No confundir con la prueba ZKP que un texto fue cifrado correctamente.

2.2.6. Combinación de las desencriptaciones parciales

Cuando se obtienen w desencriptaciones parciales, estas se combinan con la siguiente fórmula:

$$c' = \prod_{i \in S} c_i^{2\lambda_{0,i}^s} \tag{2.3}$$

donde se define:

$$\lambda_{0,i}^S = \Delta \prod_{i' \in S \setminus i} \frac{-i}{i - i'} \in Z \tag{2.4}$$

El valor de c' tendrá la forma de $c' = c^{4\Delta^2 d}$. Dado que $4\Delta^2 d = 0 \ mod \ \lambda$ y $4\Delta^2 d = 4\Delta^2 \ mod \ n^s$ se puede concluir que $c' = (1+n)^{4\Delta^2 M} \ mod \ n^{s+1}$ y derivar M aplicando el algoritmo propuesto en [10]

2.2.7. Propiedades

Paillier tiene la propiedad de ser homomórfico respecto a la suma, es decir, que si yo tengo dos valores m_1, m_2, se cumple que para la suma $m_1 + m_2$, la encriptación de esta suma, es el producto de estos dos mensajes.

$$E(m_1, r_1) \times E(m_2, r_2) \ mod \ n^{s+1} = (g^{m_1} {r_1}^n \ mod \ n^{s+1}) \times (g^{m_2} {r_2}^n \ mod \ n^{s+1}) \tag{2.5}$$

$$= (g^{m_1+m_2}(r_1 * r_2)^n \ mod \ n^{s+1}) \tag{2.6}$$

$$= E(m_1 + m_2, r_1 * r_2) \tag{2.7}$$

Esta propiedad permite que los votos puedan ser sumados sin necesidad de ser desencriptados antes.

2.3. Javascript

Javascript es un lenguaje de programación que cumple con el estándar ECMAScript. Este estándar fue creado por la European Computer Manufacturers Association Internacional (o ECMA) para definir el comportamiento de un lenguaje de propósitos generales. Frecuentemente, se definen cada año las nuevas implementaciones. Prácticamente, todos los navega-

dores web modernos incluyen un intérprete de este Javascript estándar, lo que permite la interoperabilidad de un programa entre distintos navegadores y sistemas operativos [13]. Su última versión es ECMA2023 [14].

Este lenguaje empezó principalmente como scripts de códigos, pero que debido a su masividad, eventualmente evolucionó a lo que se conocen como módulos, archivos con código que pueden ser llamados por otros archivos. Un módulo puede contener clases o funciones que pueden ser llamadas al ser importadas. Y pueden ser exportadas para ser utilizadas por otros archivos [15].

Existen varios tipos de archivos como CommonJs, utilizado por Node.js, y ES202x, que corresponden a las últimas especificaciones de Javascript. Hoy en día ES2020, que corresponde a la versión que agregó los enteros grandes [16], es utilizada por las últimas versiones de navegador, como Google Chrome, Edge, Firefox, entre otros [17]. Lo que permite que las últimas versiones de los navegadores puedan ejecutar cualquier código que involucre enteros grandes sin problemas.

Capítulo 3

Estado del arte

3.1. Sistema actual

EVoting ofrece actualmente sus servicios dentro de una plataforma web, que está conectada a varios servidores, los cuales son quienes realizan los procesos de encriptación, de autenticación y de validación. Además, está conectada a una base de datos externa, que se encarga de proporcionar los datos de la votación al sistema.

Por ejemplo, el proceso de votación de EVoting cuando se utiliza RUT + validación del serial como medio de autenticación, sé gráfica en Figura 3.1

EVoting usa el esquema de criptografía con umbral propuesto en [10]. En ese esquema, la llave privada se transforma en un conjunto de llaves parciales. Cada llave parcial permite hacer una desencriptación parcial de un texto cifrado. Cuando se tiene un umbral de desencriptaciones parciales, estas se combinan para obtener el desencriptado final del texto cifrado. En EVoting se define arbitrariamente ese umbral como 50 % + 1, por ejemplo, si se generaron 5 llaves parciales, se requerirá combinar 3 desencriptados parciales para obtener el texto descifrado final.

Previo a la elección, se debe generar la llave pública y las llaves privadas parciales.

Al momento de votar, el votante en su navegador ingresa al servidor "encriptador", el cual solicita la configuración de la votación al servidor "urna" y en base a esa configuración le despliega la papeleta correcta al votante.

A continuación, el votante escoge sus preferencias y pide cerrar el voto. El servidor encriptador despliega un resumen de las opciones marcadas junto al voto encriptado usando la llave pública de la votación. Solo una vez encriptado el voto, el servidor solicita que se identifique el votante, lo cual en caso de resultar exitoso genera un ticket de autenticación.

El navegador envía el ticket de autenticación junto al voto encriptado al servidor "urna", el cual revisa que el voto esté correctamente encriptado, que el ticket de autenticación sea válido, que el votante pertenezca al padrón de autorizados para la elección, que esta esté abierta y que no haya votado previamente. Si todo se cumple, el servidor "urna" almacena el voto en la base de datos y entrega un "recibo" al cliente, para asegurarle que su voto está bien emitido.

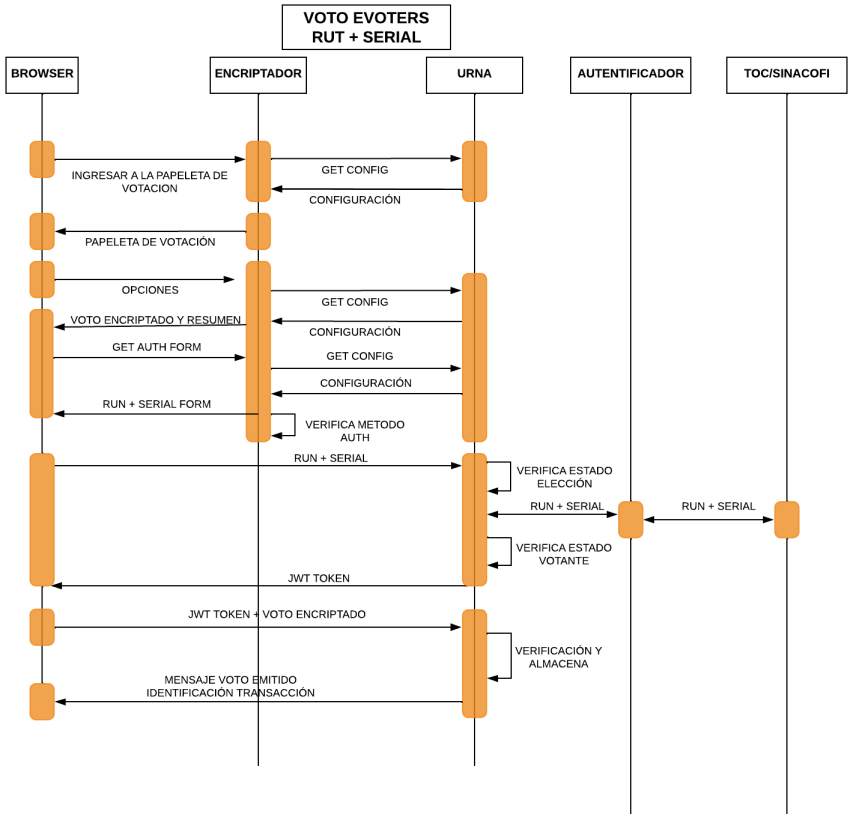

Figura 3.1: Flujo al votar

Al final de la votación, se suman todos los votos y finalmente son desencriptados con las llaves privadas, obteniendo el resultado de las elecciones.

Como se ve en este proceso, la encriptación del voto la realiza el servidor encriptador. Para que pueda realizarse en el navegador se debe contar con una implementación en `Javascript`, soportado por todos los navegadores modernos.

La implementación del algoritmo de Paillier en EVoting fue realizada completamente por los ingenieros de EVoting y está programado en Scala.

Actualmente, la implementación de la encriptación en el servidor permite trabajar con un número ilimitado de candidatos, y de cuantas marcas se realizan sobre esos candidatos, incluyendo votos nulos y blancos. También permite hasta $2^{32}-1$ votantes. Además, el servidor puede, en base a un sistema de "template", desplegar cualquier tipo de papeletas específicas para cada elección que se esté ejecutando en paralelo. La encriptación en el lado del cliente debe mantener estas propiedades.

3.2. Implementaciones de Paillier en cliente

Se encontraron dos librerías en Javascript, que implementan Paillier.

La primera llamada "paillier-bigint"que utiliza la implementación nativa de los enteros grandes (bigints) de JS [18]. Dado esto, se puede utilizar en todos los navegadores que so-

porten los bigints. Para usarse, solo se necesita llamar al módulo en un archivo JS, o bien utilizar Node-js. Lo bueno es que es fácil de usar, lo malo es que solo está implementado para la versión estándar de Paillier, y no la versión propuesta con umbral.

La segunda se llama "paillier-js", que utiliza la implementación de la librería "BigInteger.js"de Peter Olson [19]. Esta funciona tanto en navegadores que implementen BigInts como los que no, utilizando su propia implementación de estos en caso de que no se soporte. Corresponde a una implementación del algoritmo estándar de encriptación y desencriptación de Paillier, que solo necesita ser llamada para ser utilizada. Sin embargo, este proyecto está abandonado a favor de "paillier-bigint", por lo que no hay soporte.

Además, se encontraron librerías que pueden ser utilizadas con WebAssembly en el navegador. WebAssembly corresponde a un lenguaje de programación con un formato binario, que permite una variedad de lenguajes ser ejecutado en el navegador [20].

La primera corresponde a una implementación en Go [21], la cual puede correr en el navegador por medio de WebAssembly, y ser llamada por Javascript [22]. Permite realizar la suma y multiplicación homomorfica, con la implementación estándar de Paillier.

Aparte de esta, se encontró la implementación de Daylight Society, que utiliza la versión con umbral y con ZKP implementada en Ruby [23]. Esta versión es la más completa hasta ahora, puesto que utiliza los mismos elementos que EVoting. Esta al igual que la implementación en Go, no se llama directamente en el navegador web, pero puede ser compilado a WebAssembly, que puede ser utilizado por JS [24].

Por último, está la implementación de Rust, que corresponde al algoritmo de encriptación, generación de llaves y desencriptación estándar de Paillier [25]. Además está el repositorio de pruebas de ZKP, que corresponden a múltiples pruebas de ZKP utilizando este mismo código. Posee una gran variedad de pruebas, incluyendo la prueba que utiliza E-Voting [26]. Rust tiene de manera nativa compilación a WebAssembly [27].

Existen otras implementaciones de Paillier, pero estas son para usarse en el servidor: como una implementación en Python [28], Java [29], Julia [30], entre otros.

Como se ve, las implementaciones de Paillier en Javascript, no contemplan la variante con umbral que utiliza EVoting. Sin embargo, las que si contemplan esta, como la de Daylighting Society, y la de Rust, necesitan ser compiladas a WebAssembly primero.

3.3. Otros sistemas de votación electrónica por Internet

Existen otras empresas y sistemas desarrollados en la academia para votar electrónicamente por Internet con garantías de seguridad. A continuación listamos 3 que nos parecieron relevantes en el contexto de este trabajo.

3.3.1. Helios

Helios es un sistema de votación encriptada con encriptación del lado del cliente [31]. Este a diferencia del EVoting, utiliza el esquema de encriptación de ElGamal, un algoritmo de encriptación asimétrica para criptografía de llave pública, que se basa en Diffie-Hellman [32]. Al igual que EVoting utiliza ZKP para demostrar la encriptación.

El sistema en general de Helios funciona de la siguiente manera: Una persona desea votar en una elección, le pide al servidor que le traiga el folio de esta. El sistema la trae y esta persona vota y escoge sus respuestas. Estas respuestas son encriptadas.

Si se quiere, se puede volver a emitir el voto, y este se reencripta. Si se confirma las elecciones hechas en el voto, se sella el voto, y se envía el texto cifrado al servidor.

Luego el sistema pide a la persona que se autentifique. Si es correcto, el voto es asociado a esta persona. Estas respuestas son enviadas al servidor.

Después de esto se suman, y su suma es desencriptada con la llave privada y se obtienen los resultados de la elección.

El sistema de Helios funciona a través de su página web, los usuarios registrados pueden configurar y crear sus propias elecciones. Cuando se vota, se sube un boletín de los votantes y sus votos encriptados.

Cualquier persona puede auditar las elecciones. Y cualquier votante puede verificar que su voto fue contado [33].

Helios para su aplicación web utiliza Javascript con JQuery. Para el servidor utiliza Python.

3.3.2. Belenios

Belenios es otra aplicación de votación electrónica. Este es otro sistema verificable y que asegura la seguridad del voto. Su secretismo se logra, pudiendo hacer desencriptación solo con mayoría de llaves privadas. El sistema se basa en el protocolo Helios-C de Helios [34].

Belenios, al igual que Helios, utiliza ElGamal como esquema de encriptación, con la diferencia que aquí permiten votos en blancos. Para verificar las elecciones también se utiliza ZKP. También este posee votos firmados para evitar la inyección de votos (o ballot stuffing) [34].

Posee dos casos: El caso homomórfico, que corresponde a una suma homomórfica de los votos. En el cual se suman los votos encriptados y son desencriptados al final por medio de la llave privada [34].

Y el otro, el caso no homomórfico, en el cual se asocia un voto con un candidato, este es mezclado utilizando una red de mezcla. Se revela una lista de los votos de cada votante, pero ordenado de forma aleatoria, de tal forma de no asociar el votante con su voto. [34].

Luego de esto, los votos se cuentan. Este conteo puede ser realizado por cualquier método [34].

Respecto a su funcionamiento, es similar a EVoting. Se genera una llave, se reparte de forma parcial entre l personas de confianza. Cada persona de confianza posee además la llave

pública y el identificador de su llave privada.

Para que el cliente pueda votar, este recibe un correo con sus credenciales y la url, en donde está alojado el servidor. Cuando se abre la elección, el cliente se autentifica utilizando las credenciales. Se les muestra las opciones, y este selecciona una de ellas. Cuando una opción es elegida, esta es encriptada por el computador utilizando JS, y entrega un rastreador del voto. Este rastreador es un tipo de identificador del voto.

Una vez el votante ya hace su elección, este se autentifica utilizando su email, en donde recibe una contraseña. Este utiliza la contraseña y el voto es enviado. Mientras la elección esté abierta, el votante puede votar la cantidad de veces que quiera, y solo el último envío es considerado.

Al recibirlo, el servidor verifica el voto, realiza una ZKP con este, y se guarda.

Luego, cada persona de confianza entrega su encriptación parcial de los votos, y si hay $m > l/2$ llaves parciales, se obtiene el resultado final de la votación. En cualquier momento de la elección, los votantes pueden ver que su voto haya sido contado.

Belenios, utiliza Javascript, CSS, y React en el frontend. Para el backend se utiliza OCaml.

Para utilizar Belenios, se puede utilizar una plataforma en el sitio web de este sin necesidad de instalar nada. O bien utilizar el código fuente e instalarlo en un servidor propio.

3.3.3. Participa UChile

Como otro sistema está Participa UChile, el cual está basado en Helios. Es utilizado dentro de las facultades de la Universidad de Chile, y está desarrollado por académicos del departamento de Computación de la Universidad [35].

Se usa para elecciones de bajo perfil en contexto universitario, como elecciones de representantes de funcionarios, elecciones estudiantiles locales, entre otros [35]. Para participar se necesita autenticar utilizando la cuenta UChile.

Participa UChile permite auditar las elecciones en cualquier momento. Además, permite votar más de una vez, contando exclusivamente el último voto.

Se permiten los votos ponderados, blancos y nulos. La llave privada se divide en pedazos, y se entrega uno a cada *custodio de clave*, los cuales son designados de forma previa a la elección. Para poder desencriptar los votos, se necesita la mayoría de las llaves privadas [36].

Los votos pueden ser verificados en cualquier momento de la elección [36].

Capítulo 4

Diseño e Implementación de la Solución

Para poder implementar la solución definitiva, se analizaron dos alternativas: un desarrollo completo utilizando Typescrypt (Typescrypt agrega una capa de tipos explícitos a Javascript) [37] o un desarrollo usando de base el resultado de Scala-Js, un compilar de Scala a Javascript [38].

4.1. Votos

4.1.1. Estructura de los votos

Los votos en EVoting poseen la siguiente forma:

Si tenemos para una elección n candidatos posibles, el voto se forma como un entero grande con $n + 2$ bloques de 32 bits cada uno. A cada candidato se le asigna un bloque i entre 2 y $n + 1$, el primer bloque está reservado para los votos blancos y el segundo para los votos nulos. Por ejemplo, si yo voto por el candidato i con $i = 1$, el voto correctamente emitido equivale a un 1 en la posición del candidato, como se muestra a continuación:

Figura 4.1: Forma voto válido

Además de estos votos, existen los votos blancos y nulos. Los votos blancos son aquellos con preferencias sin marcar y los votos nulos son aquellos en que se ha votado por más de las preferencias posibles. Existen muchas votaciones donde la cantidad de preferencias a marcar es superior a 1 (como es el caso de las votaciones oficiales donde se puede marcar solo una preferencia por presidente, senador, diputado, alcalde, gobernador, etc.). Nos referiremos

como k a la cantidad máxima de preferencias posibles a marcar en una elección, por ejemplo, si hay 10 candidatos y se vota por 3 de estos, entonces $k = 3$.

Se considera un voto blanco por cada una de las opciones de k no utilizadas. Por ejemplo, si $k = 3$ y marco solo una opción, el voto tendrá marcada esa opción y 2 votos blancos. Siguiendo el ejemplo anterior, si voté solo por el candidato 1, el voto queda formado por:

2	0	1	···	0	0	0

Figura 4.2: Voto con marcas blancas

Por último, está el voto nulo. Este ocurre si se votan por más candidatos de los permitidos. Este voto tiene un 1 en la casilla 1, y 0 en las otras casillas.

0	1	0	···	0	0	0

Figura 4.3: Voto nulo

4.1.2. Suma homomórfica

Para mostrar como se realiza la suma homomórfica en EVoting, supondremos una votación de n candidatos con k preferencias para marcar y utilizaremos tres votos de ejemplo: v_1, v_2 y v_3.

El voto v_1 es por los candidatos $n = 1$ y $n = n$. Por lo que la casilla 2, y la casilla $n + 2$, están ocupadas por un 1.

0	0	1	···	0	0	1

Figura 4.4: Voto 1

El voto v_2 no marcó ningún candidato, por lo que la casilla 0, la que representa los votos en blanco, tiene valor 2.

2	0	0	$\cdots$	0	0	0

Figura 4.5: Voto 2

El voto v_3, solo marcó preferencia por el candidato 1, entonces la casilla 2 tendrá un 1 correspondiente al candidato y la casilla 0tendrá un 1 por la preferencia sin utilizar, es decir, opción blanco.

1	0	1	$\cdots$	0	0	0

Figura 4.6: Voto 3

Al sumar, el resultado final sería de la forma:

	2	0	0	0	0	0	0
	0	0	1	0	0	0	1
+	1	0	1	0	0	0	0
	3	0	2	0	0	0	1

Figura 4.7: Suma de votos

Esta suma nos da como resultado, que hubo 3 votos blancos, 2 votos al candidato 1 y 1 voto al candidato n. El cual es el mismo resultado, del que si lo hubiésemos contado voto por voto. Cabe hacer notar que cada uno de los votos como la suma se mantienen siempre encriptadas.

4.1.3. Pruebas ZKP

Para las pruebas de Zero Knowledge Proof utilizamos la propuesta de Taylor Fox [39], en la cual se demuestra que el mensaje está dentro de un conjunto de mensajes válidos, sin indicar cuál de estos es.

En la propuesta de Taylor, para el conjunto de mensajes válidos $\{m_i\}_1^k$ y el mensaje encriptado $c = g^m * r^n \ mod\ n^2, m \in \{m_i\}_1^n$; se calcula el conjunto $\{c_i\}_1^k$ utilizando la formula:

$$u_i = \frac{c}{g^{m_i}} \ mod\ n^2 \tag{4.1}$$

Si $m_i = m$, ocurre:

$$u_i = \frac{c}{g^{m_i}} = \frac{g^m r^n}{g^m} = r^n \ mod\ n^2 \tag{4.2}$$

4.1.3.1. Preparación

Existen dos roles en esta interacción: prover (P), el cual es el que prueba su conocimiento, y el verifier (V), el cual es quien verifica el conocimiento de P. En este caso, el prover corresponde al usuario, es decir, el votante. Y el verifier corresponde al sistema.

P selecciona aleatoriamente unos valores $e_1, e_2, e_3, ..., e_k \ \in 2^b < min(p, q)$, $z_1, z_2, z_3, ..., z_k \ \in Z_n^*$, excepto para $i/m_i = m$. Luego se selecciona un $w \in Z_n^*$.

4.1.3.2. Commitment

En base a estos valores, P calcula $a_1, a_2, a_3, ..., a_k$ tal que:

$$a_i = \frac{z_i^n}{u_i^{e_i}} \ mod\ n^2 \tag{4.3}$$

Si $m_i = m$, entonces a_i se calcula como:

$$a_i = w^n \ mod\ n^2 \tag{4.4}$$

4.1.3.3. Desafío

Para el desafío, el verificador puede generar aleatoriamente un número de b bits y enviarlo al prover para que calcule las pruebas, con una probabilidad de engañar al verificador de $P = \frac{1}{2^b}$. En el caso de EVoting el voto debe ser enviado con las pruebas ya calculadas, por lo que se usa una variante no interactiva de ZKP. En esta variante implementada en EVoting, el desafío se calcula como el hash SHA256 de la concatenación ordenada de los a_i calculados previamente, lo llamaremos $e_{\text{challenger}}$

Con $e_{\text{challenger}}$, P puede calcular los valores e_i, z_i para $m_i = m$.

$$e_i = e_{\text{challenger}} - \sum_{k=1}^{K} e_k \tag{4.5}$$

$$z_i = w * r^{e_i} \ mod\ n \tag{4.6}$$

Los conjuntos $\{a_i\}_1^k, \{z_i\}_1^k, \{e_i\}_1^k$ se envían junto al voto encriptado para la posterior verificación.

4.1.3.4. Verificación

Para verificar que el voto corresponde a un mensaje válido, es decir, que corresponde a un voto encriptado válido, se comienza con la siguiente verificación:

$$\sum_{k=1}^{K} e_k = e_{\texttt{challenger}} \ mod \ 2^b \tag{4.7}$$

Si es válida, se procede a verificar que

$$z_k^n = a_k * u_k^{e_k} \ mod \ n^2 \tag{4.8}$$

Si $m_i \neq m$, este cálculo da como resultado:

$$z_k^n = \frac{z_k^n}{u_k^{e_k}} * u_k^{e_k} \ mod \ n^2 \tag{4.9}$$

$$z_k^n = z_k^n \ mod \ n^2 \tag{4.10}$$

Si $m_i = m$, el resultado final es:

$$u_i = r^n \ mod \ n^2 \tag{4.11}$$

$$z_i = w * r^{e_i} \ mod \ n \tag{4.12}$$

$$a_i = w^n \ mod \ n^2 \tag{4.13}$$

Reemplazamos los valores de 4.11 en la ecuación 4.7, y obtenemos:

$$z_k^n = w^n * (r^n)^{e_k} \ mod \ n^2 \tag{4.14}$$

$$(w * r^{e_k})^n = w^n * (r^{e_k})^n \ mod \ n^2 \tag{4.15}$$

$$(w * r^{e_k})^n = (w * r^{e_k})^n \ mod \ n^2 \tag{4.16}$$

Notar que para cada voto encriptado es necesario hacer cálculos de 3 veces el conjunto de todos los mensajes válidos.

4.1.3.5. Votos válidos

Para una votación con n candidatos y la opción de marcar hasta k preferencias, el tamaño del conjunto de los votos válidos corresponde a:

$$votos\_validos(n,k) = 1 + \sum_{i=0}^{k} \binom{n}{i} \tag{4.17}$$

El uno equivale al voto nulo. La sumatoria corresponde a la suma de cada combinatoria dejando $0, 1, \ldots k$ blancos

Al ser una sumatoria de combinatorias, se puede ver que el resultado crece exponencialmente si k es muy grande. De hecho ya con $k = 2$ el resultado es varias veces más grandes

que con $k = 1$. Esto se puede ver en el siguiente gráfico con $k = 1$, $k = 2$ y $k = 3$, para $n = 1$ hasta $n = 100$.

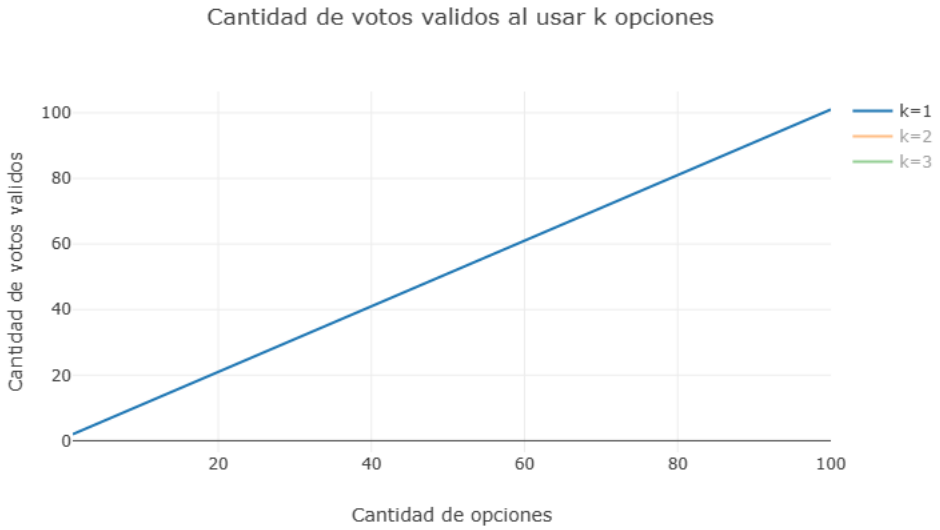

Figura 4.8: Cantidad de votos validos para $k = 1$

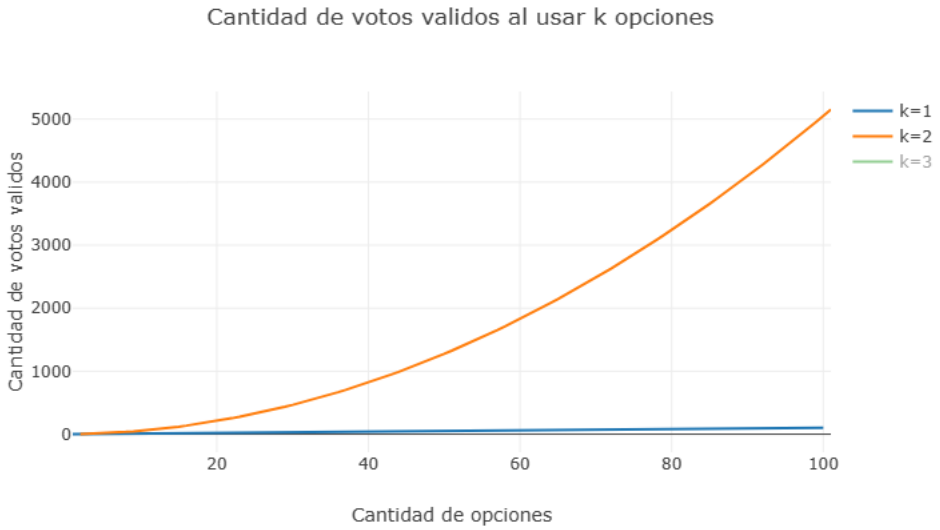

Figura 4.9: Cantidad de votos validos para $k = 1$ y $k = 2$

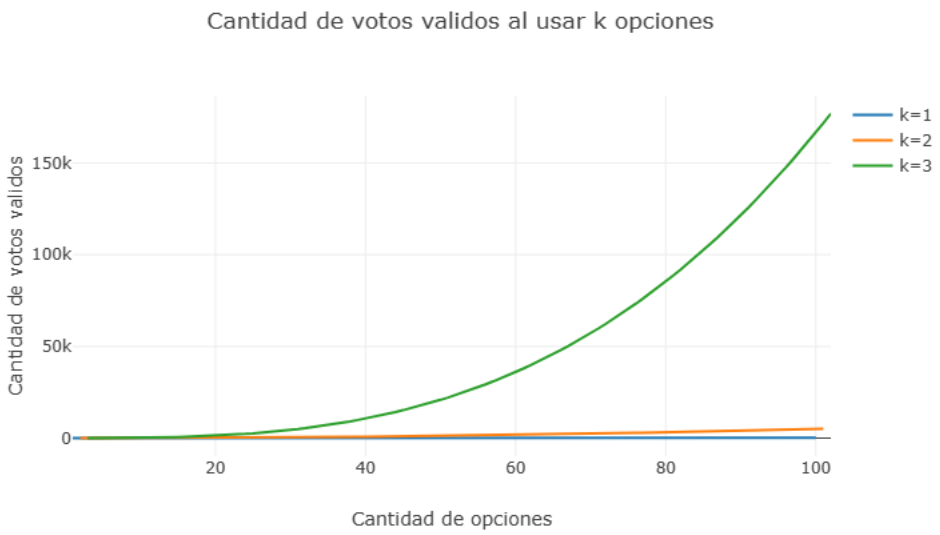

Figura 4.10: Cantidad de votos válidos para $k = 1$, $k = 2$ y $k = 3$

Puede verse que con $k = 2$ el resultado es varias veces más grande que con $k = 1$ y a su vez, $k = 3$ es mucho más grande que con $k = 2$. Además se observa que con $k = 1$ la curva es lineal, pero en la medida que aumenta k esta se va volviendo cada vez más exponencial.

Cabe recordar que la técnica de ZKP propuesta requiere un cálculo para cada uno de los votos válidos posibles.

4.2. Implementación actual

La implementación de Paillier desarrollada en Scala por EVoting contiene varias clases, que se pueden dividir en cuatro categorías: Paillier, Mensajes, Números primos y las Llaves. Estas son:

1. Primes:
    a) PrimesGenerator
2. Mensajes:
    a) PlainMessage
    b) EncyptedMessage
3. Llaves :
    a) PublicKeyLike
    b) PrivateThresholdKeyZKP
    c) KeyGeneratorZKP

4. Paillier:

    a) PaillierLike

    b) Paillier

    c) PaillierZKP

    d) EncryptionCommitmentZKP

    e) ThresholdDecryptionCommitmentZKP

`PrimeGenerator` corresponde a los generadores de números primos que se van a utilizar para encriptar. Estos números se obtienen de una lista lazy infinita (*stream* en Scala) donde se van extrayendo números aleatorios grandes hasta que se encuentran los números que cumplan con las categorías definidas en 2.2.3 y en 2.2.4.

`PlainMessage` y `EncryptedMessage` son los tipos de mensaje sin encriptar y encriptados respectivamente, en ambos casos representados como BigInt.

Las clases de llaves se utilizan para encriptar, desencriptar y generar llaves. El tipo `PublicKeyLike` representa una llave pública, el tipo `PrivateThresholdKeyZKP` representa una llave parcial y `KeyGeneratorZKP` es un Factory para generar la llave pública y las llaves privadas parciales.

Nos quedan las clases que representan el algoritmo de encriptación y desencriptación. Tenemos tres clases: `PaillierLike`, `Paillier` y `PaillierZKP`.

`PaillierLike` define el cuerpo básico de las clases Paillier, `Paillier` define las funciones que lo componen, como encriptar, sumar y desencriptar. `PaillierZKP` define el funcionamiento específico relacionado con el ZKP, como generar las pruebas de ZK.

`PaillierZKP` se define por una llave pública y una lista de votos válidamente emitidos. Aparte de esto, tiene la función `encryptWithZKP` que encripta el mensaje de texto plano junto con generar la prueba de ZK. Está la función `verifyZKP` que verifica la prueba generada por `encryptWithZKP`. Este prueba, o también llamada *commitment* es representada por la clase `EncryptionCommitmentZKP`, que posee tres valores a_s, e_s y z_s, que se definieron en 4.1.3.

El funcionamiento de este sistema, empieza generando la llave con `KeyGenerator`, que crea la llave pública y las llaves privadas parciales. Estas son creadas utilizando el objeto `PrimeGenerator`, por medio de sus funciones `getSafePrimeStream` y `getBigIntRandomStream`. Con estas llaves se crea Paillier.

Para encriptar se generan números enteros utilizando `getBigIntRandomStream`, que crea un número entero grande, que será el número r. Este número r en conjunto con la llave pública, se utilizan para encriptar.

Por último, PaillierZKP genera las pruebas ZK llamando a `getBigIntRandomStream` para generar z_k.

4.3. Solución con TypeScript

4.3.1. TypeScript

TypeScript (TS) fue la primera opción en esta memoria, debido a permitir tipeo explícito en Javascript (JS), lo que permite su una programación más segura.[37].

Si bien TypeScript es una herramienta útil, es difícil traducir el código original, puesto que este posee rasgos y características de Scala que no son directamente traducibles a JS o TS se detalla a continuación:

4.3.2. Descripción solución

La idea era escribir el código original en código JS con TypeScript, para definir tipos como se hacían en Scala.

En el código original en Scala, se hace conversiones de BigInt a String, utilizando por medio la conversión a ByteArray. Dado que es una conversión que se utiliza en todo el código, se priorizó empezar por allí.

Para esto, se definió una función que transformara un array de Bytes a un entero grande, puesto que esta función no estaba definida en JS estándar, ni se encontró ninguna librería que hiciese la conversión requerida. La función creada se llama toByteArray. Nótese que en JS, Int8Array es el equivalente a un ByteArray de Scala.

Código 4.1: Función auxiliar BigInt a ByteArray

```
protected toByteArray(value:bigint):Int8Array{
    var VstringOriginal=value.toString(2);
    var largo=Math.floor(VstringOriginal.length/8)+1;
    var array=new Int8Array(largo);

    var largoInt32=Math.floor(VstringOriginal.length/32)+1;

    var array32Int= [];
    array32Int.push(VstringOriginal.slice(0,VstringOriginal.length %32));
    // De 0 son los bits mas significativos, De largoInt32-1 son los bits menos significativos.
    for (var x=1;x<largoInt32;x++){
        array32Int[x]=VstringOriginal.slice(VstringOriginal.length %32 +32*(x-1),
 ↪ VstringOriginal.length %32 +32*(x))
    }

    var extra=Math.floor((VstringOriginal.length %32)/8)
    var stringUtilizar=array32Int[0];
    var indice=0;

    for(var x=0;x<largo;x++){
        array[x]=parseInt(stringUtilizar,2)>>8*extra;
        extra--;
        if(extra*8<0){
```

```
            indice++;
            stringUtilizar=array32Int[indice];
            extra=3
        }
    }
    return array;
}
```

También se creó una función que hace la conversión de BigInt a String utilizando la función toByteArray. Además de crearse una función que convierte un String a un BigInt.

Código 4.2: Conversión BigInt a String

```
protected encodeBase64(value:bigint):string{
    var array=this.toByteArray(value);
    return Buffer.from(array).toString('base64');

};
```

Código 4.3: Conversión String a BigInt

```
protected decodeBase64(value:string):bigint{
    var res=Buffer.from(value, 'base64');
    var array=new Uint8Array(res);
    var total=0n;
    var indice=BigInt(array.length-1);
    for (var i in array){
        total=total+BigInt(array[i])*(2n**(indice*8n));
        indice--;
    }
    return total;
};
```

Al definir estas funciones, se programó `PlainMessage`, `EncryptedMessage`, el paquete `Primes`, y por último `PrimeGenerator`.

4.3.3. Limitaciones

Al avanzar con la solución, se notaron limitaciones a TypeScript.

Primero, tiene que ver con los tipos de Scala versus los tipos de TypeScript / Javascript. En JS y en TypeScript no existe los tipos como Int, Float, Byte. Para hacer que existan estos tipos de números, hay que definirlos manualmente. Pero existen otras soluciones que realizan que el proyecto tenga las limitaciones de Scala, sin necesidad de definirlas manualmente. Esto se ve en el punto 4.4.

Segundo, la falta de librerías para los enteros grandes. Dado que la implementación de

enteros grandes era reciente, no muchas librerías la habían implementado. Si bien existían algunas que las utilizaban, eran limitadas respecto a sus aplicaciones, usadas para modo general y no para los usos específicos que se necesitaban.

Un tema importante tiene que ver con generar un Stream de enteros grandes. En el código original, esta se generaba por medio de programación funcional, utilizando evaluación *lazy* con el propósito de no llenar la pila, y ser más eficiente. Esto se realizaba con la librería fs2 [40] que no tiene un equivalente en Javascript.

También en el código de Scala de la implementación se hace un uso extensivo de las librerías de abstracción cats [41] y cats-effect [42]. Abstracciones que no existen en TypeScript ni JavaScript lo que no permite una traducción directa.

4.4. Solución con Scala-js

4.4.1. Scala-js

Scala corresponde a un lenguaje orientado a objetos (OO) con programación funcional tipeado estáticamente. Su código puede ser compilado a Bytecode y ejecutado en una máquina virtual de Java (JVM). Provee interoperabilidad con Java, por lo que comparte muchas de sus funciones y clases [43].

Scala-js es una implementación de Scala, que compila el código Scala a Javascript, por lo que puede ser ejecutado tanto en navegador como en Node-js. Además, al ser compilado en JS, puede utilizar módulos de JS [38].

Se escogió finalmente utilizar Scala-js debido a que preserva el código de Scala; por su velocidad de ejecución (más rápido que TypeScript o Javascript plano) al estar ampliamente optimizado [44], además que las pruebas iniciales mostraron también que permite utilizar las características de Scala sin problemas.

La implementación de Paillier de EVoting se construye con la herramienta de construcción SBT. Para compilar a JavaScript se agregó a la biblioteca de Pailler en Scala el plugin Scala-js en build.sbt:

Código 4.4: Habilitar plugin Scala-js

```
1   lazy val paillier=project.in(file("."))
2 .enablePlugins(ScalaJSPlugin)
```

Además, en el archivo pluginst.sbt, se agrega la siguiente línea:

Código 4.5: Habilitar plugin Scala-js 2

```
1   addSbtPlugin("org.scala-js" % "sbt-scalajs" % "1.13.0")
```

Se pueden crear módulos de JS, al cambiar la configuración del proyecto de Scala con el siguiente comando:

Código 4.6: Configuracion Scala-js

```
scalaJSLinkerConfig ~= {
  _.withModuleKind(ModuleKind.ESModule)
    .withModuleSplitStyle(
      ModuleSplitStyle.FewestModules)
},
```

Esto permite que el código de Scala se pueda dividir en distintos en módulos de JS.

4.4.2. Vite

Es una herramienta de desarrollo de software para aplicaciones web. Debida a su facilidad para utilizar módulos y juntarlos en *bundles*, y su facilidad para conectarlo con Scala-js, fue ideal para el desarrollo de esta memoria [45].

Vite se utiliza finalmente para crear un servidor, y poder subir los cambios del código fácilmente.

4.4.3. Slinky

Slinky es una API, que permite escribir en Scala-js código React como si se estuviese escribiendo en JS [46].

Esto permite crear el frontend de páginas web sin necesidad de escribir en JS, lo que permite operar mejor con la lógica de Scala, sin necesidad de intermediarios [46].

Sin bien se consideró su uso durante un momento del proyecto, eventualmente se descartó y se decidió utilizar funciones auxiliares que conectan Scala-js con JS. Principalmente por temas de diseño, para separar la capa de lógica de la capa de presentación.

4.4.4. Descripción Solución

Para implementar el código en Scala-Js, se debieron hacer algunos cambios al código original.

Dado que el código originalmente se ejecutaba en el servidor y en particular utilizando JVM, había una diferencia importante respecto al código en JavaScript, y esta era respecto a los hilos de ejecución.

El código en Scala hace un uso extensivo de multithreading para mejorar el performance. Sin embargo, Javascript solo hace uso de un *thread* de ejecución, por lo que no se permite la paralelización real del código [47].

El cambio principal consistió que todos los procesos en espera se ejecutan uno tras otro de forma secuencial. Cambiando todos los procesos, que podían ser síncronos o asíncronos, a solo procesos síncronos.

Esto es representado en el código original cambiando IO, por SyncIO.

Código 4.7: Código original

```
// Generate a Prime number using a Future: This does not compute the value, just define
   ↪ the computation in the future.
 private def generatePrimeNumber(bitLength: Int): IO[BigInt] =
   ↪ IO.blocking(BigInt(bitLength - 1, certainty, new SecureRandom()))

 ...
 // Takes the Stream and finds the safe primes
 def getSafePrimesStream(bitLength: Int): Stream[IO, BigInt] =
   Stream(getPrimeStream(bitLength).find(p => (2 * p +
   ↪ 1).isProbablePrime(certainty))).repeat.parJoinUnbounded
```

Código 4.8: Código nuevo

```
// Generate a Prime number using a Future: This does not compute the value, just define
   ↪ the computation in the future.
   private def generatePrimeNumber(bitLength: Int): SyncIO[BigInt] =
   ↪ SyncIO(BigInt(bitLength - 1, certainty, new SecureRandom()))

 ...
   def getSafePrimesStream(bitLength: Int): Stream[SyncIO, BigInt] =
   Stream(getPrimeStream(bitLength).find(p => (2 * p +
   ↪ 1).isProbablePrime(certainty))).repeat.flatten
```

El segundo cambio que ocurrió, es que al utilizar encriptación en el navegador, se utiliza una versión de Paillier con ZKP. Esta versión de PaillierZKP, ya estaba implementada por EVoting, y lo que faltaba era utilizarla. PaillierZKP comprobaba que el voto estuviese en una lista de votos válidamente emitidos. Esta lista era calculada por una función creada para esta memoria. Esta calculaba la cantidad de permutaciones, desde 0 hasta k blancos, en votos de largo $n + 2$. También calcula el voto nulo.

Código 4.9: Código para generar todas las opciones válidas

```
   @JSExport
     def generateValidMessages(n:Int, k:Int):List[PlainMessage]={
     // (amount 0)(000)
     //Obtener todas las permutaciones posibles
     val kValue =List.range(1, k+1)
     val example=kValue.map{e => ("1"*e) + ("0" * (n -e))}
     val iterator=example.map{e => e.permutations}

     //Con k blank (Voto totalmente en blanco)
     val blankV=List(k.toString+"|0|" + ("0"*(n)))

     val blankList=List.range(0,k)
```

```
13      val m=kValue.map{e => iterator(k -e).toList}
14      val z=blankList.map{e => m(e).map{f => (e).toString +"|0|" + f}}.flatten
15
16      // Con null
17      val nullVote=List("0|1|"+("0"*n))
18
19      val totalList:List[String]=nullVote.head :: blankV.head :: z
20      val possible_messages= totalList.map(e => Plaintext(encodeToBigInt(e, n)))
21      possible_messages
22  }
```

Esta crea una lista de strings que representan los mensajes válidos. Con cada string representando un voto válido. Luego, cada uno de estos string es convertido en un entero grande utilizando la función `encodeToBigInt(str,n)`. Para cada entero grande se crea un `PlainMessage` con este valor. Al hacerlo con todos los valores. finalmente, se entrega una lista con `PlainMessages`.

La función `encodeToBigInt(str,n)`, recibe un string y lo transforma en un entero grande utilizando la definición de los votos dada en 4.1.1. La función transforma cada substring del string "voto", en un entero grande que es desplazado $32 * i$ hacia la izquierda dependiendo de su posición. Eventualmente, todos los números que salen se suman y se obtiene el entero grande. El código es de la forma:

Código 4.10: Codificador de strings a BigInts

```
1   def encodeToBigInt(str:String, n:Int)={
2     //Transforma un string en un bigint
3     val strArray=str.split('|')
4     val blank=strArray(0)
5     val nullV =strArray(1)
6     val rest=strArray(2).split("")
7     val indices=rest.indices
8     var bigint=indices.map{i => BigInt(rest(i)) << (32 * (n -1 -i))}.reduce((a,b) => a+b)
9     bigint = (BigInt(blank) << (32*(n+1))) + (BigInt(nullV) << (32*(n))) + bigint
10    bigint
11  }
```

Por último, se generaban a partir de las clases y objetos de Scala, módulos de JS optimizados.

Esto se lograba, activando el plugin Scala-js, y además agregando unas etiquetas a cada función importante. Esta etiqueta `@JSExportTopLevel(funcion, name)`, define el nombre del módulo, y la función que se exporta a JS.

Se definieron dos módulos: `benchmark.js` y `encrypt.js`, que corresponden a la función de medir el tiempo y de encriptar, respectivamente.

Código 4.11: Módulo Benchmark

```
  @JSExportTopLevel("BenchmarkPlot",moduleID="benchmark")
object BenchmarkPlot{
  ...
}
```

Código 4.12: Módulo Encript

```
  @JSExportTopLevel("encrypt", moduleID="encryption")
 def encrypt(toEncrypt: String, n:Int, k:Int, l:Int, w:Int): String = {
 ...}
```

Luego cada módulo, puede ser llamado al ser importado como si fuese cualquier otro módulo de JS.

4.5. Diseño final

Para su uso en la aplicación web, se crearon dos objetos en Scala `webapp-paillier` y `BenchmarkPlot`. La primera clase es la que se conecta con la aplicación web, y posee una función importante llamada `encrypt()`, que genera el sistema Paillier, las llaves y realiza la encriptación.

Por otro lado, BenchmarkPlot genera pruebas, y es la que genera la cantidad de opciones posibles. Esta posee cuatro funciones importantes: `GenerateValidMessages`, `calculateTime`, `encodeToBigInt` y `plot`. La primera función genera los votos válidos. La segunda calcula el tiempo de generar estos votos válidos.

`EncodeToBigInt` es la transforma los strings a enteros grandes y `plot` es la conexión externa con JS.

Se crearon seis páginas web HTML. `index.html` corresponde a la entrada a las otras páginas. `benchmark.html`, corresponde a un formulario, en el cual se definen los parámetros de los gráficos, como la cantidad de candidatos y la cantidad de iteraciones. Esta envía los datos a la página `benchmarkPlot.html`. Y `benchmarkPlot.html` , se encarga de realizar los gráficos utilizando los parámetros entregados por `benchmark.html`.

Por otro lado, está `encriptacion.html` que corresponde a un formulario que define los parámetros de encriptación. Estos son enviados a la página `votacion.html`, la cual genera la encriptación.

Por último, está la página `resultados.html` que posee los resultados de encriptaciones pasadas graficados para distintos parámetros. Estos son los que se utilizan en el capítulo 5.

4.5.1. Páginas Web

En la página `encriptacion.html`, se insertan los parámetros para realizar la encriptación. Estos corresponde a n: la cantidad de candidatos, k: marcas disponibles, l la cantidad de llaves privadas, y w el umbral para desencriptar.

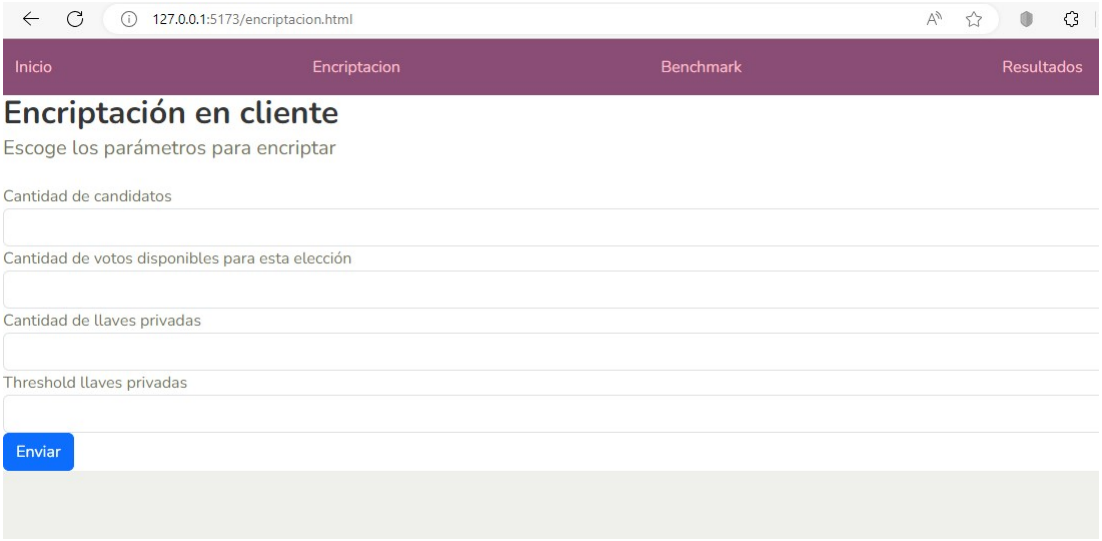

Figura 4.11: Definición de parámetros para encriptación

Luego estos datos son enviados a la página `votacion.html`, donde son utilizados para crear el sistema Paillier. Al apretar el botón se encripta el número ingresado en la barra, con los parámetros definidos en la página anterior.

Figura 4.12: Ingreso de valor a encriptar

Por otro lado, para `benchmark.html`, tenemos otro formulario en donde se ingresan los valores para generar el gráfico de tiempo.

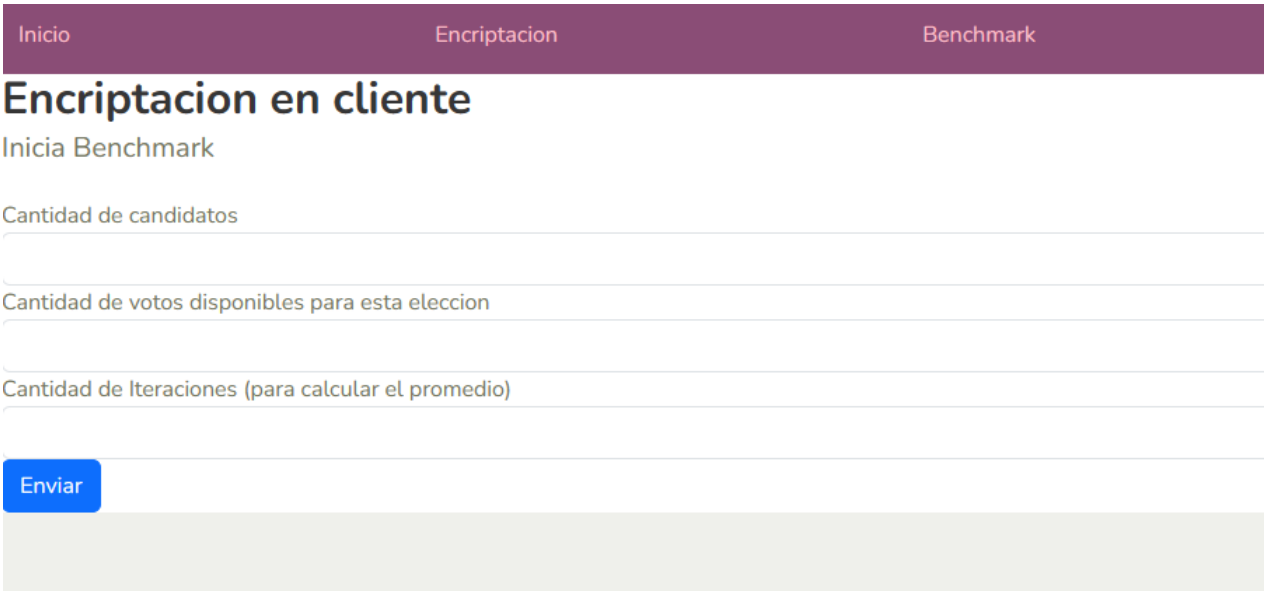

Figura 4.13: Definición de los parámetros de pruebas

Aquí al ingresar el botón, somos dirigidos a la página `benchmarkPlot.html`, donde se ve se genera un gráfico de la medición de tiempo con los parámetros especificados.

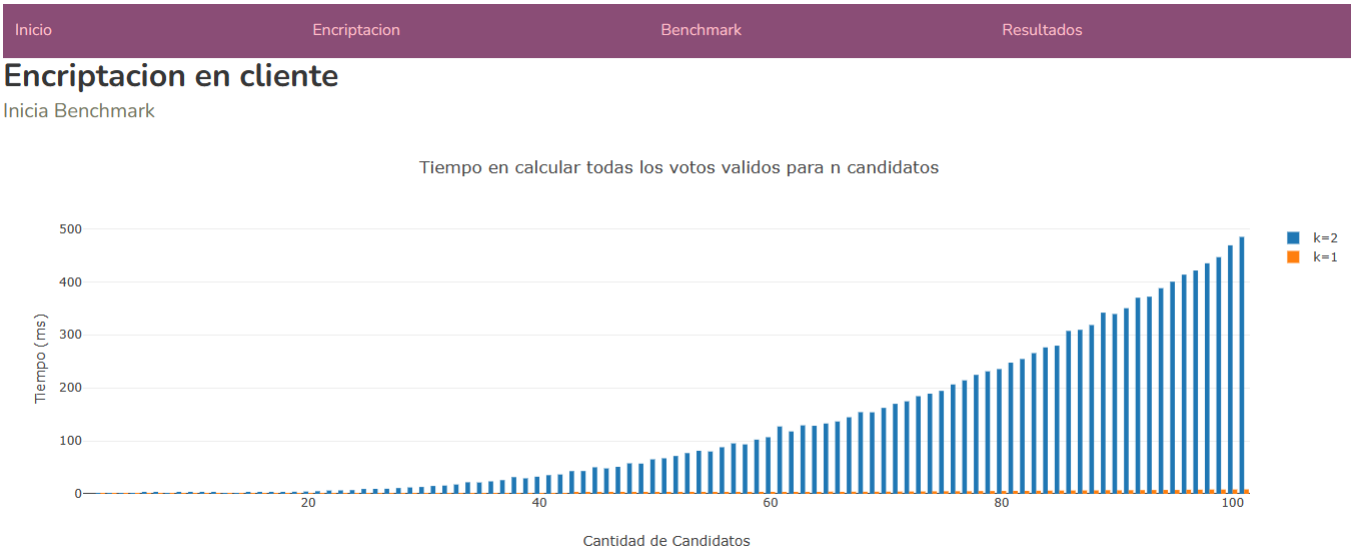

Figura 4.14: Resultados de las pruebas

Por último, está la página `resultados.html`. Aquí se guardan los resultados de pruebas anteriores.

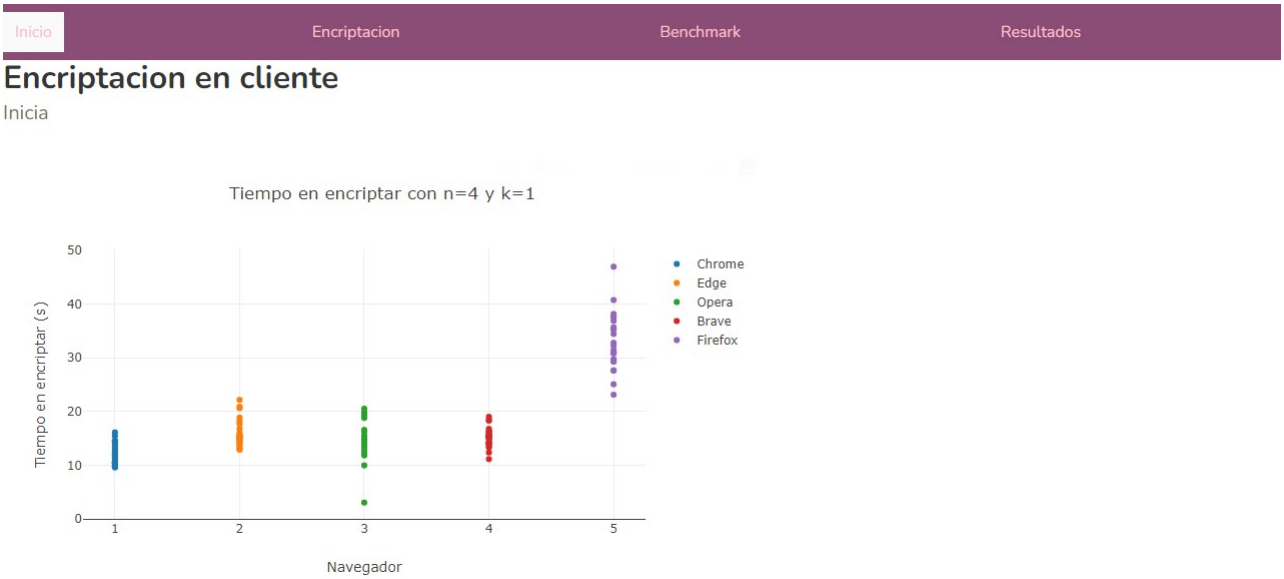

Figura 4.15: Resultados de las pruebas anteriores

4.5.2. Funciones auxiliares

Para poder utilizar los módulos de Scala.js, se crearon archivos auxiliares, que reciben los parámetros de los formularios, y llaman a los módulos de Scala-js con estos parámetros.

La página `votacion.html` tiene un script llamado `main.js`. Esta llama a `encrypt(toEncrypt,n, k,l,w)` del módulo `encryption.js`.

`encryption.js` recibe como parámetros n, k, l, w, y por último, el valor a encriptar (toEncrypt). Los primeros cuatros valores son utilizados para crear las llaves y el sistema. El valor `toEncrypt` es el valor de texto plano a encriptar.

La función `encrypt` nos entrega un JSON del texto cifrado entregado, en conjunto con la prueba.

Código 4.13: Función que llama encrypt

```
1   import {encrypt} from '../target/scala-2.13/paillier-fastopt/encryption.js'
2
3 var data=window.location.href.split("?");
4 data=data[1].split("&");
5
6 var n=Number(data[0].split("=")[1]);
7 var k=Number(data[1].split("=")[1]);
8 var lt=Number(data[2].split("=")[1]);
9 var wt=Number(data[3].split("=")[1]);
10
11 document.querySelector('button').addEventListener('click',encryptBrowser)
12
13 function encryptBrowser(){
14    var toEncrypt=document.getElementById("toEncrypt").value
```

```
    val result=encrypt(toEncryp,n,k,lt,wt)
    console.log(result)
}
```

Por el lado de Scala, **webapp-paillier** recibe los parámetros que le entrega **main.js**. Esta función genera las llaves, el sistema Paillier, y realiza la encriptación a **toEncrypt**. Esta retorna un JSON del objeto encriptado.

Código 4.14: Código para encriptar en Scala

```
@JSExportTopLevel("encrypt", moduleID="encryption")
 def encrypt(toEncrypt: String, n:Int, k:Int, l:Int, w:Int): String = {

   var possible_messages:List[PlainMessage]=generateValidMessages(n,k)
   val keysIO: SyncIO[Either[Throwable,Vector[PrivateThresholdKeyZKP]]] =
    ↪ KeyGeneratorZKP.genThresholdKeys((n+2)*32, l, w).map(Right(_))

   val plaintext= Plaintext(BigInt(toEncrypt))
   val resultIO:EitherT[SyncIO,Throwable, EncryptedMessageWithCommitmentZKP] = for {
      keys <- EitherT(keysIO)
      paillierSystemZKP <- new PaillierZKP(keys.head.publicKey,possible_messages)
      ciphertext    <- EitherT(paillierSystemZKP.encryptWithZKP(plaintext))
    } yield (ciphertext)

   val result=resultIO.value
   val res=result.unsafeRunSync()
   val res2= res match{
      case Left(err) => ""
      case Right(enc) => writeToString(enc)
    }
   val endTime=System.currentTimeMillis()
   val r=endTime-startTime
   res2
 }
```

Por otro lado, **BenchmarkPlot** tiene un script llamado **benchmark.js**. Este llama a la función **plot(n,k,iter)** del módulo **Benchmark**. Los parámetros n, k son la cantidad de candidatos, y las marcas disponibles. El parámetro **iter** corresponde a la cantidad de iteraciones necesarias para calcular el promedio.

Código 4.15: Función que llama plot

```
   import {BenchmarkPlot} from '../target/scala-2.13/paillier-fastopt/benchmark.js'
import {combinations } from 'mathjs'
import Plotly from 'plotly.js-dist-min'

```

```
var data=window.location.href.split("?");
data=data[1].split("&");

var n=Number(data[0].split("=")[1]);
var k=Number(data[1].split("=")[1]);
var iter=Number(data[2].split("=")[1]);

var time=BenchmarkPlot.plot(n,k,iter);
```

Por el lado de Scala, **plot** hace una medición del tiempo respecto a una función, en este caso respecto a calcular la cantidad de votos válidos.

Código 4.16: plot

```
  @JSExport
  def plot(n:Int, k:Int, iter:Int):Array[Double]={

  val lTime=List.fill(n+1)(0d)
  val time=calculateTime(k,n,iter,lTime)
  time.toArray
  }
```

El código de **calculateTime**, hace mediciones entre que se llama la función y esta termina. Por ejemplo utilizando la función **generateValidMessages**.

Código 4.17: CalculateTime

```
  def calculateTime(k:Int, n:Int, iter:Int, lTime:List[Double]):List[Double]={
     var t=0d
     var newList=lTime
     var newList2=lTime.take(n+1)
    for (i <- 1 to n){
       for (j <- 0 to iter){
        t=time(generateValidMessages(k, i))
        val prevValue=newList(i)
        newList2=newList.updated(i, prevValue+t)
        newList=newList2.take(n+1)
      }
     }
     newList=newList.map{e => e/(iter+1)}
     newList
   }
```

4.6. Testing

Para probar el código se utilizó las pruebas existentes adaptadas a Scala-js y se creó una prueba extra para testear a `generateValidMessages()`.

Para las pruebas ya existentes se cambió el parámetro %% en build.sbt, con %%% que refiere a que es una librería de Scala-js. Se usaron entonces las librerías de scalatest y cats-effect-testing-scalatest.

Código 4.18: Importar librerías de testing para Scala-js

```
  libraryDependencies ++= Seq(
 "org.scalatest" %%% "scalatest"                  % "3.2.10" % Test,
 "org.typelevel" %%% "cats-effect-testing-scalatest" % "1.3.0"  % Test
),
```

Ahora, respecto al testing del código agregado, se definió una prueba para probar que la función `generateValidMessages()` funciona. Lo que se quiere probar, es que este valor genera exactamente todos los posibles votos válidos. Por lo que se creó una clase de Testing llamada `ValidMessagesZKPTest` con scalatest.

Para esto, la prueba con valores pequeños comprueba que en efecto se generan todas las opciones posibles. Es decir, revisar de 1 a 1, y demostrar que en efecto todas las opciones generadas son válidas, y segundo, que no se ha perdido ninguna. Estas pruebas se hicieron para valores $n = 2$, $n = 5$, $n = 10$ y $n = 16$ con $k = 1$, $k = 2$ y $k = 4$

Para valores grandes se compara el valor numérico, definido en 4.1.1, con la cardinalidad del resultado. Si estos valores son iguales para todos los resultados, se asume, que se está generando todas las opciones posibles para cualquier input n, k dado.

Código 4.19: Prueba valores pequeños

```
   "The list of all possible messages for k=1" - {
  "have all the required elements for n=2" in {
    val listSize_2=generateValidMessages(2,1).sortBy(e => e.toBigInt)
    val supposedMessages=scala.collection.immutable.List(Plaintext(BigInt(1)),
    Plaintext(BigInt("4294967296")),
    Plaintext(BigInt("18446744073709551616")),
    ↪ Plaintext(BigInt("79228162514264337593543950336")))
    assert (listSize_2.sameElements(supposedMessages))
  }
  ...
  }
```

Por otro lado, con valores grandes se hicieron pruebas con $n = 32$, $n = 64$, $n = 128$ y $n = 256$ con $k = 1$, $k = 2$ y $k = 3$.

Código 4.20: Prueba valores grandes

```
  "have the required size for n=64 and k=2" in {
   val listSize_64_2=generateValidMessages(64,2).length
   numericValue=kRange_2.map(e =>
  ↪ fact(BigInt(64))/(fact(BigInt(64-e))*fact(BigInt(e)))).fold(BigInt(1))((x,y)=> x+y)
   assert (listSize_64_2== numericValue)
  }
```

Capítulo 5

Resultados

5.1. Tiempo en generar votos válidos para $k = 1$ y $k = 2$

Al implementar la encriptación de Paillier, lo que se probó primero, es ver el tiempo en que tomaba calcular las opciones. Esto se hizo con $k = 1$ y $k = 2$, con 2 hasta 100 candidatos, promediando entre 100 iteraciones para cada uno. Esto se hizo en Javascript con Plotly, haciendo mediciones con la función `time`.

Código 5.1: Función para medir el tiempo

```
1  def time[R](block: => R) = {
2    val t0     = System.currentTimeMillis()
3    val result = block // call-by-name
4    val t1     = System.currentTimeMillis()
5    (t1-t0).toDouble
6    }
```

Esta se llamaba después de crear la instancia de Paillier y la generación de llaves, cuando se encriptaba el texto plano. Los resultados se obtuvieron en Chrome 144.

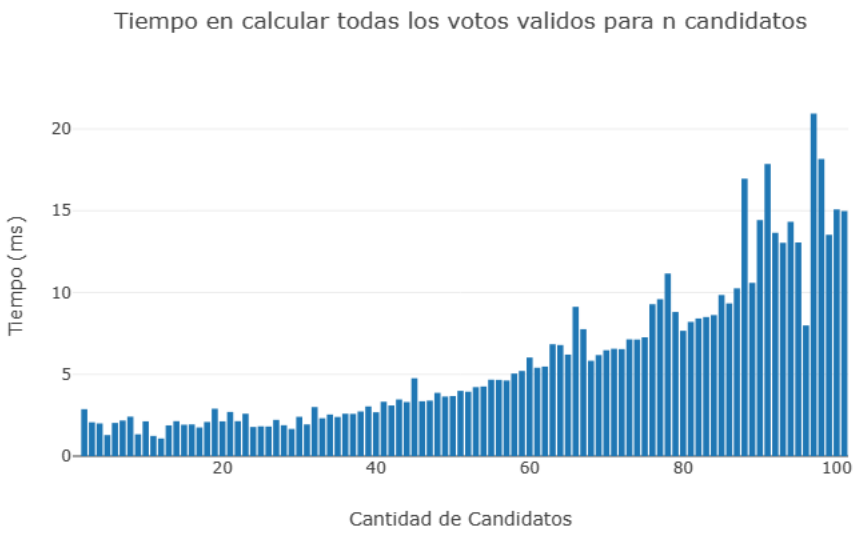

Figura 5.1: Tiempo en calcular votos válidos para $k = 1$

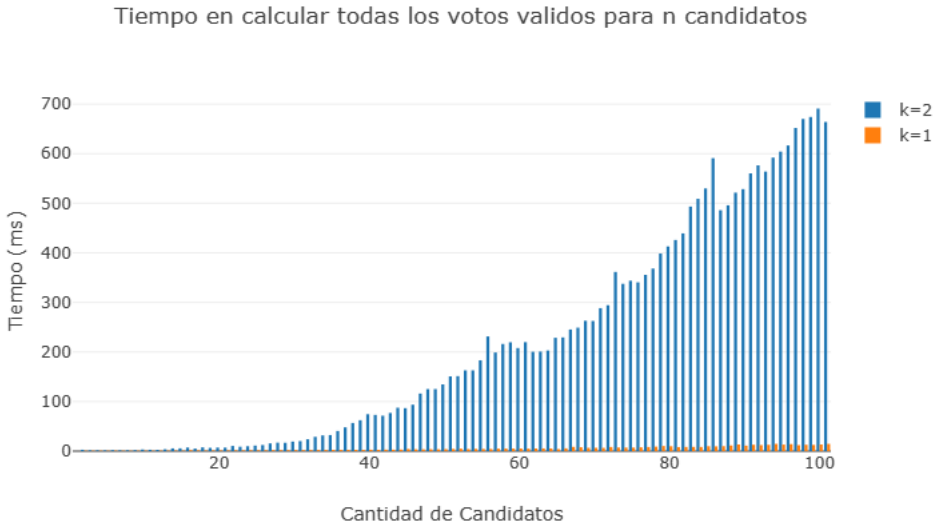

Figura 5.2: Tiempo en calcular votos validos para $k = 2$ y $k = 1$

5.2. Tiempo de encriptación por Navegador

Por otro lado se comparó el tiempo de encriptación para $n = 4$ y $n = 8$ para distintos navegadores, entre ellos Chrome 144, Edge 144, Brave 1.52, Opera 100 y Firefox 115, con 25 iteraciones cada una. Estos son los resultados:

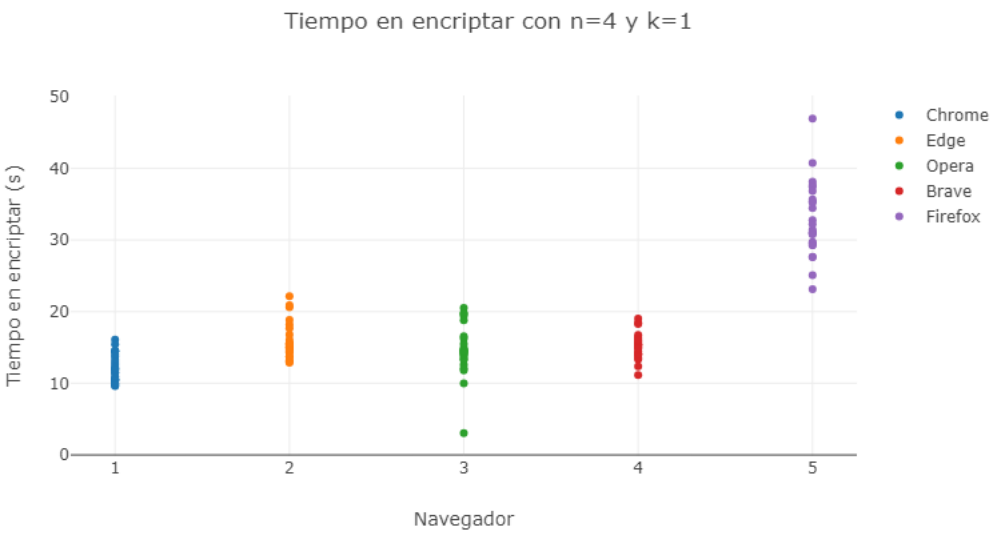

Figura 5.3: Tiempo de encriptación para $n = 4$ y $k = 1$ según Navegador

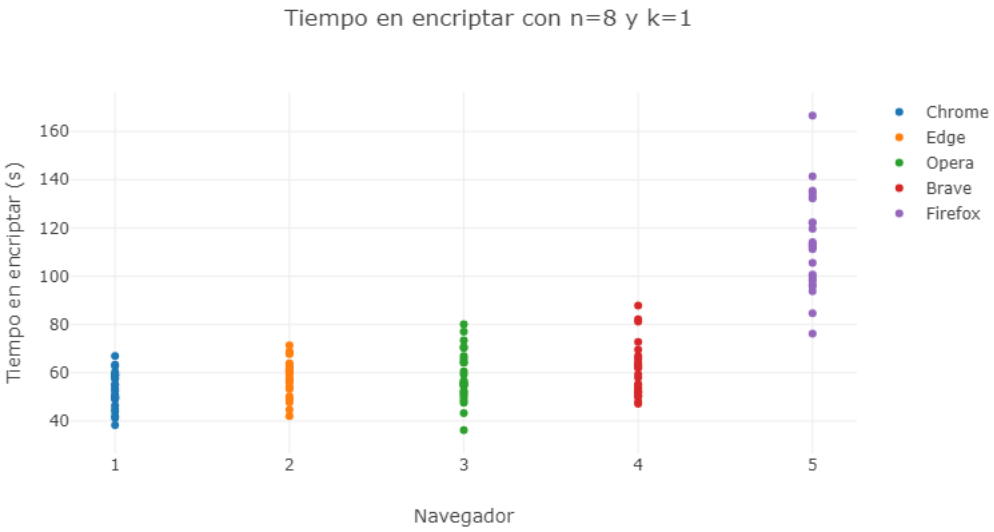

Figura 5.4: Tiempo de encriptación para $n = 8$ y $k = 1$ según Navegador

5.3. Tiempo de encriptación variando n

Seleccionado el navegador se hicieron las pruebas de tiempo de encriptación promediando el resultado de 25 iteraciones para cada n. Las pruebas se realizaron en Chrome 144. Se hizo para $k = 1$, $k = 2$ y $k = 3$. La llave es de tamaño $(n + 2) \times 32$ bits.

Aquí tenemos el resultado para cuando $n = 1$ hasta $n = 32$.

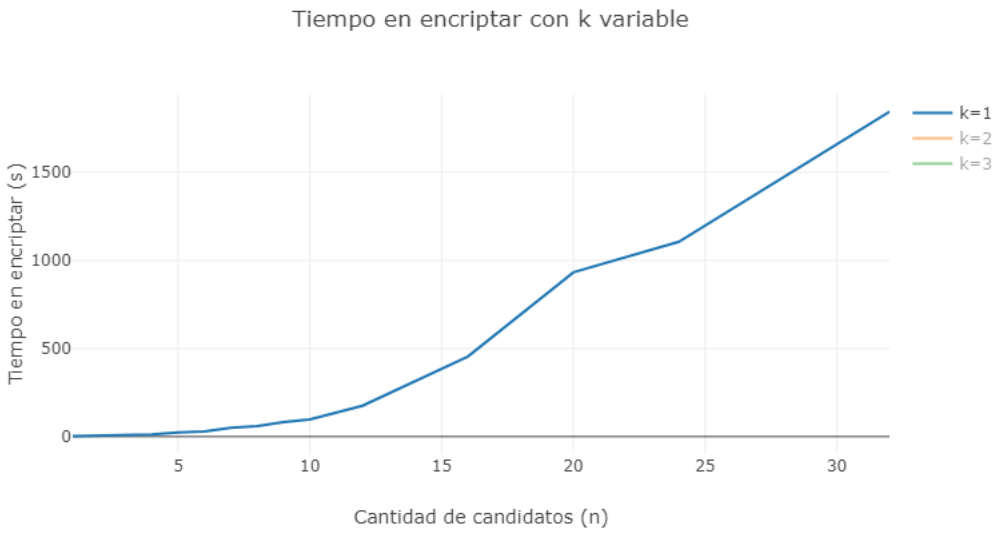

Figura 5.5: Tiempo en encriptar hasta $n = 32$ con $k = 1$

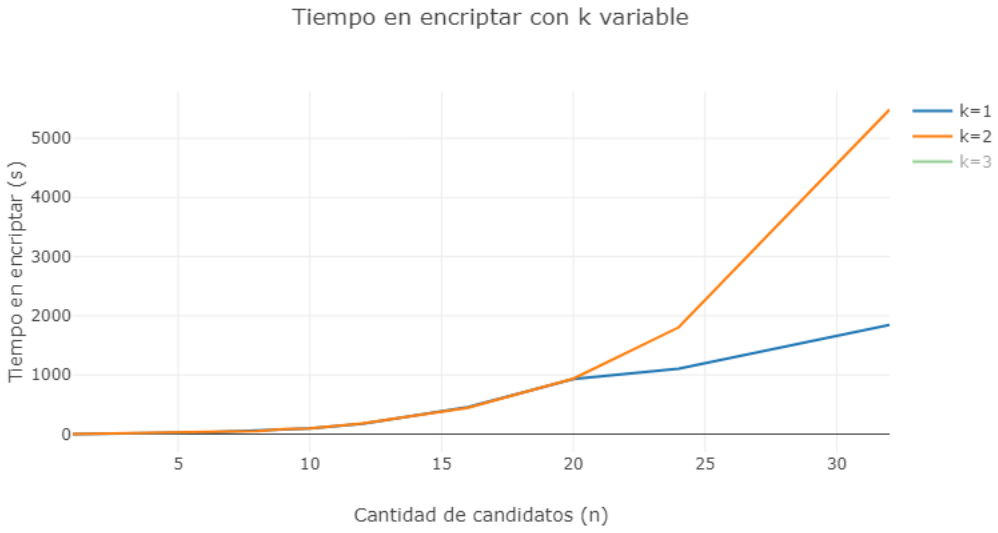

Figura 5.6: Tiempo en encriptar hasta $n = 32$ con $k = 2$ y $k = 1$

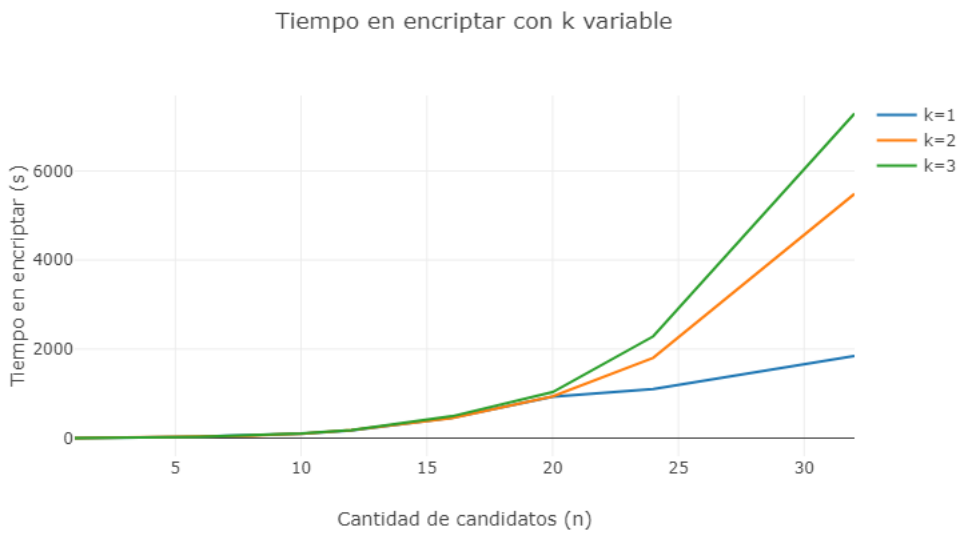

Figura 5.7: Tiempo en encriptar hasta $n = 32$ para $k = 1$, $k = 2$ y $k = 3$

5.4. Comparando con Paillier sin ZKP

También se hizo una comparación entre los resultados obtenidos con PaillierZKP versus Paillier, ambos en el navegador con Google Chrome 144. Esto se hace midiendo el tiempo que demora en encriptar, al aumentar el tamaño de la llave. Para PaillierZKP, se toma el largo de la llave $(n + 2) \times 32$ con n siendo la cantidad de candidatos.

Notar que PaillierZKP también llama la función `encryptWithRandomness` que es parte de la clase Paillier. Por lo que se puede ver la diferencia en tiempo, como cuanto tiempo se demora en calcular la prueba de ZKP.

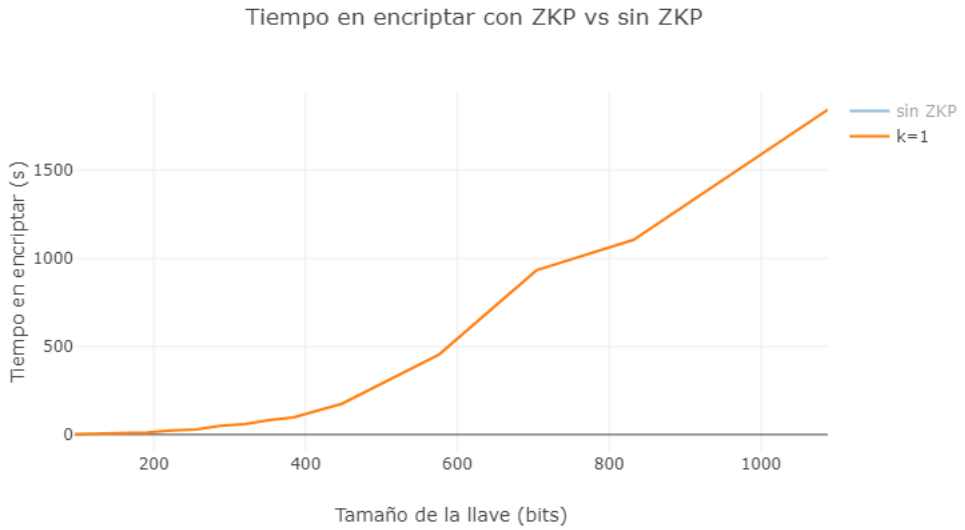

Figura 5.8: Tiempo en encriptar sin ZKP

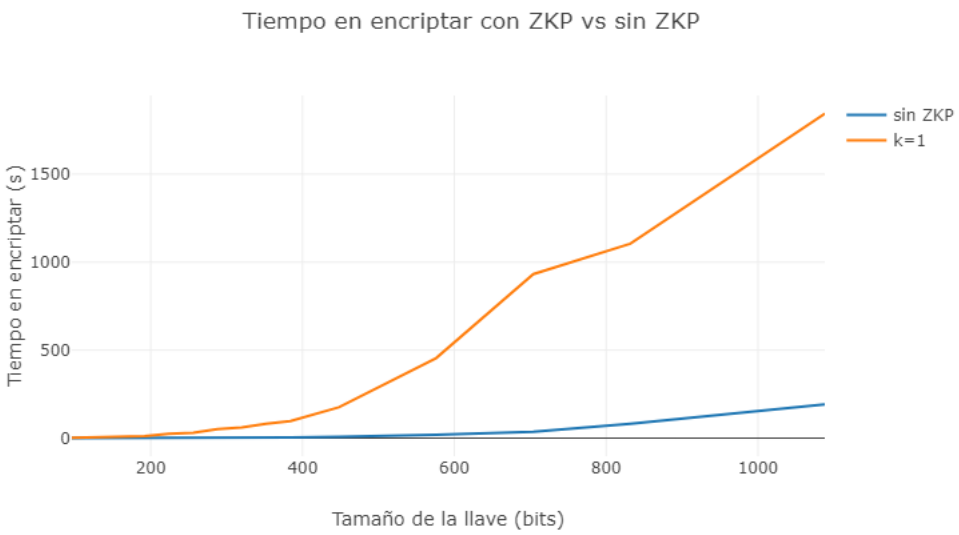

Figura 5.9: Tiempo encriptación ZKP vs no ZKP

Capítulo 6

Conclusión

En esta memoria se implementó a partir de un código escrito en Scala del algoritmo de Paillier para la encriptación de votos y pruebas de ZKP, una implementación equivalente en código Javascript que puede ser ejecutado en el navegador. Se comprobó que la implementación estaba correcta realizando varias encriptaciones y desencriptaciones haciendo match de los inputs con los outputs, además de ejecutar la verificación de las pruebas ZKP. Finalmente, se tomó la implementación en Javascript y se realizaron varias ejecuciones variando los parámetros para medir el comportamiento real en navegadores y concluir sobre la factibilidad de su uso.

6.1. Resultados de pruebas

Era esperable tener un performance más bajo en el navegador versus el servidor dada la menor capacidad de cómputo en el navegador y en particular la restricción monothread del intérprete de Javascript, pero la magnitud de la diferencia se mostró muy grande, con tiempos de espera demasiado largos para que sea una solución válida para usuarios reales.

Al comparar los resultados, tanto al cambiar de navegadores como la variación en el largo del voto (cantidad de candidatos) podemos sacar varias conclusiones.

6.1.1. Navegador

Primero, desde el punto del navegador, se puede ver que hay diferencias al momento de encriptar dependiendo de cuál navegador utilicemos.

Se aprecia que Firefox es mucho más lento, casi el doble del tiempo de los otros navegadores. Chrome es el que menos desviación posee y el que gasta el menor tiempo posible al encriptar. Seguido por Opera, que si bien posee un valor de la misma magnitud que Chrome, su desviación es mayor, por lo que su promedio podría ser mayor al observado. Esto nos indica, que si bien se puede utilizar un navegador u otro, el tiempo en que se demora en encriptar un valor en Firefox, afecta fuertemente la experiencia del usuario, por lo que es poco probable, más bien desaconsejable, que un usuario utilice Firefox para este tipo de encriptación.

6.1.2. Tiempo en encriptar

Al variar el n, aumenta el tiempo de encriptación con un comportamiento lineal para $k = 1$. Sin embargo, al aumentar k vemos que el impacto en el tiempo de encriptación va creciendo fuertemente en la medida que crece n.

Si comparamos el valor de $k = 1$ de PaillierZKP versus el valor obtenido con Paillier, se ve que la diferencia entre ambos valores es de cerca de 1.600 segundos de diferencia, lo que equivale a cerca de 26 minutos de diferencia. Considerando que para una llave de 1024 bits, el valor para Paillier-noZKP es de 192 segundos, equivalente a 3.2 minutos. Este valor es 9 veces más pequeño que el valor con ZKP.

6.2. Objetivos iniciales

Respecto a los objetivos iniciales, se logró una implementación de navegador de Paillier que funciona para todos los navegadores populares.

Sin embargo, el nivel de performance que se alcanza es muy ineficiente, no sería realista pensar que una persona esté dispuesta a esperar 3 minutos para votar, especialmente dado que solo son $n = 32$ candidatos, y esto solo considerando la versión sin ZKP. Si incluimos ZKP, esto es incluso más improbable, esperando cerca de 26 minutos por un solo voto.

Si podría considerarse útil para votaciones con pocos candidatos, y con un k pequeño, que en efecto nos comentan son la mayoría de las elecciones en EVoting. Además, esta versión, tiene la verificación que de que los votos no hayan sido alterados, algo que es un supuesto en el servidor, lo que sirve para las elecciones que tengan este requisito.

6.3. Trabajo futuro

Hay varias partes que podrían cambiarse para hacer el código más eficiente. En la parte de generación de números grandes, se optó por mantener el código lo más parecido posible al original. Una opción, sería buscar una versión que pueda lograr concurrencia.

Otra cosa que se podría cambiar, son las pruebas ZKP, en este caso se ve que para $n = 24$ en adelante, el valor de k afecta más. Para $n = 24$ con $k = 1$ corresponde a 26, y con $k = 3$ corresponde a 2.627, y tienen una diferencia de tiempo grande. Si k sigue aumentando, el tiempo de encriptación se vuelve muy mayor. Podría reducirse este tiempo para k más grandes, sin en vez de hacer pruebas que dependiesen de n y k, estas solo dependieran de n. Los votos válidos tendrían que contarse de otra forma, para que este pudiese ser lineal, en vez de exponencial. Es decir, buscar una fórmula de ZKP que en vez de usar el conjunto de mensajes válidos, pudiera bastar con una prueba de que el voto se encuentre en un rango.

También se puede optar por encriptar cada una de las opciones individualmente en vez del voto como un todo y agregar una prueba que la suma de las opciones es k como lo propone [10]. Esto requiere más encriptaciones para votos con n pequeño, ya que por cada opción hay que realizar la encriptación y la prueba ZKP que es un 0 o un 1, pero en la medida que crece n va creciendo linealmente y no exponencialmente, independiente del valor de k. Se

requerirá, además del análisis de tiempo, analizar el peso en bytes de ese voto para concluir sobre la factibilidad.

Una duda que queda es, ¿Por qué los otros sistemas de votación electrónica son capaces de obtener buenos tiempos comparados a este sistema?. ¿Será por como realizan las pruebas ZKP, o bien por la forma en que generan los números primos? ¿Hay formas más óptimas de generar valores primos, sin que demoren tanto en JavaScript?. Estas son preguntas que se podrían responder en un trabajo futuro.

Otro trabajo que se puede hacer, es en vez de utilizar Scala-js, se podría utilizar WebAssembly. Este, al igual que JS, está implementado en todos los navegadores populares modernos. Además, WebAssembly parece tener mejor optimización que JS, permitiendo que ciertas labores corran más rápido que con JS [48].

Bibliografía

[1] Liaw, H.-T., "A secure electronic voting protocol for general elections," Computers Security, vol. 23, no. 2, pp. 107–119, 2004, doi:https://doi.org/10.1016/j.cose.2004.01.007.

[2] de Chile, S. E., "Servicio electoral.", https://servel.cl/servicio-electoral-de-chile/.

[3] EVoting, "Historia | evoting.", https://www.evoting.com/nosotros/historia/.

[4] Paillier, P., "Public-key cryptosystems based on composite degree residuosity classes," en Advances in Cryptology — EUROCRYPT '99 (Stern, J., ed.), (Berlin, Heidelberg), pp. 223–238, Springer Berlin Heidelberg, 1999.

[5] Will, M. A. y Ko, R. K., "Chapter 5 - a guide to homomorphic encryption," en The Cloud Security Ecosystem (Ko, R. y Choo, K.-K. R., eds.), pp. 101–127, Boston: Syngress, 2015, doi:https://doi.org/10.1016/B978-0-12-801595-7.00005-7.

[6] Panja, S. y Roy, B., "A secure end-to-end verifiable e-voting system using blockchain and cloud server," Journal of Information Security and Applications, vol. 59, p. 102815, 2021, doi:https://doi.org/10.1016/j.jisa.2021.102815.

[7] Dilmegani, C., "Zero-knowledge proof: How it works applications in 2022.", https://research.aimultiple.com/zero-knowledge-proofs/.

[8] Diffie, W. y Hellman, M., "New directions in cryptography," IEEE Transactions on Information Theory, vol. 22, no. 6, pp. 644–654, 1976, doi:10.1109/TIT.1976.1055638.

[9] "Knowledge what? an introduction to zero knowledge - stanford code the change guides documentation.", https://codethechange.stanford.edu/guides/guide_zk.html.

[10] Damgård, I. y Jurik, M., "A generalisation, a simplication and some applications of paillier's probabilistic public-key system," Lecture Notes in Computer Science, vol. 7, 2000, doi:10.7146/brics.v7i45.20212.

[11] "Number theory - modular arithmetic.", https://crypto.stanford.edu/pbc/notes/numbertheory/arith.html.

[12] "Group theory - cyclic groups.", https://crypto.stanford.edu/pbc/notes/group/cyclic.html.

[13] MozDevNet, "Javascript technologies overview - javascript: Mdn.", https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview.

[14] INTERNATIONAL, E., https://www.ecma-international.org/publications-and-standards/standards/?order=last-change.

[15] MozDevNet, "Javascript modules - javascript: Mdn.", https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules.

[16] INTERNATIONAL, E., "Ecmascript 2020 language.", https://tc39.es/ecma262/2020/#sec-ecmascript-data-types-and-values.

[17] MozDevNet, "Bigint - javascript: Mdn.", https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt.

[18] Serrano, J. H., "paillier-bigint.", https://github.com/juanelas/paillier-bigint.

[19] Serrano, J. H., "A node.js implementation of the paillier cryptosystem.", https://github.com/juanelas/paillier-js.

[20] MozDevNet, "Webassembly.", https://developer.mozilla.org/en-US/docs/WebAssembly.

[21] Menendez, L., "Gopaillier.", https://github.com/lucasmenendez/gopaillier.

[22] Ramanathan, N., "Webassembly: Introduction to webassembly using go.", https://golangbot.com/webassembly-using-go/.

[23] Dahlin, T. F., "Paillier cryptosystem.", https://github.com/DaylightingSociety/Paillier.

[24] Kuhlmann, B., "Ruby webassembly.", https://www.alchemists.io/articles/ruby_webassembly/.

[25] Snips, "Paillier.", https://github.com/snipsco/rust-paillier.

[26] ZenGoX, "Zero knowledge paillier.", https://github.com/ZenGo-X/zk-paillier.

[27] MDN, "Compiling from rust to webassembly.", https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm.

[28] Data61, C., "Python paillier library." https://github.com/data61/python-paillier, 2013.

[29] Analytics, N., "javallier.", https://github.com/n1analytics/javallier.

[30] Thorne, B., https://github.com/hardbyte/Paillier.jl.

[31] Adida, B., "Helios voting.", https://github.com/benadida/helios-server.

[32] Elgamal, T., "A public key cryptosystem and a signature scheme based on discrete logarithms," IEEE Transactions on Information Theory, vol. 31, no. 4, pp. 469–472, 1985, doi:10.1109/TIT.1985.1057074.

[33] Adida, B., "Helios: Web-based open-audit voting," en Proceedings of the 17th Conference on Security Symposium, SS'08, (USA), p. 335–348, USENIX Association, 2008.

[34] Belenios, "Belenios specification.", https://www.belenios.org/specification.pdf.

[35] "Participa uchile.", https://participauchile.cl/.

[36] Hevia, A., Gomez, C., Burgos Kreither, C., y Apablaza, M., "Votación electrónica remota para la universidad de chile," Bits de Ciencia, p. 31–35, 2022.

[37] "Javascript with syntax for types..", https://www.typescriptlang.org/.

[38] "Scala-js.", https://www.scala-js.org/.

[39] Taylor Fox Dahlin, D. S., "Paillier zero knowledge proof.", https://paillier.daylightingsociety.org/Paillier_Zero_Knowledge_Proof.pdf.

[40] , https://fs2.io/#/guide.

[41] Allen, C., Baker, R., Bilge, A., Boykin, P. O., Brown, T., Chang, A., Esik, D., Neyens, P., Norris, R., Osheim, E., y et al., https://typelevel.org/cats/.

[42] , https://typelevel.org/cats-effect/.

[43] Scala, "Tour of scala introduction.", https://docs.scala-lang.org/tour/tour-of-scala.html.

[44] "Performance.", https://www.scala-js.org/doc/internals/performance.html.

[45] "Why vite.", https://vitejs.dev/guide/why.html.

[46] "Write react apps in scala just like you would in es6.", https://slinky.dev/.

[47] MozDevNet, "Javascript.", https://developer.mozilla.org/en-US/docs/Web/javascript.

[48] Lawson, L., "Javascript or webassembly: Which is more energy efficient and faster?," 2023, https://thenewstack.io/javascript-vs-wasm-which-is-more-energy-efficient-and-faster/.